

Luděk Skočovský  
UNIX  
POSIX  
PLAN 9



Luděk Skočovský  
**UNIX**  
**POSIX**  
**PLAN 9**



Kniha je určena pokročilým uživatelům operačního systému UNIX,  
správcům výpočetních systémů a počítačových sítí  
a studentům oborů operačních systémů

UNIX, POSIX, Plan 9

© text Luděk Skočovský

© výtvarné práce Vladimír Kokolia

odborný oponent Jan Beran

jazyková korektura Barbora Antonová

vydal Luděk Skočovský, Brno, 1998

skocovsky@hotmail.com

tisk Duo Press, Mnichovo Hradiště

náklad 1000 ks, vydání první

Plan 9 je registrovaná obchodní značka Bell Laboratories

POSIX je registrovaná obchodní značka IEEE (Institute for Electrical and Electronic Engineers)

SVID je registrovaná obchodní značka AT&T (American Telephone and Telegraph Company)

UNIX je registrovaná obchodní značka USL (UNIX System Laboratories)

X/OPEN je registrovaná obchodní značka X/Open Company Limited

X Window System je registrovaná obchodní značka MIT (Massachusetts Institute of Technology)

ISBN 80-902612-0-5

# OBSAH

	<b>ÚVOD ... 7</b>			<b>3</b>	<b>SYSTÉM SOUBORŮ ... 77</b>
<b>I</b>	<b>JÁDRO ... 11</b>			3. 1	Systém souborů z pohledu uživatele ... 80
1.1	Volání jádra ... 12			3. 1.1	Obyčejné soubory ... 82
1.2	Moduly jádra ... 14			3. 1.2	Adresáře ... 87
1.3	Přístup uživatele ... 17			3. 1.3	Symbolické odkazy ... 90
<b>2</b>	<b>PROCESY ... 19</b>			3. 1.4	Zamykání souborů ... 91
2.1	Vznik procesu ... 19			3. 1.5	Svazky ... 92
2.2	Hierarchická struktura procesů ... 21			3. 2	Systém souborů z pohledu jádra ... 93
2.3	Proces z pohledu uživatele ... 32			3. 2.1	I-uzel ... 96
2.3.1	Řídící hodnoty při vzniku dítěte ... 32			3. 2.2	Superblok ... 104
2.3.2	Ukončení dítěte ... 36			3. 2.3	Zaváděcí blok ... 109
2.3.3	Komunikace rodiče a jeho dětí ... 36			3. 2.4	Přenos dat, struktury v jádru ... 111
2.3.4	Démon ... 39			3. 2.5	Manipulace se svazky ... 112
2.3.5	Proces a program ... 41			3. 3	Speciální soubory ... 116
2.3.6	Čas a sledování dětí ... 42			3. 4	Systémová vyrovnávací paměť ... 120
2.4	Proces z pohledu jádra ... 49			3. 5	Archivy dat ... 122
2.4.1	Plánování operační paměti ... 51			3. 5.1	Program <b>tar</b> ... 122
2.4.2	Plánování procesoru ... 54			3. 5.2	Program <b>cpio</b> ... 125
2.4.3	Reálný čas ... 58			3. 5.3	Archivy programu <b>pax</b> ... 127
2.4.4	Registrace procesu jádrem ... 61			3. 5.4	Další způsoby archivu ( <b>dump</b> a <b>restor</b> ) ... 127
2.5	Shell ... 65			<b>4</b>	<b>KOMUNIKACE MEZI PROCESY ... 129</b>
2.5.1	Proces shellu ... 65			4. 1	Signály ... 133
2.5.2	Proměnné ... 70			4. 2	Roura ... 138
2.5.3	Programování ... 71			4. 3	Soubory ... 144

4. 4	Fronty zpráv ... 146	9	<b>BEZPEČNOST ... 301</b>
4. 5	Sdílená paměť ... 151	9. 1	Bezpečnost v úrovni CI ... 303
4. 6	Semaforey ... 155	9. 2	Škůdci ... 307
4. 7	Komunikace procesů v síti ... 160	9. 3	TCSEC v SVID ... 309
4. 8	Správa IPC ... 161	9. 4	Bezpečné sítě ... 315
5	<b>UŽIVATEL ... 163</b>	9. 4.1	Modemy ... 316
5. 1	Registrace uživatelů ... 163	9. 4.2	Síťové servery ... 317
5. 2	Identifikace ... 168	9. 4.3	Kontrola autenticity, Kerberos, bezpečný RPC ... 323
5. 3	Přístupová práva ... 178	9. 4.4	Ochranné zdi (Firewalls) ... 327
5. 4	Účtování ... 179	9. 5	Šifrování, fyzická bezpečnost ... 330
6	<b>PERIFERIE ... 185</b>	10	<b>SPRÁVA ... 333</b>
6. 1	Terminál ... 189	10. 1	Život operačního systému ... 333
6. 2	PROUDY - STREAMS ... 195	10. 2	Disková paměť ... 339
6. 3	Tiskárna ... 203	10. 3	Výkon operačního systému (stavba a generace jádra, periferie) ... 354
6. 4	Ostatní periferie ... 207	10. 4	Sítě ... 360
7	<b>SÍŤE ... 209</b>	10. 5	Živý operační systém (denní údržba) ... 364
7. 1	Vrstva síťová, protokol IP ... 221	11	<b>PLAN 9 ... 367</b>
7. 2	Vrstva transportní, TCP,UDP ... 229	11. 1	Základní schéma ... 368
7. 3	Vrstva procesová ... 232	11. 2	Souborové servery ... 371
7. 3.1	BSD sockets ... 235	11. 3	Provoz sítě, Internet ... 373
7. 3.2	TLI ... 239	11. 4	Programování ... 374
7. 3.3	Síťový programovací jazyk RPC ... 243	11. 5	Bezpečnost ... 375
7. 4	Síťové aplikace a jejich provoz ... 248	11. 6	Konfigurace a správa ... 376
7. 4.1	Pojmenování uzlů sítě (NIC, DNS) ... 249		<b>PŘÍLOHY ... 379</b>
7. 4.2	Síťový superserver <i>inetd</i> ... 258	A	Volání jádra ... 379
7. 4.3	Síťové periferie (tiskárny, terminály, pásky, RFS a NFS, NIS) ... 260	B	Signály ... 383
7. 4.4	Pošta ( <i>sendmail</i> ) ... 270	C	Identifikace ... 384
7. 4.5	Internet a Intranet ... 272	D	Rozložení adresářů ... 385
7. 5	Protokol IP jako aplikace (UUCP, PPP, SLIP) ... 278	E	Standardní klienty X ... 387
8	<b>X ... 285</b>		<b>LITERATURA ... 388</b>
8. 1	Základní schéma ... 288		<b>REJSTŘÍK ... 391</b>
8. 2	X z pohledu uživatele ... 290		
8. 3	X z pohledu operačního systému ... 296		

# ÚVOD

Naše doba je poznamenána masovou výrobou matematických strojů – počítačů (computers). Mezi neodborníky jsou nejznámější osobní počítače (personal computers), které jsou vzhledem ke své velikosti a určení používány pro potřeby jednoho člověka při vedení jeho osobní agendy. Vyrábí se ale i jiné počítače. Například ty, které jsou určeny pro náročné průmyslové aplikace nebo pro potřeby organizací či celých regionů. Jsou používány současně desítkami nebo i stovkami lidí, kteří čtou nebo mění uložené informace. Podle počtu lidí, kteří je v dané chvíli mohou používat současně, se při nasazování vybírá počítač vhodné síly.<sup>1</sup>

Přestože existují a jsou důležité dále i jiné typy počítačů (specializované, pro potřeby řízení výroby, sledování a řízení přírodních, umělých nebo společenských procesů, reagující ve velmi malém časovém intervalu na výskyt určitých událostí atd.), předmětem našeho zájmu bude přístup k počítačům a v nich uloženým datům, které slouží pro vědomostní potřeby současně více lidí.

Takový počítač, je-li vyroben, musí mít vytvořen software (programové vybavení), který zajistí pokud možno pohodlné a bezchybné používání hardwaru (technické vybavení, holý stroj) několika uživateli současně. Jedná se o ovládání disků, terminálů, operační paměti, tiskáren, propojení s jinými počítači atd. a poskytování jejich možností uživatelům. Takový řídicí software holého stroje nazýváme operační systém (operating system). Dnes jsou v oblasti používání počítačů známy dva hlavní problémy. První je problém návrhu holého stroje tak, aby vyhovoval požadavkům nárůstu výkonu. Druhý je návrh operačního systému, který by obecně byl schopen pracovat se stejným výsledným efektem a na různých současných nebo budoucích typech holého stroje a přitom umožňoval provoz všech možných aplikací (tj. programových systémů), které byly nebo teprve budou vymyšleny. To je jistě pro výrobce operačního systému úkol náročný a z uvedených požadavků vyplývá, že nikdy nepůjde o návrh definitivní, ale že se jedná o neustále probíhající proces dalšího rozvoje, zdokonalování nebo i vytváření nových koncepcí. Tématem této knihy je popis jednoho z nejvíce používaných operačních systémů, který tyto předpoklady splňuje, přestože není ideální. Protože je ale v současné době nejvíce rozšířený a používaný, budou jeho principy jistě akceptovány i v budoucnu, a proto je dobré o nich něco znát. Jeho jméno je UNIX.

Hlavní důvod úspěchu tohoto operačního systému je právě především ve snaze používat libovolný hardware. V praxi to pak znamená, že prostředí, ve kterém se pohybuje uživatel, programátor nebo správce systému, je shodné na počítačích od různých výrobců a s různou architekturou a koncepcí hardwaru. Říkáme, že je operační systém přenositelný mezi různým hardwarem. Tato přenositelnost ovšem znamená několikaměsíční usilovnou práci týmu specialistů, protože je nutno přeprogramovat všechny části, které operují s hardwarem. Původní myšlenka při zrodu UNIXu ale akceptovala tuto náročnost, proto jsou tyto části minimalizovány jako nejnižší vrstva operačního systému. Všechny ostatní části, které mohou být na holém stroji nezávislé, jsou pak součástí vrstev vyšších.

Protože je přenositelné prostředí, ve kterém se pohybují programátoři, jsou přenositelné i jejich programy. A to je druhý hlavní důvod jeho úspěchu, který se jednoznačně potvrdil v dnešní době při spojování počítačů a místních sítí do větších celků. Jejich vzájemná komunikace je bezproblémová, je-li komunikačním partnerem tentýž princip.

Podoba operačního systému UNIX je celá řada. Jejich počet je dán především počtem jeho výrobců, kteří jsou většinou také výrobci hardwaru, pro který je konkrétní podoba vytvořena. UNIX vznikl velmi dávno (na konci 60. let) ve výzkumných laboratořích americké firmy American Telephone and Telegraph Company (AT&T), v Bell Laboratories, která se stala vlastníkem autorských práv. Dnes je UNIX chráněná značka organizace UNIX System Laboratories (USL), která také uděluje licence pro implementaci na určitý hardware. Ve snaze dosáhnout jednoty a přesnosti při vývoji a používání UNIXu formulovala AT&T v průběhu 80. let (dnes již šestisvazkový) dokument, ve kterém jsou uvedeny podstatné a neměnitelné vlastnosti a charakteristiky UNIXu. Název tohoto dokumentu je System V Interface Definition, zkratka SVID, poslední edice je třetí, tedy SVID3 (viz Literatura). Název je odvozen od jména UNIXu firmy AT&T, který je System V (pod touto značkou AT&T začala prodávat UNIX od poloviny 80. let), poslední známá verze je System V Release 4.2 (SVR4) (viz Literatura). Dokument SVID, kterému říkáme také doporučení, protože není registrovanou normou, popisuje rozhraní (interface) jednotlivých vrstev UNIXu tak, aby byl pro výrobce závazný a pro uživatele konkrétní vrstvy daný. Dnes je běžné, že výrobci akceptují tento dokument jako kritérium kompatibility a také shodu s dokumentem v charakteristice svého operačního systému uvádějí.

Známy je také dokument X/OPEN Portability Guide (XPG3), který má podobnou funkci jako SVID a je produktem původně evropského sdružení výrobců UNIXu. Obě aktivity, jak SVID, tak XPG úzce spolupracují, takže dokumenty jsou velmi podobné, v mnoha částech zcela shodné. Sdružení X/OPEN bývá ovšem považováno za obecněji uznávanou instituci, proto SVID obvykle požaduje svolení k přetištění částí X/OPEN, stejně tak USL požaduje potvrzení kompatibility svých verzí operačního systému UNIX v kontextu X/OPEN.

SVID3 navazuje na standard (registrovanou normu) operačních systémů POSIX a oba standardy jazyka C, tj. X3.159-1989 (ANSI C) a ISO/IEC 9899-1990 (ISO C). Z pohledu normy POSIX je jeho nadmnožinou. Majitelem POSIXu (plným názvem Portable Operating System Interface for Computer Environments) je IEEE (Institute for Electrical and Electronic Engineers). Standard přitom definuje model obecného operačního systému a pro něj implementovaných programovacích jazyků pro zaručení přenositelnosti napsaných programů a aplikací.

Text této knihy vychází při popisu práce v UNIXu především z SVID3 (viz [SVID391]) a posledního vydání POSIXu (viz [POSIX94]), odkud také přebírá popis struktur a jednoznačné definice rozhraní. Filozofie a funkcionality UNIXu jako celku nebo jednotlivých komponent je výsledkem autorových praktických zkušeností nebo literární erudice a je popisována obecně tak, že vyhovuje těm verzím UNIXu, které přijímají SVID3 a POSIX, pokud nebude řečeno jinak. POSIX je přitom prezentován v zatím poslední verzi ISO/IEC 9945-2:1993(E), ANSI/IEEE Std 1003.2-1993 a SVID ve verzi třetího vydání (Third Edition)<sup>2</sup>. V textu této knihy tedy nejde jen o vágní údaje, ale o komentář přesných a závazných definic.



Přenositelná část UNIXu je napsána v programovacím jazyce C, který za tímto účelem vymyslel a implementoval jeho první verze v 70. letech jeden z autorů UNIXu Denis Ritchie. Je samozřejmé, že při vyjadřování v UNIXu používají odborníci tento jazyk, a stejně tak tomu bude i v této knize. Učebnic jazyka C je mnoho, bestsellerem je právem [KernRitch78], podrobným popisem s častými poznámkami k přenositelnosti a implementaci je [HarbStee87] a přehled jazyka uvádí také [BrodSkoc89].

Kniha je určena pokročilým uživatelům operačního systému UNIX. Přestože každý základní prvek je definován, není možné knihu úspěšně zvládnout bez znalosti prostředí některého z shellů, znalosti pojmů soubor (file), adresář (directory), přesměrování vstupů a výstupů atd. Kniha popisuje vždy práci pro zabezpečení požadavku uživatele z pohledu operačního systému, je tedy určena především programátorům a správcům systémů a sítí. Začínající a středně pokročilé uživatele odkazují na [KernPike84] a [Skoc93].

<sup>1</sup> K takovému účelu může pro potřeby několika uživatelů sloužit i počítač navržený a vyrobený jako osobní (IBM PC, Macintosh, pracovní stanice SGI, HP nebo Sun atd.), ale tyto počítače upřednostňují především podporu jednoho uživatele v daném okamžiku, proto je to řešení nouzové a ve většině případů by mělo být pouze dočasné.

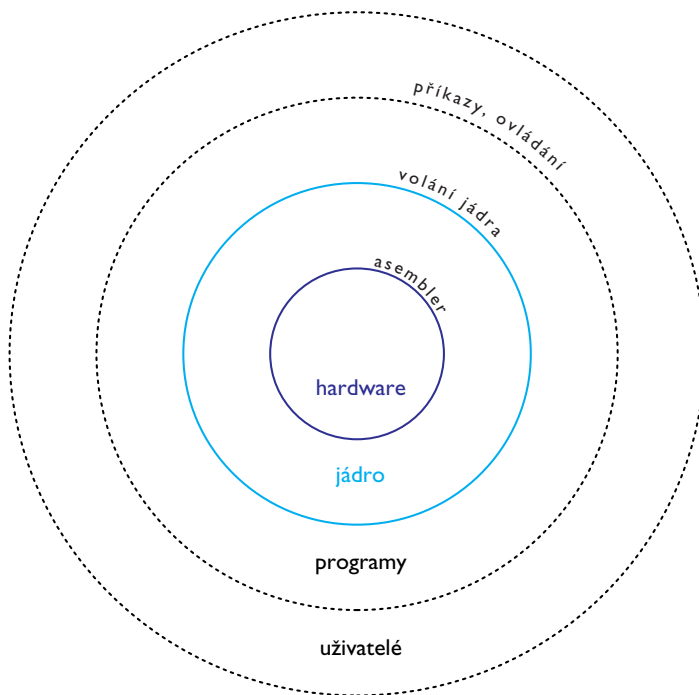
<sup>2</sup> Uvedený POSIX v bibliografii odkazuje na SVID jako na jednu z průmyslových specifikací, kterou se vyplatí sledovat.



# I JÁDRO

Jádro (kernel) v UNIXu je pro mnoho odborníků synonymem operačního systému. Není tomu přesně tak, protože jádro sice spravuje všechny výpočetní zdroje hardwaru, ale umožňuje přístup k počítači programátorům, nikoliv běžným uživatelům. Pro uživatele jsou určeny další části operačního systému, programy, které jim umožňují s jádrem komunikovat pomocí příkazového řádku nebo grafického rozhraní. Tyto programy pak převádějí jejich požadavky na rozhraní, které používají programátoři. Stejně tak správa operačního systému se opírá o řadu běžících programů, které jádru sdělují (opět v řeči programátorů) požadavky na manipulace s výpočetními zdroji v celkovém kontextu. Každopádně je ale jádro určující pro techniky přístupu k výpočetním zdrojům a manipulaci s nimi.

Jádro je program, který běží na holém stroji (tzv. standalone program). Ostatní programy, které uživatelé nebo správa systému používají, jsou sice také prováděny strojem, ale pod dohledem a řízením jádra (říkáme, že jádro je supervizorem běžících programů). Pomocné hardwarové programy (tzv. firmware, programy monitoru hardwaru) jádro zavedou do operační paměti<sup>1</sup> a předají mu řízení. Tím je operační systém nastartován. Jádro obalí hardware a od tohoto okamžiku až do zastavení jádra nelze přistupovat k hardwaru jinak než pomocí požadavků na jádro – tzv. *volání jádra* (system calls). Tím oddělí operační



Obr. 1.1 Vrstvy operačního systému UNIX

systém programátory od holého stroje. Uživatelé jsou pak od jádra odděleni pomocí aplikačních programů. Vychází nám tedy vrstvená struktura operačního systému, viz obr. 1.1.

Na obrázku jsou slovně označeny jednak jednotlivé vrstvy (jádro, programy, uživatelé) a jednak jejich rozhraní (assembler, volání jádra, příkazy, ovládání). Funkcí vrstvy jádra je převádět aktivitu programů (která je formulována voláním jádra) na příkazy assembleru stroje. Programy převádějí požadavky uživatelů (které jsou formulovány příkazy a ovládáním těchto programů) na volání jádra, kterým rozumí jádro. Pochopitelně ne všechny požadavky na určité rozhraní jsou vždy předávány dolní vrstvě, protože každá vrstva má svou funkcionalitu a pracuje s větším komfortem směrem od středu schématu (např. komunikace mezi běžícími programy může být vyřízena v jádru, jednotné rozhraní terminálů v knihovně funkci atd.), ale je zřejmé, že podstatné požadavky, např. přístup k diskům, operační paměti, perifériím atd., jsou převáděny postupně na nejnižší vrstvy. Stejně tak je jednoznačně definováno rozhraní od středu směrem k uživateli, kdy dolní vrstvy sdělují úspěch nebo neúspěch požadavku nebo sdělují výskyt událostí, které ovlivňují činnost vrstev vyšších.

## 1.1 Volání jádra

Vidíme tedy, že mezi holý stroj a uživatele je vložen operační systém. Nemá za úkol uživatele omezovat a komplikovat mu život, ale naopak mu jednak pomáhat zajišťovat jeho požadavky na hardware pohodlným způsobem, jednak jej ochránit před možným zásahem jiných uživatelů nebo i svým vlastním. Dosud známé požadavky uživatele na výpočetní systém (tj. hardware i software) jsou pořizování dat, jejich úschova, modifikace a přesun. Na základě takových požadavků současně několika uživatelů v jednom okamžiku má jádro koncepci pro správu dat i operací nad nimi. Správu dat zajišťuje část označovaná jako systém souborů (file system) a operaci nad daty systém procesů (processes). UNIX poskytuje pro práci se systémem souborů volání jádra, která se vztahují k akcím:

- vytvoření souboru, jeho otevření, čtení nebo zápis dat do souboru, uzavření souboru, zrušení souboru,
- vytvoření nebo zrušení adresáře, nastavení pracovního adresáře,
- vytvoření nebo zrušení speciálního souboru, který představuje periferii,
- test atributů existujícího souboru, adresáře nebo periferie, změna těchto atributů,
- rychlé přemístění na požadovanou pozici v otevřeném souboru,
- vytvoření zámku pro výlučný přístup k souboru.

Pro práci s procesy UNIX poskytuje především volání jádra ve významu:

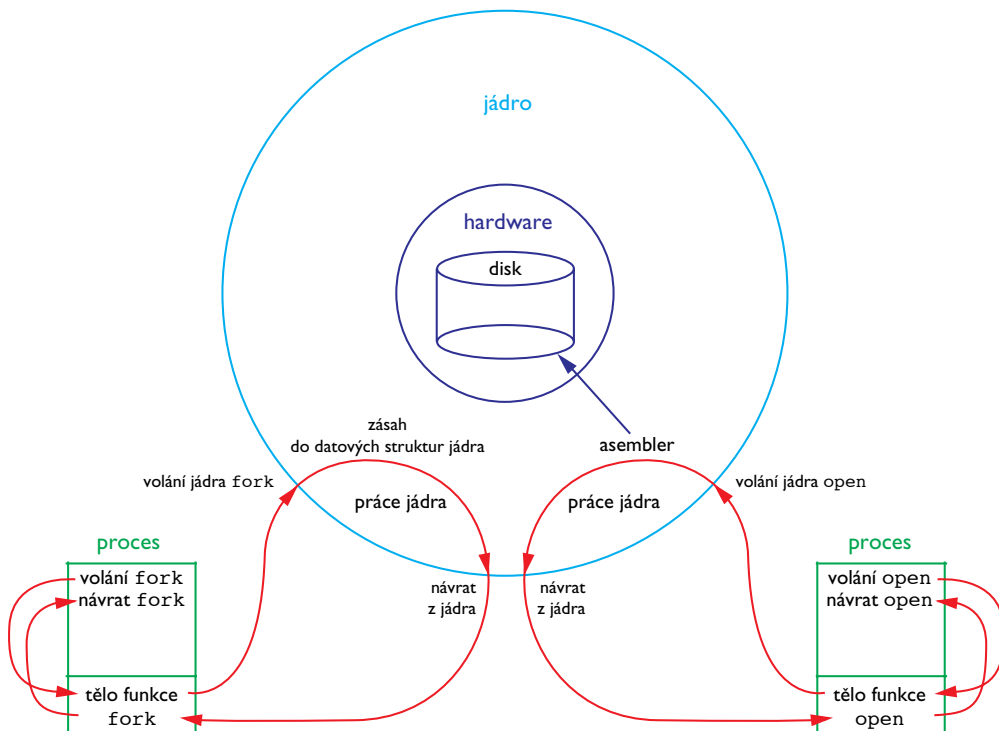
- vytvoření nového procesu, čekání na jeho dokončení, ukončení procesu,
- výměna programu, který řídí proces, za jiný,
- komunikace mezi procesy pomocí signálů, rour, front zpráv, sdílené paměti a semaforů,
- zjišťování a nastavování atributů běžícího procesu,
- změna priority běhu procesu,
- práce s datovou pamětí procesu, její rozšiřování nebo zmenšování, zamykání v operační paměti.

Zbývá hlavní volání jádra, která charakterizují UNIX a nejsou jednoznačně zařaditelná do předchozích dvou skupin, jsou pro zajištění:

- ladění programů, zjišťování spotřebovaného času stroje procesem,
- práce se systémovou vyrovnávací pamětí,
- identifikace instalace operačního systému.

Jednotlivá volání jádra jsou koncipována jako volání funkcí programem. Takto vzniklý externí symbol je při sestavování programu pokryt modulem standardní knihovny konkrétního jazyka (např. C). Obsahem připojeného modulu je převedení formulovaného požadavku na jádro do assembleru, kdy pomocí tohoto jazyka stroje je řízení předáno z uživatelského procesu do jádra s udáním jména volání jádra a parametrů formulovaných programem. Vstup do jádra znamená dále práci jádra pro proces až do chvíle, kdy je tato práce ukončena, ať už uspokojivě nebo ne (došlo k chybě při realizaci požadavku, např. při otevírání souboru nebyl nalezen odpovídajícího jména atd.). Princip práce jádra pro proces, který otevírá soubor voláním jádra `open`, a současně pro jiný proces, který vytváří nový proces voláním jádra `fork`, ukazuje obr. 1.2.

Přestože je na obrázku záměrně zobrazena situace, kdy se o práci jádra uchází dva procesy současně (a může jich být teoreticky neomezené množství), protože se jedná o víceuživatelský (a tedy i víceprocesový) operační systém, volání jádra jsou realizována jádrem postupně podle jejich příchodu. Jde tedy o frontování požadavků na jádro. Jinými slovy lze říct, že jádro je v okamžiku realizace volání jádra jiným procesem nepřerušitelné.



Obr. 1.2 Zajištění požadavků volání jádra

Formát každého volání jádra je popsán ve svazku (2) provozní dokumentace. Způsob manipulace v programu v jazyce C je tentýž, jako je obvyklé volání funkce. Programátor má k dispozici argumenty funkce pro vyjádření svého požadavku, jádro mu odpovídá především návratovou hodnotou funkce volání jádra. Návratová hodnota je vždy celočíselná (`int` nebo `long`, případně ukazatel) a má význam daný popisem v dokumentaci. Jednotně je ale sdělována kolize, tj. případ, kdy jádro požadavek nemůže vyřídit, a proto jej odmítne (špatný formát volání jádra, neexistující výpočetní zdroj, překročení mezních hodnot tabulek jádra atd.). Návratová hodnota je potom `-1`, a pokud má programátor definovanou externí proměnnou `errno` způsobem

```
extern int errno;
```

je `errno` naplněna jádrem celočíselnou hodnotou, která charakterizuje důvod odmítnutí požadavku<sup>2</sup>. Hodnoty `errno` jsou popsány v úvodní stránce provozní dokumentace `intro(2)`, případně v popisu každého volání jádra svazku (2) dokumentace. Číselné kódy chyb a jejich textové ekvivalenty jsou definovány v souboru `<errno.h>`. Poznamenejme také, že jádro naplní obsah `errno` pouze v případě, že došlo ke kolizi. V případě úspěšného provedení ponechává její obsah nezměněn. Funkce, které zvětšují komfort programátora při zpracování chyb jádra, jsou `perror`, která má být v budoucnu podle SVID nahrazována funkcí `fmtmsg`, protože `fmtmsg` umožňuje vypisovat požadované texty chyb také na systémové konzole a v definovaném národním prostředí. `perror` je ale definována v ANSI C a POSIXu, její eliminace tedy nehrozí. `perror` pracuje s `errno` a odpovídajícím textovým polem,

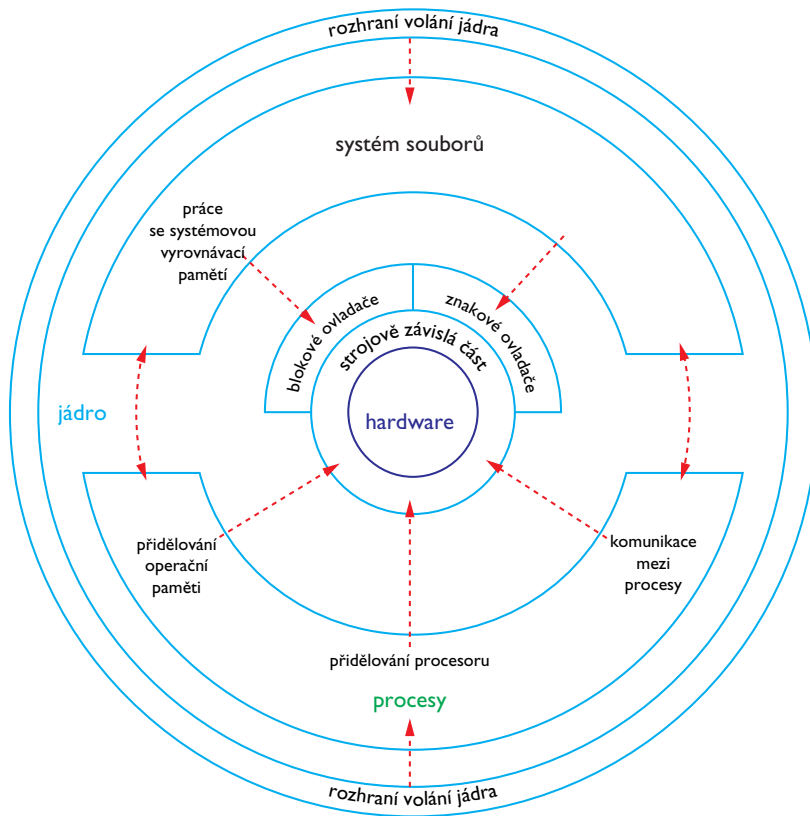
```
void perror(const char *s);
```

tiskne na standardní chybový výstup text daný parametrem `s` následovaný dvojtečkou, mezerou a systémovým chybovým textem odpovídajícím současnému obsahu `errno` (`errno` bývá modifikováno jádrem pouze při výskytu chyby!). SVID definuje (a System V.4 používá) příkaz se jménem **truss**, který umožňuje sledovat použitá volání jádra procesu.

## 1.2 Moduly jádra

Strukturu jádra ukazují obr. 1.3.

Z obrázku je patrná důležitost hlavních dvou částí jádra, a sice modulů systému souborů a modulů procesů. Jejich popis bude předmětem vždy některé z následujících kapitol, včetně interakce těchto částí. Rozhraní volání jádra přijímá požadavek běžícího programu a pro jeho zajištění aktivuje odpovídající moduly. Moduly se při své práci obracejí na hardware tak, že pomocí *ovladačů* (drivers) sdělují *strojově závislé části* manipulace s hardwarem. Vrstva hardwaru požadavek vykoná a vrací reakci (v podobě dat nebo potvrzení vykonané akce) zpět přes strojově závislou část do ovladače a vyšších modulů. Vlastní program jádra je programován v jazyce C, vyjma strojově závislé části. Ovladač je programován rovněž v jazyce C. Je to sada funkcí, přičemž aktivována je vždy ta funkce, která odpovídá požadavku (např. funkce pro čtení nebo funkce pro zápis atp.). Reakce hardwaru ale není očekávána synchronně. Tím je myšleno, že jádro se nezastaví a nečeká po dobu obsluhy jeho požadavku hardwarem, ale věnuje se jiné činnosti, např. zajišťuje jiný požadavek volání jádra pro jiný proces (jádro nemůže čekat na reakci hardwaru, která může být i neúměrně dlouhá, např. u tisku na tiskárnu by se po dobu tisku skupiny znaků čekalo na dokončení této operace). Čeká pouze proces, pro který je volání jádra realizováno. Hardware má za úkol požadavek zpracovat, neuvážnout při této akci (i to je možné) a po ukončení



Obr. 1.3 Moduly jádra

výsledky práce předat zpět jádru. Hardware používá zpětně pro komunikaci s jádrem *systém přerušení*, který jádru vnutí pozornost na ukončenou akci nebo *výjimečnou událost* (třeba kolizi hardwaru). Pozornost získá hardware tím způsobem, že jádro je při výskytu přerušení naprogramováno na akci vynuceného provedení jedné z funkcí ovladače odpovídající hardwarové periférie (jedná se o funkci končící na `_intr`, jak se o tom zmíníme později). Vykonáním této funkce dojde k návratu z hardwaru, volání jádra pak může být pro odpovídající proces dokončeno. Systém přerušení, který je vlastně rozhraním zdola nahoru mezi hardwarem a jádrem, musí ovšem mít určité uspořádání, hierarchii, která znamená prioritu ošetření vzniklého přerušení jádrem. Jádro může např. v době příchodu přerušení provádět akce v *kritických sekcích* (např. práce s vázaným seznamem některých datových struktur jádra) a přerušení by mohlo způsobit poškození obsahu takové sekce. Proto je přerušení tzv. *maskováno*, tj. je jádrem registrováno, ale odloženo na pozdější zpracování. Je to věc strojově závislé části jádra, jak je systém přerušení nastaven, protože hardware umožňuje nastavovat *hierarchii přerušení* v různých variantách. Typicky

nejvyšší priorita je nastavena pro ošetření strojových chyb, pak hodin stroje, diskových operací, sítě a konečně terminálů a tiskáren. Na nejnižší úrovni pak bývají přerušení softwaru.

Teprve na nejnižší vrstvu práce výpočetního systému, tj. hardware, jsou tedy z pohledu implementace UNIXu kladeny následující požadavky. Jednak je to práce procesoru v *privilegovaném režimu* (supervizorovém, chráněném), do kterého může být procesor přepnut (toto přepnutí je v UNIXu hlídáno jádrem, protože jádro je první, které se k hardwaru dostane při startu operačního systému). Pokud přepnut není, může procesor provádět instrukce, ale ne ty, které manipulují s periferiemi. O práci s periferiemi může softwarově požádat jádro, které přepnutí na základě zvážení požadavku provede, samo akci vykoná a nakonec procesor přepne zpět do neprivilegovaného režimu a předá řízení odpovídajícímu procesu. Další požadavek na hardware je přítomnost systému přerušení, který UNIX využívá pro asynchronní obsluhu operací nad hardwarem. Konečně třetí požadavek je na možnost tzv. *mapování paměti*. Každý program je při překladu do instrukcí stroje *překladačem* (kompilátorem) vytvářen, jako by měl běžet sám v operační paměti stroje. Při spuštění programu je ale umístěn do volného místa v paměti (do okamžiku dokončení může navíc tuto polohu několikrát změnit) a hardware pro něj zajistí prostředí, které vypadá, jako kdyby běžel v paměti sám. Říkáme, že proces je prováděn ve *virtuální paměti*. Pracuje pro něj *virtuální počítač*, protože překladač přidělil procesu *adresy virtuální*, nikoliv fyzické; na ty je převádí právě systém mapování paměti. Mapování paměti umožňuje jádru přidělit procesu paměť větší, než by bylo schopno bez této podpory. Manipulace s procesem je pak také jednodušší, např. řešení odkazu na paměťové místo, které je mimo oblast procesu, je odmítnuto strojem jednoznačně, takže proces nemůže zasahovat do adresového prostoru jiných procesů nebo dokonce jádra. Také nemůže manipulovat s adresami periférií a hardwaru vůbec, které jsou výhradně ve správě jádra, jak bylo popsáno výše.

Uvedená práce jádra pro požadavky procesů (potažmo uživatelů) koncepčně znesnadňuje práci procesů reálného času, tj. procesů, které řídí periférii tak, že při příchodu přerušení musí reagovat do několika desítek milisekund. Problém je skryt v upozornění jádra na výskyt události. Jádro přerušení zpracuje a událost odpovídajícímu procesu sdělí po uplynutí časového intervalu vyřízení předchozí fronty přerušení. Úskalí se dále objeví ze strany zpracování volání jádra, pomocí kterého a jedině kterého může proces vydat patřičný pokyn pro reakci periferie. Uvedli jsme totiž, že při zajišťování volání jádra procesu je jádro po dobu předání požadavku hardwaru jiným procesem nepřerušitelné. To je opět kvůli zajištění konzistence operací prováděných jádrem a pro zjednodušení této situace jádro raději dočasně ignoruje požadavky jiných procesů, než aby komplikovaně regulovalo vstup do kritických sekcí struktur jádra. V klasickém UNIXu proto procesy reálného času, pokud pro ně není postačující reakce do 1 vteřiny, nemají možnost spolehlivého provozu. Verze UNIX System V.4 (podle SVID) provoz procesů reálného času podporuje, a sice na základě patřičné úpravy jádra. Jaká úprava jádra má být provedena, SVID neříká. Definuje pouze volání jádra, které proces může použít, aby se prohlásil procesem reálného času. Obvykle je ale úprava jádra prováděna řadou kontrol při běhu procesů v supervizorovém režimu tak, že vždy po daném časovém intervalu (znamená to, že běh sekvence instrukcí jádra musí být změřen, aby nepřekročil požadovaný časový limit) jádro zjišťuje, zda některý z procesů reálného času nemá požadavek volání jádra. Situace se tímto hodně komplikuje, protože pokud takový požadavek skutečně nastane, jádro musí uschovat kontext svého běhu, provést obsluhu procesu (zajištěním dalšího volání jádra) a pak kontext před příchodem požadavku zpětně obnovit a pokračovat v práci pro původní proces. To není snadná úprava, uvědomíme-li si, že procesů reálného času může být několik.



### 1.3 Přístup uživatele

Na obr. 1.1 jsme uvedli vrstvu programů, která využívá rozhraní volání jádra při požadavcích na hardware. Programy z druhé strany komunikují většinou interaktivně s uživateli. Pro formulování běžných uživatelských požadavků na jádro od terminálu byl v době vzniku UNIXu vymyšlen a naprogramován interpret příkazů, který byl nazván *shell*, původní s přívlastkem Bourne podle autora tohoto programu. Bourne shell stejně jako jeho pokračovatelé (C-shell, Job Control Shell, KornShell) zachovává řádkovou komunikaci s uživatelem. I přes tento zdánlivý nedostatek zůstává podstatný a neměnný pro uživatelské prostředí. Shell je rukou dosahující do systému. Jednak lze pomocí shellu formulovat požadavky na jádro a hardware, což také zahrnuje možnost spojovat soubory nebo procesy při jejich spolupráci. Bourne shell je realizací myšlenky B. Kernighana o vytvoření komfortního prostředí práce uživatele u terminálu na konci 60. let tohoto století. V UNIXu se stalo toto prostředí součástí operačního systému. Společně s jádrem je přenášeno a implementováno na různých platformách. Bourne shell používá pro požadavky zpracování dat (třídění, výpočty standardních funkcí atd.) pomocné programy, kterým se říká *nástroje* (tools). Je jich několik set, jsou součástí operačního systému a spoluvytváří prostředí uživatele u terminálu. Uživatel je používá jako moduly zpracování dat při programování v interpretu příkazů, shellu. Při komunikaci uživatele s jádrem v následujícím textu budeme používat příkazů Bourne shellu nebo jeho pokračovatele KornShellu, který je s Bourne shellem kompatibilní (množina příkazů Bourne shellu je podmnožinou příkazů KornShellu). Manipulace uživatele s jádrem a hardwarem pomocí shellu nám bude sloužit jako ukázka k vyvolání práce typických mechanismů nižších vrstev operačního systému, kterými jsou jádro a hardware.

Řádková, nebo chcete-li textová orientace systému je typická. Přestože je dnes zřejmé, že bez *grafického rozhraní* pro uživatele (GUI, Graphical User Interface) se žádná komfortní aplikace neobejde, dolní vrstvy UNIXu zůstávají zachovány v textové podobě. V rámci horních vrstev (uživatelé, programy, viz. obr. 1.1) je v UNIXu definováno grafické prostředí *X Window System* (často označováno pouze X, viz kap. 8), v jehož prostředí jsou programovány a používány grafické aplikace. X pracuje s grafickým prostředím jako se systémem procesů, které mezi sebou navzájem komunikují, a to nejenom místně (v rámci jednoho jádra), ale také sítově. Vždy jde ale o využívání dolních vrstev (přes jádro vede cesta k hardware). Jsou známa různá grafická prostředí, která v UNIXu vznikla (Motif, Open Look), v rámci kterých každý výrobce správce systému zpřístupňuje také akce, které usměrňují práci celého operačního systému. GUI pro jejich realizaci používá volání jádra, se kterými se seznámíme. Historie přesto ukázala, že komfortní grafické prostředí není vždy nutné v každé situaci (řadě aplikací v provozu lépe vyhovuje sice obrazovkový, ale textově orientovaný přístup uživatele) a někdy naopak překáží programátorům a správcům UNIXu, kteří dolní standardizované vrstvy (popis shellu a nástrojů je součástí POSIXu) ovládají raději přímo. I z těchto důvodů je součástí každého GUI v X emulace textového terminálu *xterm* (podrobněji viz kap. 8). Zkušenosti dále ukazují, že vrstvená skladba operačního systému, kde GUI je realizováno jako aplikace na úrovni nejvyšších vrstev, je optimální a velmi efektivní při změnách systému, atypických situacích a přenositelnosti na jiný hardware.

<sup>1</sup> RAM (Random Access Memory), v překladu paměť s libovolným přístupem, tj. pro čtení i zápis. Na rozdíl od ROM (Read Only Memory), paměť pouze pro čtení. V obou případech jde o typ paměti, ve které se procesor pohybuje v přímé adresaci.

<sup>2</sup> Ne každé volání jádra má návratovou hodnotu. Např. `free`, volání jádra pro uvolnění dříve přidělené datové oblasti operační paměti, nemá žádnou návratovou hodnotu (je definována jako `void`). Programátor by proto měl vždy využívat provozní dokumentaci nebo dokumentaci standardu POSIX.

## 2 PROCESY

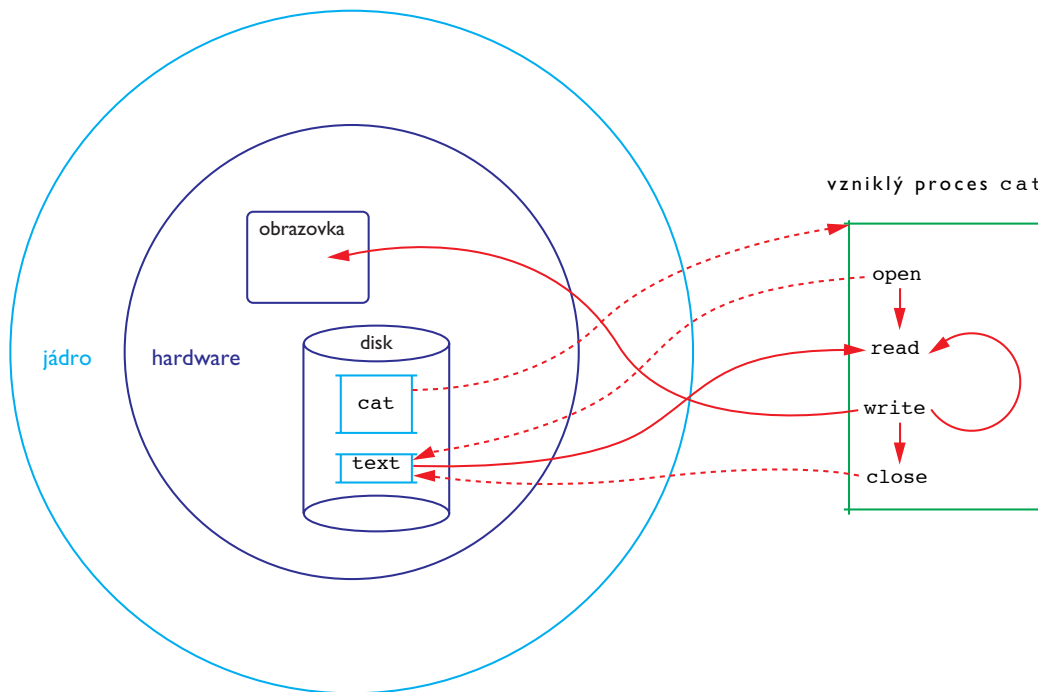
Na obr. 1.3 konce předchozí kapitoly jsou jako dvě hlavní části jádra uvedeny část systému souborů (file system) a část procesů (processes). Jedná se o klasický způsob přístupu uživatele k výpočetnímu systému, kdy jeho požadavkem je provedení manipulace s daty. Data jsou ukládána na média s trvalým záznamem (disky, pásky atp.). Změna dat je proto zaznamenána pro další zpracování za teoreticky libovolný časový interval. V UNIXu pro ukládání dat slouží systému souborů a pro manipulaci s daty systém procesů nebo jen procesy. V této kapitole popíšeme práci operačního systému pro podporu procesů.

### 2.1 Vznik procesu

*Proces* vzniká na pokyn uživatele. V rámci sezení zadává uživatel příkazy pro operační systém. Např.

**\$ cat text**

je zobrazení obsahu souboru se jménem `text` na obrazovku uživatelského terminálu. Operační systém pro zpřístupnění tohoto souboru aktivuje proces se jménem `cat`. Jeho jméno je odvozeno od jména programu, který proces po dobu jeho existence řídí. Jméno programu je shodné se jménem souboru, ve kterém je text programu uložen. Jméno našeho souboru je `cat`. Soubor je umístěn v některém veřejně

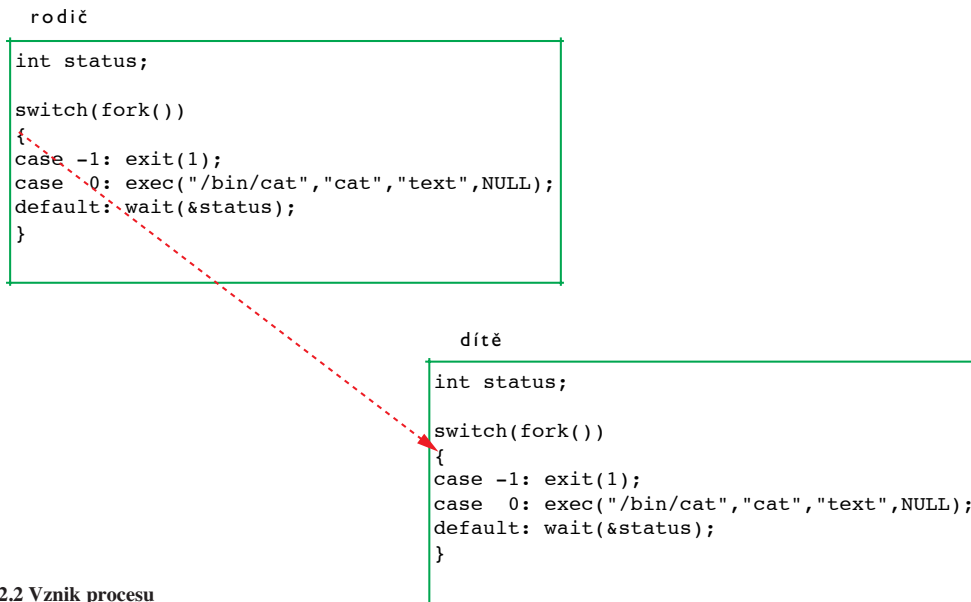


Obr. 2.1 Situace `cat text`

čitelném adresáři `/bin` nebo `/sbin`. Operační systém vytváří nový proces, který na základě instrukcí programu otevírá soubor, jehož obsah čte a zapisuje na obrazovku periferie terminálu. Proces `cat` využívá pro komunikaci s hardwarem (disk, terminál) volání jádra `open`, `read`, `close` (pro přístup k disku) a `write` (pro terminál) ve smyslu kapitoly 1 a obr. 2.1.

Po vzniku procesu je z disku ze souboru `/bin/cat` přenesen program `cat` jako řídicí část vznikajícího procesu. Proces je spuštěn, voláním jádra `open` si zpřístupňuje soubor `text` na disku a voláním jádra `read` čte jeho obsah. Voláním jádra `write` zapisuje přečtená data na obrazovku terminálu a po přečtení a zápisu všech dat souboru voláním jádra `close` soubor `text` uzavírá. Před zápisem `write` na terminál tento terminál nemusí žádným zvláštním způsobem otevírat, protože přístup k periférii terminálu je nastaven současně se vznikem procesu `cat`. Kdo ale dává pokyn k vytvoření našeho procesu? Uvedli jsme, že je to uživatel pomocí příkazového řádku. Tento příkazový řádek čte příkazový interpret shell, který oznamuje svoji přítomnost uživateli znakem `'$'`. Shell také iniciuje vznik procesu `cat`. Vzhledem k tomu, že shell je realizován také procesem, vznik nového procesu (`cat`) je na žádost volání jádra. Volání jádra, která znamenají vznik nového procesu jsou `fork` a `exec` a v našem případě je používá program řídicí proces shell. V dalším textu jej budeme označovat jako `sh` (`sh` je jméno příkazu a souboru pro Bourne shell).

Proces `sh` na základě žádosti uživatele u terminálu používá volání jádra `fork` pro vytvoření nového procesu. Jádro provádí rozšíření svých vnitřních tabulek procesů o nový proces a po této evidenci vrací řízení procesu `sh` za volání jádra `fork`. Od tohoto okamžiku již existuje nový proces. Prozatím ale není procesem `cat`, ale je kopií procesu, který žádal jeho vytvoření. Proměna nově vzniklého procesu v proces `cat` se v následujícím kroku dosáhne pomocí volání jádra `exec`. Situaci vzniku nového procesu ukazuje obr. 2.2.



Obr. 2.2 Vznik procesu

Na obrázku je proces, který žádá o vznik procesu, označen jako *rodič* (parent) a nově vzniklý proces jako *dítě* (child). Dítě je tedy kopie rodiče. Především to znamená, že oba procesy řídí tentýž program. Každý proces ale provádí jeho jinou část, tu, která je na obrázku označena tučně. Volání jádra `fork` má formát

```
#include <sys/types.h>
pid_t fork(void);
```

V jeho návratové hodnotě může program očekávat buďto hodnotu `-1` (nastala chyba), hodnotu `0` (program řídí dítě) a nebo to může být celočíselná hodnota větší než `0` (program řídí rodiče). S návratovou hodnotou pracuje programátor, který takto rozliší rodiče a dítě. Přitom pro rodiče použije část programu, která znamená čekání na dokončení dítěte (volání jádra `wait`) a pro dítě volání jádra, které promění proces na jiný, tzn. vymění mu řídicí program za jiný (volání jádra `exec`). Dítě pak pokračuje v provádění nového programu (pro nás v programu **cat**, který je uložen v souboru `/bin/cat`), a to od jeho začátku. Na obrázku je vždy odpovídající část programu pro proces rodiče nebo dítěte vyznačena tučně.

Dvoustupňový způsob vzniku nového procesu má své praktické opodstatnění, protože na zajištění určitých atomických akcí při programování lze použít nový proces, ovšem při řízení stejným programem, nebo lze změnit procesu program na jiný bez vzniku procesu nového a šetřit tak výpočetní zdroje.

V uvedeném příkladu jsou použita také volání jádra `exit` a `wait`. `exit` je žádost o ukončení procesu. V našem příkladě je používáme pro zjednodušení na ukončení rodiče, pokud se `fork` nezdaří (dítě vůbec nevzniká), parametr `exit` je návratová hodnota procesu, kterou systém registruje a může být testována programem. `wait` znamená čekání na dokončení některého z dětských procesů (procesů může být vytvořeno více). Parametr `status` je hodnota, kterou nám jádro naplní výsledkem návratové hodnoty dítěte. Návratová hodnota `wait` (pokud není `-1`) identifikuje ukončený dětský proces (je to tatáž hodnota, kterou získáme v návratové hodnotě `fork`, když dítě vytváříme).

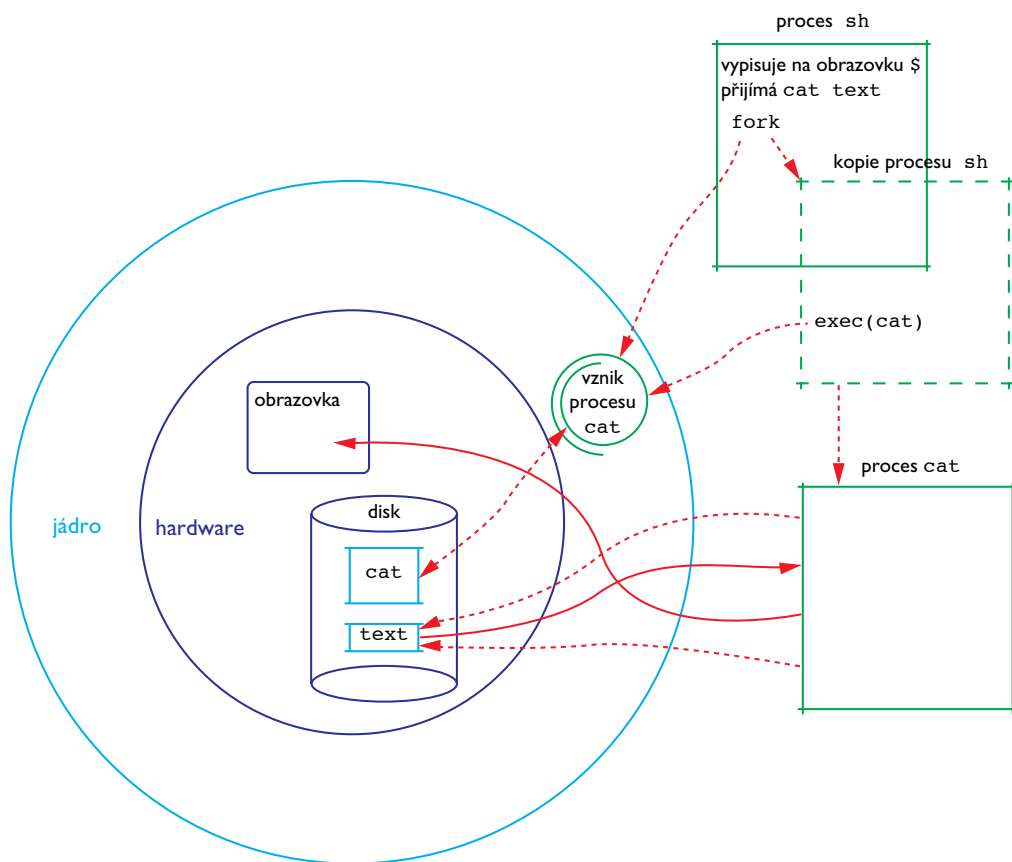
Na obr. 2.3 je uvedena celková situace vzniku procesu **cat** jeho tvůrcem – procesem **sh**.

Návratová hodnota `fork` při vzniku dětského procesu je *číselná identifikace nově vzniklého procesu* (Process Identification, PID) a je v rámci běhu operačního systému jedinečná. Je to celočíselná nezáporná hodnota, kterou jádro přiděluje tak, že použije PID naposledy vzniklého procesu zvětšenou o 1. V okamžiku startu jádra má první vzniklý proces přiděleno PID=0. Největší mezní hodnota PID je dána rozsahem typu `unsigned int`. Dosáhne-li jádro v okamžiku přidělování této mezní hodnoty, pokračuje přidělováním opět od hodnoty 0 a vynechává přitom PID procesů, které od přidělení jejich PID ještě stále běží.

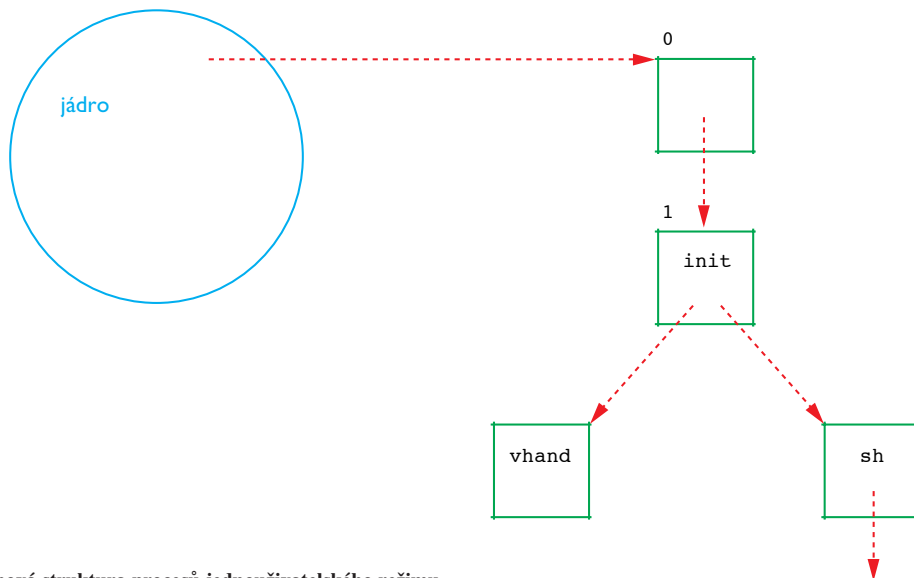
## 2.2 Hierarchická struktura procesů

Vznik nového procesu je iniciován jiným procesem. Vzniká tedy hierarchická struktura procesů z pohledu rodičů a dětí, protože každý rodič může mít teoreticky neomezený počet dětí. Každý proces přitom musí mít nějakého rodiče vyjma prvního, který je kořenem této stromové struktury běžících procesů. První proces vzniká při startu operačního systému z iniciativy jádra a má číselnou identifikaci 0. Proces je vždy pojmenován podle jména souboru s řídicím programem. První proces má obvykle

jméno **sched** (nebo **swapper**) a jeho funkce je práce pro *odkládací oblast* (swap area) ostatních procesů. Jeho první aktivita je ovšem žádost o vznik nového procesu, ten má PID=1, jeho jméno je **init** a je to proces označovaný jako prapředeek všech dalších procesů. **init** je proces, který podle tabulky `/etc/inittab` startuje nové procesy zajišťující chod operačního systému (jde o procesy zajištění frontování požadavků na tiskárnu, obsluhy požadavků sítě, oživení terminálů uživatelů nebo grafických adaptérů atd.). **init** je proces, který řídí práci dvou základních režimů operačního systému: režimu *víceuživatelského* (multi user) nebo *jednouživatelského* (single user). Režim je zajištěn během odpovídajících procesů, tj. skupinou procesů. Režim víceuživatelský znamená běh procesů, které umožňují uživatelům přihlašovat se a pracovat se svými daty. Jednouživatelský je režim, který podporuje práci pouze jednoho uživatele (privilegovaného uživatele) na periférii systémové konzoly a všechny další aktivity systému pro práci periférií jsou utlumeny (je určen pro údržbu systému). Jiný režim ale neznamená žádné omezení práce jádra pro procesy, všechny funkce jádra jsou tytéž. Proces **init**



Obr. 2.3 Situace `$ cat text`



Obr. 2.4 Stromová struktura procesů jednouživatelského režimu

rozhoduje, který z režimů nastaví podle tabulky `/etc/inittab`. Na obr. 2.4 je uvedena typická stromová struktura jednouživatelského režimu.

Podle obrázku je patrné, že proces před aktivací procesu **sh** na systémové konzole vytváří ještě některé další systémové procesy, jako např. **vhand**, **bdf flush** atp. (jména procesů se mohou v různých implementacích lišit). Pracují pro stránkování a segmentaci operační paměti ostatních procesů a jsou pochopitelně přítomny jak v jednouživatelském, tak víceuživatelském režimu. Proces **sh** vzniká na systémové konzole po zadání hesla privilegovaného uživatele (uživatele se jménem **root**). Režim víceuživatelský ukazuje obr. 2.5.

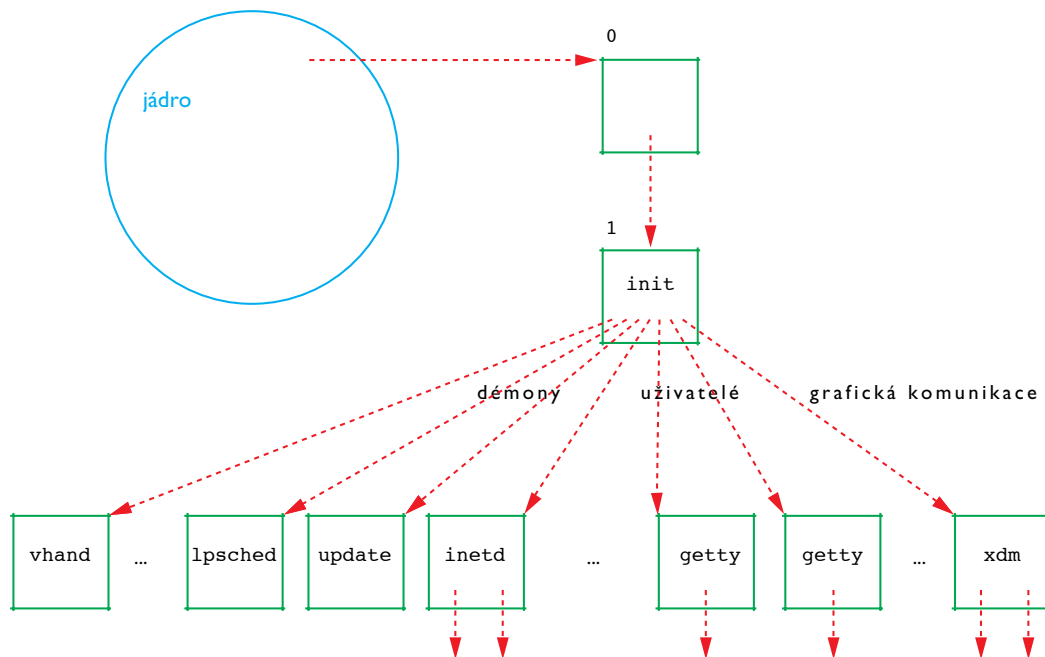
Kromě procesů práce s odkládací oblastí pracují v režimu víceuživatelském další procesy, které zajišťují chod operačního systému pro práci zúčastněných uživatelů. Jsou to např. procesy **lpsched** (fronta požadavků na tiskárnu), **update** (práce se systémovou vyrovnávací pamětí), **cron** (zajištění provedení určité akce podle data a času), **inetd** (síťový superserver) atd. Uživatelé vstupují do systému pomocí procesů **getty** (nebo **ttymon**), které jim umožňují se přihlásit a startují procesy jejich komunikace se systémem (některý z shellů). Grafický způsob vstupu uživatelů do systému je obvykle zajišťován procesem **xdm** grafického rozhraní UNIXu s názvem X-Window System (viz kap. 8).

Typický příklad použití pouze jednoho z volání jádra pro vznik procesu **exec** je při přihlašování uživatele do systému. Proces **getty** vypisuje na terminál text končící na

**login:**

Po zadání uživatelského jména proces **getty** používá volání jádra **exec**, takže se proces **getty** promění na proces **login**, který vypisuje na terminál text

**password:**



Obr. 2.5 Stromová struktura procesů víceuživatelského režimu

Uživatel mu oznamuje heslo, kterým je kryt jeho vstup do systému. Pokud je heslo kontrolované procesem **login** vyhovující, proces **login** se pomocí volání jádra **exec** mění na proces **sh** a uživatel může zadávat systému své požadavky pomocí příkazů shellu. Výhoda takového postupu je evidence stále jednoho procesu jádrem. Přitom po celou dobu uživatelského sezení je proces **sh** dítětem procesu **init**. V okamžiku uživatelského odhlášení (současným stiskem klávesy Ctrl a d nebo příkazem **exit**) tento dětský proces zaniká a jeho rodič, proces **init**, vytváří nový proces **getty** (už jiného PID), který vypisuje na obrazovku terminálu text

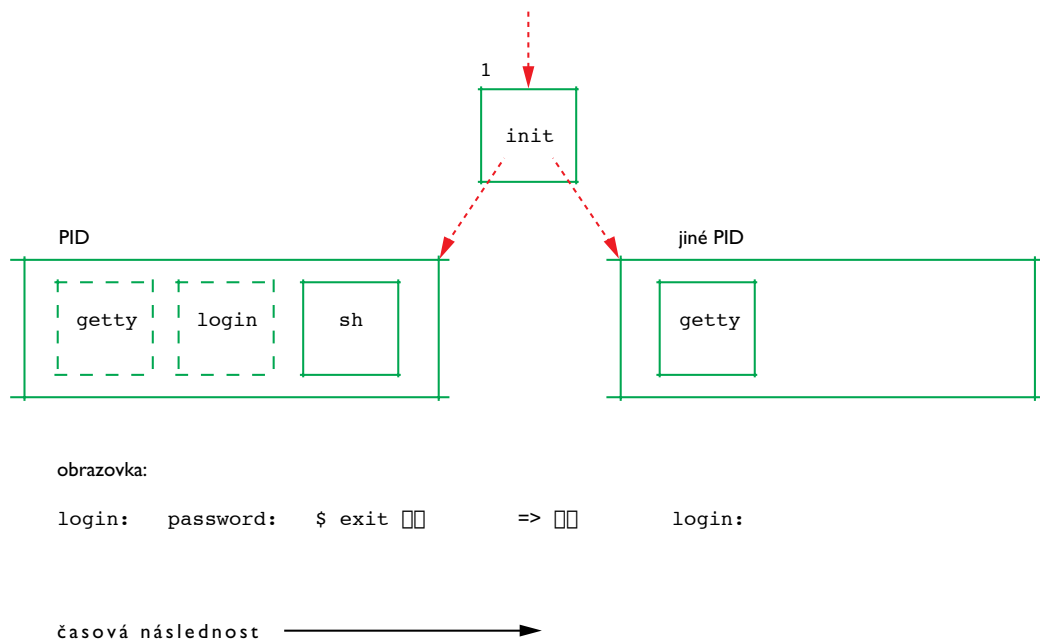
**login:**

Nový proces **getty** přitom může vzniknout již při kontrole vstupního hesla, protože není-li uživatelské heslo zadáno správně, proces **login** se nemění v **sh** pomocí **exec**, ale zaniká pomocí **exit** a **init** přebírá situaci zániku procesu s odpovídajícím PID.

Cyklus přihlášení a odhlášení uživatele ukazuje obr. 2.6.

Shell je komunikačním prostředkem uživatele s operačním systémem. Proces shellu přebírá od uživatele příkaz a iniciuje odpovídající akce. Je obvyklé, že pro zajištění uživatelské požadavky shell startuje nový proces, jak bylo popsáno v předchozím článku a ten může opět vytvářet další dětské procesy (typický je např. překlad programu, protože při spuštění překladače vzniká proces, který vytváří procesy





Obr. 2.6 Přihlašování uživatele u terminálu

jednotlivých průchodů překladu a řídí jejich činnost). Vzniká tak podstrom uživatelských procesů. Těchto podstromů je přitom tolik, kolik je současně přihlášených uživatelů.

Uživatelské procesy jsou podporovány při své činnosti procesy systémovými. Systémové procesy běží současně s procesy uživatelskými a provádí určitou systémovou funkci. Např. je to proces **lp** podle obr. 2.5. Tento proces běží trvale ve víceuživatelském režimu a vždy na požadavek uživatele na tisk na tiskárnu příkazem **lp** přebírá požadavek zařazený do fronty požadavků na tiskárnu. Pokud je tiskárna v klidovém stavu, aktivuje ji, protože ve frontě se objevil náš požadavek tisku. Systémový proces, který běží trvale v nekonečném cyklu a probouzí se k akci na základě výskytu určité události, se v UNIXu nazývá *démon* (daemon). Přestože existují v UNIXu i jiné typy systémových procesů, tj. ty, které provedou určitou akci a skončí, nebo čekají na určitou událost a skončí (typicky **getty** čeká na přihlašování uživatele), démony jsou používány jako běžný prostředek zajištění chodu určitého programového podsystému nebo přístupu k periférii atd. Správce systému často nové démony programuje a vkládá do systému sám. Každopádně jakýkoliv systémový proces se musí odlišovat od procesu uživatelského prioritou zpracování. *Priorita zpracování* je atribut procesu, který sleduje a spravuje jádro, a je přiřazována procesu v okamžiku jeho vytvoření. Proces může požádat jádro o změnu priority a zvýhodnit nebo znevýhodnit se tak oproti jiným procesům. K tomu slouží volání jádra **nice**, které může obyčejný uživatel používat pro snížení (tedy znevýhodnění) priority a privilegovaný uživatel ke

zvýšení (zvýhodnění). Démony a vůbec systémové procesy obvykle požadují vyšší prioritu zpracování (je to smysluplné, protože tak nedochází k zahlcení systému uživatelskými požadavky), proto musí být startovány privilegovaným uživatelem. Tento předpoklad je ale zajištěn hierarchií zpracování požadavků procesem **init**, protože ten je privilegovaný.

V okamžiku vytvoření procesu je nový proces kopií svého rodiče. Instrukci, kterou provádí jako první, je ta, která následuje za voláním jádra **fork** podle programu. Program je tentýž pro rodiče i dítě. Můžeme říct, že oba procesy v okamžiku vzniku dítěte jsou identické. Dítě tedy požaduje a při svém běhu používá tytéž výpočetní zdroje jako jeho rodič. Říkáme, že dítě *dědí* prostředí provádění od svého rodiče.

Předmětem *dědictví* (inheritance) dětského procesu od svého rodiče je především:

- identifikace vlastníka procesu a skupiny, do které vlastník patří,
- identifikace skupiny procesů, do níž je proces přiřazen,
- nastavení zpracování signálů, tj. komunikace mezi jádrem a procesy a mezi procesy navzájem,
- maska přístupových práv vytvářených souborů,
- pracovní a kořenový adresář,
- tabulka otevřených souborů (ta však není s rodičem sdílena, ale je pro dítě okopírována),
- identifikace terminálové skupiny.

Dětský proces má ale naopak unikátní identifikaci procesu PID. Jak už bylo řečeno, PID je unikátní označení procesu v rámci chodu operačního systému. PID přiřazuje a registruje jádro ve svých vnitřních tabulkách. Každý proces má ale právo potřebná PID svých dětí a rodičů zjišťovat a používat vzájemný synchronizační mechanismus. Proces své vlastní PID získá v návratové hodnotě volání jádra **getpid**. PID svého rodiče může získat voláním jádra **getppid** rovněž v jeho návratové hodnotě.

Dětský proces je na svém rodiči existenčně závislý. Znamená to, že pokud rodič z nějakého důvodu skončí, všechny jeho děti skončí také. Tento mechanismus je praktický, protože je-li potřeba ukončit práci celého podsystému procesů (programové systémy jsou vždy programovány jako systém procesů navzájem propojený vztahy rodičů a dětí), stačí ukončit proces, který je ve stromové struktuře procesů programového systému kořenem. Často ale můžeme požadovat existenční nezávislost procesu. Každý proces je účastníkem nějaké skupiny procesů. Jeden z procesů skupiny, v hierarchii kořen stromu, je označen jako *vedoucí skupiny* (leader of the group, někdy se mu říká rodič skupiny procesů). Proces se osamostatňuje, tj. stává se existenčně nezávislým na svém rodiči a vedoucím skupiny, voláním jádra **setpgrp**. Současně s nezávislostí se proces stává vedoucím takto nově vytvořené skupiny procesů. Vedoucím skupiny procesů je každý shell po přihlášení uživatele. Všechny jeho děti jsou na něm závislé. Zrušení kořene stromu uživatelských procesů znamená ukončení běhu všech jeho procesů. Na úrovni příkazového řádku může uživatel požadovat nezávislost nově vznikajícího procesu v době uživatelského sezení příkazem

**\$ nohup příkazový\_řádek &**

kde nově vznikající proces uvedený v části **příkazový\_řádek** je nezávislý na shellu, který jej vytváří jako dítě (uživatel se nyní může např. odhlásit bez obavy ukončení procesu spuštěného na pozadí). Program **nohup** používá fragment programu

```
setpgrp( );
exec("příkazový_řádek");
```

Dětský proces změnil své prostředí během volání jádra `setpgrp` a přepíše svůj program voláním jádra `exec`. Protože je to ale tentýž proces, prostředí běhu zůstává totožné.

PID vedoucího své skupiny proces získá z návratové hodnoty volání jádra `getpgrp`.

Synchronizační mechanismus procesů je realizován posíláním *signálů* (signals) procesům a je to metoda stejně stará jako UNIX. Proces může poslat jinému procesu signál pomocí volání jádra `kill`. Jeho název je odvozen od požadavku ukončení procesu a v počátcích UNIXu signály sloužily především k tomuto účelu. Proces v parametrech `kill` určuje typ signálu a PID procesu, kterému signál posílá.

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

je formát. Signál je celé číslo od 1 do 28. Běžně se používá textová definice, která je nastavena v souboru `<sys/signal.h>`. V příloze B uvádíme tabulku používaných signálů podle SVID. Proces žádá o signál jádro, které tento požadavek vyřizuje. Význam signálu podle tabulky uvedené v příloze naznačuje, že procesu posílá signály z vlastní iniciativy i jádro samo, a to tehdy, jestliže proces programovou chybou provádí např. neznámou instrukci, pokouší se zapisovat mimo svůj adresový prostor atp. Jádro procesu oznámí vzniklou situaci odpovídajícím typem signálu a proces, pokud nemá nastavenou reakci na tento typ signálu, obvykle končí svoji činnost (odtud `kill`). Tento význam byl v počátcích UNIXu dominantní. Tehdy také byl počet signálů 15 a jak ukazuje tabulka v příloze, proces byl implicitně ukončen. Pro použití uživatele byl tehdy důležitý signál č. 9 (`SIGKILL`), pomocí kterého mohl uživatel ukončit proces bez vyznačení jakéhokoliv důvodu (např. zacyklený proces, uváznutí na chybné operaci terminálu atd.). Signály nad hodnotou 15 byly postupně do systému vkládány za účelem rozšířit možnosti synchronizace procesů nebo s doplňovanými vlastnostmi systému. Efekt při příchodu signálu pak nemusí být jen ukončení procesu, ale třeba i ignorování (např. změna velikosti okna) nebo pozastavení (např. čekání na výstup na terminálu u procesu běžícího na pozadí).

S výjimkou signálu `SIGKILL` může proces na příchozí signál reagovat. Reakce je nastavována pomocí volání jádra `signal`, kde v parametrech zadáváme typ signálu a jméno funkce, která se provede v okamžiku příchodu signálu. Formát je

```
void (* signal(int sig, void (*func)(int)))(int);
```

Např. zachycení a zpracování signálu `SIGUSR1` funkcí `create_child` tak, že je vytvořen nový dětský proces, dosáhneme programem

```
void create_child()
{
    signal(SIGUSR1, create_child);
    fork();
}

main(void)
{
    signal(SIGUSR1, create_child);
    sleep(10);
    exit(0);
}
```

Ve funkci `main` nastavíme ihned po spuštění programu ošetření signálu `SIGUSR1` funkcí `create_child`. Poté proces neaktivně stojí 10 vteřin a skončí. Po dobu 10 vteřin může přijít signál `SIGUSR1`, kdy dojde ve funkci `create_child` k vytvoření dětského procesu. Poté je proveden návrat do `main` tak, že je pokračováno od místa přerušení, tj. proces opět čeká zbytek času. Počet dětských procesů pak odpovídá počtu signálů `SIGUSR1`, které rodič (nebo jeho děti) obdrží v průběhu 10 vteřin. Po 10 vteřinách jsou ukončeny všechny procesy, protože jejich existence je závislá na existenci výchozího procesu, a ten žije nejdéle 10 vteřin. Ve funkci `create_child` je ještě před `fork` znova použito volání jádra `signal`. Je to proto, že nastavení reakce na signál je po příchodu takového signálu zrušeno a mění se ve výchozí nastavení, kterým je u `SIGUSR1` ukončení procesu.

Přihlášený uživatel je z pohledu UNIXu nezávislý iniciátor vytváření procesů a jejich práce. Přihlášený uživatel musí mít pro svou práci k dispozici potřebné výpočetní zdroje, o které se dělí s ostatními přihlášenými uživateli. Všechny požadavky uživatel sděluje jádru prostřednictvím procesů, které vytváří. Tak pracuje každý obyčejný uživatel. UNIX totiž rozeznává dvě třídy uživatelů, obyčejné a privilegované. Privilegovaný uživatel je určen pro systémové změny, jako je např. výměna jádra, instalace nového aplikačního softwaru atd. Rozdíl mezi obyčejným a privilegovaným uživatelem je z pohledu práce procesů v uspokojení požadavků, které by mohly u obyčejného uživatele ohrozit konzistenci a chod operačního systému jako celku a narušit tak práci ostatních uživatelů. Je to např. zásah do tabulek systému v adresáři `/etc` nebo zásah do datové základny ostatních uživatelů. Při programování aplikací, které pracují se systémovými zdroji ale často obyčejný uživatel nevystačí s obvyklými možnostmi. Např. je-li vzniklý dětský proces řízen programem, který má provést zápis do některého systémového souboru (např. evidence o přístupu k periférii), musí mít dětský proces právo zápisu do odpovídajícího souboru, jiný dětský proces téhož uživatele ale nikoliv. Znamená to, že po dobu běhu určitého procesu (ale ne déle) propůjčí vlastník souboru s programem svou identifikaci uživateli, který proces spustil. Takové nastavení programu pro obyčejné uživatele provádí majitel programu (tj. majitel souboru) nastavením tzv. *s-bitu*.

Uživatelé jsou registrováni (viz kap. 5) jednoznačnou číselnou identifikací v tabulce `/etc/passwd` a jsou rozděleni do skupin. Skupiny jsou jednoznačně číselně registrovány v tabulce `/etc/group`. Každý soubor s programem má přiřazen vlastníka a skupinu souboru. Proces, který je řízen takovým programem, může zjistit *reálného vlastníka procesu* (real user), tj. uživatele, který jej spustil voláním jádra

```
#include <sys/types.h>
```

```
uid_t getuid(void);
```

a reálnou skupinu

```
gid_t getgid(void);
```

Pokud má soubor s programem kromě bitu proveditelnosti (x-bit) také nastaven s-bit, *efektivní vlastník* (effective user) a *efektivní skupina vlastníka procesu* (tj. uživatel a skupina vlastníka souboru s programem) se mohou od reálného vlastníka a reálné skupiny procesu lišit. Proces zjistí efektivního vlastníka procesu voláním jádra

```
uid_t geteuid(void);
```

a efektivní skupinu

```
gid_t getegid(void);
```

Jádro umožňuje přepínat mezi reálným a efektivním vlastníkem procesu pomocí volání jádra

```
#include <sys/types.h>
```

```
int setuid(uid_t uid);
```

a mezi reálnou a efektivní skupinou procesu pomocí

```
int setgid(gid_t gid);
```

kde `uid` a `gid` je identifikace vlastníka nebo skupiny výsledného efektivního vlastníka nebo skupiny procesu.

Proces má také k dispozici volání jádra

```
#include <unistd.h>
```

```
int access(const char *path, int amode);
```

kterým zjistí svá přístupová práva k souboru vzhledem k efektivnímu vlastníku a skupině procesu.

V parametru `amode` může používat logickou kombinaci hodnot `R_OK` pro test přístupu na čtení, `W_OK` zápis, `X_OK` proveditelnost souboru a `F_OK` jako test pouhé existence souboru.

Cestou nastavení s-bitu umožňuje privilegovaný uživatel obyčejnému pracovat řízeně se systémovým zdrojem, na který jiným způsobem nemá právo. S-bit je také používán obyčejným uživatelem k poskytování řízeného přístupu ostatním uživatelům k jeho lokálním výpočetním zdrojům (jde především o data, tj. přístup k souborům v jeho vlastnictví). Uživatel nastavuje s-bit v příkazovém řádku pomocí příkazu **chmod**, např.

```
$ chmod g,o+s prog
```

což znamená, že jak pro skupinu (**g**), tak pro ostatní (**o**) je umožněno spouštět program v souboru **prog** s efektivním uživatelem a skupinou vlastníka souboru. Proces **chmod** využívá pro toto nastavení volání jádra

```
#include <sys/stat.h>
```

```
int chmod(const char *path, int protect);
```

kde `path` je jméno souboru a `protect` přístupová práva k souboru.

Vytváří-li proces nový soubor, přístupová práva souboru jsou stanovena podle *masky přístupových práv* (file creation mode). Jedná se o bitové pole, které je použito při volání jádra systému souborů **creat** nebo **open** s bitovým součinem jejich argumentu. Výsledek určuje přístupová práva nově vznikajícího souboru. Masky přístupových práv, která je děděna z rodiče na dítě, je v shellu uživateli dostupná ke čtení i změně pomocí příkazu

```
$ umask
```

Uživatel tak může dětským procesům stanovit přístupová práva nově vznikajících souborů. Proces mění masku přístupových práv voláním jádra

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
mode_t umask(mode_t cmask);
```

Maska je implicitně nastavena na 0133 (osmičkově).

*Pracovní adresář* (working directory, current directory) je místo v systému souborů, odkud byl proces jiným procesem startován. Při spouštění procesu uživatele z příkazového řádku je to pracovní adresář jeho procesu shell, který je dán posledním použitím jeho vnitřního příkazu **cd**. Proces mění svůj zděděný pracovní adresář pomocí volání jádra

```
int chdir(const char *path);
```

kde v parametru **path** stanovuje jméno nového pracovního adresáře.

*Kořenovým adresářem* (root directory) procesu rozumíme pak výchozí adresář v hierarchické struktuře adresářů, tj. ten, pro který se vztahuje začátek cesty k souboru označením /. Obvykle je kořenový adresář procesu totožný s kořenovým adresářem celého systému souborů běžícího UNIXu, ale proces může voláním jádra

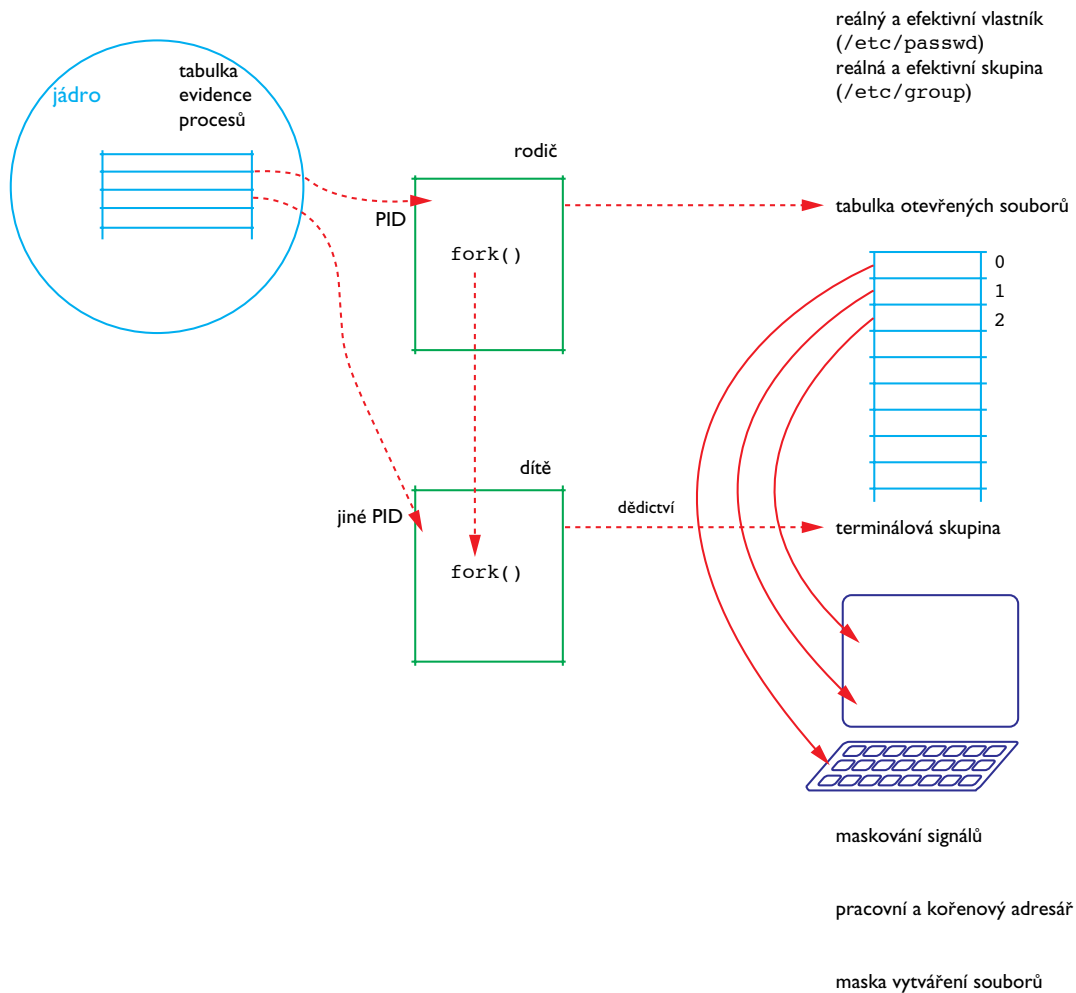
```
int chroot(const char *path);
```

změnit svůj výchozí kořenový adresář na jiný. Kořenový adresář je součástí dědictví procesu po rodiči a programátor pomocí něj může simulovat práci v dané hierarchii adresářů<sup>1</sup>.

*Tabulka otevřených souborů* (file descriptors, tabulka deskriptorů) procesu je další seznam výpočetních zdrojů, které jsou převedeny z rodiče na dítě. Dětský proces tak může pokračovat s používáním otevřených souborů rodiče. Je přitom důležité si uvědomit, že se nejedná jen o datové soubory, ale jde také o periferie, které jsou uživateli a programátorovi viditelné jako zvláštní druhy souborů. Tabulka otevřených souborů je vlastně tabulka kanálů, jejichž číselné označení je vstupním bodem této tabulky.

Použije-li proces volání jádra **open** k otevření souboru, znamená to, že je z této tabulky využita první volná pozice. Návrátová hodnota **open** je právě číslo kanálu, které proces dále používá při manipulaci s obsahem souboru (např. čtení nebo zápis dat pomocí dalších volání jádra **read** nebo **write**). Proces **init** vytváří procesy **getty** pro zpřístupnění terminálu uživateli. **getty** jako první z akcí otevírá označený terminál jako soubor, a to třemi způsoby: pro čtení, zápis a ještě jednou zápis. Znamená to, že jsou použity kanály č. 0, 1 a 2. Kanál č. 0 je otevřen pro čtení, jde tedy o klávesnici terminálu, 1 pro zápis, je to obrazovka terminálu, a 2 opět pro zápis (chybových zpráv), opět obrazovka. Tyto tři kanály dědí pak všechny dětské procesy už jako otevřené soubory. Vzniklý proces se proto již nemusí zabývat vytvářením přístupu k terminálu. Proces nemusí také otevírat soubory, které již otevřel rodič, a může s kanály různě manipulovat, tj. spojovat je, kopírovat atd. Samozřejmě je může také uzavírat voláním jádra **close** a tím je odpojovat od zpracování (typický případ odpojení všech zděděných kanálů je u procesů démonů, protože tyto potřebují nezávislost na jakémkoliv souboru nebo zařízení).

Konečně posledním předmětem dědictví dítěte je *terminálová skupina*. Číselná identifikace terminálové skupiny je PID procesu, který terminál otevřel. V normálním běhu systému je to každý shell (terminál otevřel **getty**, ale to je tentýž proces). Proces může odkazovat na takový terminál prostřednictvím souboru periferie **/dev/tty**. Identifikace terminálové skupiny se uplatňuje v rámci skupiny procesů. Je to PID vedoucího skupiny. Znamená to také, že se hodnota terminálové skupiny změní, použije-li proces volání jádra **setpgrp** pro osamostatnění se a založení nové skupiny. Číselná identifikace terminálové skupiny se sice změní na PID procesu, který provedl **setpgrp**, ale označení **/dev/tty** zůstává. Signály, které vznikají generací z klávesnice (je to SIGINT, klávesou Del nebo Ctrl-c), jsou posílány procesu s PID terminálové skupiny. V naposledy uvedeném příkladu použití funkce



Obr. 2.7 Proces a dědictví rodiče

`create_child` při zachycení signálu dochází k ukázkové situaci při výměně `SIGUSR1` za `SIGINT`. Pokud se žádný z dětských procesů neosamostatní, vytváří nové děti vždy výchozí proces. Situaci při vytváření nového procesu z pohledu dědictví z rodiče na dítě ukazuje obr. 2.7.

## 2.3 Proces z pohledu uživatele

Registrovaný uživatel má oprávnění se přihlásit (vstoupit) do systému. V okamžiku přihlášení vzniká proces interaktivní komunikace uživatele se systémem shell, který má reálnou i efektivní identifikaci vlastníka přihlášeného uživatele a který je také výchozím procesem stromu uživatelských procesů. Všechny uživatelské procesy vznikají, běží a zanikají v určitém prostředí, které je nastaveno v okamžiku přihlášení uživatele a je čitelné jak na úrovni volání jádra, tak na úrovni procesu shellu. V této části se zaměříme především na úroveň volání jádra, přestože s procesem shell budeme udržovat stálý kontext. Vlastní proces shell z tohoto pohledu bude popsán v závěrečném článku této kapitoly.

Uživatel, účastník výpočetního systému, používá proces jako nejmenší jednotku zpracování. UNIX podporuje propojování a komunikaci procesů, které programátor nebo uživatel spojuje do programových systémů a aplikací. Výhoda nezávislých částí programového systému je známá v pojmech modulárního programování. Proces je oddělen od dat jiných procesů operačním systémem. Plní jednu dílčí funkci, kterou ale vykonává spolehlivě. Jakýkoliv pokus procesu o přístup do adresového prostoru mimo jeho kompetence znamená odmítnutí operačním systémem. Celková práce programového systému se tak zprůhlední a chyby se objevují tam, kde skutečně jsou.

### 2.3.1 Řídící hodnoty při vzniku dítěte

Při programování v jazyce C program, který řídí proces, obsahuje funkci `main` jako výchozí startovací místo procesu. Funkce `main`, která je povinná, má formát

```
int main(int argc, char *argv[], char *envp[]);2
```

a ve svém těle obsahuje instrukce programu. Argumenty funkce `main` (které jsou nepovinné a mohou být nahrazeny definicí `void`) jsou parametry, které může používat proces rodiče pro přenos dat do procesu řízeného tímto programem. V prostředí shellu, bude-li jméno proveditelného souboru např. `prog`, to znamená, že příkazový řádek

```
$ prog -c baby
```

je uložen v parametrech funkce `main` programu `prog` takto:

```
argc      3
argv[0]    "prog"
argv[1]    "-c"
argv[2]    "baby"
```

Proměnná `argc` obsahuje počet parametrů příkazového řádku (včetně jména programu). Textové řetězce `argv` jsou postupně naplněny jednotlivými texty příkazového řádku.

Třetí parametr `envp` funkce `main`, který může být vynechán při použití pouze prvních dvou, je pole textových řetězců, které definují prostředí provádění procesu, v tomto případě některé proměnné shellu. Jsou to texty, jejichž obsah je ve tvaru

```
PROMĚNNÁ=OBSAH
```

tedy dvojice textových řetězců oddělených znakem '='. Jedná se o možnost krátkých textových instrukcí pro vlastní běh procesu. Shell přenáší všechny takové proměnné, které jsou označeny před vznikem dětského procesu, vnitřním příkazem `export` (podrobněji viz článek 3.6). Seznam textových řetězců



pole ukazatelů `envp` je ukončen ukazatelem na textový řetězec prázdné délky (označovaný v C jako `NULL`). Pokud programátor použije v definici funkce `main` pouze proměnné `argc` a `argv`, může získat prostředí provádění procesu ve stejném tvaru jako `envp` pomocí definice proměnné

```
extern char **environ;
```

V těle funkce `main` může programátor používat zpracování uvedených tří typů argumentů převzatých od procesu rodiče, jejich modifikace má ale pouze lokální platnost, protože sice mohou být dále přeneseny do případných dalších dětských procesů, ale nejsou zpětně viditelné rodičem.

Na úrovni volání jádra jsou argumenty funkce `main` definované v rodiči ve volání jádra `exec`. `exec` má několik tvarů (jejich definice je v `<sys/unistd.h>`).

```
int execl(const char *path, const char *arg, ...);
```

```
int execv(const char *path, char *const *argv);
```

Uvedený první tvar volání jádra `exec` používá v parametrech `argv` pouze texty, které jsou přeneseny do parametrů `main` `argc` a `argv` (v shellu příkazový řádek uživatele). Např. shell po úspěšném `fork` v dětském procesu provede

```
execl("/bin/ls", "ls", "-l", NULL);
```

pokud chce zabezpečit příkaz uživatele

```
$ ls -l
```

Parametr `path` ve volání `exec` je cesta k souboru s programem, který bude vyměněn za dosavadní a který bude řídit dětský proces, zbytek je příkazový řádek ukončený ukazatelem na prázdný řetězec. Varianta `execv` umožňuje vložit textové řetězce příkazového řádku do textového pole, náš příklad by tedy mohl mít kód

```
child_array[0]="ls";
child_array[1]="-l";
child_array[2]=NULL;
execv("/bin/ls", child_array);
```

Prostředí provádění procesu `envp` vyvezeme do dětského procesu do argumentu `envp` funkce `main` pomocí volání jádra

```
int execl(const char *path, const char *argv0, ...
        /* char const *envp */);
```

nebo

```
int execve(const char *path, char *const *argv, char *const *envp);
```

kde v prvním případě vyjmenujeme textové řetězce jako argumenty a ve druhém můžeme opět použít textové pole pro přenos prostředí `envp`. Pokud má shell pomocí vnitřního příkazu označeny pro export do dětských procesů např. proměnné `PATH`, `HOME` a `TERM`

```
$ export PATH HOME TERM
```

a pokud je jejich obsah po řadě např. `./sbin:/usr/sbin`, `"/usr2/people/petr"` a `"ansi"`, shell používá pro vytvoření dětského procesu `exec` takto

```
child_env[0]="PATH=./sbin:/usr/sbin";
```

```
child_env[1]="HOME=/usr2/people/petr";
child_env[2]="TERM=ansi";
execl("/bin/ls", "ls", "-l", child_env);
```

Konečně poslední možnost je použít

```
int execlp(const char *file, const char *argv0, ...);
int execvp(const char *file, char *const *argv);
```

kde je oproti `execl` a `execv` vyměněno pouze jméno `path` za `file`. V tomto způsobu volání proměnná `file` může obsahovat i neúplnou cestu k souboru s programem. Dětský proces vnutí do obsahu argumentu `argv[0]` v tomto případě úplnou cestu, a to podle obsahu proměnné `PATH`, definované v prostředí provádění `envp` jako seznam adresářů, ve kterých postupně hledá soubor odpovídajícího jména (jména adresářů jsou v `PATH` oddělena znakem ':').

Např. budeme-li chtít psát program, který na základě příchodu signálu `SIGUSR1` vytvoří dítě, které bude řízeno programem podle příkazového řádku textu za volbou `-c` rodiče, zdrojový kód v jazyce C bude

```
#include <stdio.h>

void create_child()
{
    signal(SIGUSR1, create_child);
    switch((fork()))
    {
        case -1: /* v případě chyby při vytváření dítěte */
        case 0: return 0; /* nebo v případě rodiče */
        default: execvp(ch_argv[ch_from], &ch_argv[ch_from]);
    }
}

void ex_func()
{
    exit(0);
}

extern int optind;
char *ch_argv[];
int ch_from;

int main(argc, char *argv[])
{
    int errflg=0, ch_from=0, c;
    ch_argv=argv;
    while((c=getopt(argc, argv, "c:"))!=EOF)
    {
        switch(c)
        {
```

```

        case 'c':
            ch_from=optind;
            break;
        default:
            errflg++; break;
    }
}
if(errflg)
{
    fprintf(stderr, "prog: usage: prog [-c ""name_of_child""]\n");
    exit(1);
}
signal(SIGUSR1, create_child);
signal(SIGTERM, ex_func);
for(;;);
}

```

Instrukce programu (tělo funkce `main`) v první části (cyklus `while`) provádí analýzu příkazového řádku pomocí standardní funkce `getopt`. `getopt` analyzuje obsah proměnných `argc` a `argv` a hledá v příkazovém řádku volbu `-c` definovanou v řetězci posledního argumentu funkce `getopt`. V případě nalezení nastavuje proměnnou `ch_from` na hodnotu indexu pole `argv`, která následuje za touto volbou. V případě použití jakékoliv jiné volby program vypisuje pomocí standardní funkce `fprintf` text jejího argumentu do kanálu č. 2 a končí. Pokud proces pokračuje dál, nastaví reakci na příchod signálů `SIGUSR1` a `SIGTERM`. `SIGTERM` je pokyn k ukončení nekonečného cyklu, který následuje jako poslední instrukce programu. Funkce `create_child`, která ošetří příchod signálu `SIGUSR1`, provádí `fork` a analyzuje výsledek provedení tohoto volání jádra. V případě neúspěchu proces pokračuje dál beze změny. V případě vytvoření dítěte je použito `execvp` pro nastavení příkazového řádku dítěte na požadovaný v argumentu, dítě tedy bude řízeno programem podle zbytku příkazového řádku rodiče za volbou `-c`. V případě, že jméno proveditelného souboru s právě uvedeným programem bude `prog`, příkazový řádek

```
$ prog -c "ls -l" &
```

znamená, že shell vytvoří proces se jménem **prog**, který bude řízen našim programem. Proces poběží v nekonečném cyklu, dokud nepřijde signál `SIGTERM`, který ho ukončí kdykoliv. V době běhu procesu **prog**, pokud mu bude zaslán signál `SIGUSR1`, vytvoří **prog** dětský proces, který bude řízen programem `ls` s volbou `-l`. Podrobnosti ohledně popisu funkcí `getopt` a `fprintf` nalezne čtenář v dokumentaci každého UNIXu, který odpovídá doporučení SVID.

Nekonečný cyklus `for(;;)` znamená čekání procesu na příchod události. Je to ale aktivní čekání, které může zbytečně zatěžovat operační systém. Programátor může použít volání jádra `pause`, které jádro požádá o pozastavení procesu v rámci systému až do příchodu signálu. Program v těle cyklu `for` tedy může obsahovat toto volání jádra

```
for(;;)pause();
```

To znamená, že opakování těla cyklu (tj. novou žádost o pozastavení) proces provede teprve po ošetření přijatého signálu.

### 2.3.2 Ukončení dítěte

Návratová hodnota funkce `main` je celočíselná hodnota. Proces, který vytvoří dítě, může vyčkat na jeho dokončení a získání návratové hodnoty funkce `main` dítěte, kterou dítě stanovuje v parametru volání jádra `exit`. K tomu rodiči slouží volání jádra `wait`, které jsme už použili v příkladu na obr. 2.2 a má formát

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *stat_loc);
```

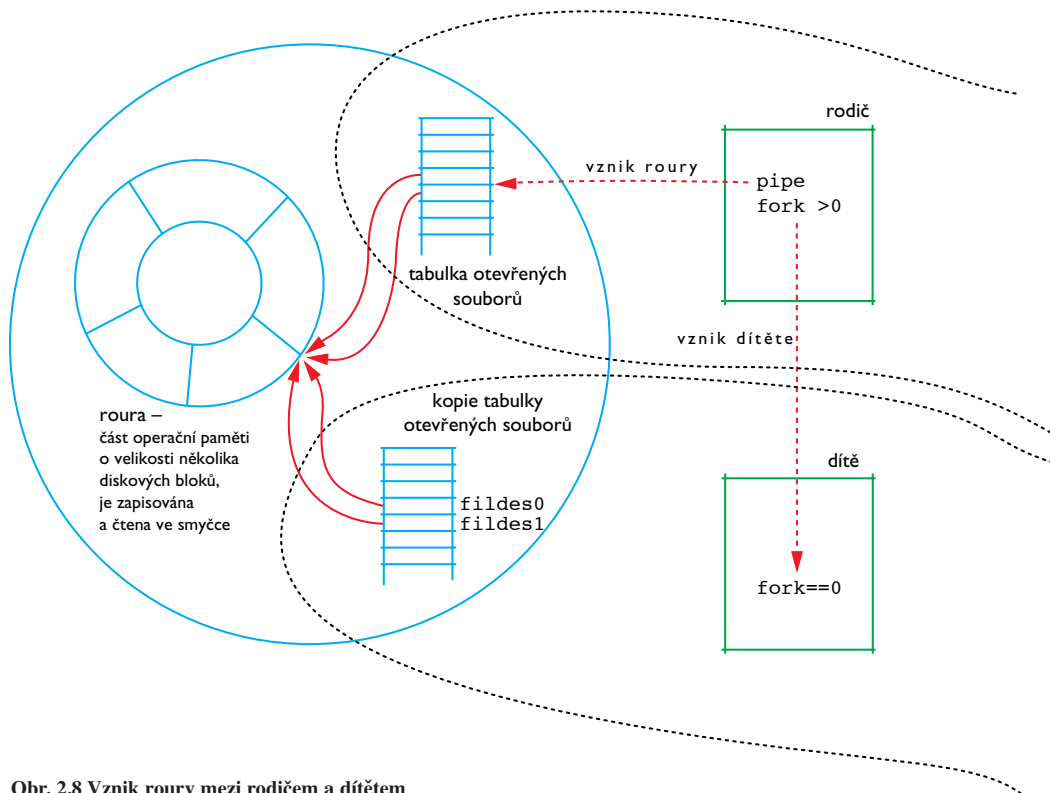
Použitím `wait` je proces *pozastaven* (tzv. zablokován) a operační systém jej *probouzí* (odblokuje) při dokončení některého z jeho dětských procesů. Návratová hodnota volání jádra `wait` je PID dětského procesu, který ukončil běh. Tak může rodič čekat na konkrétní dítě, protože očekávanou návratovou hodnotu může registrovat při vytvoření dítěte v návratové hodnotě `fork`. Parametr `stat_loc` je jádrem naplněn hodnotou, kterou použije dítě v parametru volání jádra `exit`. Tak může dítě sdělovat rodiči úspěch nebo důvod neúspěchu své činnosti. Pro pohodlnější zpracování ukončení dětí má rodič ještě k dispozici funkci `waitpid`, kde v parametru stanoví PID dítěte, o které má zájem.

Hodnoty `stat_loc` po provedení volání jádra `wait` využívá shell u řídících struktur interpretu. Uživatelé příkazového řádku běžně neviditelná návratová hodnota dětského procesu je ale snadno viditelná procesu shell, který ji může dát do požadovaného kontextu některé z podmínek **if**, **while**, **until** atd. (viz. čl. 2.5).

Proces může volání jádra `wait` použít kdykoliv po ukončení dětského procesu. Použitím `wait` převezme rodič status ukončení dítěte a tyto informace o dítěti může systém zapomenout. Pokud rodič status dítěte nepřevzme, systém stále registruje odpovídající informace o procesu, který již vlastně neexistuje. Pokud rodič vytváří stále další dětské procesy, mohou tyto informace zbytečně zatěžovat systém. Proto je proces na ukončení dětského procesu upozorněn příchodem signálu `SIGCHLD`. Použití `wait` ve funkci, která tento signál ošetří, znamená převzetí informace o ukončeném dítěti a zbytky dětského procesu jsou systémem zapomenuty. Dětský proces, který ukončil svoji činnost, ale jehož ukončení rodič zatím nepřevzal, je stále evidován jádrem v tabulce procesů a nazývá se *mátoha* (zombie).

### 2.3.3 Komunikace rodiče a jeho dětí

V úvodu článku jsme zdůraznili funkci procesu jako jednotky výpočtu programového systému. K tomu, aby proces mohl být součástí celku více procesů, vyžaduje možnosti komunikace procesu se svým okolím. Komunikace procesů v obecném slova smyslu, zejména z pohledu technologie klient – server, je předmětem kap. 5. V předchozím článku 3.3 jsme v rámci popisu hierarchie systémového stromu procesů uvedli základní možnosti komunikace rodičů a dětí, používání signálů. Vzhledem k tomu, že proces může snadno pomocí volání jádra `getpid`, `getppid`, `getpgid`, `fork` atd. (viz čl. 3.3) rozeznávat své předky a potomky, může s nimi snadno komunikovat pomocí signálů. Komunikace mezi procesy směrem shora dolů (tj. z rodiče na dítě) je možná pomocí parametrů `argc`, `argv` a `envp` volání jádra `exec`, které jsou viditelné ve funkci `main` dětského procesu. Parametry `argc`, `argv` a `envp` lze v dětském procesu modifikovat, ale tato změna se nikdy nepřenáší do oblasti, kde by byla



Obr. 2.8 Vznik roury mezi rodičem a dítětem

čitelná rodičem. Rodič může zjišťovat výsledky chování svých dětí pomocí obsahu parametru volání jádra `wait`, kam se promítá hodnota parametru volání jádra `exit` dětského procesu. Tyto možnosti ovšem nestačí ke komunikaci, kdy vyžadujeme meziprocesový tok dat mezi rodičem a dítětem. Situaci lze řešit pomocí vytvoření dočasněho souboru v některé z veřejných oblastí, např. `/tmp` nebo `/usr/tmp`, ale nastávají problémy se synchronizací operací čtení a zápisu. Přenos dat je zpomalován menší rychlostí disku a režii systému souborů. Jiná možnost je ovšem v UNIXu zajištěna pomocí volání jádra `pipe`, vytvořením roury. Roura je v UNIXu známý jednosměrný komunikační prostředek dvou procesů. Uživatel procesu shell jej používá pomocí znaku zvláštního významu `|`, takže např. dva procesy, `ls` a `wc` na příkazovém řádku spojuje rourou

```
$ ls | wc -l
```

Výsledkem je předání výstupních dat procesu `ls` na vstup procesu `wc`. Vstupem a výstupem procesu zde máme na mysli standardní vstup a standardní výstup, tedy kanály č. 0 a 1, které jsou spojeny s klávesnicí a obrazovkou terminálu. V našem příkladě je kanál č. 1 procesu `ls` přesměrován na kanál č. 0 procesu `wc`. Výstup `wc` je stále obrazovka terminálu a vstupem `ls` (pokud by jaký vyžadoval) je klávesnice. Shell tedy znak `|` interpretuje ve smyslu

## \$ PRODUCENT | KONZUMENT

kde mezi procesy PRODUCENT a KONZUMENT stojícími vlevo a vpravo od znaku | je uvedeným způsobem vytvořena roura (podrobněji viz čl. 2.5).

Proces vytváří rouru pomocí volání jádra `pipe`. Má formát

```
int pipe(int fildes[2]);
```

Jádro vytvoří kanál pro tok dat. V parametru pole o dvou prvcích proces získá od jádra označení pro zápis dat (celočíselná hodnota `fildes[0]`) a pro čtení dat (hodnota `fildes[1]`). Jedná se o jedno-směrný kanál, kdy proces může používat jak zápisový konec roury, tak čtecí konec. Deskriptory `fildes[]` jsou součástí dědictví procesů z rodiče na dítě, proto mohou procesy používat rouru jako další způsob komunikace mezi rodičem a dítětem. Modelovou situaci ukazuje obr. 2.8.

Jádro přiřadí rouře neobsazené kanály tabulky otevřených souborů procesu, který o vytvoření roury žádal. Po vytvoření dítěte je součástí dědictví i tato tabulka. Čtení i zápis do roury je tedy možný jak z procesu rodiče, tak z procesu dítěte. Vzhledem k tomu, že v okamžiku vzniku dítěte řídí oba procesy tentýž program, zajistí si tento čtení nebo zápis patřičného procesu. Program, kdy dítě pošle rodiči sto tisíc bytů dat rourou, má takovýto zdrojový text:

```
int roura[2], i;
char slabika="a", buffer;

int main(void)
{
    pipe(roura);
    switch(fork())
    {
        case -1:
            exit(-1);
        case 0: /* dítě, proces PRODUCENT */
            for(i=0; i<100000; i++)
                write(roura[0], slabika, 1);
            close(roura[0]);
            exit(0);
        default: /* rodič, proces KONZUMENT */
            while(read(roura[1], buffer, 1)==1);
            exit(0);
    }
}
```

Shell dosáhne propojení PRODUCENTA a KONZUMENTA stejným způsobem, ale jako zatím nepoužitý následný krok provede spojení kanálů roury s kanálem obrazovky a klávesnice (pomocí volání jádra `close` uzavře obrazovku i klávesnici a pomocí volání jádra `dup` kanály `fildes` spojí s kanály č. 0 a 1) a pak v procesu rodiče i dítěte provede volání jádra `exec`. Od provedení `exec` jsou procesy řízeny jinými programy. Upravené dědictví ale zůstává a procesy, aniž by jejich programy vyvinuly jakoukoliv činnost navíc, namísto na obrazovku a z klávesnice zapisují do roury nebo z ní čtou. Konkrétní příklad

s diskuzí uvedeme v čl. 2.5 a komunikaci procesů obousměrně při implementaci roury technologií STREAMS pak v části o obousměrné rouře v kap.5.

Možnosti komunikace rodiče a dětí můžeme shrnout následovně.

Od rodiče k dítěti: pomocí parametrů `argc`, `argv`, `envp` a volání jádra `exec`.

Od dítěte zpět k rodiči: parametrem `wait` v procesu rodiče je tatáž hodnota parametru `exit` dítěte.

Synchronizace rodiče a dítěte (rovnocenně rodič i dítě): pomocí signálů. Voláním jádra `kill` proces posílá signál. Způsob zpracování signálu nastaví proces pomocí volání jádra `signal`. Proces získá potřebné PID rodiče pomocí volání jádra `getppid`, vedoucího skupiny `getpgid`, dítěte `fork` a `wait`. Typy signálů jsou uvedeny v příloze B.

Tok dat (rovnocenně rodič i dítě): rourou. Rouru vytvoříme voláním jádra `pipe`. Přenos dat dosáhneme voláním jádra `write` a `read` jako při v/v operaci s otevřeným souborem.

### 2.3.4 Démon

*Démon* (daemon) je systémový proces, který běží v nekonečném cyklu. Je připraven na výskyt určité události a když nastane, proces se probouzí a vykoná určitou akci, která podporuje operační systém v jeho běhu. Událostí může být uplynutí určitého časového kvanta (např. démon **cron**), požadavek obsluhy síťového spojení (síťový superserver, démon **inetd**) nebo např. příchod požadavku tisku (démon **lpsched**). Démon po probuzení provede odpovídající akci a opět čeká na výskyt události, kterou ošetřuje. Démon je startován obvykle při vstupu operačního systému do víceuživatelského režimu, protože zajišťuje podporu práce přihlášených uživatelů, a naopak v jednouživatelském režimu může správci systému překážet při globální manipulaci se systémovými zdroji. Démon bývá proto spouštěn v rámci evidence tabulky `/etc/inittab` nebo některého scénáře pro shell, který s touto tabulkou souvisí. Je ale také možné, že ho spouští privilegovaný uživatel z příkazového řádku svého sezení. Každopádně takový proces démon musí přežít konec sezení každého uživatele nebo jiné systémové akce prováděné v rámci běhu systému. Démon končí svoji činnost teprve na základě pokynu procesu **init** nebo jiného procesu, kterému k tomu dal oprávnění správce systému, nebo na základě přímého požadavku privilegovaného uživatele.

Démon je konstruován na základě následujících pravidel.

Je evidován jako dětský proces procesu **init**. Toho dosáhne použitím volání jádra `setpgrp`. Získá tak nezávislost, pokud jej vytvoří jiný proces než **init**. Rovněž tak je dosaženo odpojení řídicího terminálu, tj. v případě, že bude démon spuštěn z příkazového řádku, reakce ovladače terminálu na něj nemají vliv (např. při vypnutí terminálu, podrobněji viz kap 6).

Odpojí se od všech kanálů vstupu a výstupu. Démon na každý kanál tabulky otevřených souborů aplikuje volání jádra `close`.

Běží v nekonečném cyklu, tj. používá řídicí strukturu `for(;;)` nebo `while(1)`.

Při své práci čeká neaktivně na výskyt události, která je mu obvykle sdělována příchodem nějakého signálu. Proces se vzdá soupeření o systémové zdroje použitím volání jádra `pause`. Teprve příchod jakéhokoliv signálu jej z tohoto stavu probouzí. Po dobu od `pause` do příchodu signálu se jádro nesnaží udržet proces v operační paměti, ani jej neuvažuje při výběru procesu, kterému svěří čas procesoru. Proces ale stále žije se všemi svými atributy.

Démon končí svoji činnost přijetím smluveného signálu od oprávněného procesu. Smluveným signálem je SIGTERM. Tento signál vyvede démon z nekonečné smyčky a proces končí svoji činnost. Ve funkci ošetřující SIGTERM může démon uskutečnit některé závěrečné akce své činnosti.

Pokud vytváří dětské procesy, přebírá status jejich ukončení, aby mátohy nezahltily systém. Znamená to, že reaguje na signál SIGCHLD a používá volání jádra `wait`.

Následuje příklad zdrojového textu jednoduchého démonu, který se probouzí vždy po intervalu 20 vteřin a spouští proces se jménem **baby**.

```
end_of_daemon()
{
    exit(0);
}

end_of_child()
{
    int status;
    wait(&status);
    return 1;
}

action()
{
    switch(fork())
    {
        case -1: return 0;
        case 0: execl("/usr/local/bin/baby", "baby", NULL);
        default: return 1;
    }
}

main(void)
{
    int fd;
    setpgrp();
    for(fd=0; fd<OPEN_MAX; fd++)close(fd);
    signal(SIGTERM, end_of_daemon);
    for(;;)
    {
        signal(SIGCHLD, end_of_child);
        signal(SIGALRM, action);
        alarm(20);
        pause();
    }
}
```



Pro správce systému je důležitý systémový démon **cron**, který obsluhuje požadavky spouštění procesů v určitém čase. Dokáže také spouštět procesy opakovaně vždy po uplynutí určitého časového kvanta. Jeho popis a manipulace s ním je uvedena v závěru čl. 5.2.

### 2.3.5 Proces a program

Překlad zdrojového textu v jazyce C do kódu stroje a jeho sestavení programátor provádí příkazem **cc**. Příkaz **cc** je kompilace podle rozhraní označovaného často jako Kernighan and Ritchie C, podle autorů jazyka definovaného koncem 70.let. Později vzniklý standard jazyka C se od původní definice liší, proto POSIX definuje pro překlad a sestavení programu ve zdrojovém jazyce C za tímto účelem příkaz **c89**. **cc** i **c89** je dostupný ve většině současných systémů. V obecném použití mají shodné příkazové řádky a často, jak POSIX doporučuje, je **c89** možné používat v rámci volby příkazu **cc**.

`/bin/cc` je program, který řídí jednotlivé fáze překladu programu ve zdrojovém kódu do podoby proveditelného souboru. Fáze překladu jsou preprocesor (textový makroprocesor), jednotlivé průchody překladu, optimalizace, assembler a sestavení. Fáze vznikají jako dětské procesy procesu **cc**. Programátor přitom může odpovídající volbou překlad v některé z fází zastavit a zkoumat výsledný mezikód např. po provedení náhrady textových maker nebo po převodu do assembleru stroje. Programátor může používat kompilátor jednoduchým způsobem

```
$ cc file.c
```

kde v souboru se jménem `file.c` očekává **cc** zdrojový text programu. Takto použitým příkazovým řádkem požadujeme provedení všech fází překladu. Výsledkem je tedy soubor s proveditelným kódem, který je uložen v souboru se jménem `a.out`. Programátor může použít volbu **-o** pro změnu výsledného jména souboru, ale při ladění programu se to nedoporučuje, protože nastavení sekvence adresářů, které příkazový interpret shell prohledává, často preferuje především systémové adresáře. Může se proto stát, že jméno programu, které uživatel použije, je shodné s některým systémovým nástrojem (kterých je několik stovek) a při spuštění se použije řídicí program takového nástroje, nikoliv testovaný program (viz odst. 2.5.2, proměnná `PATH`). Struktura obsahu binárního souboru `a.out` je dána implementací konkrétního UNIXu a je dostupná ve svazku (4) provozní dokumentace. Strukturu především rozumí operační systém (tj. jádro), protože obsah `a.out` používá při volání jádra `exec` pro další řízení procesu. AT&T zavedla jednotící strukturu `a.out` v rámci popisu binárních spustitelných souborů *coff* (common object file format), která ale není součástí SVID nebo dokonce POSIXu. Formát *coff* je vyjádřením používaných částí procesu z pohledu mapování paměti a přístupu k segmentům běžícího procesu. Programátor strukturu `a.out` může ovlivnit pouze zadáváním voleb v příkazu **cc** (nebo **ld**, což je spojovací program a je poslední fází **cc**) ve smyslu zadání typu výsledné binární formy, která je dána konkrétním typem procesoru a projeví se v záhlaví `a.out` jako součást *magického čísla* (magic number). Podrobnosti ohledně formátu *coff* a jeho následníka *elf* nalezne programátor v provozní dokumentaci.

Proces je jádrem vytvořen a programátorem viditelný podle obr. 2.9 v části adresového prostoru samotného procesu a v části evidence jádrem. Pro evidenci jádro používá struktury, které jsou součástí jádra a nejsou na obrázku vyznačeny, protože jsou programátorovi běžně neviditelné. Jsou popsány v následujícím čl. 2.4.

Jednotlivé segmenty struktury procesu na obrázku jsou *text*, *data* a *zásobník*. Text je kopie programu ze souboru *a.out*. Textový segment je část procesu, která, pokud není řečeno jinak, je určena pouze ke čtení. Samotný proces při řízení tímto kódem ji nemůže modifikovat. Je ale možné podle způsobu sestavení (v příkazovém řádku **cc**) vytvořit v *a.out* magické číslo, které obecně znamená možnost přepisu i textového segmentu. Ztrácí se přitom ale výhoda udržování jednoho textového segmentu v systému pro všechny procesy řízené jedním programem. Každý proces musí mít tedy svůj unikátní textový segment. UNIX implicitně udržuje pro více procesů jednoho programu tentýž text. Část *data* procesu má omezenou velikost danou možnostmi adresace konkrétního operačního systému, případně konkrétního hardwaru. Textový segment je umístěn od virtuální adresy 0, zatímco od opačné hodnoty, tj. od maximální velikosti adresového prostoru procesu, je umístěn prostor pro zásobník (stack) procesu. Jsou to *data* procesu, která se dynamicky mění podle práce procesu, tj. podle dočasně alokovaných dat např. při volání funkcí parametry funkce a její proměnné atd. Velikost zásobníku proto narůstá podle potřeb procesu směrem k datovému segmentu (je to volná paměť, kterou čerpá zvětšující se zásobník); v případě nárůstu velikosti zásobník po hranici datového segmentu dochází k porušení přístupu k segmentu a proces je jádrem ukončen. Rané verze UNIXu umožňovaly programátorovi pohybovat hranicí mezi datovým segmentem a zásobníkem pomocí volání jádra *brk* nebo *sbrk* a tím měnit proporce mezi datovým segmentem a zásobníkem podle aktuální potřeby běžícího procesu. *brk* i *sbrk* není definováno v SVID ani POSIXu. Namísto těchto volání jádra se doporučuje používat volání jádra

```
void *malloc(size_t size);
```

a

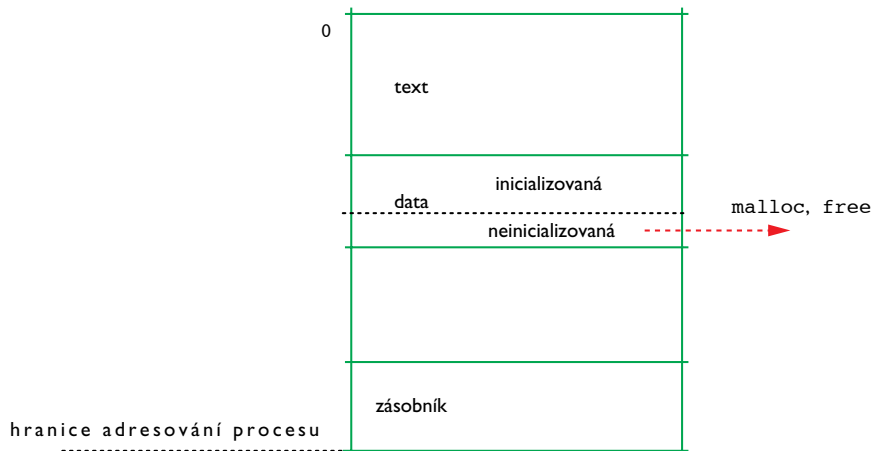
```
void free(void *ptr);
```

která dynamicky přidělují (*malloc*) a uvolňují (*free*) procesu paměťový prostor, jehož velikost je stanovena parametrem *size* v počtu bytů. *malloc* vrací ukazatel na tuto paměť, která je obvykle implementována mimo prostor procesu na obr. 2.9. *free* pak tuto paměť uvolňuje odkazem na ukazatel dříve přidělené paměti.

Jakoukoliv snahu dostat se mimo povolený adresový prostor segmentů procesu nebo jiné korektně přidělené paměťové oblasti procesu jádro při provádění programu kvalifikuje jako porušení odkazu paměti procesem. Odkaz neprovede a procesu posílá signál *SIGSEGV* nebo *SIGBUS*, který implicitně znamená ukončení procesu s výpisem obsahu paměti do souboru *core* pro možnou analýzu příčiny havárie procesu (např. prostředkem *adb*).

## 2.3.6 Čas a sledování dětí

Proces může evidovat čas, který strávil v systému on i jeho děti. Jádro tuto aktivitu podporuje voláním jádra *times*. Doba života procesu je měřena od jeho vzniku po zánik, proces za svého života přitom může být zpracováván procesorem (user time – uživatelský čas), jeho požadavky může zabezpečovat jádro (system time – systémový čas) a také proces může pouze čekat na zpracování, když procesor provádí jiný proces. Součet těchto tří časů je čas života procesu (real time – reálný čas), který ovšem může být proměnlivý podle počtu současně běžících procesů v systému. Proto jádro eviduje každou uvedenou část času života procesu odděleně. Formát volání jádra je



Obr. 2.9 Struktura procesu

```
#include <sys/times.h>
```

```
clock_t times(struct tms *buffer);
```

Proces, který použije toto volání jádra v jeho návratové hodnotě, získá celkový čas svého dosavadního života v počtu tiků hodin<sup>3</sup> stroje (tzv. reálný čas života procesu). Struktura `buffer` je naplněna dílčími hodnotami tohoto času a hodnotami časů ukončených dětských procesů. `tms` je definována v `<sys/times.h>` a obsahuje položky (obsah každé položky je v počtu tiků hodin stroje, jejichž počet ve vteřině je dán hodnotou `CLK_TCK`, která je definována v `<time.h>`)

```
clock_t tms_utime;      /* uživatelský čas volajícího procesu */
clock_t tms_stime;      /* systémový čas volajícího procesu */
clock_t tms_cutime;     /* součet uživatelských časů všech ukončených
                        dětských procesů */
clock_t tms_cstime;     /* součet systémových časů všech ukončených
                        dětských procesů */
```

Čas života dítěte je přitom zahrnut do součtů, pokud rodič použil volání jádra `wait` nebo `waitpid` a registroval tak konec dětského procesu. Časy `tms_cutime` a `tms_cstime` jsou součty časů všech dětských procesů a všech jejich případných dalších dětí atd. Proces tedy může měřit časy svých dětských procesů a čas svůj.

Uživatel v příkazovém řádku má k dispozici příkaz **time**, kterým může sledovat uvedené časy pro vybraný proces. Formát je

**time** command\_line

Měřený proces zadáváme v části **command\_line**. Proces **time** jej provádí standardním voláním jádra **fork** a **exec** a po ukončení tohoto dětského procesu vypíše **time** uživatelský, systémový a reálný čas na standardní chybový výstup. Např.

```
$ time cc prog.c
```

```
real  0m0.69s
```

```
user   0m0.14s
```

```
sys    0m0.35s
```

```
$
```

Pro ilustraci si uvedeme fragment zdrojového textu pro příkaz **time**

```
#include    <stdio.h>
#include    <sys/time.h>

extern int errno;
clock_t pred, po;
struct tms pred_ch, po_ch;
int status;

int main(int argc, char *argv[])
{
    switch(fork())
    {
        case -1:    perror("time: sorry");
                    exit(1);
        case 0:    if(execvp(argv[1], &argv[1])==-1)
                    {
                        perror("time: sorry");
                        exit(1);
                    }
        default:    pred=times(&pred_casy);
                    wait(&status);
                    po=times(&po_casy);
                    fprintf(stderr,
                        "real\t%0dm0.%02d\n",
                        (po - pred)/CLK_TCK/60,
                        (po - pred)%CLK_TCK%60);
                    fprintf(stderr,
                        "user\t%0dm0.%02d\n",
                        (po_ch.tms_cutime - pred_ch.tms_cutime)/CLK_TCK/60,
                        (po_ch.tms_cutime - pred_ch.tms_cutime)%CLK_TCK%60);
```

```

    fprintf(stderr,
        "sys\t%0dm0.%02d\n",
        (po_ch.tms_cstime - pred_ch.tms_cstime)/CLK_TCK/60,
        (po_ch.tms_cstime - pred_ch.tms_cstime)%CLK_TCK%60);
}
}

```

V příkladu používáme standardní funkci `perror`, která tiskne na standardní chybový výstup text z parametru, a chybu volání jádra (viz čl. 1.1). V případě dětského procesu (`case 0`) provádíme `execvp`, které hledá soubor jména v `argv[1]` podle nastavené cesty k adresářům prostředí proměnné `PATH`. Zbytek příkazového řádku jsou argumenty dětského procesu. Rodič (`default`) čeká na dokončení vytvořeného dětského procesu. Pomocí `times` přitom získá čas jeho vytvoření (v návratové hodnotě) a jeho ukončení (v návratové hodnotě druhého `times`). Jejich rozdílem je reálný čas dítěte. Čas uživatelský a systémový je rozdílem hodnot položek naplněných struktur `pred_ch` a `po_ch`.

Příkaz **time** souvisí s porovnáním výkonu různých implementací UNIXu na různém hardwaru. Souvisí také s laděním psaných programů.

Reálný čas stroje je uživateli viditelný příkazem

**\$ date**

kdy příkaz na standardním výstupu vypisuje formátované datum a čas, např. `Sun Sep 15 03:08:41 1996` znamená, že podle hodin stroje je právě neděle 15. září 1996, 8 minut a 41 vteřiny po 3. hodině ráno.

Operace získání času je procesu dostupná pomocí volání jádra

```
#include <time.h>
```

```
time_t time(time_t *tlock);
```

`time_t` je typ obvykle definovaný jako `long`. Volání jádra vrací počet vteřin, který uplynul podle hodin stroje od 1. ledna 1970 greenwickského času. Pokud parametr `tlock` není `NULL`, je návratová hodnota také uložena do obsahu proměnné `tlock`.

Příkaz **date** může použít také privilegovaný uživatel pro nastavení času hodin stroje. **date** za tím účelem používá volání jádra

```
int stime(const time_t *tp);
```

Argument `tp` je ukazatel na hodnotu, která je opět počtem vteřin od 1. ledna 1970 greenwickského času. Touto hodnotou je přepsán čas stroje.

Pro větší komfort práce s uvedenými funkcemi jádra nabízí standardní knihovna funkce:

```
char *ctime(const time_t *clock);
```

Převádí hodnotu počtu vteřin v `clock` na textový řetězec typu `Sun Sep 15 03:08:41 1996`.

```
struct tm *localtime(const time_t *clock);
```

Naplní položky struktury `tm` hodnotami měsíce, dne, hodiny, minuty atd., a to podle počtu vteřin v `clock`. Respektuje přitom nastavenou časovou zónu vzhledem ke greenwickskému času.

```
struct tm *gmtime(const time_t *clock);
```

Má tutěž funkci jako funkce `localtime`, ale přitom ignoruje místní časovou zónu. Hodnoty v návratové hodnotě funkce `gmtime` jsou tedy vztaženy ke greenwichskému času.

```
char *asctime(const struct tm *tm);
```

Převádí hodnoty struktury `tm` do textového řetězce typu `Sun Sep 15 03:08:41 1996`.

Používaná struktura `tm` je definována v souboru `<time.h>` a obsahuje položky data a času

```
int tm_sec;      /* vteřin od 0 do 59 */
int tm_min;      /* minut od 0 do 59 */
int tm_hour;     /* hodin od 0 do 23 */
int tm_mday;     /* den v měsíci v rozsahu 1 - 31 */
int tm_mon;      /* měsíc v rozsahu 0 - 11 */
int tm_year;     /* rok, aktuální hodnota zmenšená o 1900 */
int tm_wday;     /* den v týdnu, 0 - 6 (0 je neděle) */
int tm_yday;     /* den v roce, 0 - 365 */
int tm_isdst;    /* má nenulovou hodnotu, pokud je respektován
                  letní čas */
```

Časová zóna je nastavena správcem systému na hodnotu počtu hodin vzhledem k časové zóně vztažené ke greenwichskému poledníku. Správce vkládá informace o časové zóně do systémového souboru v adresáři `/etc` (např. `/etc/TIMEZONE`), který definuje obsah proměnné TZ standardního prostředí provádění procesu (viz odst. 2.5.2). Funkce `localtime`, `asctime` a další funkce, které pracují s informacemi o časové zóně, využívají obsah proměnné TZ z prostředí `envp` parametrů funkce `main`. Textové označení časové zóny je tříznakové. Greenwichská časová zóna má označení UTS (dříve GMT). Programátor může rozdíl v počtu vteřin nastavené časové zóny od UTS získat přístupem k externí proměnné (je rovněž naplněna podle TZ), kterou definuje

```
extern time_t timezone;
```

Letní čas je viditelný v externí proměnné

```
extern int daylight;
```

a

```
extern char *tzname[2];
```

obsahuje text časové zóny jak pro zimní, tak letní čas. Obsah `tzname` je naplněn obsahem TZ použitím funkce

```
extern void tzset(void);
```

která je také volána vždy při použití funkce `asctime`.

UNIX System V rozšířil (podle vzoru systémů BSD) získání a nastavení času hodin stroje o volání jádra `gettimeofday` a `settimeofday`, protože prozatím uvedená volání jádra pracují na úrovni celých sekund. Formát je

```
#include <sys/time.h>
```

```
int gettimeofday(struct timeval *tp);
```

pro získání času a

```
int settimeofday(struct timeval *tp);
```

pro jeho nastavení (je dostupné pro privilegovaného uživatele). `tp` přitom ukazuje na strukturu `timeval`, v níž je uveden čas rozdělený na vteřiny od 00.00 hodin 1. ledna 1970 a na mikrosekundy poslední vteřiny:

```
long tv_sec;      /* sekundy od 00.00 1.1. 1970 UTC */
long tv_usec;     /* zbytek mikrosekund */
```

Volání jádra `gettimeofday` a `settimeofday` jsou implementována ve všech současných systémech. SVID je obsahuje, ale upozorňuje na nesoulad s POSIXem. Pro použití v budoucnu slibuje definici návaznou na POSIX, tj. definice funkcí `clock_gettime`, `clock_settime`, které mají podobný formát, kromě identifikace hodin, na které se práce s časem vztahuje, a volání jádra `clock_getres`, které práci s hodinami stroje doplňuje a je implementačně závislé. V SVID je také definována používaná funkce (prvně implementovaná v systémech BSD) `adjtime` pro korekci času na úrovni mikrosekund. Přestože jej současná verze POSIXu nedefinuje, explicitně slibuje její zavedení v budoucnu.

Rodič může sledovat činnost svých dětí pomocí volání jádra `ptrace` definovaného i v SVID. Je základem pro trasování programů v tzv. dynamickém ladění, kdy se využívá možnosti zjišťovat v pozastaveném procesu informace o obsahu jeho dat, případně tento obsah měnit. Takové zkoumání dětského procesu v *bodech přerušení* (breakpoints) využívají programy **adb** (ladění na úrovni assembleru) a **sdb** (úroveň zdrojového textu v C). Podle vyjádření SVID jsou ale zastaralé a budou nahrazeny jinými, dokument ale dosud neuvádí kterými, přestože ladicí prostředky **adb** i **sdb** má každý průmyslový UNIX. POSIX `ptrace` ani jiné možnosti dynamického ladění programů prozatím nedefinuje.

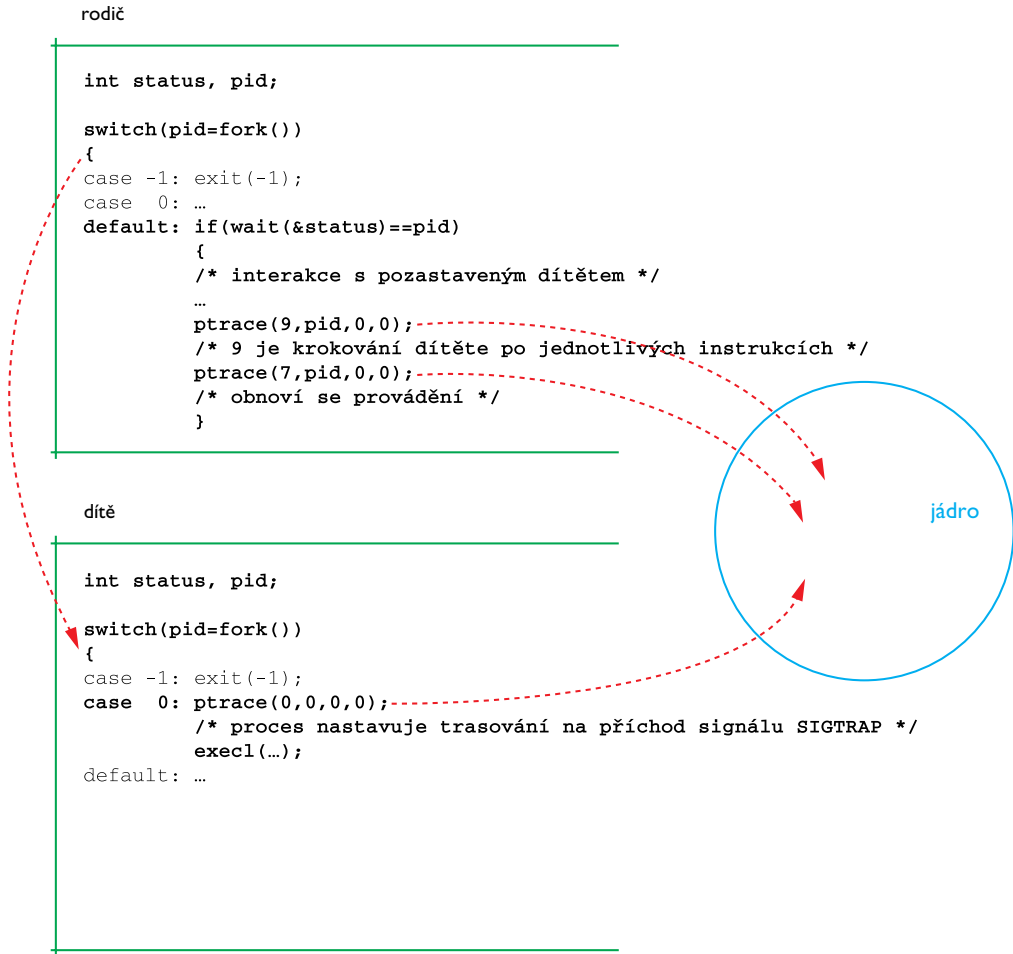
Technika práce s voláním jádra `ptrace` opět vychází ze spolupráce rodiče a dítěte. Rodič má výhradní právo dětskému procesu upravit prostředí provádění před výměnou textového segmentu dítěte pomocí `exec`. `ptrace` využívá signál `SIGTRAP`, na základě kterého je dětský proces pozastaven, a rodič voláním jádra `ptrace` zjišťuje potřebné informace. Dítě ale musí deklarovat svoji reakci na `SIGTRAP` pozastavením tak, že použije `ptrace` před vlastním prováděním procesu. Formát je

```
#include <sys/types.h>
```

```
int ptrace(int request, pid_t pid, int addr, int data);
```

a dítě je po `fork` ještě za řízení rodiče používá s hodnotou `request` rovnu 0. Při tomto použití jsou obsahy ostatních parametrů ignorovány (přestože jsou vyžadovány) a dítě provádí `exec` s maskováním signálu `SIGTRAP` na pozastavení procesu. Od chvíle takového nastavení je dítě pozastaveno vždy příchodem `SIGTRAP` nebo vždy před vykonáním každého `exec`. Pozastavení dítěte sleduje rodič voláním jádra `wait`. Pozastavené dítě může prohlížet, měnit a dávat pokyn k jeho pokračování pomocí volání jádra `ptrace`. U rodiče jsou pak používány i ostatní parametry `ptrace`. `request` je celočíselná hodnota, která znamená typ operace s dětským procesem. Např. hodnota 1,2 nebo 3 je požadavek přenosu slova z místa daného `addr` dítěte do rodiče (v návratové hodnotě, `data` je bez významu), 4,5 nebo 6 naopak zápis slova `data` do místa daného `addr` dítěte, 7 je pokyn k dalšímu provádění atd. Schematicky situaci včetně fragmentů kódu procesů ukazuje obr. 2.10.

Volání jádra `ptrace` je hardwarově závislé. Hodnota `request` 3 nebo 6 umožňuje čtení a zápis dat do systémové oblasti procesu, proto má volání jádra privilegovaný charakter (obyčejnému uživateli jej



Obr. 2.10 Trasování dítěte

zpřístupní pouze s-bit aplikačního programu) a nedoporučuje se používat obecně pro komunikaci procesů.

SVID kromě `ptrace` také definuje volání jádra `profil`, které má rovněž ladicí charakter. Na rozdíl od `ptrace` jde o statické ladění. `profil` umožňuje shromažďovat informace o procesu a v určitém tvaru je ukládat např. do souboru, odkud mohou být zobrazovány uživateli. `profil` dokáže evidovat čas spotřebovaný určitou částí procesu podle textu programu a vztahuje se pouze k volajícímu procesu, nejde tedy o sledování dětí.



```
void profil(unsigned short *buff, unsigned int bufsiz,
           unsigned int offset, unsigned int scale);
```

je formát, kde `bufsiz` je velikost oblasti čítačů `buff`, které naplňuje jádro po každém tiku hodin stroje v měřítku daném `scale`. Měřítkem rozumíme jemnost sledování, např. po jedné nebo více instrukcích stroje. Pokud je `scale` 0 nebo 1, volání znamená vypnutí měření chodu procesu.

Volání jádra `profil` je používáno makrem `MARK`, pomocí kterého si můžeme označovat měřená místa v programu. Definována je také funkce `monitor`, která vytváří pohodlnější prostředí pro měření spotřebovaného času procesu částmi programu. Konečně na úrovni uživatelské lze `profil` používat při ladění programů, protože překladač jazyka C má volbu `-p`, která ve výsledku překladu, programu `a.out`, znamená používání funkce `monitor` pro sledování každé funkce programu při běhu procesu. Výsledek proces ukládá do souboru s implicitním jménem `mon.out`. Formát souboru `mon.out` je čitelný programem `prof`, který umí data zobrazit v textově čitelné podobě.

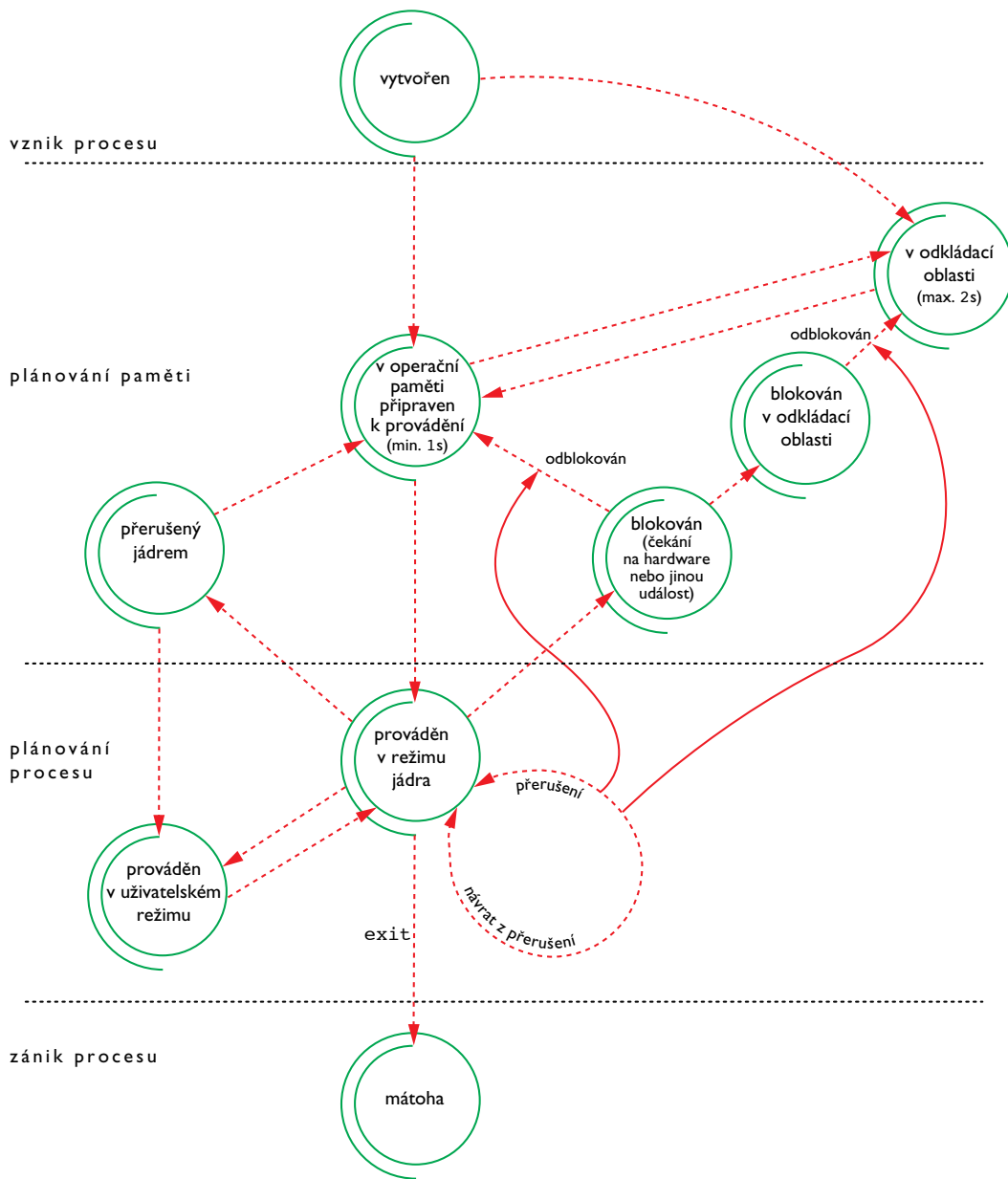
Programátor má pro vývoj programu k dispozici řadu prostředků vývoje programu. Jejich podrobný popis nalezne čtenář v odpovídající části provozní dokumentace konkrétního UNIXu. Většina je standardizována podle POSIXu a SVID.

## 2.4 Proces z pohledu jádra

Jádro je supervizorem všech akcí, které probíhají ve výpočetním systému. Iniciátory akcí jsou procesy uživatelské i systémové. Jádro zajišťuje požadavky procesů, sleduje jejich činnost a distribuuje mezi nimi výpočetní zdroje. Každý proces vzniká za účelem uskutečnění určité akce. Ideálně zpracovávaný proces má zajištěny všechny požadované výpočetní zdroje tak, aby zpracování akce proběhlo co nejrychleji. O to ale usiluje každý proces v operačním systému. Jádro proto musí přidělování výpočetních zdrojů plánovat a přidělovat podle určitých pravidel. Základní výpočetní zdroje, které proces potřebuje ke svému zpracování, jsou operační paměť a procesor. Proces musí být zaveden do operační paměti a poté může být prováděn. V operační paměti může být zavedeno současně více procesů, ale procesorem může být prováděn pouze jeden proces uložený v operační paměti<sup>4</sup>. Operační paměť a procesor jednotlivým procesům přiděluje jádro pomocí svých modulů plánování paměti a plánování procesoru. Teprve za svého běhu proces vyžaduje další výpočetní zdroje, jako např. přístup k diskům, terminálu, síti atd., což vyjadřuje pomocí volání jádra.

Jádro zajistí provádění několika procesů současně. Proces nečeká na ukončení všech procesů, které byly spuštěny před ním. Každý nově vzniklý proces je procesem konkurujícím ostatním registrovaným procesům. V průběhu provádění procesorem může být jako každý jiný jádrem přerušen, aby na určité časové kvantum umožnil provádění dalších konkurujících procesů. Uvedme si nyní schéma práce jádra pro zajištění zpracování každého procesu od jeho vzniku až do ukončení. Každý proces za svého života je jádrem evidován v různých stavech. Na obr. 2.11 je každý stav označen spirálou, přechod mezi stavy je vždy vyjádřen červenou směřovanou čarou. Šedá čárkovaná čára odděluje kompetence modulů plánování operační paměti a plánování procesoru. Obrázek ukazuje klasickou strukturu, která zde není obohacena o procesy vyžadující odezvu v reálném čase.

Proces je vytvořen na žádost jiného procesu voláním jádra `fork`. Pokud má jádro dostatek místa v tabulkách pro evidenci procesů, neodmítne vytvoření procesu a eviduje jej ve stavu *vytvořen*. Proces tak vstupuje do oblasti kompetence plánování paměti.



Obr. 2.11 Stavy procesu

### 2.4.1 Plánování operační paměti

Zjednodušený pohled na obsazení operační paměti v kontextu plánování paměti je uveden na obr. 2.12.

Jádro usiluje o přidělení operační paměti procesu ihned po jeho vzniku. Podporuje tak interaktivní příkazy, které iniciují vznik nových procesů. Pokud je v operační paměti dostatek místa pro zavedení procesu, je proces zaveden a je zviditelněn pro modul plánování procesoru. Pokud je operační paměť plně obsazena, jádro spouští algoritmy, které zkoumají všechny procesy umístěné v operační paměti, aby vybral vhodného kandidáta pro vyjmutí a nahrazení nově vzniklým procesem. Pokud jádro neuspěje (ve výjimečných případech), je proces umístěn na disk do odkládací oblasti (swap area), kde společně s ostatními zde odloženými procesy soupeří o operační paměť. Z operační paměti může být každý proces odložen do odkládací oblasti aniž dojde k jeho provádění, protože některý z konkurujících procesů má přednost. UNIX podporuje v práci interaktivní procesy, tj. nově vzniklý proces má snahu jádro vždy umístit v operační paměti. Proces, který zde již určitou dobu pobývá, se jádru (modulu plánování paměti) jeví jako uspokojen, a proto jádro procesu paměť odebere. K zajištění průchodu procesu systémem a současně k tomu, aby proces neuváznuv v odkládací oblasti, jádro dodržuje pravidlo umístění procesu v operační paměti po dobu nejméně 1 vteřiny a nejvíce 2 vteřiny v odkládací oblasti. Časové konstanty se mohou mírně lišit v jednotlivých implementacích, ale uvedený poměr 1:2 je vždy zachován. 1 vteřina je také časový údaj, ve kterém je proces nejméně jednou prováděn procesorem. Obsluha uživatele, který je majitelem procesu, je tedy celkově limitována 1 vteřinou od zadání po zahájení provádění.

*Odkládací oblast* (swap area) je spravována procesem č. 0. Odkládací oblast je umístěna na disku a její velikost je stanovena při instalaci operačního systému. Takto vzniká *primární odkládací oblast*. Je pro ni vymezena zvláštní oblast disku mimo oblast uživatelských dat. Mnohdy její velikost nedostačuje rostoucímu využívání výpočetního systému. Aby se nemusel systém nově instalovat, může správce systému definovat pomocí volání jádra `swapctl` (je uvedeno v SVID, ale ne v POSIXu) další část některého z disků (opět mimo oblast uložení dat) jako rozšíření odkládací oblasti.

Příkaz **swap** (není uveden v SVID) je převedení možnosti volání jádra `swapctl` na úroveň příkazů správce systému. K vytvoření další odkládací oblasti používáme **swap** s volbou **-a**. Vzniká tak sekundární odkládací oblast. Sekundárních oblastí může mít operační systém připojen několik. Pro efektivní práci se doporučuje pro ně využívat vždy jiný disk, protože proces č. 0 využívá primární i *sekundární odkládací oblast* současně (více viz kap. 10). Konečně může správce systému v případě nedostatku odkládací oblasti i nového prostoru na discích použít tzv. *virtuální* (nebo dynamickou) *odkládací oblast* (rovněž příkazem **swap**, někdy **swapon**). Místo na disku pro odkládání procesů je zde čerpáno z volné kapacity datové části disku. Odebráno je přitom vždy tolik, kolik je nezbytně potřeba, a po zmenšení nároků je část opět uvolněna pro alokaci uživatelskými daty. Výhodou virtuální odkládací oblasti je její velikost, která je limitována pouze volnou kapacitou použitého svazku. Nevýhodou je rychlost přidělení, která je zatížena režii systému souborů. Přehled využívané primární, sekundárních a virtuálních odkládacích oblastí můžeme získat příkazem

**\$ swap -l**

Volání jádra, které příkaz **swap** používá, má v SVID formát

```
#include <sys/stat.h>
```

```
#include <sys/swap.h>
```

```
int swapctl(int cmd, void *arg);
```

kde `cmd` může být buďto `SC_ADD` pro připojení odkládací oblasti, `SC_REMOVE` pro odpojení, `SC_LIST` pro získání informací o odkládací oblasti a `SC_GETNSWP`, kdy zjistíme celkový počet odkládacích oblastí. `arg` pro `SC_ADD` nebo `SC_REMOVE` je struktura `swapres` (viz `<sys/swap.h>`), která obsahuje položky určující jméno souboru pro část disku, začátek a velikost odkládací oblasti. Při použití `SC_LIST` zadáváme v `arg` ukazatel na strukturu `swaptab` (viz tamtéž), kterou jádro naplní podrobnými informacemi o umístění a stavu odkládací oblasti.

Proces privilegovaného uživatele (tj. proces efektivního uživatele `root`) může požádat jádro o zamknutí v operační paměti a získat tak výhodu ve vytíženém systému oproti ostatním procesům. Používá k tomu volání jádra `plock`, které má formát

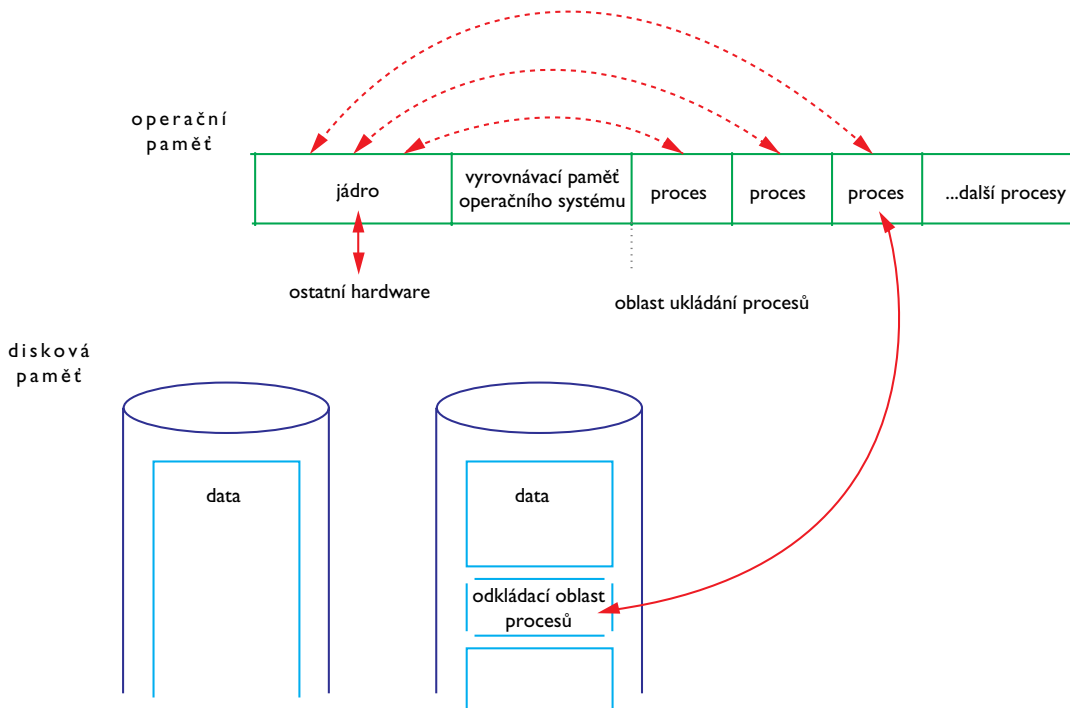
```
#include <sys/plock.h>
```

```
int plock(int op);
```

kde parametr `op` může nabývat hodnoty (viz obr. 2.9) `TXLOCK` a jde o zamknutí textového segmentu, `DATLOCK` zamknutí datového segmentu, `PROCLLOCK` zamknutí textového i datového segmentu a `UNLOCK` ruší zámky.

V případě, že operační systém používá při správě paměti také metodu stránkování na žádost (viz dál), je `plock` přesunuto na úroveň funkce, protože každý segment procesu je pak chápán jako množina stránek (pages). Přitom je možné zamykat jednotlivou stránku.

V současných implementacích UNIXu je běžně využívána při správě paměti také metoda stránkování na žádost. Jde o známou metodu teorie operačních systémů, kdy paměťový prostor, který každý registrovaný proces vyžaduje v operační paměti, je rozdělen na úseky, kterým se říká stránky (pages) tak, že proces může být prováděn, aniž jsou v paměti uloženy všechny stránky procesu. Potřebná operační paměť pro běh procesu se tedy snižuje. Proces za jeho provádění ovšem dříve nebo později narazí na chybějící stránku. Pak jde o tzv. výpadek stránky, kdy je proces přerušen a je uplatňována žádost o chybějící stránku (ve vnitřních strukturách jádra). Chybějící stránka je vyhledána v odkládací oblasti a je snaha ji do operační paměti uložit. Přitom se v případě nedostatku operační paměti naopak odloží stránka, která není v operační paměti v daném okamžiku nutná. Tento jednoduchý princip je při hlubším zkoumání komplikován řadou faktorů. Jeden z nich je např. stanovení stránek, které lze odsunout u všech registrovaných procesů uložených v operační paměti tak, aby následující přidělení procesoru nevyvolalo výpadek stránky. K tomu se používá stanovení množiny platných stránek procesu, tj. stránek nezbytně nutných pro následný běh procesu. Jádro při uplatňování stránkování na žádost pro uvolnění nikoliv nezbytných stránek procesů z operační paměti využívá systémové procesy, které za tímto účelem aktivuje a které spouští při startu operačního systému proces `init` jako jedny z prvních procesů (jejich start bývá vyjmut z řízení tabulkou `/etc/inittab`) a jejich zánik nastává teprve se zastavením operačního systému. Podobně jako procesy č. 0 a 1 přežívají všechny stavy operačního systému. Procesy, které používá jádro pro podporu stránkování na žádost, se nazývají *zloději stránek* (page stealer) a jejich jméno v systému bývá různé (např. `vhand`); správce systému nemá obvykle výraznou možnost tento mechanismus ovlivnit. V současných implementacích je běžné současné použití jak odkládání celých procesů, tak stránkování na žádost, což teoretikové označují jako systémy hybridní.



Obr. 2.12 Obsazení operační paměti

Ke zlepšení výkonu práce přidělování paměti přispívá reentrantnost (reentrant code) textových segmentů procesů. Každý program, který je přeložen a sestaven pro řízení procesu, je v UNIXu uložen v proveditelném souboru tak, že v hlavičce programu (hodnotou magického čísla, viz 2.4.4) je poznamenána možnost používání textového segmentu několika procesy současně. Znamená to, že při vzniku nového procesu, který je řízen již používaným programem, nemá proces unikátní textový segment, ale je sdílen tolika procesy, kolik jich je tímto programem řízeno. Např. procesy **sh** přihlášených uživatelů mají všechny sdílený textový segment a šetří tak obsazení operační paměti. Tato implicitní vlastnost běhu procesů může být ale vypnuta tak, že uživatel, který sestavuje program, použije odpovídající volbu sestavujícího programu **ld** (označení volby se v implementacích liší). Tehdy má každý proces, byť řízen tímž programem, jedinečný textový segment.

Neodmyslitelnou součástí modulů plánování paměti je mapování virtuálních adres běžícího procesu na adresy fyzické. Každý program je přeložen a sestaven tak, že reference v rámci jeho adresového prostoru začínají tak, jak je uvedeno na obr. 2.9, od adresy 0, jako by měl být uložen a prováděn v operační paměti samotný. Jsou to tzv. virtuální adresy. Vzhledem k tomu, že současně je v operační paměti uloženo jádro i několik procesů současně, reference v rámci adresového prostoru procesu jsou převáděny na skutečné uložení procesu v paměti, tj. na fyzické adresy. Převod z virtuálních adres na

fyzické se nazývá mapování paměti procesu. Informace o fyzických adresách procesu jsou přitom evidovány v tabulkách jádra pro každý registrovaný proces. Hodnoty fyzických adres se pochopitelně mění se změnou umístění procesu v operační paměti, která nastává nejméně při odložení procesu na disk a po jeho zpětném zavedení. Další komplikace přináší metoda stránkování na žádost, kdy je proces rozdělen na stránky a tyto jsou umísťovány v paměti nezávisle na sobě, tj. proces v paměti nemá souvislé obsazení. Opět musí existovat mapování adres pro jednotlivé stránky procesu. Tyto informace opět eviduje jádro pro každý proces. Souhrn všech informací, které proces provázejí od vzniku do jeho zániku a které jsou evidovány jádrem, je uveden v odst. 2.4.4.

Používání virtuálních adres namísto přímo fyzických vytváří flexibilní model správy operační paměti a jednoduchý způsob generace programů vývojovými prostředky programátora (kompilátory a sestavovací programem). Ve vlastním provozu znamená zvýšení režie, ale umožňuje snadné používání odkládání procesů a stránkování na žádost.

Správce systému (jako uživatel `root`) může pracovat s procesy jako s množinou stránek, které jsou mapovány z virtuální na fyzickou paměť pomocí volání jádra `mmap`. SVID jej definuje k označení stránek procesu jako privátní nebo sdílené, textové nebo datové, možnost jejich přepisu, provádění nebo pouze čtení nebo jejich zamykání a odemykání v operační paměti. Tím je přenesen mechanismus reentrantnosti textového segmentu na úroveň stránek, stejně tak pro zamykání v operační paměti je `mlock` z vrstvy volání jádra přesunuto na vrstvu funkce. Potřebné definice pro práci s `mmap` nalezne programátor v `<sys/mman>`. Pro usnadnění práce jsou mu k dispozici makra a funkce `mlock`, `munlock`, `mlockall`, `munlockall`, `mmap`, `munmap` a `msync`. POSIX uvedené volání jádra a funkce cituje, ale zatím přesně nedefinuje.

## 2.4.2 Plánování procesoru

Je-li proces ve stavu v *operační paměti*, soupeří o čas procesoru. Soupeření znamená, že jádro ze všech registrovaných procesů v tomto stavu vybírá vhodného kandidáta, kterému na určité časové kvantum přidělí procesor. Vybraný proces je pak podle obr. 2.11 převeden do stavu *prováděn v režimu jádra*, který se při provádění procesorem střídá se stavem *prováděn v uživatelském režimu*. V ideálním případě je proces prováděn do volání jádra `exit`, které ukončí život procesu. Přestože je proces pro uživatele ukončen, jádro jej ještě stále udržuje ve svých vnitřních tabulkách ve stavu *mátoha* (zombie) do chvíle, kdy jeho rodič odebere z jádra (voláním jádra `wait` a `times`) návratový status a časovou statistiku (viz odst. 2.3.6) nebo kdy rodič zanikne. Stav *prováděn v režimu jádra* znamená, že jádro provádí některé z volání jádra, které proces používá. *Prováděn v uživatelském režimu* je veškerý zbylý výpočet, na který proces nepotřebuje jádro a jeho podporu. Do stavu *blokován* se může proces dostat při vykonávání volání jádra, kdy při komunikaci s periferií nemůže jádro čekat na dokončení operace nad ní (čtení z disku, dokončení tisku na tiskárnu, komunikace s terminálem), proto proces převádí do tohoto stavu a vybírá ke zpracování jiný proces. Do stavu *blokován* se proces může dostat také např. z důvodu čekání na příchod signálu nebo dokončení dětského procesu, tedy čekání na událost, která souvisí se zpracováním jiného procesu nebo jádra. Jádro v době, kdy je náš proces ve stavu *blokován*, může mezitím vybrat ke zpracování procesy jiné a provést řadu jiných systémových akcí. Pokud přijde přerušení z periferie nebo nastane očekávaná událost, je proces odblokován a stává se procesem soupeřícím o procesor, jestliže mezitím nedošlo k jeho odložení z operační paměti a je pod řízením modulů správy

paměti. Jádro přerušení od hardwaru řadí do fronty a zpracovává podle priorit, jak bylo uvedeno v kap. 1. Pomocí systému přerušení hardwaru a jeho obsluhy jádrem dochází asynchronně k převodu zablokováných procesů do jiných stavů. Jádro ale může procesu odebrat procesor, protože je nutné provádět i jiné procesy. Přejed do stavu *přerušný jádrem* může nastat ze dvou důvodů. Jednak je to vždy při ukončení volání jádra (návrh z jádra do stavu *prováděn v uživatelském režimu*) a jednak pokud proces vyčerpal nejvyšší hodnotu časového kvanta, po kterou může být proces prováděn, a tou je 1 vteřina. Jádro pak v další práci provede výběr pro zpracování z procesů, které jsou ve stavu *připraven k běhu v operační paměti a přerušný jádrem*.

Vynucené zablokování procesu může proces dosáhnout pomocí volání jádra `pause`. Proces je propříště odblokován na základě přijetí signálu (voláním jádra `alarm` si např. může proces naplánovat příchod signálu `SIGALRM` za určitý počet vteřin). Pro větší flexibilitu jsou definována další dvě volání jádra pro práci s časovým intervalem života procesu.

```
#include <sys/time.h>
```

```
int getitimer(int which, struct itimerval *value);
```

Umožňuje získat informace o časových intervalech, které jsou dány parametrem `which`. Časový interval je popsán ve struktuře `itimerval`, která obsahuje dvě položky

```
struct timeval it_interval; /* časový interval */
struct timeval it_value;    /* jeho aktuální hodnota */
```

Popis struktury `timeval` najde programátor v `<sys/time.h>` nebo v definici volání jádra `gettimeofday` (viz odst. 2.3.6), protože se jedná o zpracování časového intervalu na úrovni mikrosekund.

Parametr `which` může být v současné době použit (podle SVID) ve třech variantách:

`ITIMER_REAL` pro sledování reálného intervalu života procesu; procesu je zaslán signál `SIGALRM` po jeho uplynutí,

`ITIMER_VIRTUAL` pro sledování intervalu života procesu ve stavech *prováděn v režimu jádra*, procesu je po jeho uplynutí zaslán signál `SIGVTALRM`,

`ITIMER_PROF` pro sledování intervalu života procesu jak ve stavu *prováděn v režimu jádra*, tak *prováděn v uživatelském režimu*, používá se pro sledování práce procesu, po uplynutí časového intervalu je procesu zaslán signál `SIGPROF`.

Proces může pracovat s uvedenými třemi časovými intervaly maskováním příchodu odpovídajících signálů, ale také pomocí volání jádra

```
int setitimer(int which, struct itimerval *value,
              struct itimerval *ovalue);
```

může jednotlivé intervaly nastavovat. Pokud není `ovalue` `NULL`, předchozí nastavení se po úspěšném provedení jádra uloží do struktury, na kterou `ovalue` ukazuje.

POSIX definuje obecná volání jádra `timer_create`, `timer_delete` pro vytvoření a zrušení časového intervalu, `timer_gettime`, `timer_settime` a `timer_getoverrun` pro práci s časovým intervalem. Jde o zobecněný přístup k časovým intervalům. Nejsou pouze tři pevně dané, ale je možné si vytvářet potřebný počet časových intervalů a zadávat podmínky jejich odpočítávání. Pokud

dojde v SYSTEM V k jejich implementaci, `getitimer` a `setitimer` budou implementovány jako funkce nebo makra.

Výběr procesu pro zpracování se uskutečňuje podle procesu s nejvyšší dynamickou prioritou. *Dynamická priorita* každého procesu je v okamžiku výběru pro každý proces vypočtena z

*výchozí priority* (uživatelské, user priority)

času procesu, který už proces odčerpál.

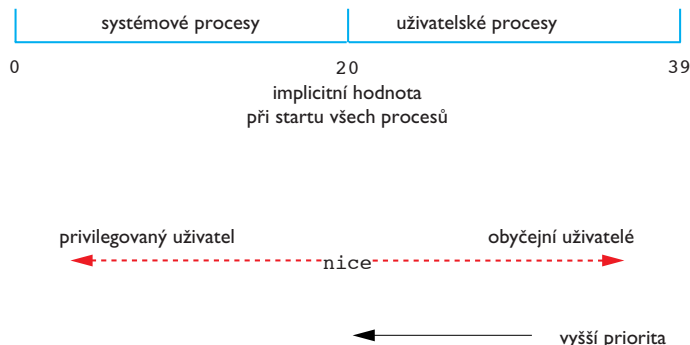
Je přitom zohledněna situace, na základě které se proces dostal do stavu *připraven k běhu v operační paměti* nebo *přerušený jádrem*. Jádro totiž udržuje několik front procesů usilujících o procesor. Základní rozdělení je na frontu procesů systémových a frontu procesů uživatelských. Ve frontě systémových procesů jsou evidovány procesy, které jsou životně důležité pro systém, jako např. proces č. 0 nebo zloděj stránek, ale také procesy, které jsou zablokovány a čekají na událost zvláštního významu, jako je např. dokončení diskové operace, dokončení operace nad sítí atp. To je fronta systémových procesů, které jsou v dalším zpracování nepřerušitelné. Dále ještě existuje druhá fronta systémových procesů přerušitelných, tj. čekání na dokončení terminálové operace, dokončení práce dětského procesu atd. Systémové procesy mají vždy vyšší výchozí prioritu než procesy uživatelské. Teprve nenajde-li jádro žádný systémový proces, který by usiloval o procesor, vybírá z fronty procesů uživatelských.

Dynamická priorita procesu má hodnotu kladného celého čísla. Nižší hodnota znamená vyšší prioritu. Nejvyšší priorita je tedy 0. Dynamická priorita je vypočítávána ze vztahu

$\text{priorita} = (\text{poslední využití procesoru}) / 2 + \text{uživatelská priorita}$

U každého procesu je přitom při výběru vhodného kandidáta upravena hodnota posledního využití procesoru (v počtu tiků hodin počítače) tak, že je dělena dvěma a teprve uvažována.

Hodnota uživatelské priority je dána rovněž celou kladnou hodnotou vyšší než 0. Rozsah jejích hodnot je v původním UNIXu dán v intervalu od 0 do 39 (viz obr. 2.13), obecně dnes podle SVID od 0 do  $\text{NZERO} * 2 - 1$ , kde  $\text{NZERO}$  je implicitní hodnota, se kterou procesy vstupují do systému, pokud není



Obr. 2.13 Uživatelská priorita



řečeno jinak. Hodnota NZERO je hraniční hodnota mezi systémovými a uživatelskými procesy. Všechny procesy s touto a vyšší hodnotou jsou neprivilegované procesy. Privilegovaný uživatel může výchozí hodnotu snížit a dostat se tak do oblasti systémových procesů.

Hodnotou výchozí uživatelské priority může proces změnit voláním jádra `nice`. Přitom pouze privilegovaný proces může mít výchozí hodnotu uživatelské priority zvýhodněnu. Volání jádra má formát

```
int nice(int incr);
```

kde `incr` je hodnota, o kterou je snížena výchozí priorita, tj. hodnota, o kterou se zvýší NZERO. Při výpočtu dynamické priority je pak proces vždy znevýhodněn. Pokud v `incr` nebyla použita záporná hodnota, která znamená odpočet od NZERO a tedy zvýšení priority. Hodnota NZERO je definována v `<limits.h>` nebo `<sys/param.h>`. Pokud by programátor použil v `nice` hodnotu, která by znamenala překročení horní nebo dolní hraniční hodnoty, priorita je nastavena na odpovídající limitní hodnotu.

V souvislosti s voláním jádra `nice` může uživatel používat příkaz **nice** v interaktivní komunikaci s procesem shell. Ve formátu

```
$ nice [-increment] command
```

je `increment` hodnota, která odpovídá `incr` z uvedeného volání jádra. Není-li zadána, je použita hodnota NZERO/2 (zaokrouhlena nahoru, běžně 10). `command` je příkazový řádek, jemuž odpovídající proces poběží se změněnou hodnotou výchozí priority. Příkaz **nice** je implementován pomocí volání jádra `nice` a volání jádra `exec`.

Volání jádra `nice` je implementováno s uvedeným významem ve všech známých verzích UNIXu tak, jak bylo zavedeno v jeho raných verzích. Za uvedeného mechanismu ale není možné měnit uživatelskou prioritu běžícím procesům. POSIX `nice` proto nedoporučuje používat (a sám ho nedefinuje) a zavádí nová volání jádra `sched_setparam`, `sched_getparam`, pro nastavení a získání priority běžícího procesu a `sched_getscheduler` a `sched_setscheduler` pro nastavení, případně získání strategie plánování procesoru. Přestože tato volání jádra vyčerpávajícím způsobem definují strategii přidělování procesoru, v praxi jsou v UNIXu téměř nepoužívaná (respektive čekají teprve na svou implementaci). Implementace UNIXu (včetně System V) používají volání jádra `setpriority` a `getpriority` převzatá ze systémů BSD, která umožňují změnu výchozí priority jiných procesů. POSIX také definuje všeobecně používaný příkaz **renice**, pomocí kterého se prodlužuje nastavování uživatelské priority procesů na uživatelskou úroveň a který je k nalezení v každém současném UNIXu. SVID jej ovšem neuvádí. SVID definuje obecné schéma práce nejenom procesů sdílení času `prctl`, ale i jiných typů procesů, které je diskutováno v následujícím odst. 2.4.3. Příkaz **renice** má formát

```
$ renice [-n increment] [-g | -p | -u] ID ...
```

Parametr `increment` má stejný význam, jako byl uveden u příkazu **nice**. ID je identifikace procesu (nebo procesů), ke kterým se změna vztahuje, **-p** pro PID procesu, **-g** pro PID skupiny procesů (PID vedoucího skupiny procesů) a **-u** pro všechny procesy určitého uživatele (může být použito jak jméno, tak číselná identifikace uživatele podle `/etc/passwd`).

V současné praxi v UNIXu tak, jak bylo uvedeno, kromě možností volání jádra `nice`, `getpriority` a `setpriority`, kterými upravuje proces svoji vlastní uživatelskou prioritu nebo prioritu jiných

procesů, nemá správce systému mechanismus pro ovlivnění práce modulů správy procesoru. Přestože celkový dosud popsáný mechanismus správy paměti a procesů je spravedlivý a neumožňuje mimořádné zvýhodnění vybraného procesu (např. procesu reálného času), má správce přesto k dispozici poměrně silné prostředky. Je to zamknutí v operační paměti (volání jádra `plock`), maximální zvýšení výchozí priority (volání jádra `nice`) a použití s-bitu. Takto lze program, který bude provádět i obyčejný uživatel, zvýhodnit oproti všem ostatním procesům v systému. Pokud je ale operační paměť malá a provoz v systému a síti frekventovaný, je to opatření velmi neprozíravé.

V operačních systémech podporujících práci procesů v reálném čase jsou dnes k dispozici prostředky, které umožňují manipulovat s prioritou běžících procesů, a to nejen pracujících v reálném čase. Takový UNIX je rozšířen o volání jádra `pricntl` (SVID) a systémové příkazy **`pricntl`** (SVID) a **`dispadm`** (pouze SVR4), které umožňují ovlivňovat práci modulů jádra plánování procesů. POSIX je nedefinuje, přestože se o zpracování dat v reálném čase zmiňuje. Závazné definice ponechává na příští vydání této normy a strategii přidělování procesoru pro procesy sdílení času řeší definicí uvedených volání jádra začínajících na `sched_`. Procesy reálného času v SYSTEM V jsou předmětem následujícího odstavce 2.4.3.

## 2.4.3 Reálný čas

Procesem reálného času rozumíme proces, který zajistí odpověď na přerušení od hardwaru v takovém časovém intervalu, aby dokázal řídit probíhající přirozené procesy reálného světa. Stanovení takového časového intervalu je pochopitelně věc diskuze. V současné době vzhledem k praktickým potřebám a vývoji praktické computer science je nicméně požadován v rozsahu 50 - 150 milisekund. Ten je postačující pro řízení technologických linek v automatizovaných provozech i k ovládání vojenského letounu.

UNIX ve svém prvotním vzniku a vývoji nebyl plánován pro zpracování procesů reálného času. Jeho tvůrci nejen nepředpokládali takové rozšíření, jakého se mu dostalo, ale jejich ambice končily u návrhu a realizace pohodlného operačního systému programátorů ve sdílení času, tj. zajistit pro programátory a uživatele sedící u terminálů odezvu v čase přijatelném pro interakci stroje a člověka. To ovlivnilo návrh modulů plánování procesů, který zajistí zpracování požadavku nejpozději do 1 vteřiny od příchodu přerušení od hardwaru a v předchozím textu kapitoly jsme se této koncepci věnovali především<sup>5</sup>. S rozšířením UNIXu do různých oborů lidské činnosti se sjednotil přístup uživatelů a prostředí výpočtu. Zákonitě vznikl požadavek implementace i v jiných způsobech zpracování než je pouze sdílení času. Práce v UNIXu při ošetření požadavku v reálném čase poprvé implementovala v první polovině 80. let firma Hewlett Packard, když uvedla na trh operační systém UNIX s názvem HP-UX. Velká část produkce této firmy je totiž zaměřena na používání počítačů jako doplňku k měřicí a řídicí technice. Bez úpravy pro reálný čas pro ni tedy vývoj UNIXu neměl smysl. V průběhu 80. let pak vzniklo ještě několik systémů UNIX s úpravou pro reálný čas. Tečku za nestabilitou v této oblasti přineslo 3. vydání SVID a prezentovaný návrh UNIX SYSTEM V.4 začátkem let devadesátých. SVID sice nedefinuje, jak má implementace procesů reálného času v UNIXu být provedena, ale stanovuje volání jádra, pomocí kterého lze měnit strategii přidělování procesoru. SVID se ve své definici liší od posledního vydání POSIXu (které je mladší), protože POSIX definuje pro změnu plánování procesoru (ale nikoliv reálného času) volání jádra začínající na `sched_`. Jejich implementace v UNIX SYSTEM V neznamená zánik implementací podle současného znění SVID, pouze jeho rozšíření. Dnešní situace

v praxi je dána stavem výkonu hardwaru, na kterém má být UNIX provozován. Současně používat operační systém pro sdílení času a procesů reálného času znamená poškodit v provozu jedno nebo druhé. Počítač, řídicí technologický postup, vyžaduje trvale vysoký zájem operačního systému a pracující uživatelé budou muset být stále v defenzivním postavení, což zpomaluje jejich interakce a ruší při práci. Opačný výsledek je zcela nežádoucí. Pozdní obsluha výskytu události reálného procesu může znamenat havárii či vysoké znehodnocení celkového reálného procesu. Výrobci UNIXu proto neimplementují obě varianty současně. Nabízejí varianty UNIXu jak pro sdílení času, tak pro reálný čas. Sami pak přitom nedoporučují variantu pro reálný čas používat pro interaktivní práci uživatelů na jiných problémech. Jejich práce by dnes byla poznamenána vysokými prodlevami.

Implementace doporučení SVID pro řízení procesů reálného času není jednoznačně zveřejňována. Každopádně je jasná koncepce, podle které jsou procesy běžící v operačním systému rozděleny do tří kategorií: procesy reálného času, systémové a uživatelské. Nejvyšší prioritu má přitom kategorie procesů reálného času před systémovými procesy, a ta před zbylými uživatelskými. Procesy systémové a uživatelské jsou plánovány stejnou strategií sdílení času. Je pochopitelné, že proces reálného času musí být také přednostně odblokován jádrem, aby mohl být naplánován k provádění. Současně také každé zpracování volání jádra kterýmkoliv procesem (stav *prováděn v režimu jádra*) musí být jádrem měřeno. Pobyt procesu v jádru přitom nesmí přesáhnout kritickou hodnotu reakce procesu reálného času, který by v daném okamžiku usiloval o přístup k řízené periférii, tj. o vstup do jádra. V klasickém UNIXu není problém dosáhnout přednostní odblokování procesu a zvýhodněná priorita zajistí procesu dobré provádění. Opravdu úzké místo pak zůstává v přerušení práce jádra procesem, což je pro původní UNIX věc nemožná. Jádro je proto přeprogramováno obvykle tak, že při svém provádění obsahuje řadu tzv. měřicích bodů, kdy je přerušeno jeho provádění a zjišťováno, zda nevypřel kritický časový okamžik či zda některý z procesů registrovaný jako proces reálného času nepožaduje vstup do jádra. Pokud ano, je jádro v měřicím bodu přerušeno, je uschován jeho kontext a je provedena obsluha žádoucího procesu. Kontext je poté obnoven a jádro pokračuje ve svém provádění k dalšímu měřicímu bodu. Připustíme-li, že i při obsluze požadavku reálného času dosáhne jádro měřicího bodu a že registrovaných procesů reálného času může být obecně neomezené množství, implementace je prakticky odsouzena k výrazným omezením a v případě nedostačujícího hardwaru dokonce k neúspěchu.

SVID definuje (a SVR4 implementuje) volání jádra `priocntl` jako základní nástroj řízení plánování procesoru. Procesy jsou rozděleny do *tříd* (classes). Strategie plánování procesoru může přitom být pro každou třídu jiná. Současně jsou definované dvě hlavní třídy: *třída reálného času* (real-time) a *třída sdílení času* (time-sharing). Volání jádra, které může používat pouze privilegovaný uživatel, má formát

```
#include <sys/types.h>
#include <sys/procset.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>
#include <sys/tspriocntl.h>
```

```
long priocntl(idtype_t idtype, id_t id, int cmd, ... /* arg */);
```

Parametr `idtype` společně s `id` stanoví skupinu procesů, na kterou je volání jádra aplikováno. Např. použitá textová definice `P_PGID` v `idtype` znamená, že v `id` je PID vedoucího skupiny procesů.

Platnost `priocntl` bude tedy vztažena ke všem procesům jeho skupiny. `idtype` je jedna z definic

P\_PID (proces), P\_PPID (rodičovský proces), P\_PGID (skupina procesů), P\_SID (sezení), P\_CID (třída), P\_UID (uživatel), P\_GID (skupina), P\_ALL (všechny existující procesy) a vyjadřuje smysl uvedené identifikace procesu (nebo procesů) v `id`. Volání jádra dále pokračuje identifikací požadované operace `cmd` a podle jejího významu jsou použity argumenty `arg`, které operace vyžaduje. Operace je jedna z definic `PC_GETCID` nebo `PC_GETCLINFO` (získáme identifikaci a atributy třídy dané identifikace v `arg` do struktur zadaných také v `arg`), `PC_SETPARMS` (nastaví procesům třídu a určující parametry pro přidělování procesoru), `PC_GETPARMS` (získáme identifikaci třídy a prováděcí parametry daných procesů).

Používané struktury a definice v `prionctl` programátor nalezne v `<sys/prionctl.h>`. Stanovená třída reálného času má své definice navíc v `<sys/rtpriocntl.h>`, třída sdílení času v `<sys/tpriocntl.h>`. Každá třída má přitom jinak stanovenou strategii přidělování procesoru.

Třída sdílení času principiálně pracuje tak, jak bylo popsáno v předchozím odstavci. Je možné procesům nastavovat výchozí prioritu a interval provádění.

Třída reálného času je navržena a využívána tak, že procesy této třídy mají určenou prioritu v rozsahu od 0 do `x`, kde `x` je konfiguračně závislé (je čitelné jako `rt_maxpri` v `<sys/rtpriocntl.h>`), přitom vyšší `x` znamená vyšší prioritu. Procesy této třídy jsou zvýhodněny při plánování oproti procesům jiných tříd. Proces s nejvyšší prioritou této třídy, který je *připraven k provádění*, je naplánován a prováděn. Objeví-li se proces této třídy jako *připraven k provádění* a má vyšší prioritu reálného času než proces *prováděný*, je prováděný proces přerušen a je naplánován tento nový proces. Jako obecně v každé třídě je možné nastavit procesům interval provádění a tím dosahovat další zvýhodňování procesů.

Nastavené hodnoty pomocí `prionctl` jsou děděny z rodiče do dětského procesu jako jeden z atributů procesu.

V SVID je také definován privilegovaný příkaz **`prionctl`**, pomocí kterého lze manipulovat s procesy a jejich atributy a přesunovat je z jedné třídy do druhé. Příkaz

**`$ prionctl -l`**

např. vypíše seznam všech právě registrovaných tříd běžících procesů; pomocí volby **`-d`** lze zobrazovat parametry plánování procesů, **`-s`** naopak nastavovat. Dvojici parametrů `idtype` a `id` z volání jádra zastupuje volba **`-i`**, která se používá ve spojení s volbami **`-d`** a **`-s`**. Konečně volbou **`-e`** dokážeme spustit dětský proces daný příkazovým řádkem uvedeným v argumentech tak, že vzniklé procesy jsou registrovány v zadané třídě se zadanými parametry provádění. Např. proces s číselnou identifikací 11621 převedeme do třídy procesů reálného času s prioritou 150 příkazem

**`$ prionctl -s -c RT -p 150 -i pid 11621`**

SVR4 má pro implicitní vyjádření strategie plánování procesů příkaz **`dispadmin`**. Kromě získání informací o třídách a procesech třídy (které má velmi podobný formát jako **`prionctl`**) je možné pomocí

**`$ dispadmin -s file -c class`**

v souboru `file` stanovit plánovací kritéria (interval provádění, výpočet priority) pro třídu `class`. Příkaz není uveden v SVID.

## 2.4.4 Registrace procesu jádrem

Od vzniku až po zánik jádro eviduje procesy ve vnitřních datových strukturách `proc` a `user`. Struktura `user` obsahuje všechny informace o uživatelské části procesu, který je zaveden v operační paměti. Je to rozsáhlá datová struktura, kterou má každý systém definovanou v `<sys/user.h>`. Struktura `proc` eviduje všechny informace o procesu, které jádro potřebuje pro plánování procesoru. Je definována v `<sys/proc.h>`. Kromě těchto hlavních struktur potřebuje jádro ještě evidovat mapování virtuálních adres na fyzické. Je to tzv. tabulka regionů (text, data, zásobník) procesu pro uživatelský režim (*provádění v uživatelském režimu*, User Mode Region Table), tabulka regionů jádra (*provádění v režimu jádra*, Kernel Region Table) a tabulka umístění stránek procesu v operační paměti, tzv. tabulka stránek (Page Table Entry). Všechny datové struktury jádra (a nejen týkající se procesů) jsou úzce svázány s implementací konkrétního UNIXu. Jejich popis a obsah není proto uveden v žádném doporučení nebo dokonce standardu. Informace uváděné v tomto odstavci nejsou proto konkretizovány v podobě jmen položek struktur jazyka C. Nicméně obecný princip, který byl popsán v předchozích částech kapitoly, se v obsahu struktur zrcadlí. Slovní popis proto budeme používat více než jazyk C.

Každý přihlášený uživatel v UNIXu má právo získat seznam procesů jeho sezení nebo dokonce seznam všech procesů, které jsou registrovány v systému. Používá k tomu příkaz **ps**. Při použití bez voleb získá pomocí něj uživatel seznam všech procesů jeho sezení (přesněji procesů používaného terminálu). Např.

```
$ ps
PID  TTY    TIME  CMD
6029 ttys0 0:39   ksh
7349 ttys0 0:00   ps
```

je seznam procesů, které přísluší uživateli terminálu. První řádek je záhlaví výpisu. V následujícím seznamu znamená každý řádek informaci o jednom z procesů. `PID` je číselná identifikace procesu, jak bylo popsáno. `TTY` je jméno terminálu, který je procesu přiřazen. `TIME` je reálný strávený čas procesu v systému (součet všech časů běhu procesu, viz 2.3.6) a `CMD` jméno příkazu procesu. Informace získané při tomto použití mají pro uživatele význam především jako seznam běžících procesů a jejich číselné identifikace, kdy uvažuje o jejich zrušení pomocí některého ze signálů (příkazem **kill**).

Privilegovaný uživatel často používá volbu **-e** příkazu **ps**, protože tak získá (ve stejném formátu) seznam všech procesů registrovaných v systému. Má pak oprávnění zasahovat do běhu jakéhokoliv procesu zasláním některého ze signálů. Podle POSIXu je to ale volba **-A** a pro seznam všech procesů, které jsou spojeny při svém běhu s některým terminálem, pak volba **-a**.

Z pohledu hlubší analýzy atributů běžících procesů v souvislosti se správou paměti a správou procesoru je možné používat volby **-f** (full, úplný výpis) nebo **-l** (long, dlouhý výpis), např.

```
$ ps -f
UID  PID  PPID  C  STIME      TTY  TIME  COMMAND
skoc 6029 151   1 10:09:13 ttys0 0:39  -ksh
skoc 7349 6029 6 10:48:47 ttys0 0:00  ps -f
```

kde jsou nové sloupce výpisu: `UID` je identifikace uživatele (jméno), `PPID` je číselná identifikace procesu rodiče, `C` využitelnost procesoru a `STIME` čas vytvoření procesu. Konečně nejvíce obsažný výpis je s volbou **-l**, např.

\$ ps -l

	F	S	UID	PID	PPID	C	PRI	NI	SZ	WCHAN	TTY	TIME	COMMAND
08	S		1110	6029	151	0	30	20	348	88216920	ttys0	0:39	ksh
08	R		1110	7349	6029	6	63	20	312		ttys0	0:00	ps

Oproti předchozímu výpisu je ve sloupci UID uvedena číselná identifikace uživatele (koresponduje s jeho jménem podle `/etc/passwd`). Ve sloupci PRI je uvedena dynamická priorita, v NI výchozí priorita. Sloupec SZ obsahuje velikost obrazu paměti procesu (v počtu diskových bloků), WCHAN je důvod, pro který je proces zablokován nebo spící (proces běží, je-li tato položka prázdná). Ve sloupci s označením S je uveden stav procesu. Mohou se zde objevit znaky s uvedeným významem:

- O proces je *prováděn v uživatelském režimu*,
- S *blokován (spící)*,
- R proces je ve stavu *připraven k provádění v operační paměti*,
- I je ve stavu *vytvořen*,
- Z proces je ve stavu *mátoha*,
- T trasovaný proces, je pozastaven na pokyn trasujícího rodiče,
- X čeká na přidělení další operační paměti.

Sloupec F je příznak dalších atributů procesu. Položka je uváděna v šestnáctkové soustavě a je logickým součtem používaných hodnot obvykle daných výrobcem. Obecně definované hodnoty jsou tyto:

- 00 proces byl ukončen, takto označená položka tabulky procesů je uvolněna pro další použití,
- 01 systémový proces, je trvale v operační paměti,
- 02 proces je trasován rodičem,
- 04 signál trasujícího rodiče proces pozastavil,
- 08 proces je právě v operační paměti,
- 10 proces je právě v operační paměti zamknut, čeká na dokončení operace.

Uvedené příklady korespondují s SVID. Formát příkazu **ps** i jeho výpis se v různých implementacích UNIXu liší, protože záleží na označení pojmů a způsobu správy paměti a správy procesoru. Typický příklad je v dokumentaci uváděná volba **-c**, která je používána v případě implementace možnosti ovlivnění strategie plánování procesů (volání jádra a příkaz `prctl`). Formát výpisu je pak obohacen o uvedení typu třídy (sloupec CLS), do které proces patří. Priorita má pak jiný význam podle skupin (u procesů reálného času je priorita vyšší, je-li vyšší její číselná hodnota) atd.

Příkaz **ps** není ve své činnosti podporován žádným zvláštním voláním jádra. Obsah datových struktur jádra týkající se procesů, získává čtením odpovídající části operační paměti jádra. Zní to zvláště, ale privilegovaný uživatel může číst virtuální paměť jádra z obsahu souboru `/dev/kmem` (jedná se o soubor přístupu k periférii, viz kap. 3). Ještě do nedávné doby **ps** prohledával obsah tohoto souboru, který se měnil podle aktuálního stavu datových struktur jádra a z informací zde jednoznačně umístěných tabulek procesů vytvářel výpis evidovaných procesů jádrem. SVID dnes uvádí adresář se jménem `/proc`, který má operační systém využívat pro odkazy na aktuální informace týkající se jednotlivých procesů. Adresář v implementacích obvykle obsahuje jména souborů, která odpovídají číselné identifikaci PID procesů. S informacemi zde dostupnými pracuje příkaz **ps** především pomocí volání jádra `open` a `ioctl`. Správce konkrétního systému se v této oblasti musí spolehnout na provozní dokumentaci, protože organizace obsahu adresáře `/proc` je implementačně závislá. Každopádně je ale cesta

čtení obsahu adresáře `/proc` způsobem získání obsahu datových struktur jádra týkajících se jednotlivých procesů. Definice a formát používaného zápisu v tomto adresáři je uložen v souboru `<sys/procfs.h>`, případně v `<sys/proc.h>` a `<sys/user.h>`, tj. struktury `proc` a `user`.

Jádro, které rozhoduje o přidělování a odebírání procesoru, potřebuje mít při opětovém přidělení procesoru dostatek informací, aby mohlo pokračovat v dříve přerušené činnosti procesu. Tyto potřebné informace nazýváme *kontext procesu*. Součástí kontextu procesu jsou obsahy strojových registrů, textový segment procesu, datový segment procesu, zásobník uživatelského režimu, zásobník režimu jádra, struktura `proc`, struktura `user` a odkazy do tabulky regionů pro mapování paměti. Části kontextu, které jsou uloženy v jádru, jsou uschovávány vždy po odebrání procesoru do paměti v jádru, která je typu fifo (zásobník), podle priorit přerušení od hardwaru nebo softwaru.

Struktura `proc` (viz `<sys/proc.h>`) je alokována pro každý vzniklý proces. Obsahuje tyto položky:

- stav procesu (blokován, běžící v režimu jádra atd.),
- příznaky manipulace (zamknut v paměti, odložen, v paměti atd.),
- priority,
- další parametry přidělování procesoru,
- dosud nezpracované signály,
- PPID (tj. PID rodiče),
- adresa a velikost částí odkládaných na disk,
- ukazatel na strukturu `user` nebo další informace s procesem související (např. tabulka mapování regionů),
- ukazatel do zřetěženého seznamu registrovaných procesů,
- čas, který zbývá do případně nastaveného signálu `SIGALRM`.

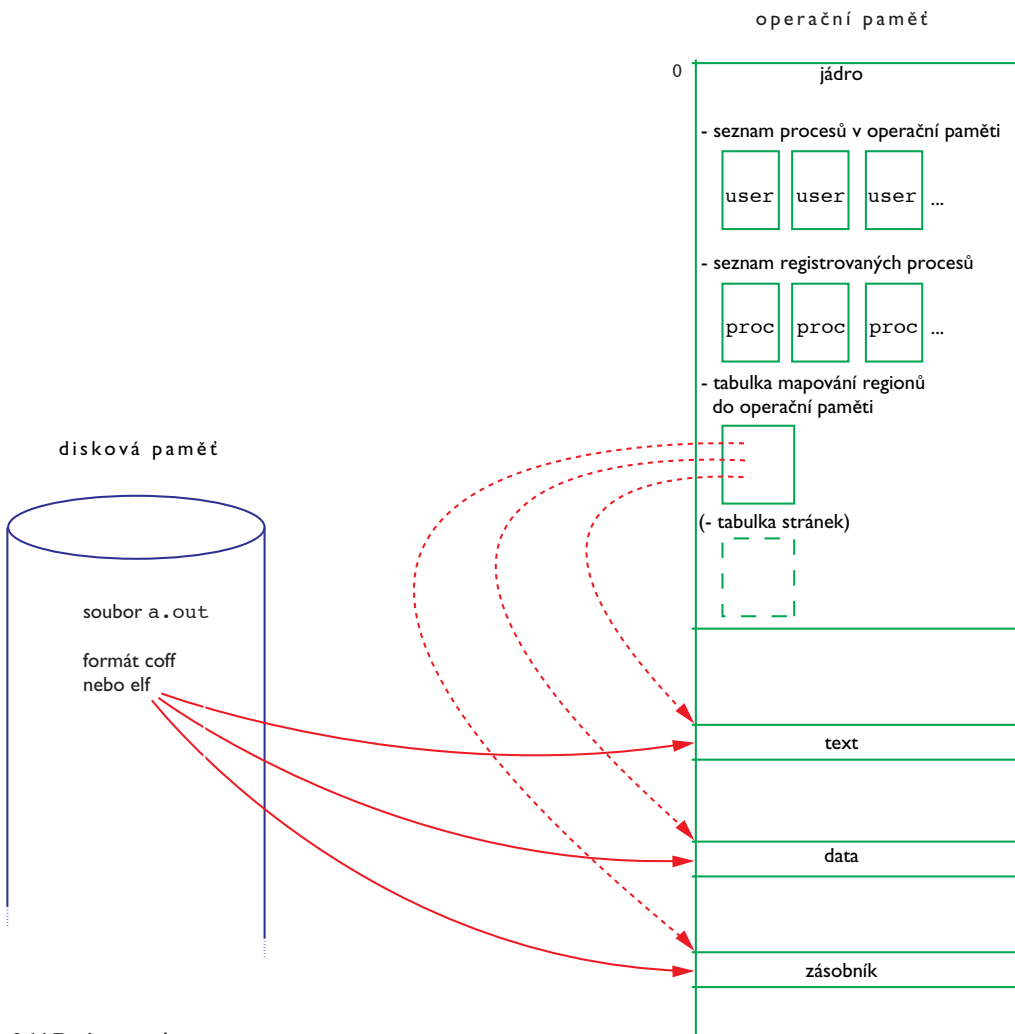
Struktura `user` (viz `<sys/user.h>`) je inicializována vždy při zavedení procesu do operační paměti. Obsahuje:

- poslední chybu některého volání jádra (`errno`),
- efektivní a reálnou identifikaci uživatele a skupiny uživatele,
- ukazatel na strukturu `proc`,
- návratovou hodnotu volání jádra,
- adresu vyrovnávací paměti a počet bytů v/v operací,
- ukazatel na pracovní, kořenový a nadřazený adresář,
- tabulku otevřených souborů,
- argumenty aktuálního volání jádra,
- velikost textového, datového segmentu a velikost zásobníku,
- pole popisu zpracování signálů,
- uživatelský a systémový čas procesu a jeho dětí,
- adresy a velikost systémových vyrovnávacích pamětí procesu nad diskovou pamětí,
- masku přístupových práv při vytváření souborů,
- data inicializace terminálu,
- zásobník kontextu jádra.

V době inicializace jsou informace nutné k vytvoření procesu (např. text programu, data atd.) čerpány z obsahu proveditelného souboru s obecným jménem `a.out` (obr. 2.14). Jeho formát je uveden ve



čtvrtém svazku provozní dokumentace. V rámci SVR4 je definován obecný formát obsahu tohoto souboru. Formát nese označení `coff` (`c`ommon `o`bject `f`ile `f`ormat) nebo nověji `elf` (`e`xecutable and `l`inking `f`ormat). Informace v něm obsažené jsou především text programu, data a jejich inicializace, definice použitého modelu paměti atd. Záhlaví souboru obsahuje tzv. magické číslo (magic number), což je vyjádření typu práce při inicializaci informací z `a.out` do operační paměti, kdy je proces vytvářen. Žádný z používaných formátů `a.out` nebo struktur `proc` nebo `user` není obecně stanoven v SVID nebo dokonce v POSIXu.



Obr. 2.14 Registrovaný proces



## 2.5 Shell

Shell (obr. 2.15) je proces, který zpřístupňuje vlastnosti jádra na úrovni příkazů uživatele.

Příkaz zadaný uživatelem na terminálu je vstupním pokynem provedení akcí, které požaduje shell od jádra. Často je to vytvoření nového procesu (pomocí volání jádra `fork` a `exec`), kterému shell předá v parametrech `argv` požadavky uživatele, nebo jde o jiné akce požadavků na hardware, komunikace s ostatními procesy atd. Plně červené čáry na obr. 2.15 vyjadřují úzkou komunikaci každého procesu s jádrem, kdežto přerušované virtuální efekt této komunikace, protože komunikace procesu s jiným procesem je vždy realizována i evidována jádrem (volání jádra `kill`, `signal`, `pipe` atd.), příbuzenské vztahy procesů rovněž atd.

Shell je z pohledu uživatele textově orientovaný příkazový interpret. Začátečník ho mylně chápe jako správce souborů (file manager) a hledá prostředí snadné manipulace a orientace v systému souborů. Přestože byly takové programy správy souborů v UNIXu naprogramovány a každý program správce oken (window manager) grafického prostředí UNIXu (X-Window System) jej obsahuje, správce systému nebo programátor při práci v systému používá stále textovou komunikaci. Důvodem je, že abstrakci, kterou je nucen si uživatel UNIXu vytvořit při práci v systému, lze velmi obtížně vyjádřit graficky při zachování snadného ovládání. Příkladem jsou všechny doposud popsané akce jádra s procesy v této kapitole. Pochopení této abstrakce a práce v ní pomocí Bourne shellu, C-shellu nebo KornShellu je podmínkou iniciace začátečníka UNIXu.

Shell je také programovací jazyk na úrovni procesů. Je řídicím procesem ostatních procesů uživatelevo sezení a na základě výsledků jejich práce má k dispozici řídicí struktury pro rozhodování nad dalším vývojem své práce podle algoritmu zapsaného uživatelem. Řídicí struktury v shellu přitom může používat uživatel jak interaktivně, tak pomocí textového souboru s programem. Shell tedy umožňuje psát systémové programy, kde moduly zpracování jsou jednotlivé procesy.

Shell je výchozím procesem uživatelevo sezení. Je vytvořen jako výsledek přihlášení uživatele. Všechny ostatní uživateleovy procesy jsou vytvářeny na jeho pokyn, skončí-li svoji činnost, sezení zaniká. V souboru `/etc/passwd` nastavuje správce systému shell každého uživatele. Přestože shell může být v různé modifikaci (Bourne shell jako `/bin/sh`, KornShell jako `/usr/bin/ksh`) nebo dokonce může být nahrazen některou aplikací, jeho funkce řídicího procesu uživatelevo sezení, tak jak je popsáno ve zbytku této kapitoly, zůstává stále stejná.

Ve zbylém textu tohoto článku nebudeme detailně uvádět seznam všech příkazů shellu. Uživatel detaily nalezne v provozní dokumentaci určitého shellu. Zde se zaměříme více na princip a možnosti, které uživatel nebo programátor využívá. Pro označení procesu shell budeme často používat jen označení **sh**, protože pro systém je podstatné obsaženo již v Bourne shellu.

### 2.5.1 Proces shellu

První shell uživatelevo sezení je dítětem procesu **init** a nemůže se tohoto vztahu zbavit. Všechny další procesy, které shell vytváří, jsou evidovány jako jeho děti. Uživatel může pomocí příkazu **nohup** (viz čl. 2.2) prohlásit proces ve zbytku příkazového řádku za nezávislý na svém rodiči a založit si tak svoji vlastní skupinu procesů.

Shell přijímá příkazy uživatele. Příkaz je první slovo příkazového řádku shellu a může se jednat o jeho *vnitřní příkaz* (build-in command), tj. příkaz, který shell realizuje sám, na který nevytváří žádný dětský proces. Jména všech vnitřních příkazů musí uživatel dobře znát. Jsou uvedena v provozní dokumentaci. Pokud shell nerozezná příkaz uživatele jako vnitřní příkaz, hledá ve smluvených adresářích proveditelný soubor jména stejného, jako je jméno příkazu. Nalezne-li jej, vytváří nový dětský proces řízený programem z tohoto souboru (nejprve voláním jádra **fork** a v dalším kroku **exec**).

Proces shellu může být využit jako proces, jehož program bude vyměněn za jiný řídicí program. Shell to provede pomocí volání jádra **exec** na základě pokynu uživatele vnitřním příkazem **exec**. Po ukončení práce procesu daného příkazovým řádkem v parametrech **exec** ukončí proces svoji činnost a sezení zaniká. Sezení zaniká také zadáním vnitřního příkazu **exit** (shell použije volání jádra **exit**), který má stejný význam jako kombinace kláves Ctrl-d, kterou se uživatel obvykle odhlašuje. Klávesu (nebo kombinaci kláves), na základě které proces shellu ukončí sezení, lze ovšem změnit na jinou (příkazem **stty** a voláním jádra **ioctl**). **exit** je věc neměnná.

Vytvoří-li shell proces na základě pokynu uživatele, shell čeká na jeho dokončení, a teprve pak umožní uživateli zadávat další vstupní příkaz (dětský proces běží na popředí, foreground). Dětský proces běží synchronně s čekajícím rodičem. Pokud uživatel ukončí příkazový řádek pro dětský proces znakem **&**, pak shell nečeká na dokončení tohoto dítěte (dětský proces běží na pozadí, background). Pro uživatele to znamená, že může zadávat další příkazový řádek. Znak výzvy k zápisu dalšího příkazového řádku je většinou **\$** (uživatelem měnitelný text, viz odst. 2.5.2), např.:

```
$ cc prog.c &
```

```
1628
```

```
$ _
```

kde vypsaná číselná hodnota před **\$** je hodnota PID dítěte, která je vypisována procesem shellu do kanálu č. 2 (pozor, shellu, ne dětského procesu!). Dětský proces spuštění na pozadí běží asynchronně se svým rodičem, tj. shell nehlídá jeho ukončení voláním jádra **wait**. Uživatel může synchronizaci vyžadovat dodatečně vnitřním příkazem **wait**:

```
$ wait 1628
```

```
—
```

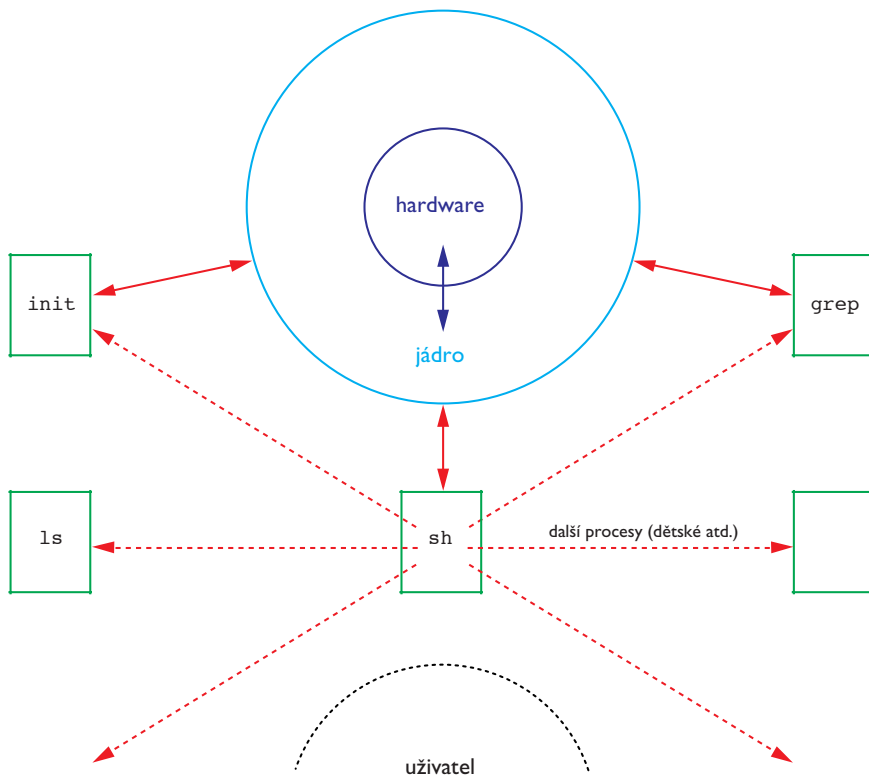
Shell pak nabízí zadání dalšího příkazového řádku až po ukončení dítěte. **wait** bez parametrů je synchronizace na všechny spuštěné děti.

PID dítěte může uživatel použít při zasílání signálů. Zadává-li uživatel vnitřní příkaz **kill**, kde v parametru určí typ signálu a PID procesu, kterému je signál určen, shell přebírá tento požadavek a převádí jej na volání jádra **kill**. Signál tedy posílá shell. Uživatel může také opačně proces shell připravit na příchod signálů. Používá k tomu vnitřní příkaz **trap**, např. příkazem

```
$ trap 'echo Přišel signál č. 15' 15
```

maskujeme příchod signálu č. 15 tak, že je proveden příkazový řádek ohraničený znaky **'**. Shell vnitřní příkaz **trap** realizuje pomocí volání jádra **signal**. Maskuje tak svůj proces pro příchod signálů od jiných procesů.

Uživatel může zjistit PID dětí shellu ze standardního chybového výpisu při jejich spouštění na pozadí. Seznam dětí a jejich PID může také zjistit příkazem **ps**. PID posledního dítěte spuštěného na pozadí



Obr. 2.15 Shell

eviduje shell také v obsahu jeho proměnné `!`. PID pro samotný shell je obsahem proměnné `?` (viz následující odst. 2.5.2). V kontextu odst. 2.5.3 pak může program shellu využívat právě uvedených metod synchronizace podstromu procesů řídicího procesu shell.

Pro pohodlnější práci uživatele u terminálu umožňuje kterýkoliv současný shell (C-shell, KornShell, Job Control Shell atd.) také synchronizaci procesů, jejich pozastavování a opětné pokračování, přesun z popředí na pozadí atd., a to pomocí vnitřních příkazů **jobs**, **bg**, **fg**, (příp. **stop** a **suspend**). Výhodou je také implicitní pozastavování procesů běžících na pozadí v případě jejich požadavku v/v operace s terminálem, kdy se texty výpisu na terminál jednotlivých dětí nesmíchají.

Shell umožňuje uživateli snadno měnit v/v kanály dětských procesů. Použitím znaků zvláštního významu `>`, `>>`, `<`, `<<` a `|` může uživatel v příkazovém řádku zadávat *přesměrování vstupu* nebo *přesměrování výstupu* dětského procesu z terminálu do souboru nebo roury. Mechanismus, který shell

využívá k zajištění těchto požadavků, je manipulace s tabulkou otevřených souborů (tabulkou kanálů) procesu dítěte, která je od volání jádra `fork` až po `exec` zcela v moci rodiče. Když ten pomocí volání jádra `open` a `close` (případně `dup`) změní význam kanálů, dětský proces převezme toto nasměrování, a dokud sám tento význam nezmění, v/v je upraven podle shellu. Např. příkazový řádek uživatele

```
$ ls > adr
```

shell realizuje fragmentem programu

```
#include <fcntl.h>
#include <stdio.h>

...
if(fork()==0)
{
    close(1);
    open("adr", O_WRONLY | O_CREAT, 0644);
    execl("/bin/ls", "ls", NULL);
}
```

Uzavřením položky č. 1 tabulky kanálů voláním jádra `close` kanál uvolníme pro další použití, přičemž 1 je použita jako první vhodný kanál při následujícím `open`. Obrazovka terminálu (kanál č. 1) je tak změněna na soubor `adr` a program `ls`, za který je `sh` (už ale jako dítě) vyměněn, data místo na obrazovku terminálu vypisuje do souboru.

Komplikovanější situace nastává při spojení dvou procesů rourou. Volání jádra `pipe` (podle odst. 2.3.3) obsadí první dva volné kanály tabulky. Shell musí ještě stanovit, který kanál bude pro vstup a který pro výstup. Použije proto opět volání jádra `close`, ale teprve po přidělení roury. Pomocí volání jádra `dup` kanály roury spojí s kanálem č. 0 a 1. Např. příkazový řádek

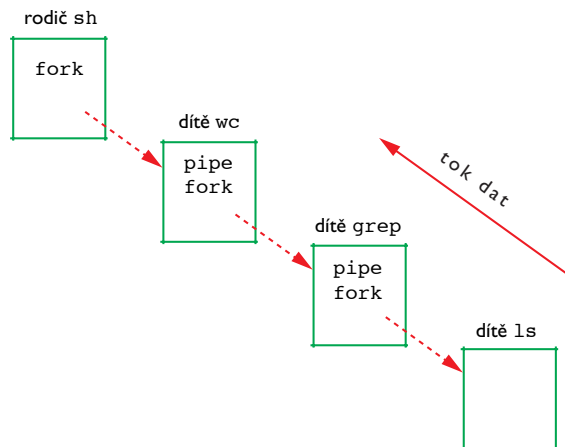
```
$ ls | grep .c | wc
```

realizuje shell tak, že vytváří děti `ls`, `grep` a `wc` postupně v obráceném pořadí zápisu podle obr. 2.16.

`ls` je posledním v řadě vytvořených dětských procesů, protože může jako první ukončit svoji činnost. Pokud skončí, žádný dětský proces s ním již existenčně není spojený. Kolona příkazů spojených rourou proto není ve svém výsledném efektu ohrožena. Rodič v každém dítěti po jeho vytvoření a před voláním jádra `exec` provede manipulaci s kanály č. 0 nebo 1 podle fragmentu:

```
pipe(fd);
...
close(0);
dup(fd[0]);
close(1);
dup(fd[1]);
exec( ... );
```

Uzavřeme vždy odpovídající kanál terminálu (0 pro klávesnici a 1 pro obrazovku) a voláním jádra `dup` spojíme vždy odpovídající konec roury s uvolněným kanálem. Dětský proces v `exec` pak pracuje se stejnými kanály, ale jejich nasměrování zdědil po rodiči. Fragment zdrojového textu pro shell uvedené kolony procesů tedy bude



Obr. 2.16 Kolona procesů `ls | grep .c | wc`

```

#include    <stdio.h>

...
int fd_lsgrep[2], fd_grepwc[2];
...
switch(fork())
{
case -1:
    exit(1);
case 0:
    pipe(fd_grepwc);
    switch(fork())
    {
case -1:
        exit(1);
case 0:
        pipe(fd_lsgrep);
        switch(fork())
        {
case -1:
            exit(1);
case 0:
                close(1);
        }
    }
}

```

```

        dup(fd_lsgrep[1]);
        close(fd_lsgrep[0]);
        execl("/bin/ls", "ls", NULL);
    default:
        close(0);
        dup(fd_lsgrep[0]);
        close(fd_lsgrep[1]);
        close(1);
        dup(fd_grepwc[1]);
        close(fd_grepwc[0]);
        execl("/bin/grep", "grep", ".c", NULL);
    }
default:
    close(0);
    dup(fd_grepwc[0]);close(fd_grepwc[1]);
    execl("/bin/wc", "wc", "-l", NULL);
}
}
..

```

V příkladu každý nepoužívaný konec roury uzavíráme voláním jádra `close`, protože jinak blokuje kanál pro použití jiným procesem, který jej potřebuje pro určitou v/v operaci.

## 2.5.2 Proměnné

Každý shell disponuje možností práce s daty ve formě textových řetězců. Jedná se o *proměnné shellu*. Uživatel u terminálu mohou připadat jako jednoúčelové textové proměnné, jejichž obsah lze naplnit způsobem

**\$ PROMĚNNÁ=obsah**

nebo textem standardního výstupu procesu

**\$ PROMĚNNÁ=`příkaz`**

nebo ze standardního vstupu vnitřním příkazem

**\$ read PROMĚNNÁ**

V další práci lze jejich obsah testovat (např. vnitřním příkazem **test**) a používat v příkazových řádcích odkazem **\$PROMĚNNÁ**. Proměnné v shellu jsou také nositeli informací z rodiče do dětských procesů. Např. vnitřní proměnná **TERM** obecně definuje typ alfanumerického terminálu pro obrazovkové operace (přesun kurzoru, vyčištění obrazovky atd.) a její jméno a obsah je přitom přenášen z rodiče do dítěte pomocí pole textů **envp** ve volání jádra **exec** do parametrů funkce **main** dětského procesu, jak bylo uvedeno v odst. 2.3.1. Ne všechny uživatelem definované proměnné jsou automaticky přenášeny do dětských procesů. Pokud není řečeno jinak, mají pouze místní platnost, tj. jsou definovány pro použití pouze v rámci procesu **sh**, kde vznikly. Použitím vnitřního příkazu **export** uživatel v jeho parametrech stanovuje jména proměnných, které jsou zařazeny pro globální použití, tzn. od okamžiku použití

**export** jsou viditelné v poli **envp** každého dětského procesu. Je zvykem, že každý proces je programován tak, aby při řízení dětského procesu propouštěl všechny proměnné, které sám našel v poli **envp**. Seznam proměnných shellu, které jsou označeny pro vývoz do dětských procesů, získá uživatel použitím příkazem **set** bez parametrů. **set** je důležitý a silný příkaz. Jeho možnosti jsou definovány v SVID a každá implementace je respektuje. Rovněž tak je v dokumentaci uveden seznam všech vnitřních proměnných shellu, které mají zvláštní význam pro práci uživatele u terminálu (důležitá je např. **PATH**, která obsahuje seznam adresářů, které shell prohledává – a žádné jiné – hledá-li proveditelný soubor s programem pro proces vnějšího příkazu nebo **LANG** pro definici národního prostředí atd.). Seznam všech proměnných shellu uživatelského sezení definuje *prostředí uživatele* u terminálu.

Proměnnou shellu není potřeba deklarovat. Vzniká při definici jejího obsahu. Proměnnou lze zrušit vnitřním příkazem **unset**.

### 2.5.3 Programování

V úvodu kapitoly jsme uvedli význam volání jádra **exit** pro ukončení procesu. Každý proces používá toto volání jádra. Dokonce i když není zapsáno jako součást zdrojového kódu programu, je doplněno překladačem a sestavujícím programem (v jazyce C před pravou složenou závorkou funkce **main**). Jeho formát je

```
void exit(int status);
```

nebo

```
void _exit(int status);
```

(POSIX definuje pouze druhou variantu a **exit** uvádí pouze jako odkaz na knihovní funkci jazyka C se stejným významem). Volání jádra nemá návratovou hodnotu, protože jejím uskutečněním proces končí. Parametr **status** je celočíselná hodnota, která je předána rodiči procesu. Otec ji získá voláním jádra **wait** a může podle její hodnoty určit způsob ukončení dětského procesu. Shell obsah proměnné **status** přebírá a dokáže analyzovat. Je to hodnota, která je testována v řídicích strukturách shellu, tj. ve vnitřních příkazech **if**, **while** a **until**. Principiálně je totiž řídicí struktura shellu test úspěchu nebo neúspěchu provedeného dětského procesu. Úspěch (logická hodnota pravda – true) je reprezentována číselnou hodnotou 0 ve **status**, jakákoliv jiná hodnota je neúspěch (logická nepravda – false)<sup>6</sup>. Znamená to, že kupříkladu v zápisu

```
$ if příkaz then echo Dobře else echo Špatně
```

je **příkaz** příkazový řádek, na základě kterého vytváří shell dětský proces **příkaz**. Po jeho ukončení shell převezme jeho návratový status (tj. **status**) a řídicí struktura **if** rozhoduje o pokračování práce shellu zpracováním příkazu

```
echo Dobře
```

pokud je návratový status dítěte 0. Při jiných hodnotách pokračuje provádění shellu příkazem

```
echo Špatně
```

Na místě **příkazu** může být sekvence příkazů, tj. příkazy odděleny znakem **;** a mohou být ohraničeny v závorkách **{** a **}**. Testován je návratový status posledního ze sekvence příkazů. Algoritmus zpracování

příkazů shellu programátor tedy vyjadřuje podmínkami úspěchu dětských procesů jakožto dílčích modulů zpracování dat.

Programování v shellu znamená, že uživatel využívá řídicích struktur shellu. Může je přitom používat i v rámci svého sezení, tj. zadávat řídicí struktury jako vnitřní příkazy shellu jeho sezení. Např.

```
$ for i in *
> do
> if test -d $i then ls -ld
> done
```

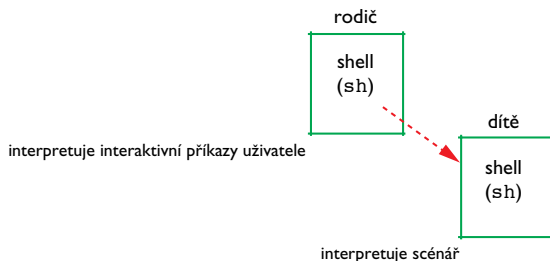
je zadání cyklu pro všechny soubory pracovního adresáře (jejich jména jsou postupně přiřazována do obsahu proměnné `i` v každé iteraci cyklu). V těle cyklu je uveden test na atribut souboru. Pokud je adresářem, vytiskne se na standardní výstup příkazem `ls` jeho jméno a atributy. Příklad tedy tiskne jména a atributy všech podadresářů pracovního adresáře. Znak `>` je interaktivní upozornění shellu na nedokončený příkazový řádek.

Uživatel může také sekvenci příkazů uložit textovým editorem do souboru a nabídnout shellu tento soubor ke zpracování. Takovému souboru říkáme příkazový soubor. V UNIXu se používá označení *scénář* (script). Scénář shell interpretuje na základě příkazu

```
$ sh scénář
```

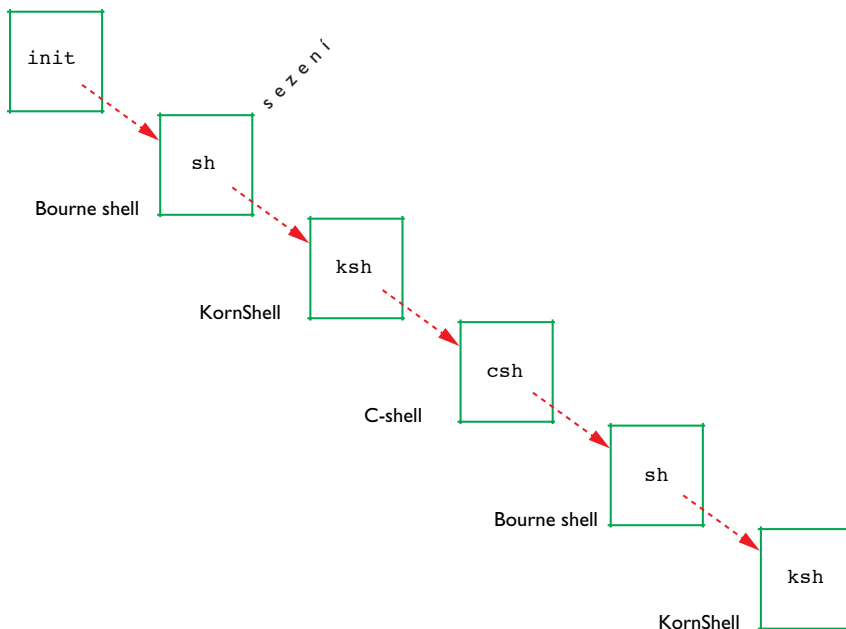
V tomto konkrétním případě vzniká situace, kdy shell sezení uživatele (který s námi komunikuje vždy výpisem znaku `$` nebo `>`) startuje nový dětský proces dalšího shellu, který **scénář** interpretuje. Po vyčerpání všech příkazů souboru **scénáře** dětský proces končí a rodič pokračuje ve zpracování interaktivních příkazů uživatele (rodič čekal na dokončení dítěte, protože příkazový řádek pro interpretaci **scénáře** nebyl ukončen znakem `&`). Situaci ukazují obr. 2.17.

Zpracování scénáře novým procesem má řadu výhod. Dětský proces dědí všechny atributy svého rodiče. Stejně tak jsou dítěti exportovány proměnné shellu rodiče – prostředí provádění je tedy stejné. Přitom vznik dalších proměnných nebo modifikace obsahu proměnných získaných od rodiče má lokální platnost. Tyto změny se nikdy nepřenáší do procesu rodiče a při ukončení dítěte zanikají. Prostedí provádění procesů shell sezení proto není nikdy ovlivněno dětskými shelly, pokud to rodič explicitně nevyžaduje.



Obr. 2.17 Interpretace scénáře





Obr. 2.18 Spuštění více interaktivních procesů shell v sezení

Dětský proces shell má také návratový status. Je to parametr jeho vnitřního příkazu **exit**. Pokud ve scénáři není uveden nebo pokud **exit** nemá parametr, je návratový status přebírán z posledního provedeného dětského procesu scénáře. Nic nebrání testovat návratový status dětského shellu v rodiči v některé z řídicích struktur. Lze tedy psát

```
$ if sh scénář then ...
```

Uživatel může také použít pro vznik scénáře závorky ( a ). Sekvence příkazů, která je závorkami ohraňována, je provedena nově vytvořeným dětským procesem shellu. Např. v

```
$ if ( cd $HOME/src; make all ) then aplikace
```

bude příkaz **aplikace** proveden v případě, že příkaz **make** provedený dětským shellem vrátí úspěch.

Uvádět jméno příkazu **sh** před každým scénářem je nepohodlné. Navíc řada programů v UNIXu je napsána ve formě scénářů a uživatel by musel jejich seznam znát. Proto je možné souboru se scénářem přiznat atribut proveditelnosti (který je souboru **a.out** přiřazen v okamžiku vytvoření) pomocí příkazu **chmod** (viz čl. 2.2), shell rozpozná obsah souboru pro textové zpracování a vytváří dětský shell, který scénář interpretuje. Vzhledem k tomu, že syntaxe jednotlivých shellů se někdy může lišit (někdy i hodně, C-shell má jinou gramatiku než Bourne shell nebo KornShell), každý shell respektuje ve scénáři

řích první řádek komentáře (pokud je uveden) jako definiční pro cestu k souboru s proveditelným souborem toho shellu, který tento scénář má provádět. Např.

```
# /usr/bin/ksh
# následuje text scénáře pro KornShell
...
```

Znak # uvádí pro každý shell ve scénáři řádek komentáře. V příkladu je řečeno, že scénář bude implicitně prováděn KornShellem.

Používání dětských procesů shellu pro interpretaci scénářů lze používat do libovolné hloubky vnoření. Nic nebrání psát také rekurzivní scénáře (scénář startuje dětský shell, který interpretuje opět tentýž scénář).

Pokud uživatel použije vnitřní příkaz . , explicitně vyjádří požadavek interpretace scénáře nikoliv novým dětským procesem shellu, ale procesem shellu, kterému je vnitřní příkaz . určen. Zpracování scénáře je provedeno jako vstup pro aktuální shell. Shell přitom nekončí s provedením posledního příkazu scénáře, ale pokračuje v další činnosti. Např.

\$ . scénář

je provedení **scénáře** procesem shellu sezení.

Termín shell sezení zde není zcela přesný, protože uživatel může vyvolat pro interaktivní komunikaci další (třeba i jiný) shell v rámci jednoho sezení, např.

```
$ ksh
$ csh
% sh
$ ksh
$
```

je postupné spuštění nových dětských procesů podle obr. 2.18.

V rámci sezení v příkladu pracuje 5 procesů shell, postupně čekajících na dokončení vždy svého prvního dítěte. Návrat k výchozímu procesu shellu sezení (rodiči, vedoucímu této skupiny procesů) lze

```
$ exit
$ exit
% exit
$ exit
$
```

(C-shell má znak interaktivní komunikace s uživatelem %).

Vznikající shell uživatelova sezení v okamžiku přihlašování uživatele (typ shellu určí správce v tabulce /etc/passwd) obvykle akceptuje některý scénář smlouveného jména (pro Bourne shell nebo KornShell je to soubor se jménem .profile). Ještě před výpisem znaku interaktivní komunikace (zde \$) takový scénář interpretuje stejným způsobem jako na základě zadání vnitřního příkazu . . Jde o inicializační scénáře, které nastavují lokální prostředí práce uživatele u terminálu (export některých místních proměnných pro používané konkrétní aplikace uživatele, nastavení typu terminálu atd.).

Programátor v shellu má pro lepší komfort práce s proměnnými možnost používat (dnes již vnitřní) příkazy **test** a **expr**. Příkaz **test** má synonymum závorek [ a ], ve kterých uzavřený seznam parametrů příkazu **test** má tentýž význam. Např.

```
if [ $PROM=`expr $I + 1` ] then echo Už je to tady
```

vypisuje text **Už je to tady** v případě shody obsahu proměnné **PROM** a obsahu proměnné **I** (třeba jako invariantu cyklu) zvýšené o hodnotu 1 (**expr** zpracuje oba textové argumenty a na standardní výstup vypíše textově výsledek).

V oblasti proměnných jsou dále zajímavé tzv. poziční parametry scénáře, tj. argumenty příkazového řádku, kdy příkaz je realizován scénářem. Jsou to proměnné **0**, **1**, **2** atd., přitom v obsahu mají případné parametry příkazového řádku, tj. např.

```
$ scénář první druhý třetí
```

bude v textu scénáře obsah proměnné **1** naplněn textem **první**, **2** textem **druhý** a **3** textem **třetí**. Proměnná **0** obsahuje jméno scénáře (text **scénář**). Celý příkazový řádek bez jména scénáře je viditelný v proměnné se jménem **\***.

Uživatel tak pomocí programovacích možností shellu v UNIXu má k dispozici nástroj pro tvorbu nových programů nebo programových celků, kde jsou jednotlivé manipulace s daty realizovány pomocí procesů. Tak jsou navzájem jednotlivé operace odděleny operačním systémem, což zvyšuje jejich bezpečnost. Uživatel k tomu dopomáhá také velké množství tzv. nástrojů programátora (tools), což jsou programy, které provádí manipulace s daty zadané v příkazovém řádku. Nástrojů programátora je velké množství a velká část jich je již zahrnuta v POSIXu.

<sup>1</sup> POSIX volání jádra **chroot** odmítá z důvodů nepřenositelnosti.

<sup>2</sup> POSIX definuje pouze

```
main(int argc, char *argv[]);
```

třetí parametr nedefinuje, je uvedeno používání externí proměnné definované jako

```
extern char **environ;
```

SYSTEM V dovoluje používat oba způsoby.

<sup>3</sup> Angl. clock ticks, je obvykle setina vteřiny. SVID definuje tik hodin stroje jako  $1 / \{ \text{CLK\_TCK} \}$  vteřiny, CLK\_TCK je definovaná konstanta, obvykle tedy 100. POSIX uvádí konstantu  $\{ \_SC\_CLK\_TCK \}$  téhož významu.

<sup>4</sup> Výjimka může být u víceprocesorových architektur. Běžný způsob využití více spřažených procesorů řízených jedním operačním systémem obvykle není distribuce procesů na jednotlivé procesory, ale analýza proudu zpracování všech požadovaných instrukcí a distribuce jeho částí bez ohledu na to, kterému procesu patří. V současné době je tato partie především předmětem výzkumu operačních systémů.

<sup>5</sup> Nejde o to, že by přerušení nebylo přijato operačním systémem. Je zpracováno a předáno procesu, proces je odblokován a souperí o přístup k procesoru. Situace může být ale neprůchozí v okamžiku naplánování procesu, které může trvat delší časově kvantě. Také jiný proces, který je právě ve stavu *prováděn v režimu jádra*, je nepřerušitelný, takže i tady dochází k prodlevě.

<sup>6</sup> Oproti zpracování návratových hodnot funkcí v jazyce C přesně naopak.



### 3. SYSTÉM SOUBORŮ

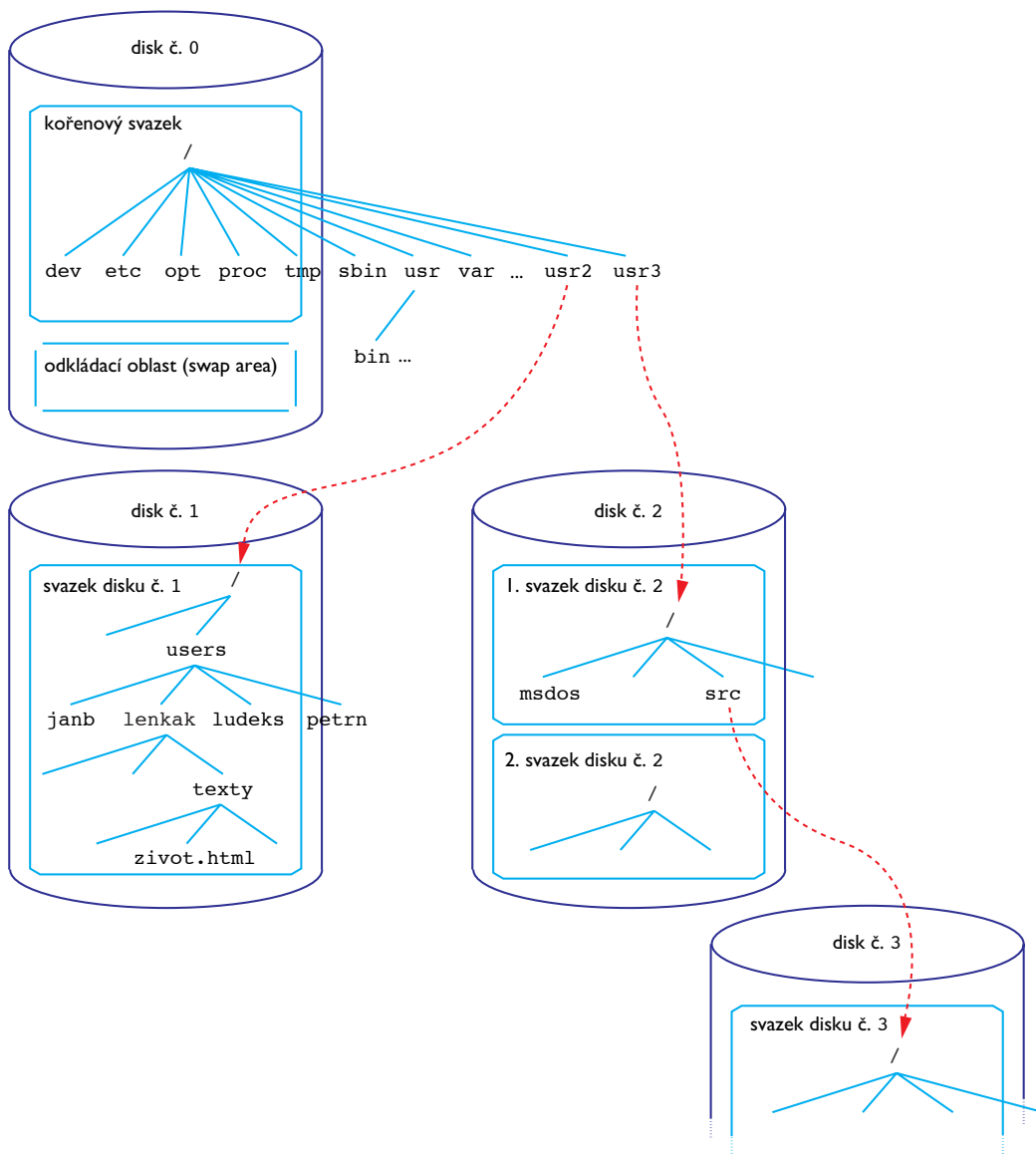
*Systém souborů* (file system) je část softwaru, která umožňuje práci procesů s daty na externích médiích. Proces používá volání jádra systému souborů, která umožňují přístup k hardwaru s daty na externích médiích pro zápis nebo čtení. Současné technologie hardwaru ukládají takto informace po zápisu čitelné i s odstupem několika let, a to i bez spojení se zdrojem elektrického napětí. Systém souborů přitom využívá médií, která zpracovávají informace po blocích dat nejlépe s náhodným (random) přístupem, tj. diskovou paměť. Moduly jádra pracující pro systém souborů mají vysokou důležitost, stejnou jako moduly pracující pro procesy.

Pod anglickým pojmem file system rozumíme jednak technologii přístupu k diskům, jednak způsob organizace dat na disku. V českých zemích se pro druhý význam vžil termín *svazek*. První překládáme jako systém souborů.

Datová základna systému souborů zahrnuje všechna potřebná systémová data (program jádra, programy systémových procesů, nástroje programátora, tabulky registrace uživatelů atd.), provozní data (tabulky pro práci systémových procesů, evidence právě přihlášených uživatelů atd.) a uživatelská data. Datová základna je uživateli viditelná jako hierarchická struktura adresářů, ve kterých jsou uloženy soubory s daty. Struktura adresářů má tedy v prvním přiblížení stromový charakter. Každý adresář může obsahovat libovolný počet podadresářů a ty rovněž další podadresáře atd. do libovolné hloubky. Strom adresářů má výchozí adresář s označením /. Je označován termínem *kořen* (root) nebo *kořenový adresář* (root directory) a disk (svazek), na kterém je uložen, jako *kořenový svazek* (root file system). Kořenový svazek je velmi důležitý, protože obsahuje všechny důležité programy a data nezbytná pro chod operačního systému. Každý svazek je ovšem omezen kapacitou disku, na kterém je uložen. Systém adresářů proto pokračuje na dalších discích (svazcích) tak, že jsou svazky spojeny s některými koncovými adresáři kořenového svazku. Na již připojené svazky lze dále připojovat další a další svazky do libovolné hloubky. Situaci ukazuje Obr. 3.1.

Každý svazek na dalším z disků obsahuje výchozí adresář (někdy se mu říká kořenový adresář svazku), který je na obrázku označován také znakem /, ale nikdy pod tímto jménem pro žádného uživatele nebude viditelný, protože každému uživateli je dostupný teprve po připojení na strom svazků. Výchozí adresář svazku na disku č. 1 podle obrázku bude uživateli přístupný cestou /usr2, protože je s tímto adresářem spojen. Výchozí adresář 2. svazku disku č. 2 nebude uživateli vůbec přístupný, protože spojení jeho výchozího adresáře se stromem svazků nebylo zatím provedeno. Připojení svazků určuje správce systému. Na obrázku jména adresářů /usr2 a /usr3 jsou dána správcem systému, ostatní adresáře kořenového svazku jsou uvedeny podle konvencí SVID a jsou neměnné. Rovněž je na disku č. 0 uvedena oblast pro odkládání procesů (swap). Nejedná se o svazek, operační systém k této části disku přistupuje jinými algoritmy. Disk č. 2 obsahuje svazky dva, velké disky bývají často rozdělovány na menší logické části z důvodů snazší údržby a bezpečnějšího oddělení dat. Svazek disku č. 3 je připojen na adresář src 1. svazku disku č. 2, disk č. 3 bude tedy využíván cestou /usr3/src.

Každý adresář může obsahovat soubory různých typů. Hlavní z nich, vyjma podadresářů, jsou obyčejné soubory a speciální soubory. Obyčejné soubory jsou nositeli vlastních dat, speciální soubory znamenají přístup k perifériím. Každý uživatel má správcem systému přidělen určitý adresář, kterým začíná oblast jeho uživatelských dat. Takový adresář nazýváme *domovský adresář* (home directory) a v oblasti jeho



Obr. 3.1 Strom adresářů, topologie svazků

podstromu může uživatel libovolně vytvářet a rušit podadresáře a jiné typy souborů. Zbývající část systémového stromu adresářů je mu obvykle pro zápis a změny nedostupná. V UNIXu je pro čtení zvykem zveřejňovat všechna systémová data libovolnému uživateli (jsou omezení, která souvisí s bezpečností výpočetních systémů, viz. kap. 9).

*Speciální soubory* (special files) jsou rozhraním přístupu k perifériím. Speciální soubory (jsou obsahem adresáře /dev) většinou obyčejný uživatel nepoužívá, protože s nimi pracují démoni, které přebírají od uživatele požadavky ve formě příkazového řádku a přes speciální soubor periférii aktivují. Např.

**lp** **sched** je démon tisku na tiskárně. Uživatel zadává tisk pomocí příkazu **lp**, který kontaktuje démon. Správce systému pracuje se speciálními soubory velmi často, protože přes jejich obsah vidí hardware (otevření speciálního souboru znamená aktivaci ovladače periferie jádra). Připojení nového typu periferie souvisí s vytvořením dalšího speciálního souboru.

Jsou také další typy souborů. Obecně je navíc rezervována možnost definice dalších typů souborů. Nové typy souborů vznikaly (a mohou nadále vznikat) při realizaci nových prvků operačního systému. Např. při vzniku komunikace procesů pojmenovanou rourou (named pipe neboli FIFO, viz čl. 4.2) vznikl nový typ souboru, a sice typ roura. Postupně při popisu vlastností UNIXu, o které byl rozšířen v průběhu vývoje, budeme v dalším textu knihy objevovat takovéto nové typy souborů. V této kapitole se ale soustředíme zejména na popis ukládání dat v obyčejných souborech, jejich organizaci do adresářů a na principy přístupů k perifériím tak, jak je datová základna především uživatelem využívána a operačním systémem zpřístupňována.

*Svazek* (file system) je systémová jednotka implementace datové základny. Má svoji vnitřní organizaci, se kterou pracuje jádro, říkáme, že jádro používá algoritmy systému souborů a data tak zpřístupňuje. Základní princip organizace svazku bude popsán ve čl. 3.2. Popis bude výchozí, tj. vztahující se obecně na různé typy svazků. V průběhu vývoje UNIXu totiž došlo (a vývoj stále není u konce) ke vzniku efektivnějších typů svazků, než je kdysi zavedl UNIX 7th Edition (tzv. Version 7) nebo první verze UNIX SYSTEM V. Příklady implementace různých používaných typů svazků uvedeme. Současné systémy se snaží rozeznat navzájem různé typy svazků a respektovat jejich ukládání dat. Znamená to, že svazek je připojitelný ke stromu svazků operačního systému a uživateli k dispozici, přestože to není hlavní, výrobcem podporovaný typ svazku. Rozumná je jistě snaha ujednacení typů svazků mezi různými výrobci, ale vzhledem k zatím neuzavřenému vývoji je věcí budoucnosti. SVID stanoví pouze obecně termín typ svazku (type of file system), ale jeho organizaci ponechává nedefinovanou. Konečně je dnes zvykem ve výkonných systémech podpora svazků jiných operačních systémů (disketa vytvořená v prostředí MS DOS bude dostupná jako svazek). Takto mnozí výrobci řešili migraci dat svých vlastních operačních systémů po implementaci UNIXu na svých hardwarových platformách.

NFS (Network File System) je způsob zpřístupnění svazku z jiného počítače. Přestože se svazkům typu NFS nebo RFS (Remote File System) budeme věnovat u podpory sítí v kap. 6, je dobré si již nyní uvědomit princip distribuovaných systémů souborů. Počítač v síti oznámí export části svého stromu adresářů tak, že definuje jméno adresáře, kterým začíná takto exportovaná datová oblast. Jiný počítač sítě provede spojení některého svého adresáře s kořenovým adresářem datové oblasti exportované do sítě. Odkaz uživatele na místní adresář pak znamená přístup k datům vzdáleného počítače. Spojení je provedeno stejnými příkazy jako pro spojení místního adresáře a místního svazku, přestože exportovaná data nemusí začínat výchozím adresářem některého svazku. Rovněž současně dochází k exportu všech dalších svazků, které jsou ke svazku, na kterém je umístěn exportovaný adresář, připojeny.

Za zvláštní typ svazku je také považován svazek `/proc`. Jak už bylo řečeno v předchozí kapitole, je to zvláštní datová oblast (silně implementačně závislá), která je připojením datové základny k dynamicky se měnícím provozním datům jádra. Takovým připojením dostává proces možnost přístupu k systémovým datům standardními prostředky podpory přístupu k systému souborů. Uvedené příklady pojmenované roury, NFS a svazku `/proc` ukazují na obecnou filozofii získávání přístupu k datům operačního systému UNIX.

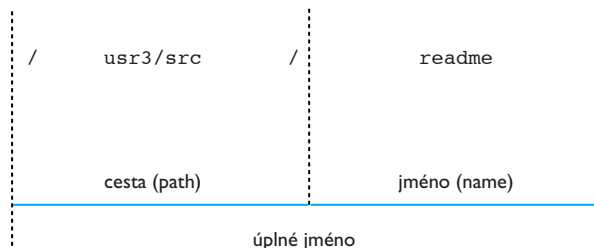
Všechny připojené svazky běžícího UNIXu z důvodu zrychlení přístupu k datům jsou zpřístupňovány pomocí *systémové vyrovnávací paměti* (buffer cache). Každý požadavek na přístup k datům připojených svazků na discích znamená, že data jsou z disku kopírována do operační paměti, odkud jsou teprve procesem čtena nebo kde jsou modifikována. Data všech připojených svazků jsou tedy procesům virtualizována přes definovanou oblast v operační paměti. Teprve zvláštní operací dochází k přepisu změných dat fyzicky na disky. Tato operace je volání jádra `sync`, které určitý systémový démon (jménem **update**, někdy **syncer**) provádí každých 30 vteřin. Každopádně je vyrovnávací paměť křehkým článkem zpracování dat v UNIXu v počítačích, které jsou zatíženy častým výpadkem zdroje elektrického napětí. Rovněž tak je důležité definovat její velikost odpovídající zatížení operačního systému, protože při její velikosti pod hranici kapacity současně zpracovávaných dat v operačním systému jádro musí neustále prohledávat data v ní uložená a hledat vhodného kandidáta k přepisu na disk. Současná velikost systémové vyrovnávací paměti je dána konfigurací jádra. Její vytížení (stejně jako vytížení dalších komponent operačního systému) je měřitelné např. prostředky **sa1**, **sa2**, **sadc** a **sar** (viz kap. 10).

## 3.1 Systém souborů z pohledu uživatele

Přístup uživatele k diskovým datům je přístup procesu, který pro uživatele pracuje. Soubor je uložen na disku v organizaci svazku, kde jsou evidovány také *atributy souboru*, tj. například jeho vlastník a přístupová práva, která byla souboru přidělena v okamžiku vzniku souboru. Proces je spuštěn uživatelem, je v jeho vlastnictví (i první proces uživatele sezení je v jeho vlastnictví). Podle označení uživatele je pak umožněn nebo odmítnut přístup procesu k datům souboru. Proces pro přístup k souboru používá *jméno souboru*. Vlastní jméno souboru je alfanumerický text. V některých systémech je omezen na délku 14 znaků, ve většině implementací na 255 znaků. Pro znak `.` (tečka) ve jménu souboru je vyhrazen jeden znak z rozsahu a je uvažován operačním systémem jako kterýkoliv jiný znak, přestože některé programy (mezi ně patří i shell) znak `.` ve jménu souboru respektují jako znak zvláštního významu. Stejně jako vlastní jméno souboru je důležitá i jeho *cesta* (path) k němu. Jde o posloupnost od kořenového adresáře až po adresář, ve kterém je soubor registrován. Textové označení souboru včetně takové posloupnosti adresářů se nazývá *úplné* (celé, absolutní) *jméno souboru*, např. (podle obr. 3.1) soubor s vlastním jménem `readme` v adresáři `src`, který je podadresářem `usr3` v kořenovém adresáři `/`, je odkazován celým jménem `/usr3/src/readme`. Uživatel (proces) přitom nemusí určovat svazky, napříč kterými je cesta k souboru vedena. Náš příklad je názorně uveden na obr. 3.2<sup>1</sup>.

Proces může soubor odkazovat úplným jménem, ale není to vždy nutné. Proces je totiž startován s určením jeho pracovního adresáře. Je to adresář, který má uživatel při práci v shellu nastaven vnitřním příkazem **cd**. Vytvořený proces pak může používat tzv. relativní jméno souboru, tj. jméno, které obsahuje seznam adresářů a vlastní jméno souboru. Např. je-li podle obr. 3.1 pracovní adresář procesu



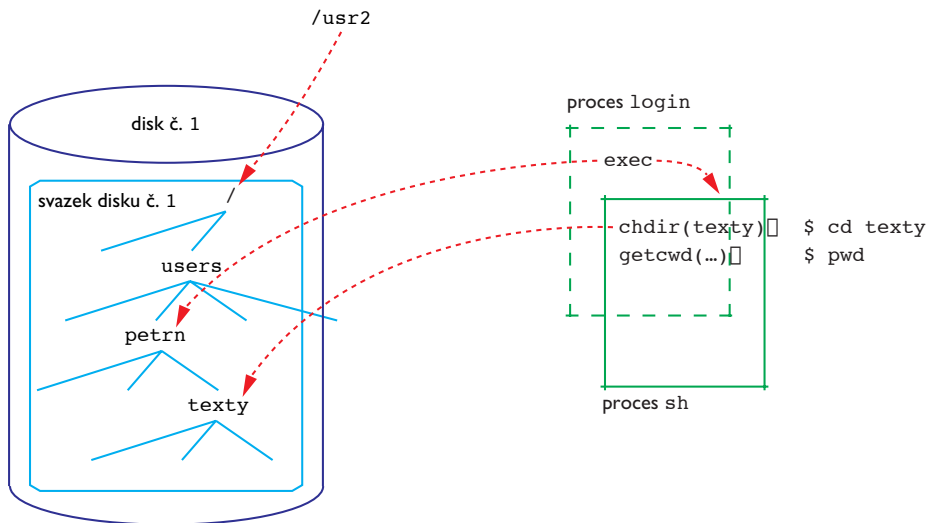


Obr. 3.2 Příklad jména souboru

/usr2/users/petrn a proces bude odkazovat na soubor vlastního jména `zivot.html`, který je uložen v podadresáři `texty`, může použít relativní jméno `texty/zivot.html` namísto úplného jména `/usr2/users/petrn/texty/zivot.html`. Relativní jméno souboru je tedy charakterizováno tak, že neobsahuje jako první znak `/`. Každý proces může měnit svůj pracovní adresář voláním jádra

```
int chdir(const char *path);
```

kde `path` je jméno adresáře, který je od okamžiku úspěšného provedení volání jádra nastaven jako pracovní. `path` přitom opět může být relativní nebo absolutní vzhledem k tomu, že adresář je zvláštní druh souboru.



Obr. 3.3 Shell uživatel v systému souborů

Volání jádra `chdir` používá shell při realizaci vnitřního příkazu **cd**. Výpis jména pracovního adresáře v shellu získá uživatel vnitřním příkazem **pwd**. Shell získá jméno svého pracovního adresáře voláním jádra

```
char *getcwd(char *buf, size_t size);
```

Po úspěšném volání je výsledek uložen v poli `buf`, na které také ukazuje návratová hodnota volání jádra. Parametr `size` je doplňující velikost znakového pole `buf`.

Výchozí pracovní adresář shellu po přihlášení uživatele je nastaven přihlašovacím procesem **login** podle obsahu souboru `/etc/passwd` (viz kap. 5), a to podle položky domovského adresáře. Domovský adresář každého uživatele vytváří správce systému při registraci uživatele. Je pak ve vlastnictví uživatele a začíná jím uživatelská datová oblast, kde může uživatel podle libosti vytvářet nové a rušit nebo měnit dříve vytvořené soubory (viz obr. 3.3).

Jak vyplývá z uvedeného, proces nemusí evidovat pracovní adresář a v přístupu k souboru používat úplná jména souborů. Úplné jméno souboru složí moduly jádra podpory systému souborů z pracovního adresáře a jména, které je použito v příslušném volání jádra.

## 3.1.1 Obyčejné soubory

*Obyčejný soubor* je termín pro označení souboru s daty. Data souboru jsou kódována v ASCII (American Standard Code for Information Interchange), tj. na 8 bitech jeden znak. Obyčejné soubory jsou často podle obsahu rozlišovány na textové (pouze znaky zobrazitelné na obrazovce terminálu nebo tiskárně) a binární (znaky celého rozsahu). Z pohledu jádra a systému souborů jde však vždy pouze o sekvenci bytů. Systém souborů nevytváří žádné kontrolní součty nebo řazení dat obyčejného souboru do záznamů se zvláštní strukturou. Pokud je soubor určitým způsobem strukturován, tuto strukturu spravuje vždy nadřazená aplikace vyšší vrstvy operačního systému (tj. procesy), ne jádro. I přesto je někdy používán termín záznam u textových souborů pro označení sledu bytů typicky zobrazovaných jako textový řádek souboru (data mezi začátkem souboru nebo znakem `\n` a koncem souboru nebo `\n`).

Proces otevírá obyčejný soubor voláním jádra

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
int open(const char *path, int oflag ... );
```

`path` je přitom jméno souboru vztažené k pracovnímu adresáři procesu (nebo absolutní cesta). `oflag` je způsob přístupu k souboru, může být použito `O_RDWR` pro čtení i zápis současně, pouze pro čtení `O_RDONLY` nebo pouze pro zápis `O_WRONLY`. Návratová hodnota v případě úspěchu je ukazatel do tabulky otevřených souborů procesu. Každý proces má po svém vzniku jako součást struktury *user* (viz odst. 2.4.4) přiřazenu tuto tabulku s názvem *deskriptory otevřených souborů*. Je to místo, odkud jádro odkazuje na atributy a obsah souboru otevřeného voláním jádra `open`. Z kap. 2 také víme, že každý proces po svém vzniku dědí tuto tabulku po svém rodiči. Dítě tedy může manipulovat s rodičem otevřenými soubory stejně jako rodič, přístup k otevřeným souborům může být proto vícenásobný. Víme také, že první tři položky této tabulky (položky 0, 1 a 2) slouží pro přístup k terminálu a jsou středem zájmu procesu shell, když vytváří uživatelem požadovanou rouru. Jádro v případě volání jádra `open`

prohledává tuto tabulku volajícího procesu a hledá první volnou položku. Je-li rodičem interaktivní uživatelův shell, bude první návratová hodnota pravděpodobně 3, protože proces dědí tabulku s prvními třemi položkami obsazenými terminálem (periferie je speciální soubor, viz čl. 3.3). Zkuste si příklad

```
main(void)
{
    printf("%d\n", open("/etc/passwd", O_RDONLY));
}
```

Uzavření souboru znamená pro uživatele ztrátu vazby tabulky deskriptorů souborů na soubor. Není-li soubor otevřen žádným dalším procesem, jsou uvolněny i ostatní struktury jádra. Pokud proces skončí a neuzavře otevřené soubory, jádro provede tuto činnost za něj. Proces uzavírá soubor voláním jádra

```
int close(int fildes);
```

kde `fildes` je položka tabulky otevřených souborů. Je to celočíselná hodnota, kterou proces získal z návratové hodnoty volání jádra `open`. Položka tabulky uvedená ve `fildes` je tak uvolněna a je použita v příštím `open`, pokud je v pořadí volných položek první.

```
main(void)
{
    close(1);
    printf("%d\n", open("/etc/passwd", O_RDONLY));
}
```

tiskne hodnotu 1. Vyzkoušejte...

Položku tabulky otevřených souborů (deskriptor souboru) proces používá ke čtení obsahu souboru nebo k zápisu nových dat do souboru. Používá k tomu volání jádra `read` a `write`. Jejich formát je velmi podobný:

```
ssize_t read(int fildes, void *buf, size_t nbyte);
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

`fildes` je deskriptor souboru. `buf` je ukazatel na datovou oblast procesu, jejíž obsah má být naplněn daty ze souboru (`read`) nebo zapsán do souboru (`write`). Parametrem `nbyte` zadává proces požadovaný počet přenášených bytů. V případě úspěchu je v návratové hodnotě jádrem uveden počet skutečně přenesených bytů. Proces tak např. u `read` může testovat konec souboru. Např.

```
main(int argc, char *argv[])
{
    int fd;
    unsigned char io_buf;
    if((fd=open(argv[1], O_RDONLY))==-1)exit(1);
    while(read(fd, &io_buf, 1)==1)printf("%c", io_buf);
    close(fd);
    exit(0);
}
```

je čtení dat souboru (jméno souboru je první parametr příkazového řádku programu) po znacích a jejich tisk na obrazovku terminálu. Ve chvíli, kdy není přenesen požadovaný jeden byte, je čtení ukončeno,

soubor je uzavřen a proces končí. Příklad volání jádra `write` bude analogický, běžně ale nedochází k rozdílu mezi návratovou hodnotou a `nbyte`.

Uvedený příklad je velmi neefektivní. Pro každý znak souboru proces žádá jádro o přenos dat. Ušetří tak sice prostor datového segmentu, ale rychlost zpracování dat procesem se výrazně sníží, protože proces bude zatížen častou režíř jádra. Je proto lépe definovat datovou oblast pro přenos dat na více znaků a ve volání jádra požadovat jejich přenos současně. Naopak zde zase čerpáme více prostoru z datového segmentu, ale práce procesu se zrychlí. Rozumným kompromisem mezi frekvencí použití volání jádra a počtem přenášených slabik současně je velikost datového bloku disku, protože operační systém pracuje se svazky po těchto blocích (viz vyrovnávací paměť zmíněná v úvodu kapitoly). Vzhledem k tomu, že programátor by měl být od implementace oddělen, standardní knihovna jazyka C mu poskytuje sadu funkcí, které manipulují se strukturou typu `FILE` (je definována v `<stdio.h>`), která obsahuje deskriptor souboru, oblast pro přenos dat mezi souborem a datovým segmentem procesu a další potřebné položky pro správu přenášených dat. Programátor pak používá namísto `open` a `close` funkce `fopen` a `fclose`, které manipulují s ukazatelem na `FILE`, namísto `read` a `write` funkce `getc`, `putc`, `fscanf`, `fprintf` a mnohé další, které zpřístupňují data z oblasti struktury `FILE`. Komentovaná skupina funkcí je označována v popisu jazyka C jako *v/v proudy* (i/o streams) a jejich seznam nalezne čtenář v provozní dokumentaci svazku s označením (3). Dnes jsou již standardizovány v POSIXu.

Proces zjišťuje možnost přístupu k souboru voláním jádra `access`. Jeho formát jsme uvedli v čl. 2.2, kde jsme také diskutovali jeho výsledky podle reálného a efektivního vlastníka procesu. Z pohledu dat svazku souvisí `access` s vytvářením souborů. Při vytváření jsou atributy každého souboru také jeho vlastník, skupina a přístupová práva, která jsou k vlastníkovi a skupině vztažena. Vznik nového souboru iniciuje vždy proces. Vlastníkem takového souboru je uživatel, který je evidován jako efektivní vlastník procesu. Totéž platí pro skupinu. Vlastníka i skupinu může proces s efektivním vlastníkem shodným s vlastníkem souboru měnit voláním jádra

```
#include <sys/types.h>
```

```
int chown(const char *path, uid_t owner, gid_t group);
```

kde hodnoty `owner` a `group` jsou číselné vyjádření nového vlastníka a skupiny souboru. Unikátní číselná registrace uživatelů je vedena v souboru `/etc/passwd` a skupin v `/etc/group`. V shellu je změna vlastníka prováděna příkazem **chown** a skupiny **chgrp**. Oba programy využívají uvedeného volání jádra.

Přístupová práva vznikajícího souboru jsou stanovena podle nastavené masky přístupových práv a bitů přístupových práv ve volání jádra `open` nebo `creat`.

Voláním jádra `creat` vzniká soubor.

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
#include <fcntl.h>
```

```
int creat(const char *path, mode_t mode);
```

Soubor ale také vzniká voláním jádra `open`. Na místě tří teček ve výše uvedeném formátu může být použit parametr shodný s `mode` v `creat`. Na místě `oflag` musí pak být použita hodnota `O_CREAT`. Proto použití

```
creat(path, mode);
```

je shodné s

```
open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

Hodnota `O_TRUNC` je používána vždy tehdy, když soubor otevíráme, ale zkracujeme jeho obsah na nulovou délku. V případě, že soubor jména použitého v `open` dosud neexistuje a v parametru `oflag` není použito `O_CREAT`, volání jádra bude neúspěšné. Programátor proto musí testovat existenci a přístup k souboru před jeho použitím nebo vytvořením. K tomu mu slouží kombinace `O_CREAT|O_EXCL`. Při použití knihovny v/v proudů jazyka C je ale použití funkce `fopen` v tomto smyslu komfortnější.

Přístupová práva vytvářeného souboru jsou dána argumentem `mode`. Každý soubor svazku má jako jeden ze svých atributů slovo přístupových práv. Povolené hodnoty bitů slova přístupových práv jsou podle SVID a POSIX definovány

<code>S_IRWXU</code>	je povolení současně čtení, zápisu, prohledávání (u adresáře) nebo provádění (u ostatních typů souborů), a to pro vlastníka souboru; přitom lze použít pouze <code>S_IRUSR</code> pouze pro bit čtení,
<code>S_IWUSR</code>	pouze pro zápis a <code>S_IXUSR</code> pouze pro procházení adresářem nebo proveditelnost,
<code>S_IRWXG</code>	pro povolení čtení, zápisu i prohledávání nebo proveditelnosti pro skupinu souboru, analogicky je bit pro čtení <code>S_IRGRP</code> , pro zápis <code>S_IWGRP</code> a proveditelnost nebo průchod adresářem <code>S_IXGRP</code> ,
<code>S_IRWXO</code>	jsou všechny bity přístupu pro ostatní, analogicky je <code>S_IROTH</code> čtení, <code>S_IWOTH</code> zápis a <code>S_IXOTH</code> proveditelnost,
<code>S_ISUID</code>	je s-bit pro uživatele a
<code>S_ISGID</code>	je s-bit pro skupinu.

SVID dále definuje také bit `S_ISVTX` s označením rezervy.

*Slovo přístupových práv* (file access mode word) vzniká při vytváření souboru podle argumentu `mode` (jádro přitom ještě respektuje nastavenou masku přístupových práv vytvářených souborů procesu, její binární doplněk je binárně vynásoben s `mode`, znamená to, že maskou označené bity se v slově přístupových práv při vytvoření nikdy nenastaví). Proces dědí masku přístupových práv od svého rodiče a zjišťuje nebo mění její nastavení voláním jádra `umask` (formát viz čl. 2.2). Slovo přístupových práv souboru může být později změněno voláním jádra `chmod` (formát viz opět čl. 2.2). Na úrovni příkazů některého z shellů je používán příkaz **chmod**. Běžné nastavení masky v shellu (zjistitelné vnitřním příkazem **umask**) je `S_IXUSR|S_IWGRP|S_IXGRP|S_IWOTH|S_IXOTH` (zakázaný zápis a proveditelnost pro skupinu a ostatní a zakázaná proveditelnost pro vlastníka). Použijeme-li proto při `creat` kombinaci

```
close(creat("jentak", S_IRWXU|S_IRWXG));
```

dostáváme v pracovním adresáři nově vzniklý soubor, jehož výpis je

```
$ ls -l jentak
```

```
-rw-r----- 1 sko users 0 Nov 14 10:57 jentak
```

a teprve dalším voláním jádra můžeme přístupová práva souboru pomocí `chmod` změnit.

Volání jádra `access` je test přístupu procesu k souboru podle efektivního vlastníka a skupiny procesu.

Proces může také pomocí `access` testovat pouze existenci souboru, např. fragment

```
if(access("jentak", F_OK)==0)printf("Jsme\n");  
else printf("Nejsme\n");
```

tiskne text `Jsme`, pokud soubor jména `jentak` v pracovním adresáři procesu existuje, a `Nejsme`, pokud neexistuje.

Požaduje-li proces informace atributů souboru tak, jak jsou uloženy ve svazku, může k tomu použít volání jádra `stat`. Získá tak všechny důležité atributy souboru. Volání jádra má formát

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
int stat(const char *path, struct stat *buf);
```

pro identifikaci souboru podle jeho jména `path`. Pro otevřený soubor je volání jádra

```
int fstat(int fildes, struct stat *buf);
```

kde ve `fildes` uvádíme odkaz do tabulky otevřených souborů. Obě volání jádra v úspěšném provedení naplní položky struktury `buf` informacemi atributů požadovaného souboru. `stat` získává hodnoty všech položek struktury `buf` z části svazku, které se říká *oblast i-uzlů*. Každý soubor svazku jakéhokoliv typu má přiřazen *i-uzel* (i-node). Reference jménem souboru znamená odkaz na jeden i-uzel. Písmeno *i* je odvozeno od slova *index*, protože odkaz (číslo i-uzlu) je opravdu indexem do této oblasti. Spojení jména a čísla i-uzlu souboru je provedeno v adresáři. Odkaz do oblasti i-uzlů je unikátní, ale pouze v rámci jednoho svazku. Položky struktury `buf` (je definována ve `<sys/stat.h>`) jako struktura `stat`) jsou

```
mode_t st_mode; /* typ souboru a slovo přístupových práv */
```

```
ino_t st_ino; /* číslo i-uzlu */
```

```
dev_t st_dev; /* identifikace svazku */
```

```
dev_t st_rdev; /* doplňující (raw) identifikace svazku,  
pouze SVID */
```

```
nlink_t st_nlink; /* počet odkazů na i-uzel */
```

```
uid_t st_uid; /* vlastník */
```

```
gid_t st_gid; /* skupina */
```

```
off_t st_size; /* velikost souboru v bytech */
```

```
time_t st_atime; /* datum a čas posledního přístupu */
```

```
time_t st_mtime; /* datum a čas poslední modifikace */
```

```
time_t st_ctime; /* datum a čas změny statutu souboru */
```

```
long st_blksize; /* velikost bloku, pouze SVID */
```

```
long st_blocks; /* počet bloků velikosti st_blksize, pouze SVID */
```

Je zřejmé, že např. program `ls` s volbou `-l` získává informace o souborech požadovaného adresáře právě pomocí `stat`. Ostatní činnost je formátování obsahu struktury `stat` na standardní výstup. Pro

rozlišení typů souborů (položka `st_mode`) může proces používat makra. Např. `S_ISDIR(m)` vrací nenulovou hodnotu, pokud `m` typu položky `st_mode` je adresářem, `S_ISREG(m)` je testem `st_mode` na obyčejný soubor atd. (viz provozní dokumentace k obsahu souboru `<sys/stat.h>`). `st_mode` také současně obsahuje slovo přístupových práv, jak bylo popsáno u volání jádra `open` a `creat`.

### 3.1.2 Adresáře

*Adresář* je soubor, který má v označení typu souboru `st_mode` definici adresáře. Znamená to, že adresář má všechny atributy souboru a má také datovou část. Datová část adresáře je seznam dvojic jmen souborů a čísel i-uzlů. V souboru `<dirent.h>` je uvedena definice položky adresáře. Jde o definici struktury `dirent`, která má položky

```
ino_t d_ino;                /* číslo i-uzlu */
char d_name[NAME_MAX];     /* jméno souboru */
```

přítom číselný typ i-uzlu `d_ino` je definován v `<sys/types>`. Délka jména souboru `NAME_MAX` je systémový parametr a bývá u moderních systémů běžně 255, což je největší možná délka jména souboru. Je-li kratší, je jméno souboru v položce `d_name` ukončeno znakem binární nuly (NULL). Data souboru typu adresář jsou čitelná moduly jádra. Uživatel má možnost v některých typech svazků obsah adresáře otevřít a číst jako soubor a použít přitom odpovídající strukturu `DIR`, nebo lze v shellu použít příkaz

#### \$ od adresář

což je osmičkový výpis obsahu souboru (zde typu adresář). Nové typy svazků ale neumožňují běžnými metodami přístupu k souboru s adresářem manipulovat a operační systém nabízí skupinu funkcí pro čtení adresáře

```
#include <sys/types>
#include <dirent.h>

DIR *opendir(const char *dirname);
struct dirent *readdir(DIR *dirp);
void rewinddir(DIR *dirp);
int closedir(DIR *dirp);
```

Typ `DIR` je definován v `<dirent.h>` pro přístup k adresáři jako k proudu položek (tzv. directory stream). Použití funkcí je mnemonické. Při otevření adresáře pomocí `opendir` je ukazatel čtení pomocí `readdir` nastaven na první položku adresáře. Ukazatel je přesunut vždy na následující položku adresáře použitím `readdir`. `rewinddir` nastaví ukazatel na první položku. Obsah adresáře (proud jeho položek) je rozšiřován o novou položku při vytváření souboru. Je-li adresář nově vytvořen, připojí novou položku k obsahu adresáře. Položky nejsou obecně nijak tříděny, `readdir` proto čte vždy fyzicky následující položku adresáře. Je obvyklé, že svazky jsou implementovány tak, že naopak při rušení položky adresáře (rušení souboru) se provede tzv. vynulování položky, což je přepis čísla i-uzlu (`d_ino`) na 0. Při zavádění nové položky adresáře je tedy prohledáván celý adresář a jako první v pořadí je alokována taková dříve uvolněná položka (tzv. prázdná položka, empty directory entry, podle POSIXu empty name). `readdir` proto čte následující neprázdnou položku adresáře. Výpis datové části obsahu pracovního adresáře proto můžeme zajistit např. programem

```
#include <sys/types.h>
#include <dirent.h>

main(void)
{
    DIR dirp;
    struct dirent d, *dp;
    dp=&d;
    dirp=opendir(".");
    while((dp=readdir(dirp))!=NULL)
    {
        printf("%ld %s\n", dp->d_ino, dp->d_name);
    }
    closedir(dirp);
}
```

za předpokladu, že `d_ino` je typu `long`.

Adresář vzniká voláním jádra

```
#include <sys/types.h>
#include <sys/stat.h>

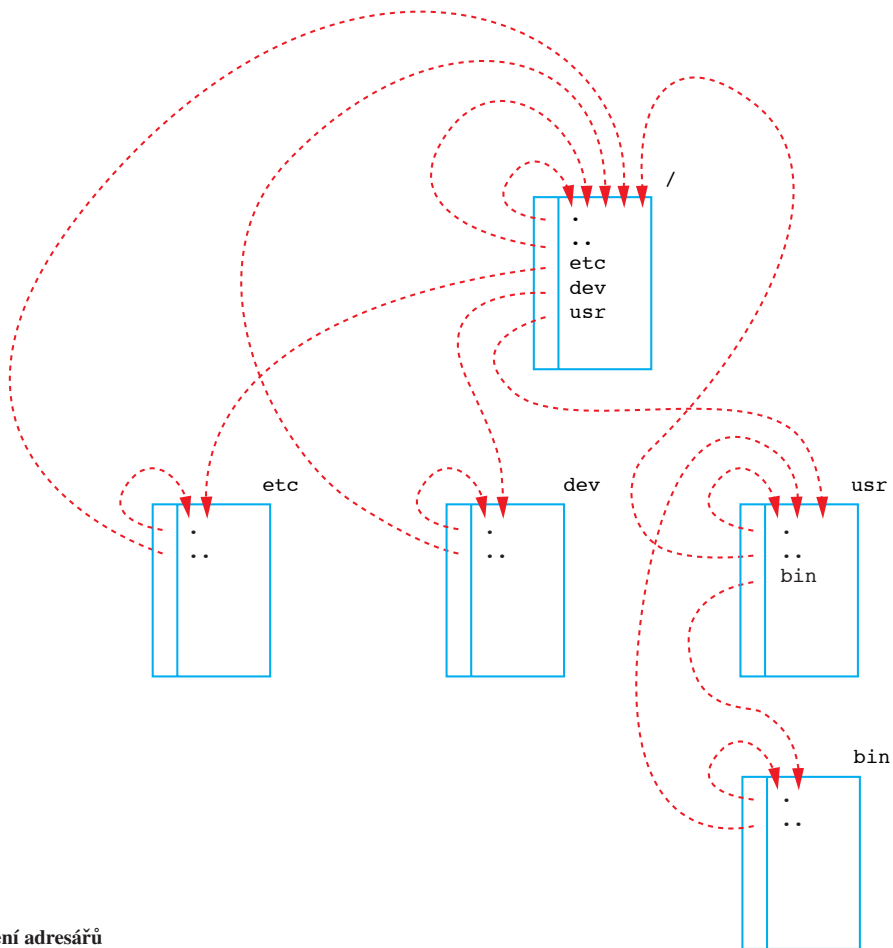
int mkdir(const char *path, mode_t mode);
```

kde `path` je položka `d_name` struktury `dirent` adresáře, ve kterém nový adresář vzniká (jméno vznikajícího adresáře), protože i zde jde o založení nové položky adresáře. `mode` jsou přístupová práva adresáře, která jsou uložena v nově alokovaném i-uzlu označeného v `d_ino` (měnitelná rovněž pomocí `chmod`).

Při vzniku adresáře jádro v jeho datové části zajistí existenci dvou položek. Jejich jméno je `.` (tečka, dot) a `..` (tečka-tečka, dot-dot). Jádro přidělí těmto jménům i-uzly a jejich čísla vloží do `d_ino` jejich položek adresáře. Položka se jménem tečka-tečka má přiděleno číslo i-uzlu adresáře, ve kterém je nově vzniklý adresář evidován (nadřazený adresář, rodičovský, parent directory), jméno tečka má číslo i-uzlu adresáře nově vzniklého (sama sebe). Tímto mechanismem jádro zajišťuje konzistenci propojení adresářů mezi sebou do stromové struktury. Pouze jádro má právo zápisu do datové části adresáře. Procesy ke změnám v adresáři používají uváděná volání jádra. Situaci ukazuje obr. 3.4.<sup>3</sup>

Podle obrázku a mechanismu vytváření a údržby stromové struktury adresářů vidíme, že některé i-uzly jsou vícenásobně odkazovány, tj. tatáž čísla i-uzlů jsou použita u různých jmen souborů. Např. i-uzel adresáře `/usr` na obr. 3.4 je odkazován celkem třikrát, a to z adresáře `/`, adresáře `/usr/bin` (jako na svůj nadřazený adresář u jména tečka-tečka) a z adresáře `/usr` (jako odkaz sám na sebe u jména tečka). Adresář `/` má na obr. 3.4 dokonce odkazů 5 (víte proč?). Pro kontrolu je proto obsahem každého i-uzlu i počet odkazů (number of links) na tento i-uzel. UNIX dále přitom poskytuje volání jádra `link`, které umožňuje uživateli vytvářet další odkazy na již existující položku adresáře. Neprivilegovaný uživatel má ovšem možnost použít toto volání jádra pouze na jiné soubory než adresáře (pochopitelně je to věc efektivního vlastníka procesu), a to z důvodů zajištění další konzistence stromu adresářů. Obyčejný uživatel může proto použít `link` (nebo příkaz `ln` v shellu) např. na obyčejné soubory a mít tak tentýž soubor odkazován pod různými jmény.





Obr. 3.4 Propojení adresářů

```
int link(const char *existing, const char *new);
```

je formát volání jádra. Jádro pro existující soubor `existing` vytvoří nový odkaz pod jménem `new`. Je lhostejné, je-li soubor `new` vytvořen jako položka téhož nebo jiného adresáře. Jádro provádí vytvoření nové položky v cílovém adresáři s číslem i-uzlu stejným, jako je u `existing`.

Každý soubor (jakéhokoliv typu) je tedy evidován jako odkaz na i-uzel v některém adresáři. Proto při rušení souboru říkáme, že rušíme tento odkaz na i-uzel. Položka adresáře zaniká voláním jádra

```
int unlink(const char *path);
```

V parametru `path` zadáváme jméno souboru. Jádro požadovanou činnost provede tak, že změní číslo i-uzlu u položky adresáře odpovídajícího jména souboru na 0. Vzniká tak prázdná položka adresáře

a zaniká odkaz na i-uzel (index č. 0 není v oblasti i-uzlů obsazen). V dalším kroku jádro sníží hodnotu počtu odkazů doposud odkazovaného i-uzlu o 1 a testuje, zda počet odkazů není roven 0. Je-li tomu tak, i-uzel již není odkazován z žádného jiného adresáře a jádro i-uzel i datovou část souboru uvolní pro další použití. Je-li položka adresáře adresářem, jádro musí vykonat změnu i v jiných i-uzlech, než je ten, na který je odkazováno z rušené položky, proto je pro prázdný adresář (obsahující pouze neprázdné položky tečka a tečka-tečka) používáno volání jádra

```
int rmdir(const char *path);
```

Jakýkoliv jiný typ souboru vyjma adresáře rušíme pomocí `unlink`.

Uživatel v shellu má možnost použít k rušení adresáře příkaz **rmdir**, každý jiný soubor ruší (má-li oprávnění zápisu) příkazem **rm**. Zrušit celý podstrom uživatel dokáže příkazem **rm -r**, kdy příkaz používá volání jádra `unlink` pro všechny položky adresáře, které nejsou podadresářem, a po vyprázdnění adresáře použije `rmdir`.

Přejmenování souboru v UNIXu znamená vytvořit nový odkaz na tentýž i-uzel pomocí `link` a starý zrušit pomocí `unlink`. UNIX poskytuje příkaz **mv** uživatele shellu, který je v klasickém UNIXu takto implementován. POSIX ovšem (z pochopitelných důvodů neznalosti pojmu i-uzel a jeho odkazů) zavádí volání jádra, které definuje i SVID a pro kompatibilitu a skrytí atomické operace do jádra implementuje každý dnešní UNIX. Volání jádra

```
int rename(const char *old, const char *new);
```

přejmenuje soubor se jménem `old` na `new`.

### 3.1.3 Symbolické odkazy

Uvedený způsob vícenásobných odkazů na soubor nebo adresář je v UNIXu možný v rámci jednoho svazku. Pokusí-li se proces vytvořit odkaz na tentýž i-uzel z adresáře na jiném svazku, dojde k chybě. I-uzel, jak již bylo řečeno, je odkazován svým číslem, a to je pořadí v oblasti všech i-uzlů na jednom svazku. Odkaz na i-uzel na jiném svazku téhož čísla znamená odkaz na jiný i-uzel. Vzhledem k tomu, že tak vzniká nepřijemné omezení, byl v UNIXu zaveden nový typ souboru, symbolický odkaz (symbolic link)<sup>4</sup>.

*Symbolický odkaz* (symbolic link) je označení typu souboru. Při vzniku symbolického odkazu je tedy alokován nový i-uzel. Jeho typ je nastaven na symbolický odkaz a v datové části je uložena cesta a jméno souboru, který je takto odkazován. Symbolický odkaz vzniká na tomtéž nebo jiném svazku, než je odkazovaný soubor (pozor na jinou topologii připojení svazků). Přestože uživatel v shellu používá tentýž příkaz pouze s novou volbou (**ln -s**), samotný proces pracuje se symbolickým odkazem pomocí nových volání jádra. Symbolický odkaz vzniká pomocí

```
int symlink(const char *existing, const char *new);
```

kde parametry mají tentýž význam jako u volání `link`. Obsah symbolického odkazu čteme zvláštním voláním jádra

```
int readlink(const char *path, char *buf, int bufsiz);
```

Cesta se jménem odkazovaného souboru je uložena do pole `buf`, jehož velikost zadáváme v `bufsiz`. Zrušení symbolického odkazu je použitím `unlink` (v shellu **rm**), protože jde o zrušení odkazu na i-uzel

z adresáře. Zánikem datové části i-uzlu typu symbolického odkazu zaniká i odkaz na jiný i-uzel, třeba i na jiném svazku. Vyzkoušejte, prozkoumejte, dávejte pozor na přístupová práva.

Podle výše uvedeného znamená v UNIXu vznik souboru vytvoření položky v adresáři a novou alokaci i-uzlu. Naplnění obsahu i-uzlu určuje typ souboru, přístupová práva a ostatní atributy důležité pro manipulaci se souborem. Obsah alokovaného i-uzlu závisí na použitém volání jádra, tj. `creat`, `open`, `mknod` nebo `symlink`. Privilegovaný proces může také použít volání jádra `mknod`, které nově alokuje i-uzel na svazku. Je definováno v SVID, ale ne v POSIXu. `mknod`, jak už jeho název napovídá, pouze vytváří nový i-uzel zadaného typu a odkaz na něj (jméno souboru). Přestože je v i-uzlu stanoven typ souboru, jádro neprovede další akce, které s vytvořením souboru odpovídajícího typu mnohdy úzce souvisí a správnou manipulaci se souborem podmiňují (např. při vytvoření i-uzlu adresáře je ještě nutné v datové části vytvořit položky tečka a tečka-tečka). Pomocí `mknod` vznikají např. speciální soubory, kterým je věnován samostatný článek 3.3, formát volání jádra a jeho popis uvádíme v odst. 3.2.1, kde i podrobně analyzujeme obsah i-uzlů.

### 3.1.4 Zamykání souborů

Uvedli jsme, že každý soubor je vícenásobně přístupný, čili je možné jej otevřít a pracovat s ním současně z několika různých procesů (rodič a dítě je běžný případ). Vzhledem k tomu, že každé volání jádra je operace atomická, nemůže se stát, aby dva procesy zapisovaly data do jednoho souboru současně, ale z praktických důvodů (proces může pro zápis dat jdoucích za sebou použít více volání jádra) by měly být operace nad obsahem souboru řízeny nějakým obecně dostupným mechanismem, tj. práce procesů s daty by měla být synchronní. Práci více procesů s obsahem souboru je možné synchronizovat pomocí některé formy komunikace mezi procesy (např. signály nebo semafore, viz kap.4), nejvíce bezpečná a triviální je ale varianta zamykání samotných souborů nebo jejich částí (tzv. segmentů).

Proces může vytvořit zámek nad souborem, který dříve otevřel. Znamená to, že zámek nevzniká na samotném svazku, ale v datových strukturách jádra, kde je zámek registrován (viz obr. 3.5). Zámek je přiřazen k souboru, který může mít otevřeno více procesů, ale zámek je registrován na jeden proces, který když zanikne, zanikají i všechny jeho zámky.

Volání jádra `fcntl` je určeno nejenom pro zamykání souborů (dokumentace hovoří obecně o řídicích operacích nad souborem). Má formát

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fildes, int cmd, ...);
```

`fildes` je deskriptor otevřeného souboru, `cmd` je činnost nad otevřeným souborem. Při zamykání použijeme na místě `cmd` jednu z možností:

- `F_GETLK`      získání informací o prvním nalezeném zámku nad souborem,
- `F_SETLK`      vytvoření zámku, pokud zámek nelze vytvořit, protože existuje jiný, `fcntl` vrací řízení procesu,

F\_SETLKW totéž co F\_SETLK, jádro ale blokuje proces do okamžiku, kdy je předchozí zámek zrušen.

Jako parametr (na místě tří teček ve formátu) používáme pak odkaz na strukturu flock, která má položky

```
short l_type;      /* jedna z možností F_RDLCK (zámek pro čtení),
                   F_WRLCK (zámek pro zápis), F_UNLCK (zrušení zámku)
                   */
short l_whence;    /* segment je stanoven
                   SEEK_SET od začátku souboru,
                   SEEK_CUR od současné pozice,
                   SEEK_END od konce souboru */
off_t l_start;     /* začátek segmentu v počtu bytů od l_whence */
off_t l_len;       /* délka segmentu, 0 znamená do konce souboru */
pid_t l_pid;       /* PID vlastníka zámku – je naplněna při F_GETLK */
```

Rozsah zámků pro čtení se může překrývat, pro zápis nikoliv. Fragment programu, kdy zamykáme celý obsah souboru s deskriptorem fd, je pro čtení i zápis:

```
int fd;
struct flock l;
...
l.l_type=(F_RDLCK | F_WRLCK);
l.l_whence=l.l_start=l._len=0;
fcntl(fd, F_SETLKW, &l);
...
```

## 3.1.5 Svazky

Obvyčejný uživatel se nemusí o topologii připojených svazků příliš starat. Jednak ji nedokáže ovlivnit a jednak uvedené mechanismy nerozlišují fyzické umístění souborů na svazcích. Změna adresáře nebo kopie dat napříč svazky je transparentní. Situace, ve které bude donucen se rozložením stromu adresářů do svazků zabývat, je použití přejmenování souborů nebo vytvoření dalšího obvyčejného odkazu na soubor, je-li nový odkaz založen na jiném svazku. Je použitým voláním jádra nebo následně příkazem upozorněn na přítomnost více svazků a musí použít pro řešení např. symbolické odkazy (přestože uživatelovu oblast dat by měl správce systému plánovat tak, aby k podobné situaci nedocházelo – víte jak?). Aby se obvyčejný uživatel měl možnost s topologií svazků seznámit, může používat informativní část příkazu **mount** nebo příkaz **df**. Oba příkazy ve svém výpisu seznamují s připojením svazků na kořenový svazek, **df** navíc informuje o dosud neobsazené kapacitě každého svazku. Např.

```
$ /etc/mount
/dev/root on /
/proc on /proc
/dev/dsk/1s7 on /usr2
/dev/dsk/2s6 on /usr3
/dev/dsk/3s7 on /usr3/src
```

\$ df

Filesystem	Type	blocks	use	avail	%use	Mounted on
/dev/root	efs	3830376	2551885	1278491	67%	/
/dev/dsk/1s7	efs	3686020	1449885	2236135	39%	/usr2
/dev/dsk/2s6	efs	1623452	914158	709294	56%	/usr3
/dev/dsk/3s7	efs	3826812	836871	2989941	22%	/usr3/src

V příkladu používá obyčejný uživatel celou cestu k příkazu **mount**, protože není zvykem nastavovat v proměnné prostředí **PATH** adresář **/etc** pro obyčejného uživatele. Výpis obou příkazů navazuje na obr. 3.1 z úvodu kapitoly. Vždy jeden řádek výpisu popisuje jeden připojený svazek. První informace na řádku je jméno speciálního souboru svazku (viz 3.3). Je to označení části disku, na které je umístěn svazek, a obyčejného uživatele nemusí zajímat. Důležitá je poslední informace na řádku. V obou příkazech je to adresář, který je s odpovídajícím svazkem spojen, takže uživatel rozezná hranice jednotlivých svazků. Ostatní informace výpisu příkazu **df** informují o typu svazku, o jeho velikosti v diskových blocích a počtu použitých a volných bloků každého svazku.

Uvedené příkazy využívají volání jádra **mount** a **ustat**, která jsou sice uvedena v SVID, ale ne v POSIXu. Obě volání jádra mají systémový charakter a prozkoumáme je v následujícím článku.

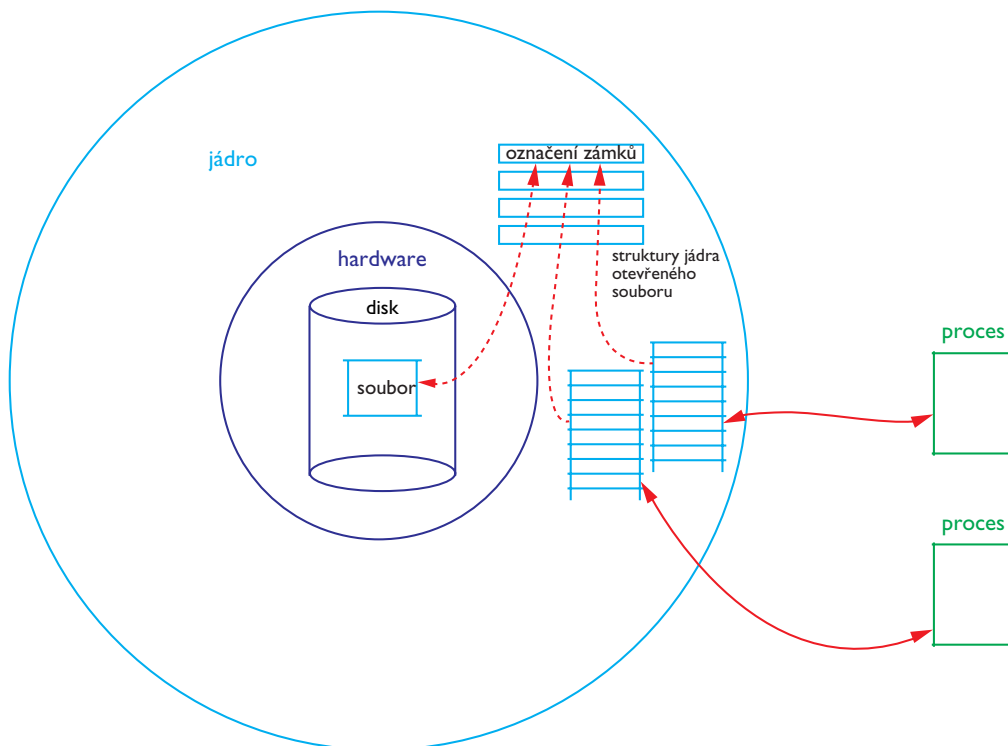
## 3.2 Systém souborů z pohledu jádra

Jádro poskytuje procesům služby uvedené v předchozím článku za podpory modulů práce systému souborů. Tato část jádra pracuje se svazky, což je organizace dat disků. Disk (obr. 3.7) je blokové zařízení, které je jádrem viditelné jako magnetická plocha, rozdělená do adresovatelných částí tak, že čas přístupu ke každé z nich je dán časovým intervalem s definovanou odchylkou, která dosahuje pouze díl časového intervalu. Technologie takového přístupu je dosažena pomocí přístupu několika hlav současně k otáččenému systému ploch použitého média nad sebou. Snímání a zápis informací na médium je organizováno pomocí řadiče (controller), který skládá nebo distribuuje požadovanou informaci na jednotlivé hlavy. Tento zatím nejvýkonnější známý způsob ukládání dat na externím médiu je dnes různým způsobem modifikován a kombinován pro dosažení maximální přenosové rychlosti pro spojení s přímou (operační) pamětí, kde je prováděn výpočet. Adresovatelné části disku jsou označovány jako sektory nebo fyzické bloky. Jejich zpřístupnění modulům systému souborů zajišťuje ovladač (driver), který je určen způsobem přístupu k disku (tj. typem řadiče). Moduly systému souborů tak mohou fyzické bloky disku organizovat i do větších částí, které nazýváme diskové bloky operačního systému (v dalším textu pouze diskové bloky). Fyzické bloky jsou číslovány od 0, nejvyšší možné číslo bloku vynásobené velikostí bloku je kapacita disku. Disk je v UNIXu ovladačem poskytován jako několik sekcí, které může jádro využít. *Sekce* jsou části disku označené počátečním a koncovým fyzickým blokem, do sekce patří všechny fyzické bloky s označením v intervalu mezi nimi. Sekce se mohou překrývat, protože můžeme požadovat rozdělení disku na několik částí (tj. na několik svazků nebo na svazek a odkládací oblast atd.), anebo můžeme požadovat práci s celým diskem najednou. Např. můžeme mít rozdělen disk na celkem 7 sekcí označených **a - g** (příklad je převzat ze systému BSD), jak je dále uvedeno. Rozdělení disku na sekce vychází z pojmu cylindr. Je to část disku, která je dostupná všemi hlavami při vystavení raménka na určitou pozici (stopu), a to při provedení jedné otáčky disku

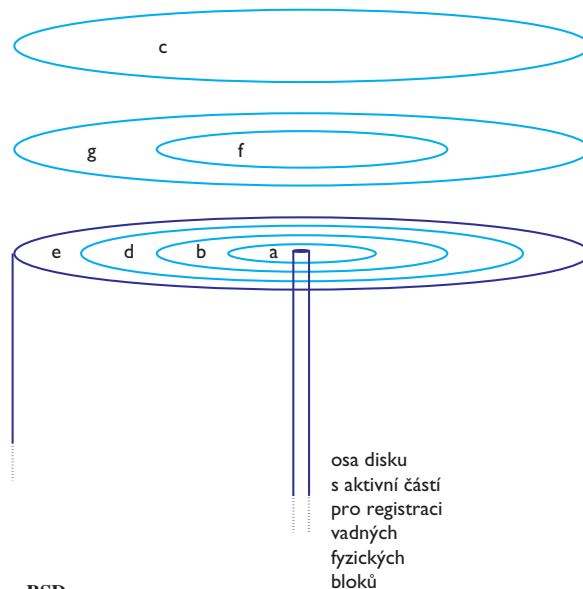
(počet cylindrů je tedy roven počtu stop). Je proto efektivní využívat cylindry stop jdoucí za sebou. Umístění sekcí a - g může proto být zobrazeno podle obr. 3.6.

Správce v tomto případě může použít celý disk sekcí c, může disk mít rozdělen na dvě sekce f a g nebo může používat sekce a, b, d, e, ale také a, b, g nebo e, d, f atp. Z takové úvahy vychází způsob rozdělování disků na sekce v současných systémech. Na rozdíl od BSD systémů bývá obvykle stanoveno číselné označování sekcí (od 0 výše). Např. IRIX používá celkem 16 sekcí: disjunktní sekce jsou 0, 1 a 6, sekce 7 je využití celého dostupného prostoru, 8 je záhlaví disku, ostatní sekce jsou určeny pro vnitřní potřebu systému souborů (např. pro záznam vadných fyzických bloků disku). V BSD je definice nového způsobu rozdělení disku do sekcí (rozsah platnosti sekcí) dáno popisem v ovladači a regeneraci jádra. IRIX podobně jako ostatní průmyslové systémy používá systémový program (v IRIXu **fx**), který zapisuje informace o rozdělení na jinak nedostupné místo na disku.

Svazek je organizován v určité sekci (viz také obr. 3.1). Správce systému jej vytváří pomocí příkazu **mkfs** tak, že zadává označení sekce disku (sekce jako celek je přístupná pomocí speciálního souboru) a velikost vytvářeného svazku v diskových blocích. Velikost diskového bloku je dána typem svazku.



Obr. 3.5 Soubory zamykané procesy



Obr. 3.6 Příklad sekcí disku v BSD

Přestože je velikost sekce disku dána tabulkou uloženou na disku (dříve v záhlaví ovladače disku) v počtu fyzických bloků, správce systému může dále použít pouze část sekce. Tato flexibilita se dnes už využívá málo. Pochází z raných verzí systému, ve kterých ovladač poskytoval disk jako jednu sekci a jeho rozdělení na části zadával správce systému pomocí dále strukturovaných speciálních souborů. Dnes je používán speciální soubor pro označení jedné sekce dané délky (více viz čl. 3.3). Např.

```
# mkfs /dev/dsk/1s7 2000000
```

je vytvoření svazku v 7. sekci druhého disku (disky počítáme 0, 1, 2 atd.) o velikosti 200000 diskových bloků, tedy ne fyzických bloků. S fyzickými bloky pracuje operační systém na úrovni ovladače disku a velikosti sekcí, ale při organizaci sekce disku do svazku pomocí **mkfs** již pracujeme na úrovni logických bloků. Velikost logického diskového bloku je přitom dána typem svazku. Je tedy zřejmé, že při vytváření svazku v sekci musí správce systému znát velikost diskového bloku vzhledem k fyzickému, a to z provozní dokumentace. Velikost diskového bloku se pohybuje od 512B až po 4KB. Speciální soubor je v příkladu použit **1s7**, což je konvence SVID. Další podrobnosti o vytváření svazků uvedeme v čl. 3.2.5.

Oproti tendenci rozdělovat disky do sekcí a v oddělených svazcích pracovat s daty stojí snaha vytvářet rozsáhlé diskové kapacity z více fyzických disků současně, tj. vytvářet svazek na více discích současně. Této technologii se říká *logické svazky* (logical volumes) nebo také *pruhované svazky* (striped disks, každý pruh svazku je jedna sekce jiného disku). Jde o využívání disků jako částí svazků na úrovni obsluhy modulů jádra správy systému souborů. Svazek je vytvořen na několika discích současně. Přístup k datům takového svazku je přitom optimalizován vzhledem k přístupovým metodám jednotlivých

disků. Jednou z výhod je také možnost použití zdvojeného zápisu dat (mirroring) a zvýšení bezpečnosti ukládaných dat. Technologii logických svazků poprvé představil systém OSF/1 (předchůdce Digital UNIX). Dnes je možné používat pruhované svazky v každém moderním systému (IRIX, AIX, HP-UX). Pro vytváření a správu takto organizovaných svazků přitom používá správce systému několik programů s označením Správce logických svazků (Logical Volume Manager, LVM). SVID ani POSIX jej neuvádí, tyto programy mají v různých systémech různá jména. Pro vytváření logických svazků je nejvíce používaný název programu **mk1v**, o kterém se také zmíníme ve čl. 3.2.5. S organizací sekcí několika disků do jednoho svazku souvisí také možnost rozšiřování kapacity svazku. Velikost svazku je totiž po jeho vytvoření pomocí **mkfs** pevné délky, ale mnohdy vzniká potřeba volnou kapacitu rozšířit, a to nikoli pouze pro některý z jeho adresářů připojením dalšího svazku, ale pro všechny adresáře svazku. V rámci práce LVM je tento aspekt podporován u každého současného UNIXu. Předpokládá ovšem použití vhodného typu svazku, kterým může být např. princip svazku typu **bsd** (viz odst. 3.2.2).

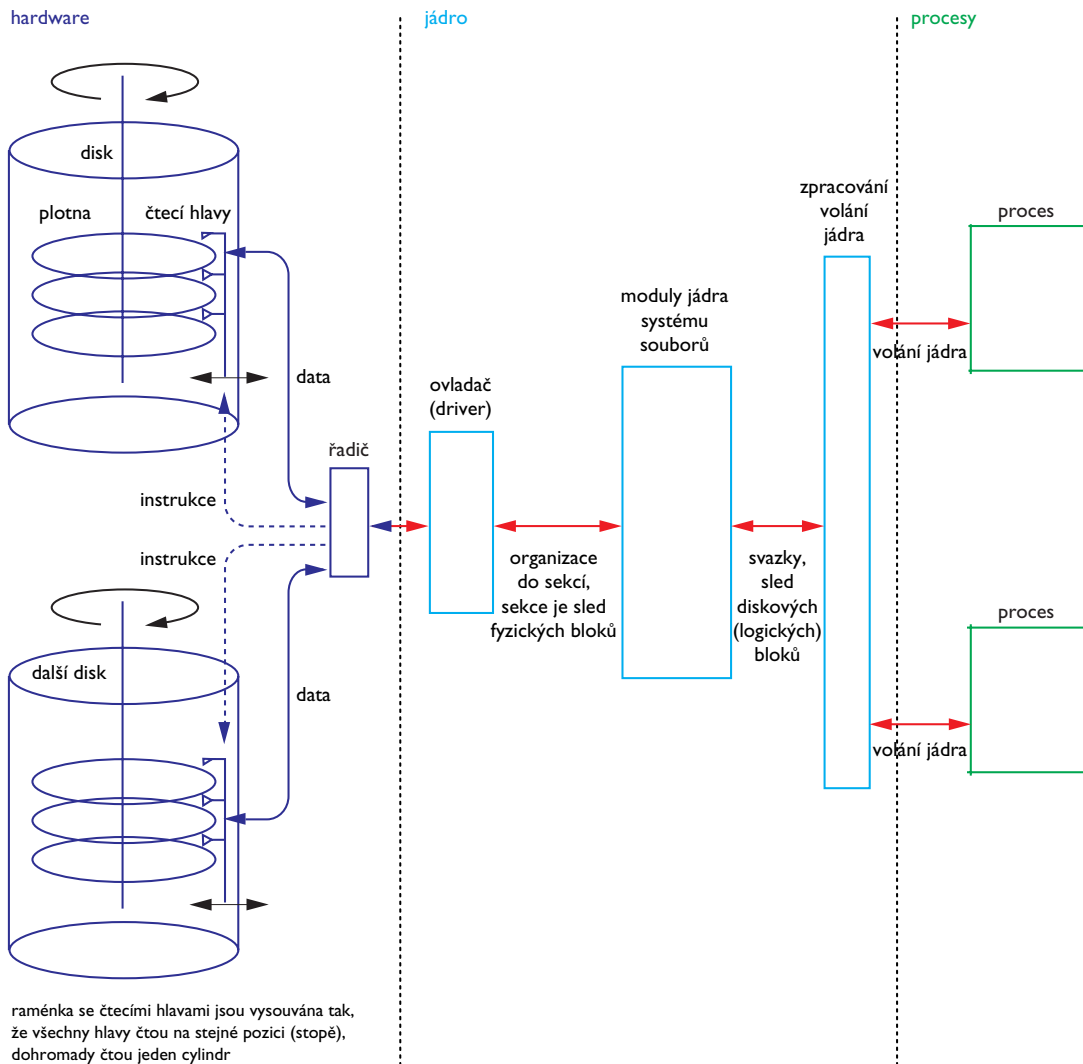
Organizace diskových bloků do svazku je dána konkrétním operačním systémem, mnohdy dokonce i jeho verzí. V průběhu života operačního systému UNIX vznikla celá řada různých typů svazků, kdy se snažili výrobci co nejvíce zrychlit a optimalizovat přístup k diskům, protože data na nich jsou často modifikována a přístup k nim je velmi častý (UNIX je diskový operační systém). Dnes jsou známy dva hlavní modely svazků. Jsou označovány jako **sys5** (velikost bloku od 512B) a **bsd** (velikost bloku až 4KB nebo dokonce 8KB). První z nich je původní, a přestože má v názvu arabskou cifru, znamená 5 v jeho názvu SYSTEM V. Je to výchozí typ svazku pro všechny později navržené. Název sloví jeho částí a jejich organizace zůstává principiálně stejná, charakteristická pro UNIX, přestože o organizaci svazku ani SVID ani POSIX nehovoří. Při popisu základních pojmů svazku vyjdeme proto z organizace **sys5**, kterou i SVR4 stále doporučuje, ale prakticky používá málo. Dnes nejvíce používaný typ svazku je **bsd** (tzv. Berkeley Fast Filesystem), o který projevíli výrobci zájem teprve začátkem 90. let, přestože vznikl již v polovině let 80. Jde o zvýšení propustnosti oproti **sys5**, zvýšení bezpečnosti před ztrátou dat, menší náchylnost k havarijním stavům a snadnou údržbu. Přestože výrobci označují své svazky jinými zkratkami (např. **ufs** pro Digital UNIX, **bfs** pro HP-UX, **EAFS** pro SCO UNIX, 4.2 pro SunOS, **ufs** pro Solaris, **efs** nebo pro IRIX, **ext2** pro Linux atd.), u většiny z nich jde jen o aplikaci modelu **bsd**. Do budoucna ovšem nic nebrání i velmi radikální změně při objevu nové technologie zpracování přístupu k diskům, přestože důležitá budou také nová vydání standardů (každý současný UNIX rozumí např. datům na CD-ROM jako svazku podle ISO9660). Pohled jádra na disky v uvedeném přiblížení ukazuje obr. 3.7.

Základní pojmy svazku pro pochopení jeho organizace dat jsou *i-uzel*, *superblok*, *datový blok* a *zaváděcí (boot) blok*.

## 3.2.1 I-uzel

*I-uzel* (inode – index node) je místo na svazku, které soustřeďuje všechny informace potřebné pro manipulaci se souborem. Kromě všech atributů (přístupová práva, vlastník atd., ale ne jméno) obsahuje také odkazy na datové bloky souboru. Každý soubor libovolného typu má jádrem přiřazen právě jeden i-uzel. I-uzel může být ale odkazován z více míst svazku. Odkaz znamená registraci položky v adresáři s číslem odpovídajícího i-uzlu (inode number nebo i-number), což je vytvoření jména souboru. I-uzel při zavádění takové položky v adresáři přitom může být nově alokovan nebo může být použit již existující





Obr. 3.7 Disk, přístupové moduly jádra

i-uzel (volání jádra `link`); soubor tak má více různých jmen, říkáme také odkazů (links). Velikost i-uzlu je v `sys5` 64B, v `bsd` 128B, není ale důvod zakazovat jeho prodlužování. I-uzly jsou soustředěny do souvislé oblasti svazku, kde jsou odkazovány z adresářů. Odkaz je pořadí i-uzlu v oblasti, odtud název index node. V `sys5` je oblast i-uzlů umístěna před datové bloky podle obr. 3.8.

I-uzel obsahuje atributy souboru:

- typ,
- počet odkazů (jmen),
- vlastníka,
- skupinu vlastníka,
- přístupová práva,
- velikost v počtu bytů,
- 13 adres datových bloků,
- datum a čas posledního přístupu k obsahu souboru,
- datum a čas poslední změny obsahu souboru,
- datum a čas poslední změny i-uzlu.

Uvedené atributy kromě adres datových bloků může proces získat voláním jádra `stat`, jak bylo uvedeno v čl. 3.1.1.

Podle typu souboru stanovuje jádro přístupové metody manipulace se souborem, provádí nebo odmítá provádět akce požadované procesem. Znamé a používané typy souboru jsou (*označení* je text používaný ve výpisech příkazů, např. příkazu `ls`):

<b>označení</b>	<b>typ souboru</b>
-	obyčejný soubor
d	adresář
l	symbolický odkaz
b	speciální blokový soubor (přístup k periférii)
c	speciální znakový soubor (přístup k periférii)
p	pojmenovaná roura

Typ souboru je čitelný z položky `st_mode` struktury `buf` ve volání jádra `stat` (nebo `fstat`).

Uvedené typy souboru stanovuje SVID a POSIX<sup>5</sup>. Typ souboru (i-uzlu) je obecně v UNIXu velmi důležitý. Přestože je konstruován a používán zejména pro ukládání dat na discích, tj. pro práci s datovými soubory nebo adresáři, je využíván také pro účely přístupu procesů k dalším systémovým zdrojům. Např. práce se speciálními soubory znamená aktivaci ovladače příslušené periférie a práce s ní. Je zřejmé, že obsah i-uzlu bude v tomto případě v části odkazů na datové bloky pozměněn na informace potřebné ke spojení s příslušnou periférií (délka a další obsah i-uzlu ale zůstává nezměněn!). Rovněž tak je dalším příkladem typ souboru pojmenovaná roura (viz kap. 4), jejíž implementace přinesla nutnost komunikace obecně libovolných procesů mezi sebou. Procesy jako odkaz na místo společné komunikace uvádějí jméno souboru, jehož i-uzel je pojmenovaná roura. I zde jádro zajišťuje nikoliv práci s daty svazku, ale aktivuje algoritmy pro zajištění toku dat mezi procesy v operační paměti. Obecně je ale odkaz na smluvené jméno souboru nejvhodnějším způsobem pro navázání komunikace mezi libovolnými procesy. Myšlenka využití i-uzlu při implementaci odkazu procesů na nově zaváděné systémové zdroje je používána velmi často. Jako další příklad, který prozatím není uveden v doporučujících doku-

mentech, je typ souboru `s`, což je označení pro síťovou schránku (`socket`) domény UNIX jako jednoho ze způsobů komunikace procesů (viz. kap. 7). SVID při popisu aktivace dalších modulů jádra při komunikaci procesu s periferiemi v tzv. PROUDECH (STREAMS, viz čl. 6.2) rovněž definuje možnost tzv. pojmenovaného PROUDU v kontextu definice nového typu i-uzlu.

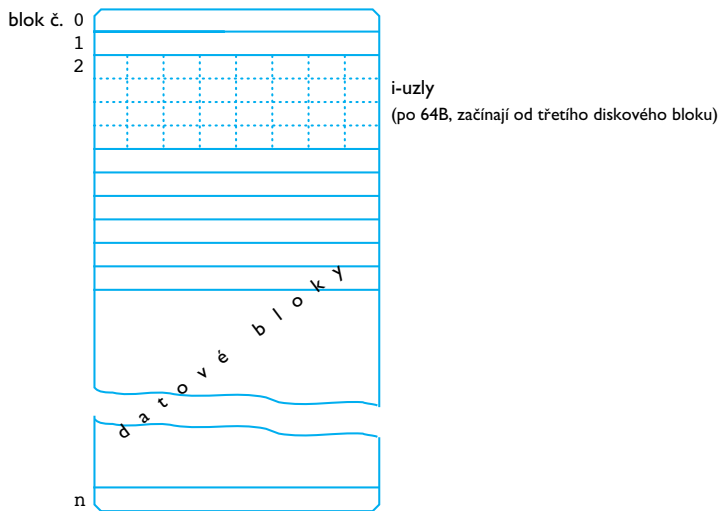
Proces obyčejného uživatele typ souboru ovlivňuje pouze stanovením konkrétní operace volání jádra při vytváření obyčejného souboru (`creat`, `open`), adresáře (`mkdir`) nebo symbolického odkazu (`symlink`). Pro vytváření speciálních souborů bylo kdysi zavedeno volání jádra `mknod` (je privilegované pro jiné použití než pro vznik pojmenované roury), které ale už tehdy umožňovalo vytvářet i-uzel libovolného typu. Pomocí `mknod` se standardně vytváří znakové a blokové speciální soubory podle uvedeného formátu, ale nic nebrání vytvářet i-uzel libovolného typu:

```
#include <sys/types.h>
#include <sys/stat.h>

int mknod(const char *path, mode_t mode, dev_t dev);
```

Volání jádra vytváří i-uzel a jméno souboru `path`, které je v odpovídajícím adresáři vytvořeno. Číslo i-uzlu přiřadí jádro, `mode` je binární součet typu i-uzlu a přístupových práv. SVID definuje konstanty `S_IFIFO` pro pojmenovanou rouru, `S_IFCHR` pro speciální znakový soubor, `S_IFDIR` pro adresář, `S_IFBLK` pro speciální blokový soubor a `S_IFREG` pro obyčejný soubor. Je zřejmé, že při použití `mknod` např. pro vytvoření i-uzlu typu adresář musí volající proces zajistit naplnění datové části souboru položkami `.` a `..` pro zajištění konzistence stromu adresářů svazku. Vzhledem k tomu, že pak celé vytvoření adresáře není operace atomická, je zavedeno volání jádra `mkdir`. Rovněž tak se pro nové typy i-uzlů zavádí vždy nové volání jádra. Jejich definice je ale často komplikována kompatibilitou na doporučení a standardy, které akceptují nové vlastnosti až po ověření jejich praktického významu. Proto je `mknod` často používáno pro nové typy i-uzlů, jejichž popis ještě nebyl zahrnut do závazných dokumentů (SVID a POSIX). Standard POSIX `mknod` vůbec nedefinuje, protože nijak nedefinuje implementaci systému souborů ani termín i-uzel. Pro vznik souborů nových typů definuje vždy nová volání jádra (např. pojmenovaná roura vzniká podle POSIXu voláním jádra `mkfifo`, viz kap. 4). U `mknod` v parametru `mode` používá proces také definici přístupových práv tak, jak byly uvedeny u volání jádra `creat` a `open` (viz odst. 3.1.1), rovněž tak při respektování nastavené masky přístupových práv pro vytváření souborů (volání jádra `umask`). Konečně `dev` je parametr, který je používán při vytváření speciálních souborů a budeme se mu věnovat v následujícím čl. 3.3. Při jiném použití je ignorován.

Počet odkazů je položka i-uzlu, ve které je registrován celkový počet jmen souboru. Jsou to všechny odkazy (odkaz je číslo i-uzlu) z adresářů celého svazku, kde je každému odkazu přiděleno jméno souboru. Při vzniku obyčejného souboru je počet odkazů 1. Jak ukazuje obr. 3.4, vznikne-li adresář, počet odkazů je 2 (pro udržení konzistence je odkaz na i-uzel také v položce se jménem `.` ve vzniklém adresáři) a se vznikem jeho podadresářů počet odkazů dále narůstá. Počet odkazů v i-uzlu lze zvyšovat voláním jádra `link` (pro adresáře musí být proces privilegovaný), což u obyčejných souborů znamená pracovat s tímto souborem pod různými jmény (i-uzel je pak tentýž, proto pozor na vlastnictví souboru a jeho přístupová práva při odkazech z adresářů různých uživatelů!). Volání jádra `unlink` pak snižuje počet odkazů v i-uzlu o 1, ruší spojení mezi některým ze jmen souboru a i-uzlem. Pokud byl před provedením `unlink` počet odkazů v i-uzlu 1, dosažená 0 jako počet odkazů znamená zrušení souboru, tj. uvolnění datových bloků i použité diskové paměti i-uzlu (viz odst. 3.2.2). Jak jsme již uvedli v odst.



Obr. 3.8 Oblast i-uzlů ve svazku sys5

3.1.3, takto konstruovaný systém vícenásobných odkazů na i-uzly může být spolehlivý v rámci jednoho svazku, protože i-uzly jsou odkazovány svým umístěním v oblasti i-uzlů každého svazku, a to v každém novém svazku vždy opět od 0 (alokace ale vždy začíná až od čísla 2, víte proč?). To znamená, že různé soubory na různých svazcích mohou mít stejná čísla i-uzlů. Proto byla v UNIXu zavedena metoda tzv. *symbolických odkazů* (volání jádra `symlink` a `readlink` viz odst. 3.1.3), kdy pro vícenásobný odkaz na obsah souboru je alokován nový i-uzel, který je typu 1 (symbolický odkaz), má své vlastní atributy (znova pozor na vlastnictví a přístupová práva, majitel původního souboru může být jiný, než který vytváří odkaz!) a v datových blocích obsahuje cestu na odkazovaný soubor. Obyčejný (angl. označovaný jako *hard*) odkaz a symbolický odkaz na tentýž soubor je na obr. 3.9.

Dnes je zvykem používat symbolické odkazy pro aplikační úroveň. Jsou totiž při výpisech uživatelských programů dobře viditelné. Obyčejné odkazy jsou ponechávány pro skrytou funkci zajištění konzistence adresářů a jiné systémové akce (např. vícenásobné odkazy na speciální soubory), přestože je to jenom úzus. Pro rozpoznání souborů jako symbolických odkazů má uživatel v rozhraní shellu označení `l` u typu souboru v příkazu `ls -l` a u jména souboru uveden i obsah datové části symbolického odkazu (cesta k odkazovanému souboru). Rozpoznávat obyčejné vícenásobné odkazy na i-uzel může uživatel pomocí výpisu příkazu `ls -li`, kdy je u každého jména souboru vypisováno také odpovídající číslo i-uzlu. Získat jména souborů včetně cesty téhož čísla i-uzlu můžeme příkazem `find`, např.

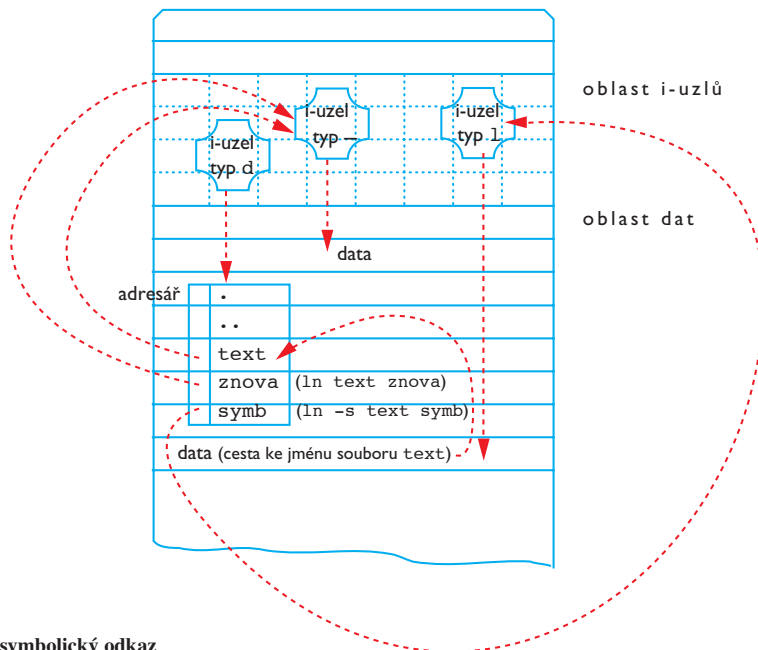
```
$ find /usr2 -inode 1621 -print
```

vyhledá všechny soubory, jejichž číslo i-uzlu je 1621, a vytiskne na standardní výstup úplná jména těchto souborů. Prohledávání bude v podstromu od adresáře `/usr2`, který ovšem může znamenat více napojených svazků (s odkazem na obr. 3.1, adresářem `/usr2/src` začíná svazek na disku č. 3), proto mohou čísla i-uzlů být shodná i u různých souborů.

Vlastník a skupina vlastníka i-uzlu jsou hodnoty číselného označení majitele a skupiny souboru. Oba atributy jsou naplněny v okamžiku vzniku souboru (alokace i-uzlu) podle efektivního vlastníka a efektivní skupiny vytvářejícího procesu. Proces s odpovídajícími přístupovými právy může později tyto hodnoty měnit (voláním jádra `chown`). V operačním systému je tabulka převodu jmen registrovaných uživatelů na jejich číselné ekvivalenty uložena v souboru `/etc/passwd` a skupin v `/etc/group`. Obě tabulky využívají všechny procesy komunikující s uživateli. Uživatel zadává uživatele nebo skupinu jménem, ale jádro je oba akceptuje číselným označením (pozor na změny v `/etc/passwd` nebo `/etc/group`, viz kap. 5). Každý i-uzel proto eviduje vlastníka i skupinu číselným označením. I-uzel také obsahuje slovo přístupových práv, což jsou určující hodnoty pro manipulaci se souborem pro vlastníka, uvedenou skupinu a ostatní. Smysl slova přístupových práv byl uveden u volání jádra `open`, `creat` a `stat` v odst. 3.1.1. Při změně vlastníka nebo skupiny musí obyčejný uživatel pečlivě zvažovat výsledek, protože předáním souboru do vlastnictví jiného uživatele vytváří nový smysl slova přístupových práv a může se stát, že získat zpět vlastnictví souboru již nebude možné.

Obsahem i-zlu jsou také informace o vlastním obsahu souboru. Je to velikost v počtu bytů a 13 adres datových bloků (implementace svazků typu `bsd` mívají počet adres rozšířen). Pro obyčejný soubor je to seznam bloků s daty, ať už textovými nebo binárními (jádro uvažuje obsah souboru jako sekvenci bytů bez vnitřní struktury). Pro adresář je to rovněž seznam bloků s daty, které obsahují tabulku dvojic čísel i-uzlů a jmen všech položek adresáře (adresář je implementován jako obyčejný binární soubor s příznakem adresáře). Další typy i-uzlů, jako např. speciální soubory, pojmenované `roury` atd., pokud nevyužívají datovou část svazku, mají v odpovídající části i-uzlu informace nikoliv o odkazech na datové bloky a velikost souboru, ale jiné hodnoty odpovídající danému typu a využití i-uzlu. Dále byl uvedený mechanismus přidělování a údržby datových bloků souboru implementován v prvních veřejných verzích UNIXu a principiálně zůstává zachován dodnes. Položka velikost souboru v i-uzlu je určující k označení délky souboru v počtech bytů. 13 adres je 13 informací s odkazem na datové bloky, tj. do části svazku, která následuje za oblastí i-uzlů (viz obr. 3.8). Každý odkaz (adresa) má velikost 3 byty (24 bitů), odkazy jsou číslovány a označovány 0 až 12 (viz obr. 3.10).

Adresy 0 - 9 jsou adresy přímé. Vedou k diskovým blokům, v nichž jsou uložena data souboru. Je-li tato část vyčerpána, je při dalším rozšíření obsahu souboru využívána adresa 10 (nepřímá první úrovně), při jejím obsazení je současně alokován jeden diskový blok a ten je použit pro uložení adres datových bloků. Jsou-li vyčerpány všechny adresy takového bloku, je v i-uzlu použita adresa 11 (nepřímá druhé úrovně) a současně jsou alokovány dva diskové bloky pro adresaci vlastních dat. Blok, který je adresován z adresy 11 i-uzlu, odkazuje na diskové bloky, v nichž je uložena teprve adresace bloků s daty. Jsou-li vyčerpány všechny adresy bloků s daty, jádro alokuje nový blok s adresami, odkaz na něj je ve druhé adrese bloku adresy 11 i-uzlu. Teprve po vyčerpání všech možností adresace přes dvě úrovně diskových bloků s adresami je použita adresa 12 i-uzlu (nepřímá třetí úrovně), kdy získání adresy diskového bloku s daty pro jádro znamená nejprve projít tři adresové diskové bloky. Největší možná kapacita souboru je pak závislá na velikosti diskového bloku. Předpokládejme, že diskový blok je schopen obsahovat  $n$  adres, kapacita je pak  $10+n+n^2+n^3$  diskových bloků, je-li  $n=256$  a velikost diskového bloku



Obr. 3.9 Obyčejný a symbolický odkaz

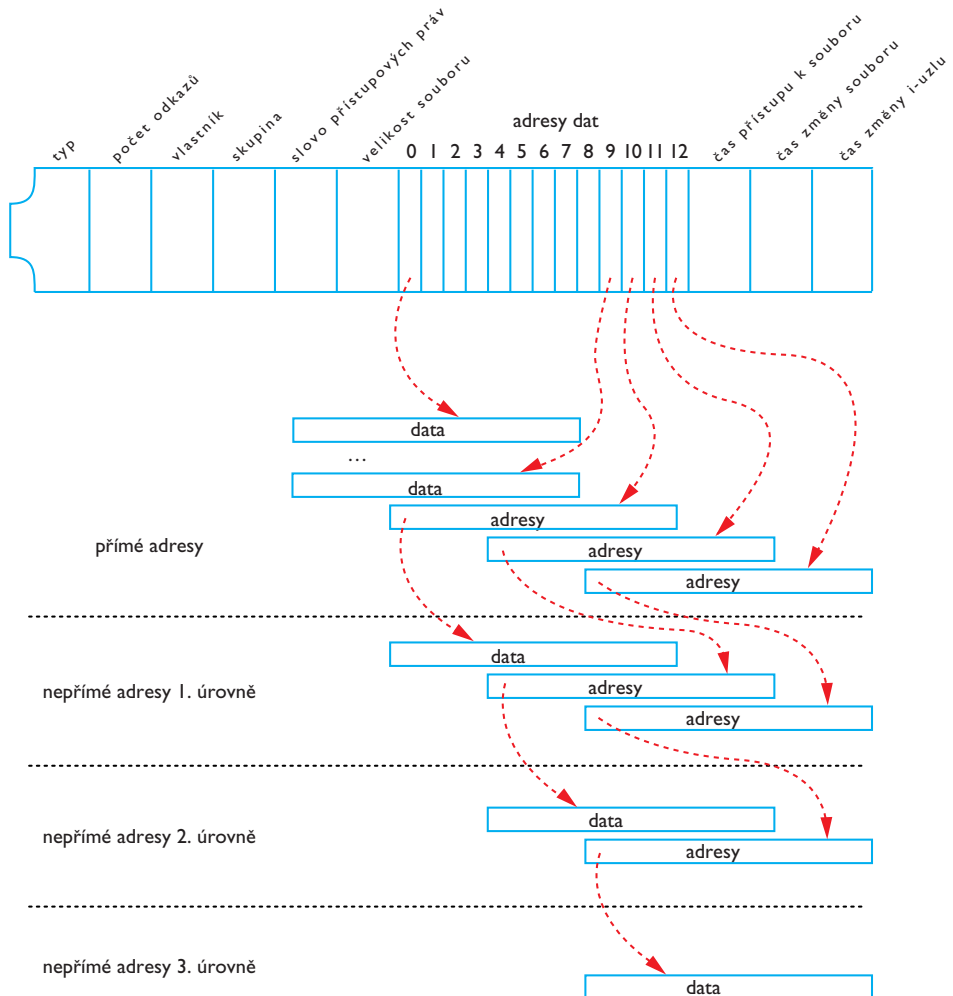
je 1K, dosahujeme velikosti až 16GB. Tato hodnota je ovšem omezena položkou velikosti souboru v i-uzlu, která má 4B (32 bitů) a v níž je možné uložit hodnotu do 4GB. Je zřejmé, že pokud v budoucnu bude velikost souboru požadována nad tuto hodnotu, musí se změnit nejenom velikost diskového bloku ale také velikost i-uzlu. Připustíme-li, že bude velikost diskového bloku za uvedené situace pouze 512B (u mnoha systémů tomu tak je), dostáváme se pod hodnotu 4GB. Svazky s touto kapacitou bloku jsou ale konstruovány jako typu `bsd`, kdy je snížena režie při využití bloku pro adresaci a je možné používat i zde  $n=256$  (viz odst. 3.2.2).

Způsob využívání datových bloků disku při rozšiřování velikosti obsahu souboru je efektivní při používání menších souborů, protože přístup ke vzdálenějším blokům souboru je zatížen režii modulů systému souborů ruku v ruce s algoritmy práce s vyrovnávací pamětí systému. Statistiky ale ukázaly, že v UNIXu jsou používány informace rozdělené do více krátkých souborů, jak k tomu nutí způsob programování a využívání modulárních technik pomocí nástrojů programátora (tools). Výhoda používané alokace dat na disku je v relativně rychlém přístupu i ke vzdáleným částem souboru. Proces tak po otevření souboru může oznámit místo dalšího přístupu k souboru (pro čtení nebo zápis) pomocí volání jádra `lseek`. Jádro snadno vypočte pořadí diskového bloku, který je nutno procesu zpřístupnit, což provede stejným počtem diskových operací jako při sekvenční manipulaci se souborem. Volání jádra má formát

```
#include <sys/types.h>
#include <unistd.h>
```

```
off_t lseek(int fildes, off_t offset, int whence);
```

Při použití (`off_t` bývá definováno jako `long`) na descriptor `fildes` otevřeného souboru jádro provede nastavení pozice pro operaci volání jádra `read` nebo `write` na pozici danou `offset` počtem bytů, a to od začátku souboru (je-li na místě `whence` použito `SEEK_SET`), od současné pozice (`SEEK_CUR`) nebo od konce souboru (`SEEK_END`). Hodnota `offset` je vždy připočtena a lze ji používat i v záporných hodnotách podle typu `offset`. V návratové hodnotě v případě úspěchu `lseek`



Obr. 3.10 I-uzel: systém adres datových bloků

vrací výsledné nastavení v souboru v počtu bytů od začátku souboru. Tak např. zjištění současného umístění pro operaci nad obsahem souboru získáme v návratové hodnotě

```
lseek(fildes, 0L, SEEK_CUR);
```

SVID definuje také funkci `fseek` pro nastavení v souboru otevřeného s přístupem i/o streams. Formát je

```
#include <stdio.h>
```

```
int fseek(FILE *strm, long int offset, int whence);
```

kdy parametry mají adekvátní význam. Pro snadnější práci pak je ještě k dispozici funkce `rewind` pro nastavení na začátek souboru (je to `(void)fseek(strm, 0L, SEEK_SET)`) a pro zjištění současné pozice funkce `ftell`.

Informace o časech přístupu k obsahu souboru a i-uzlu jsou poslední nekomentované položky každého i-uzlu. Jejich význam je zřejmý – jsou modifikovány moduly jádra pro systém souborů v okamžiku odpovídající události. Informace jsou uloženy v počtu vteřin od začátku epochy (podle POSIXu), který je stanoven na 0 hodin 0 minut 0 vteřin 1. ledna 1970, viz volání jádra `time` a `stime` v odst. 2.3.6 a `stat` v odst. 3.1.1. POSIX nezná pojem i-uzel, položka `st_ctime` struktury `buf` je proto u volání jádra `stat` komentována jako hodnota časového údaje pro změnu statutu souboru, v UNIXu je to změna obsahu i-uzlu.

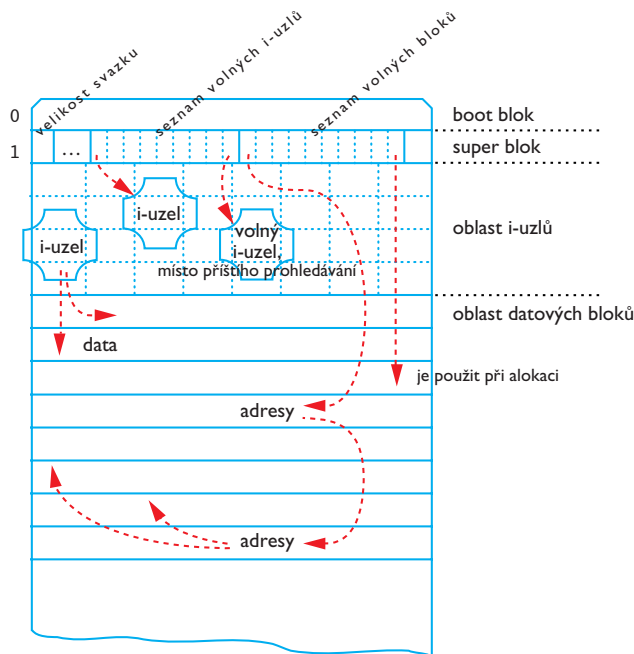
## 3.2.2 Superblok

Na obr. 3.8 je zobrazena oblast i-uzlů od třetího diskového bloku svazku. V klasické implementaci svazku `sys5` jsou obsahem druhého bloku úhrnné informace o svazku, jedná se o tzv. superblok (překládáný také jako blok popisu svazku). *Superblok* je také součástí svazků typu `bsd`, má ovšem pozměněný obsah a jeho kopie je umístěna několikrát v různých částech při vytvoření svazku. Superblok obsahuje důležité informace svazku, které se dynamicky mění s každou změnou v ostatních částech svazku (tj. oblasti dat i i-uzlů). Jeho deformace nebo úplné zničení má paralyzující vliv na další korektní činnost svazku. Pečlivě se proto o něj starají všechny současné typy svazů (např. také udržováním několika jeho aktuálních kopií). Superblok v `sys5` obsahuje tyto informace:

- velikost svazku v počtu bloků,
- počet volných datových bloků,
- seznam adres volných datových bloků a adresu bloku se seznamem dalších volných datových bloků,
- velikost oblasti i-uzlů v počtu bloků,
- počet volných i-uzlů,
- seznam volných i-uzlů a ukazatel na seznam dalších volných i-uzlů,
- pole zámků pro seznamy volných datových bloků a i-uzlů,
- příznak modifikace svazku.

Podle tohoto seznamu je zřejmé, že superblok je místo, kde moduly jádra systému souborů evidují každé nové přidělení volné části svazku. Při vytváření souboru je požadován některý z neobsazených i-uzlů a dále určitý počet datových bloků. Naopak při zmenšení souboru nebo zrušení i-uzlu dochází ke zpětnému procesu – uvolnění dříve obsazeného datového bloku nebo i-uzlu, a opět dochází k modifikaci položek superbloku. Situace je na obr. 3.11.





Obr. 3.11 Správa volného prostoru svazku typu sys5

Superblok obsahuje seznam adres volných datových bloků, které přiděluje pro odkazy z i-uzlů. Takový seznam je pochopitelně omezené délky. První v řadě z tohoto seznamu adres (je to zásobník adres) neodkazuje na volný datový blok, ale na datový blok, ve kterém jsou uloženy další adresy volných datových bloků. Dojde-li k vyprázdnění (použití) všech adres ze seznamu superbloku, jsou načteny další adresy z odkazovaného bloku adres a přidělování může pokračovat. Program **mkfs** se při vytváření svazku snaží vytvářet seznam adres volných bloků svazku tak, aby respektoval umístění bloků vzhledem k otáčkám disku. Jde o tzv. gap (díra) nebo interleave factor (faktor prokládání), což jsou parametry **mkfs**, ve kterých může správce systému stanovit rozmístění fyzických bloků na disku vzhledem k rychlosti otáčení disku a rychlosti zpracování dříve přečteného bloku (při umísťování bloků svazku ihned za sebou by zpracování ztratilo celou otáčku disku a docházelo by ke zbytečným prodlevám). Proto u přidělování adres ze seznamu volných bloků nově vytvořeného seznamu je posloupnost datových bloků souboru poměrně optimální, přestože přidělování je zajišťováno po jednotlivých blocích a o přidělení z téhož svazku může usilovat několik procesů současně. Uvolňování datových bloků při zmenšování obsahu nebo rušení celého souboru probíhá tak, že adresa právě uvolněného datového bloku se vloží do seznamu v superbloku, takže v následujícím požadavku alokace je přidělen. Teprve není-li možné uvolněný blok vložit do seznamu superbloku (protože je plný), je seznam přesunut do datového bloku se

seznamem adres a adresa tohoto bloku je vložena na první pozici seznamu v superbloku následována adresou uvolněného datového bloku. Bloky se seznamem adres jsou zřetězeny ve vázaném seznamu.

Při vzniku nového souboru jádro alokuje některý z volných i-uzlů. Každý volný i-uzel je ve svazku rozpoznatelný, protože v části typu obsahuje nulu. Jádro při postupném čtení oblasti i-uzlů lehce nalezne volný i-uzel a může jej přidělit. Pro lepší průchodnost svazku je určitý počet prvních takových i-uzlů odkazován ze seznamu v superbloku (jde to dobře, protože číslo i-uzlu je indexem do oblasti i-uzlů). Seznam i-uzlů superbloku vznikne při **mkfs** a je z něj postupně odebíráno. Je-li vyprázdněn, jádro prohledá oblast i-uzlů a nově naplní seznam. Prohledávání je vždy od umístění i-uzlu, který byl při posledním vytváření seznamu vložen do seznamu jako poslední, po naplnění seznamu jádro přiděluje i-uzly od začátku seznamu. Při uvolňování i-uzlu se jádro zajímá o seznam v superbloku. Je-li v něm volné místo, číslo uvolněného i-uzlu do něj vloží. Je-li seznam plný, testuje umístění v oblasti i-uzlů naposledy evidovaného i-uzlu v seznamu. Je-li uvolněný i-uzel blíže k začátku oblasti i-uzlů, je vyměněn za testovaný v seznamu. Při vytváření nového seznamu superbloku je pak prohledávání oblasti i-uzlů od místa prvního volného i-uzlu.

Při alokaci nebo dislokaci i-uzlu nebo datového bloku musí moduly jádra provádět také zamykání superbloku pro dosažení výlučného přístupu při požadavcích více procesů, a proto jsou součástí superbloku i pole zámek. Je pochopitelné, že v takovém případě jsou uvedené algoritmy zatíženy odchylkami.

Při alokaci nebo dislokaci i-uzlů a datových bloků jsou také měněny položky počtu volných i-uzlů a bloků svazku v superbloku.

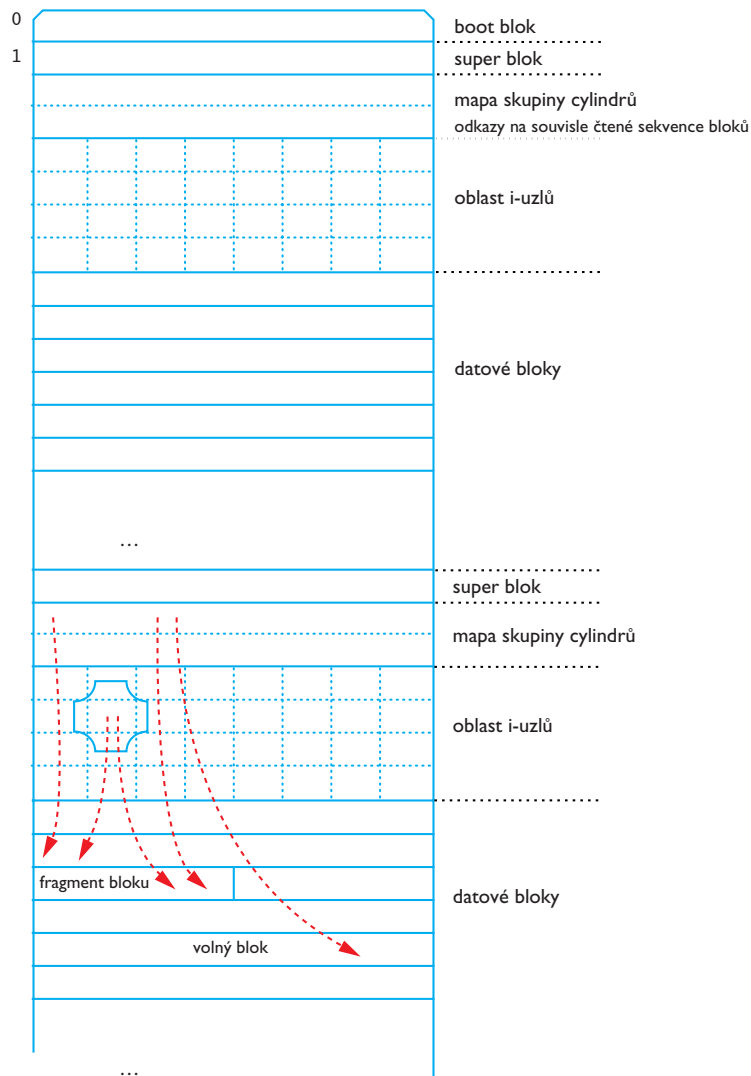
Přestože při návrhu popsaného mechanismu práce se superblokem byla snaha dosáhnout optimální rychlosti tak, aby se zamezovalo vícenásobným diskovým operacím, typ svazku má řadu nevýhod, které časem snižují jeho průchodnost. Když pomineme nevýhodu okamžitého použití uvolněných částí při snaze zachraňovat omylem smazaná data, nejdůležitější problém vzniká ztrátou optimálního seřazení volných datových bloků při delším využívání svazku. Program kontroly svazků **fsck** pro tento typ svazků proto zavedl možnost reorganizovat adresaci volných bloků pomocí volby **-s**. Principiálně jde ale vždy o dodatečné operace nad odpojenými svazky. Celkově je také svazek křehký, protože informace pro konzistenci uložených dat jsou evidovány bez jakéhokoliv zabezpečení (např. kopií některých vazeb nebo jejich zdvojení atd.). Vnitřní implementace svazků je proto jedním z míst, kde výrobci UNIXu dodnes nepřestávají vyvíjet nová schémata uložení dat. Jedním z podstatných obecných typů práce jádra se svazky, který je dnes nejvíce využíván, je struktura podle systému BSD, často označovaná jako svazek typu **bsd**.

Jak již jsme uvedli, z BSD pochází myšlenka rozdělení disku na sekce po cylindrech. Tato koncepce nezůstala pouze při snaze oddělit sekvence fyzických bloků disku od sebe definicí různých sekcí disku. Po sobě jdoucí fyzické bloky cylindrů disku jsou dále uvažovány po jednotlivých plotnách (viz obr. 3.7), resp. jejich částí, protože sekce disku je dána rozsahem od stopy ke stopě (počet cylindrů) každé plotny, kterou čte hlavička disku. Svazky typu **bsd** se snaží organizovat uložení dat na disku tak, aby při výpadku jedné čtecí hlavy nebo části plotny nebyl svazek sekce paralyzován výpadkem náhodné části svazku. Dále je kladen důraz na rychlost přístupu k datům disku. Tento typ svazku na rozdíl od výchozích myšlenek UNIXu sestupuje na úroveň znalosti provozních charakteristik uvažovaného disku. Praxe ovšem ukázala, že nejde o návrat ke strojové závislosti operačního systému, protože každý disk má princip přístupu k datům stejný (podle obr. 3.7). Rozložení částí svazku ukazuje obr. 3.12.

Sekce je rozdělena na skupiny cylindrů (cylinder groups). Každá skupina cylindrů má kopii superbloku a svoji část i-uzlů. Každá kopie superbloku vzniká při vytvoření svazku (pomocí **mkfs**) a je používána vždy pro část svazku skupiny cylindrů. Některé informace každého superbloku mají globální charakter a jsou využívány jako záloha teprve po poškození prvního superbloku. Součástí každého superbloku jsou ale také informace např. o umístění a velikosti mapy skupiny cylindrů atd., tj. informace pro danou skupinu cylindrů specifické. Alokace i-uzlů probíhá vzhledem k optimálnímu rozložení i-uzlů na disku tak, že je v oblasti i-uzlů odpovídající skupiny cylindrů vyhledán první vhodný. Při alokaci volných bloků skupiny cylindrů je využívána bitová mapa volných bloků skupiny cylindrů. Bitová mapa volných bloků je konstruována tak, že jsou evidovány sekvence volných bloků každého cylindru, které lze číst po sobě nejrychleji možným způsobem (vzhledem k otáčkám disku a zpracováním procesorem). Superblok proto neobsahuje seznam volných i-uzlů, ale ani seznam volných datových bloků. Pro registraci obsazených nebo volných datových bloků se používá bitová mapa bloků skupiny cylindrů, kde je každý volný blok registrován s odpovídajícím příznakem. Tato mapa bloků je prohledávána při požadavku alokace nového datového bloku. Přitom je vyhledáván nejbližší vhodný blok pro sekvenci přístup k souboru vzhledem k blokům již pro soubor alokovaným. Superblok proto obsahuje poněkud odlišné informace oproti svazku typu **sys5**. Je to např.

- velikost svazku ve fyzických blocích (sektorech, basic blocks),
- počet skupin cylindrů ve svazku,
- počet sektorů na jednu stopu,
- počet hlav na jeden cylindr,
- datum a čas poslední modifikace svazku,
- jméno svazku,
- počet volných bloků svazku,
- počet volných i-uzlů svazku,
- umístění kopie superbloku,
- adresa skupiny cylindrů,
- velikost skupiny cylindrů,
- velikost bitové mapy bloků v bytech,
- umístění bitové mapy bloků,
- příznak poškození svazku,
- místo dalšího rozšiřování kapacity svazku.

Superblok může obsahovat ještě další informace, které jádro používá pro interní potřebu správy každého svazku a které se mohou lišit nejen podle výrobce, ale i podle typu svazku. Kopie superbloku jsou rozmístovány tak, že při výpadku některé z ploten nebo hlav je možné zbylou část svazku rekonstruovat a využít, a to na základě umístění kopie superbloku na každou plotnu. Velikost logického bloku svazku v původních implementacích **bsd** byla až 8KB. Pro ekonomické využití diskového prostoru bylo možné použít pouze část (tzv. fragment) logického bloku, jehož velikost je definována (osmina velikosti bloku). Bylo (a je) pak nutné udržovat registraci o dostupných fragmentech, což je předmětem bitové mapy bloků dané skupiny cylindrů. Svazek **bsd** je efektivní při jeho využití do přibližně 80% kapacity. Jeho průchodnost se pak výrazně snižuje, protože svazek musí začínat využívat datové bloky různých skupin cylindrů. Správce systému přitom může stanovit hranici zaplnění svazku (informace je uložena v superbloku), kterou může překročit pouze privilegovaný uživatel (doporučuje se používat hranice 90%



Obr. 3.12 Struktura svazku typu bsd

kapacity svazku). Svazky tohoto typu vzhledem k jejich konstrukci do jednotlivých skupin cylindrů se dají poměrně snadno rozšiřovat vazbou na superblok (položka místo dalšího rozšiřování kapacity svazku). Je to případ, kdy svazek pokračuje v další sekci disku, nebo při používání pruhovaných svazků.

I-uzel má ve svazku `bsd` velikost 128B a obsahuje navíc další odkazy na datové bloky (přímých odkazů je např. 10). Se vznikem svazku typu `bsd` se objevila také první implementace symbolických odkazů (viz odst. 3.1.3). V i-uzlu byla vyhrazena položka pro evidenci datové části i-uzlu symbolického odkazu, mnohé typy svazků současných výrobců UNIXu tento způsob používají dodnes, protože přístup k obsahu souboru je rychlejší.

Uvedený princip svazků typu `bsd` je dnes používán přednostně dokonce i u implementací verzí operačního systému UNIX SYSTEM V. Výrobci se přitom snaží dosáhnout co největší průchodnost svých svazků používáním co nejefektivnějších přístupových algoritmů a organizací bitových map disků a cylindrů skupin. Každopádně jde o prověřenou technologii, která stále více vytlačuje původní klasický model svazku `sys5`.

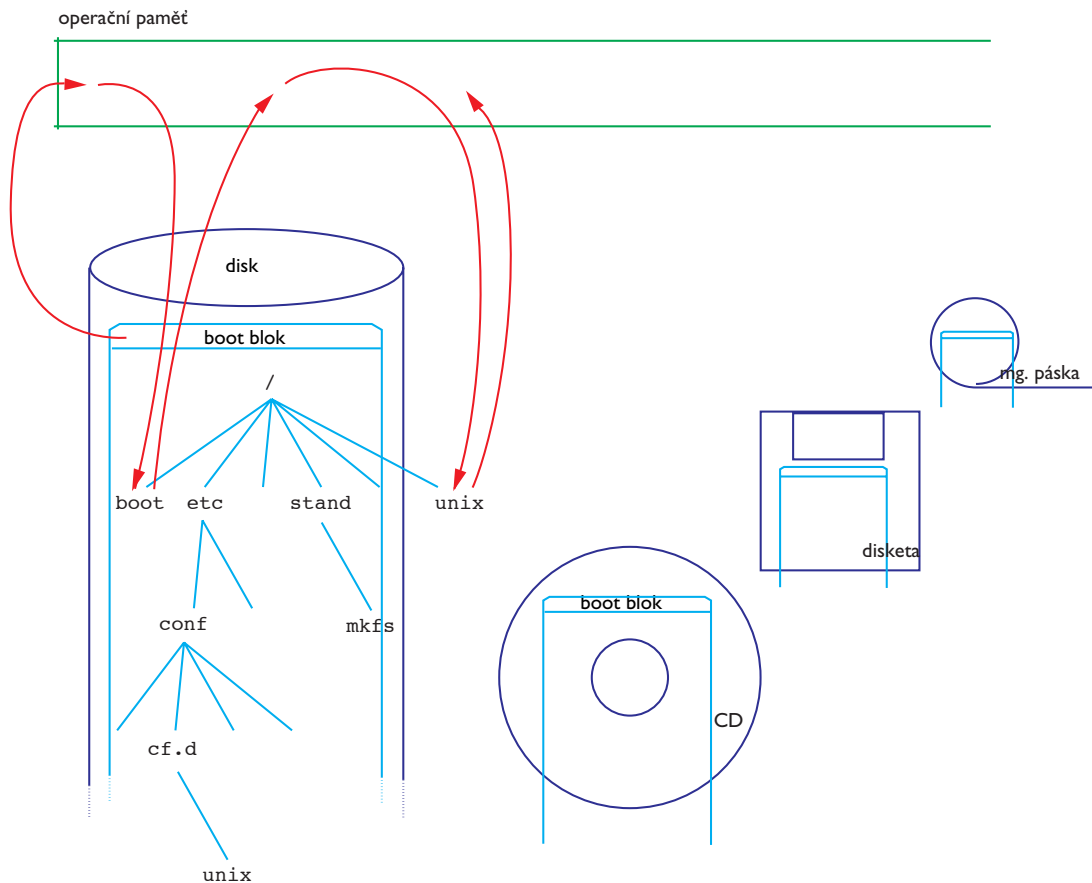
### 3.2.3 Zaváděcí blok

První blok (blok č. 0) každého svazku je rezervován pro podporu zavádění jádra do operační paměti v době startu operačního systému a je nazýván *zaváděcí blok* (boot block). Přestože je skutečně využit pouze u kořenového svazku, při vytvoření každého svazku je rezervován. Oblast ukládání vlastních dat proto začíná až druhým diskovým blokem. Obsahem zaváděcího bloku je program, který umí vyhledat v hlavním adresáři svazku soubor se jménem `unix` (někdy `hp-ux`, `vmunix` atp.). Ten obsahuje program jádra sestavený tak, že je schopen být prováděn na holém stroji<sup>6</sup>. Program zaváděcího bloku (bootstrap) musí být schopen jádro uložit do operační paměti a předat mu řízení, tj. dát pokyn procesoru k provádění instrukcí od místa paměti, která obsahuje startovací adresu jádra. Další činnost operačního systému je pak věcí činnosti jádra. Program zaváděcího bloku někdy nahrazuje činnost firemních řídicích programů hardwaru, které jsou uloženy v neměnných pamětech stroje (PROM, Programmable read-only memory) a jejichž činnost je především testovací. Program zaváděcího bloku nahrazuje ale tu činnost, kdy je nutné sdělit stroji zavedení a start jádra z jiného místa než z první sekce prvního disku, např. při instalaci, tj. použít např. periférii CD-ROM jako svazku s distribuční verzí UNIXu (v IRIXu např. nazývaném `miniroot`). Správce systému ale může chtít spustit pro vyzkoušení také jiné jádro, než je uloženo v souboru `unix`, např. při úpravách rozměrů tabulek jádra nebo po připojení ovladače nové periferie do jádra. Takové jádro u většiny systémů vzniká v souboru `/etc/conf/cf.d/unix`. Dříve než jeho obsah správce překopíruje do `/unix`, je dobré takové nové jádro testovat a ponechat zatím původní jádro pro využití v případě, že nové jádro ještě není funkční. Zavést do paměti a předat řízení libovolnému samostatně proveditelnému programu z některého ze svazků nebo jiné periferie umí právě obvykle pevně daný software dodávaný s holým strojem. Pokud tomu tak není (platforma PC apod.), umí program zaváděcího bloku plnit tuto funkci zástupně. Vzhledem k tomu, že v takových případech je zaváděcí blok příliš malý na to, aby jím obsahovaný program dokázal interaktivně plnit požadavky operátora, zajímá se mnohdy program zaváděcího bloku pouze o obsah souboru `/boot`, který dokáže z libovolného svazku některého z disků vybrat požadovaný soubor, zavést jej do paměti a předat mu řízení. Program souboru `/boot` obecně vypisuje na operátorskou konzolu text

Boot

:

a čeká na interakci operátora. Operátor může odpovědět zadáním cesty k souboru se samostatně proveditelným souborem. Program přitom akceptuje i označení disků v předponě úplného jména souboru, např.



Obr. 3.13 Zaváděcí blok a samostatně proveditelné programy

```

Boot
: /etc/conf/cf.d/unix
nebo
Boot
: hd(40,2)/unix

```

Příklad je převzat z SCO UNIXu (platforma PC), kde hd je označení periferie pevného disku, číselné hodnoty v závorce odpovídají číselné identifikaci disku (jsou shodná s identifikací periferie podle odpo-

vídajícího speciálního souboru, viz čl. 3.3), /unix je jádro svazku na uvedeném disku (třeba zálohovaný kořenový svazek). Nebo

Boot

: /stand/mkfs

je zavedení a start verze samostatně proveditelného souboru s programem pro vytváření svazků, a to z nastaveného disku s kořenovým svazkem.

Zaváděcí blok a programy manipulace před zavedením jádra operačního systému UNIX ukazuje obr. 3.13.

### 3.2.4 Přenos dat, struktury v jádru

Přístup procesu k souboru je prostřednictvím jména souboru. Proces otevírá soubor s daty (tzn. obyčejný soubor) voláním jádra `open`. Jeho návratová hodnota (viz odst. 3.1.1) je ukazatel do tabulky otevřených souborů procesu (viz čl. 2.2), která je součástí struktury `user` (viz odst. 2.4.4) každého procesu.

Položka tabulky otevřených souborů je přidělena pro nově otevřený proces. Její obsah přitom odkazuje do tabulky všech otevřených souborů v jádru. Tabulka všech otevřených souborů je evidence platná pro všechny procesy registrované jádrem. Se souborem může pracovat několik procesů současně, jádro přitom eviduje pouze jedno spojení se souborem. Přístup k souboru je pochopitelně kryt také přístupovými právy, ale pokud se procesy se stejnými možnostmi přístupu k souboru navzájem nedomluví (např. zamykání souboru viz odst. 3.1.4 nebo použitím semaforů viz čl. 4.6), jádro vyřizuje operace se souborem v časové sekvenci jejich výskytu a může se stát, že obsah souboru pak bude překvapivý. Registraci otevřeného souboru jádrem s přístupem dvou procesů ukazuje obr. 3.14.

Na obrázku má proces A obsazeny položky tabulky otevřených souborů (kanály) č. 3 a 4. Kanál č. 4 ukazuje přitom do téhož místa jako kanál č. 3 procesu B. Kanály č. 0, 1 a 2 obou procesů jsou alokovány standardně pro přístup k terminálu. Při `open` jádro tabulku otevřených souborů obou procesů procházelo a našlo tyto kanály použité, a proto je vynechalo (jinak by je přidělilo).

Tabulka všech otevřených souborů jádra je registrace přístupu procesů k souborům s různými přístupovými právy a obsahuje položky:

- přístupové příznaky souboru, tj. zamknutí souboru, modifikace souboru,
- počet procesů používajících soubor,
- ukazatel do tabulky i-uzlů všech otevřených souborů,
- ukazatel místa přístupu k obsahu souboru, tj. aktuální vzdálenost od začátku souboru v bytech pro následující čtení nebo zápis.

Tabulka i-uzlů všech otevřených souborů v jádru je spojení jádra a dat svazku. Při prvním `open` souboru s odpovídajícím i-uzlem je obsah i-uzlu ze svazku kopírován do této tabulky. Navíc je položka tabulky obohacena zejména o tyto informace:

- přístupový status, tj. zamknutí i-uzlu, některý proces čeká na odemknutí i-uzlu, obsah i-uzlu v jádru byl změněn, obsah souboru byl změněn aj.,
- počet procesů, které i-uzel využívají,
- identifikace zařízení, na kterém je soubor umístěn,
- adresu diskového bloku, který byl naposledy přečten z disku.

Při otevření souboru zatím neregistrovaného jádrem dojde k přečtení odpovídajícího i-uzlu ze svazku a je zpřístupněn první datový blok. Je alokována položka tabulky i-uzlů jádra a položka tabulky všech otevřených souborů. Naplněním všech těchto položek je provedeno vzájemné propojení a proces je odkazem ze své tabulky otevřených souborů připojen v jádru na požadovaný soubor. Při čtení nebo zápisu dochází k modifikaci obsahu položek tabulek a informací na disku. Současně se změnami ve struktuře svazku dochází ke změnám nejenom v i-uzlu a datových blocích, ale i v superbloku a návazných částech svazku. Podrobné prozkoumání vazeb, pokud o ně čtenář má zájem, nalezne v [Bach87].

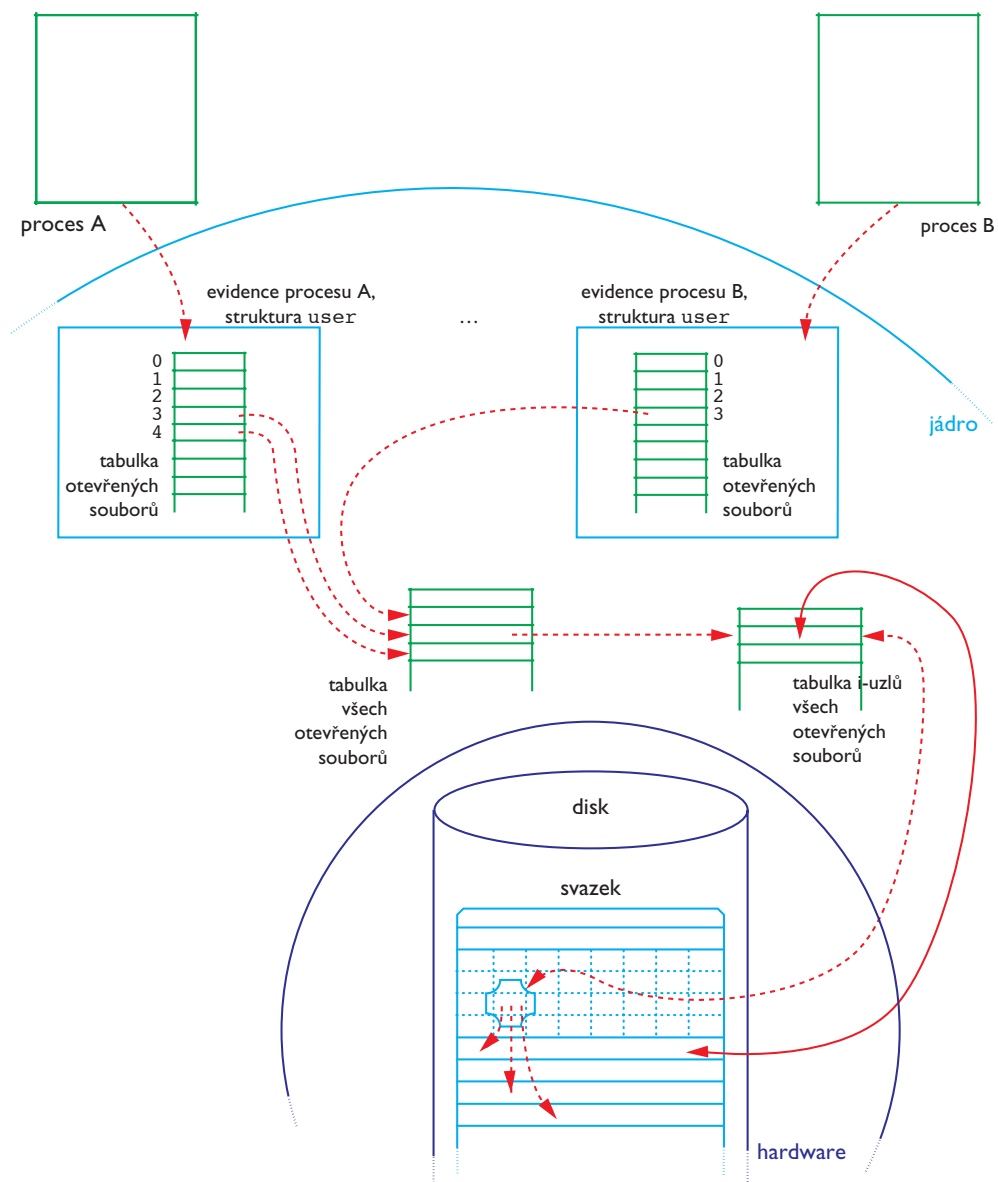
Tabulka i-uzlů všech otevřených souborů systému v jádru je používána také pro přístup jádra k jiným typům souborů, než jsou soubory obvyčejné. Každý registrovaný proces je spojen se svým pracovním, kořenovým a nadřazeným adresářem. Odkaz na ně je veden ze struktury `user`, protože jádro musí znát výchozí adresář při prohledávání stromu adresářů v odkazech na jména souborů zadávaná ve voláních jádra. Všechny takové adresáře musí jádro evidovat jako přístupné, a to včetně jejich obsahu (jde ale opět o i-uzel a datovou část adresáře jako několik diskových bloků). Jádro musí také zajistit možnost čtení obsahu kteréhokoliv adresáře, pokud o to proces požádá (dříve běžným volání jádra `open`, dnes funkcemi `opendir`, `readdir`, `rewinddir` a `closedir`). I přístup jiných typů souborů včetně těch, které nově vznikají, je vždy realizován pomocí této tabulky i-uzlů v jádru. Jaké algoritmy pro obsluhu jádro při práci s obsahovou částí takového souboru použije, je dáno typem souboru. Tuto teorii budeme konkretizovat např. při manipulaci se speciálními soubory (čl. 3.3) nebo u popisu implementace obvyčejné nebo pojmenované roury v kap. 4.

## 3.2.5 Manipulace se svazky

Pro operační systém UNIX je disk jednou z periferií. Každý disk je proto viditelný a přístupný pomocí určitého speciálního souboru. Speciální soubory disků jsou vytvořeny a pojmenovány tak, že každé rozhraní, tj. typ přístupu k disku (jde o typ řadiče, který musí mít svůj vlastní ovladač) vytváří vlastní skupinu speciálních souborů. V ní jsou viditelné jednotlivé disky (jsou číslovány podle pořadí připojení v hardwaru) a na každém disku jeho jednotlivé sekce. Každá sekce na disku v určité skupině má svůj speciální soubor. Také celý disk ve skupině má svůj speciální soubor. Skupina má svůj způsob číslování a pojmenování speciálních souborů (dále viz 3.3). Sekce disku je prostor pro vytvoření svazku. Speciální soubory disků má k dispozici uživatel `root`, svazek vytváří pomocí příkazu `mkfs` s odkazem na speciální soubor sekce za znalosti její velikosti. Při čtení provozní dokumentace k tomuto příkazu správce systému objevíme i další parametry příkazového řádku, které umožňují zadávat např. typ vytvářeného svazku nebo charakteristický přístup k disku v rozložení čtení sektorů stopy podle rychlosti přenosu dat z disku (gap a interleave factor). Po vytvoření je ale svazek stále dostupný pouze pomocí speciálního souboru a obvyčejný uživatel nemá právo manipulace s jeho obsahem. Struktura svazku je navíc srozumitelná poměrně komplikovaným algoritmům, které používá jádro. Přístup uživatele je proto zajištěn přes požadavky volání jádra. Privilegovaný uživatel proto po přípravě sekce disku, tj. vytvořením svazku, sděluje jádru pomocí příkazu `mount`, kterou část disku je možné používat pro zajištění požadavků procesů. Příkaz `mount` provede spojení svazku a některého z adresářů systémového stromu adresářů. Např.

```
# mount /dev/dsk/ls7 /usr2
```





Obr. 3.14 Registrace otevřeného souboru jádrem

je připojení svazku sekce speciálního souboru `/dev/dsk/1s7` k adresáři `/usr2` (viz obr. 3.1). Adresář `/usr2` musí být určen celou cestou a musí existovat. Ve speciálním souboru musí být nalezena organizace bloků podle některého ze systémem podporovaných typů svazků. Pokud tomu tak není, je připojení odmítnuto. Je odmítnuto jádrem, protože příkaz **mount** je realizován voláním jádra `mount`. Jádro vytváří a podporuje spojení mezi svazkem a zadaným adresářem. Volání jádra, které není obsaženo v POSIXu (je implementačně závislý element), má podle SVID formát

```
#include <sys/types.h>
#include <sys/mount.h>

int mount(const char *fs, const char *path, int mflag,
          const char *fstype, const char *dataptr, int datalen);
```

Jméno speciálního souboru je uvedeno v parametru `fs`, jméno adresáře, na který má být svazek připojen, pak v `path`. Parametr `mflag` je doplňující parametr, který může obsahovat logický součet několika hodnot, jejichž význam je například, zda má být svazek připojen pouze pro čtení, nebo slouží (s hodnotou `MS_DATA`) k indikaci přítomnosti parametrů `fstype`, `dataptr` a `datalen`, které zadávají typ jiného než implicitního svazku. Není-li hodnota `MS_DATA` uvedena, zbylé parametry nejsou respektovány a za připojovaný typ je považován svazek typu kořenového svazku.

Od okamžiku úspěšného provedení volání jádra `mount` je kořenový adresář zadaného svazku spojen se zadaným adresářem a je dostupný podle přístupových práv kořenového adresáře připojeného svazku všem ostatním uživatelům. Obyčejný uživatel proto nepoužívá disky pomocí speciálních souborů a pohybuje se pouze ve stromu adresářů, o jehož rozložení na svazky rozhoduje správce systému. Jádro ve svých datových strukturách udržuje tabulku přiřazení adresářů a speciálních souborů a každý odkaz procesu na jméno souboru přes některé volání jádra je zkoumán také podle této tabulky. Tak je definován svazek, ve kterém se soubor nachází (aktuální obsah této tabulky je uchováván také v `/etc/mnttab`, ale ta je udržována pouze pro informativní účely některých příkazů, např. **df**).

Správce systému také provádí odpojení svazků od stromu adresářů. Tato operace s každým dříve připojeným svazkem musí být provedena před zastavením operačního systému nebo před provedením manipulace globálního charakteru se svazkem. Takovou operaci rozumíme tu, která je aktivována privilegovaným uživatelem na svazek zadáním jména speciálního souboru a má zápisový charakter (např. **fsck**). Privilegovaný uživatel odpojí svazek od stromu adresářů příkazem **umount**. V jeho parametru zadává jméno speciálního souboru (tím je identifikován svazek), např.

```
# umount /dev/dsk/1s7
```

Jádro odpojí svazek a v další práci přístupový adresář pro obyčejné uživatele už neznamena práci s daty odpovídajícího svazku. Jádro odmítne provést operaci odpojení, pokud je svazek nějakým způsobem využíván, např. je-li evidován proces, který má jako pracovní, kořenový nebo nadřazený adresář v odpojovaném svazku, nebo je-li na některý adresář svazku připojen další svazek. Jádro odpojí svazek na základě volání jádra

```
int umount(const char *spec);
```

kde na místě `spec` je uvedeno jméno speciálního souboru.

Přestože obyčejný uživatel nepotřebuje znát připojení svazků, může i on tyto informace získat. Informativní výpis o způsobu připojení svazků ke stromu adresářů provádí příkaz **mount** bez parametrů. Obsa-

zenou a volnou kapacitu připojených svazků získá i obyčejný uživatel pomocí příkazu **df**. Příklady jsme uvedli v odst. 3.1.5. Příkaz **df** používá k získání informací o připojených svazcích voláním jádra **ustat**, nebo lépe novým voláním jádra **statvfs**. Dnes přítomné, ale v dalším vývoji nepodporované **ustat** umožňuje získat informace o celkovém počtu všech volných bloků, i-uzlů a jména připojeného svazku. Vzhledem k tomu, že to nejsou všechny mnohdy potřebné informace o dnešních typech svazků, SVID definuje volání jádra

```
#include <sys/types.h>
#include <sys/statvfs.h>

int statvfs(const char *path, struct statvfs *buf);
```

které naplní obsah struktury **buf** podrobnými informacemi o svazku. Její položky jsou např. velikost diskového bloku, celkový počet volných bloků, počet volných bloků dostupný neprivilegovaným uživatelům, celkový počet i-uzlů, počet volných i-uzlů, jméno a typ svazku, způsob jeho připojení atd. Podrobný popis je uveden v provozní dokumentaci.

Přístup správce ke každému svazku jako ke speciálnímu souboru vyžaduje odpojený svazek (provedený **umount**). Přestože na tento fakt klade důraz provozní dokumentace každého příkazu pro údržbu svazků, nevylučuje jejich použití i na připojené svazky. Ve většině případů pochopitelně nemůže celý proces globální manipulace skončit úspěšně, protože s připojeným svazkem může v době manipulace pracovat některý z běžících procesů, který mění data svazku, nehledě na používání systémové vyrovnávací paměti nad připojenými svazky pro urychlení práce procesů (viz čl. 3.4). Přehled základních používaných programů pro globální manipulaci s odpojenými svazky podle SVID uvádí následující seznam.

**backup**, **bkexcept**, **bkhistor**y, **bkoper**, **bkreg**, **bkstatus**, **fdisk**, **fdp**, **ffile**,  
**fimage**, **incfile**, **migration** – úschova a obnova dat svazku,  
**diskusg** – využití svazku uživateli,  
**fsck** – kontrola a oprava svazku,  
**fsdb** – ladění struktury svazku,  
**fstyp** – rozpoznání typu svazku,  
**ncheck** – generace jmen souborů a jejich čísel i-uzlů,  
**restore**, **rsnotify**, **rsoper**, **rsstatus** – obnova svazku nebo jeho části,  
**urestore**, **ursstatus** – obnova některých souborů a adresářů archivovaného svazku,  
**volcopy**, **labelit** – kopie svazku s kontrolou jeho jména.

Kromě uvedených příkazů správce systému používá každá výkonná implementace UNIXu i další příkazy, které jsou systémově závislé a mají podobné funkce, rovněž tak správa pruhovaných svazků není prozatím standardizována.

Některé sekce disků při běžném provozu neobsahují žádnou strukturu svazku a jsou vyhrazeny pro *odkládací oblast procesů* evidovaných jádrem (swap area, viz odst. 2.4.1 a obr. 3.1). Znamená to, že speciální soubor některé sekce není inicializován jako svazek. Sekce disku je pouze využita v odpovídající velikosti (opět můžeme použít pouze její část) pro odkládání procesů z operační paměti na disk. Je to předmětem činnosti jádra (jeho modulů přidělování operační paměti procesům) a procesu č. 0, proto je taková sekce přidělena v době instalace operačního systému odkazem na speciální soubor sekce některého disku. Odkládací oblast je podmínkou instalace UNIXu a v případě, že nedostačuje její velikost, může správce systému připojit další sekci některého z disků opět s funkcí odkládací oblasti.

Uskuteční to příkazem **swap** (viz odst. 2.4.1), systém pak začne používat obě sekce jako odkládací oblast současně (primární i sekundární). V případě nedostatku volných sekcí disku (příprava sekce pro odkládací oblast není sice složitá, ale uvolnit sekci může být pracné vzhledem k úschově a přesunu dat) může správce stanovit obyčejný soubor některého připojeného svazku, který bude systém uvažovat jako výchozí místo další odkládací oblasti (tzv. dynamická nebo virtuální odkládací oblast). Procesy jsou pak odkládány do dynamicky odebírané oblasti volných datových bloků svazku, což může být výhodné z hlediska stanovení a neurčitě velikosti takové odkládací oblasti. Prostor na svazku pro běžné použití může být ale omezen. Virtuální odkládání je ale také zatíženo algoritmy práce se svazkem. Naopak v případě primární a sekundární odkládací oblasti si přístup k sekci organizuje optimálně systém. Proto by měla být virtuální swap používána nouzově v případech, kdy není jednoduché provádět mnohdy časově náročnou činnost uvolňování sekce disku. Příklad práce s odkládací oblastí uvádíme v kap. 10.

Organizace sekce disku pro odkládání procesů udržuje jádro v tzv. mapě odkládací oblasti. Oblast při odkládání procesu je alokována v počtu bloků souvisle v sekci z důvodů rychlosti přenosu procesu mezi operační pamětí a diskem. Mapa odkládací oblasti je seznam dvojic popisu částí volného místa v oblasti, dvojice je adresa bloku začátku oblasti a velikost oblasti v blocích. Jádro při pokusu odložit proces na disk prohledává mapu a hledá vhodný souvislý úsek. Pokud jej nalezne, použije jej a pokud jej nepoužije celý, registruje v mapě změnu volného úseku. Naopak při uvolňování jádro redukuje mapu. Pokud uvolněná oblast je v sousedství některé volné části, provádí spojení takových úseků. Teprve pokud jádro při hledání volného úseku nenalezne souvislý úsek odpovídající velikosti, pokusí se mapu přeargumentovat tak, aby dosáhlo souvislého úseku. Není-li takto získaný souvislý úsek dostatečně velký, jádro vypisuje na konzolu počítače zprávu `no swap` a odmítne proces z operační paměti odložit.

Odpovídající algoritmy jsou využívány také u dynamické odkládací oblasti. Nenalezení souvislého úseku dostatečné velikosti ale vyvolá alokaci ze seznamu volných datových bloků. `no swap` se objevuje, je-li seznam volných datových bloků nedostačující. Současně ale dochází ke kolizi alokace volných datových bloků pro procesy pracující s tímto svazkem při rozšiřování obsahu svých souborů, takže situace může být havarijní.

## 3.3 Speciální soubory

Operační systém UNIX zpřístupňuje periferie výpočetního systému ve *speciálních souborech* (special files). Speciální soubory jsou ve velké většině konstruovány pro základní přístup správce systému k periférii. Je zvykem, že manipulace se speciálním souborem jsou svěřeny systémovému procesu (serveru periferie) a uživatelé využívají periférii odkazem na takový proces, a to prostřednictvím svých procesů (klientů). Pro práci se speciálním souborem tiskárny je např. při startu systému aktivován proces **lp***psched*, a ten spravuje frontu požadavků na tisk od uživatelů. Uživatelé do fronty připojují požadavky pomocí procesu, který vznikne příkazem **lp** a který požaduje realizaci tisku od **lp***psched*. Základní schéma je tedy oddělit uživatele od speciálního souboru. Speciální soubor je přístup k periférii bez zajištění strategie frontování požadavků či jakéhokoli jiného zajištění atomické operace s periférií. V mnoha případech je ale periferie pod přímou správou jádra (např. disky) a správce pouze konfiguruje jeho činnost (např. příkazem **mount**).

Vnitřní struktura speciálního souboru vychází opět z pojmu i-uzel. Adresář `/dev` a jeho podadresáře obsahují jména speciálních souborů. Každé jméno má přiřazeno číslo i-uzlu. Konvence pojmenování

speciálních souborů není pevně stanovena, protože tyto soubory nemají vliv na přenositelnost programů, a proto se jména v adresáři `/dev` u různých výrobců liší. SVID zavádí jména:

`/dev/console` pro systémovou konzolu,  
`/dev/null` jako prázdné zařízení (kanál pro nežádoucí data nebo zdroj prázdných souborů),  
`/dev/tty` pro řídicí terminál.

Dále v SVID nalezneme odkazy na jména speciálních souborů disků, a to v konvenci, kterou jsme obecně používali v této kapitole. Soubory jsou uloženy v adresářích `/dev/dsk` a `/dev/rdsk`. Vlastní jména souborů jsou konstruována postupně jako označení pořadí disku (0 je první disk), písmeno `s` jako sekce (section) a označení sekce, např. `/dev/dsk/0s1` je první sekce prvního disku. Toto značení je ale někdy nepraktické a výrobci často zahrnují do jména také způsob rozhraní (typ řadiče) disků. Např. IRIX pro disky SCSI používá konvenci `dks` s označením pořadí a pozice SCSI. Např.

`/dev/dsk/dks0d5s7` je sedmá sekce disku na 5. pozici prvního SCSI. Konstrukce jmen speciálních souborů vzhledem k jejich významu popisuje každá implementace v provozní dokumentaci svazku (7). V adresáři `/dev` nalezneme takto popis hardwaru (`/dev/mem` je operační paměť, `/dev/kmem` je operační paměť jádra), přestože přítomnost speciálního souboru ještě neznamená, že odpovídající periferie je připojena (viz dále). POSIX se speciálními soubory nezabývá.

Využívání speciálního souboru znamená aktivaci ovladače (driveru) periferie. Ovladače periférií jsou součástí jádra (viz obr. 1.3). Každý ovladač je skupina funkcí psaných v jazyce C, které jádro aktivuje podle typu požadavku pro manipulaci s periférií. Procesy otevírají, čtou, zapisují a zavírají periférii (tj. speciální soubor) běžným voláním jádra `open`, `read`, `write` a `close`, zvláštní manipulace požadují pomocí volání jádra `ioctl`. Podle použitého volání jádra je pak aktivována odpovídající funkce ovladače, která zajistí provedení periferní operace. Skupina funkcí každého ovladače je evidována v tabulce pod určitým pořadovým číslem. Toto pořadové číslo jednoznačně určuje ovladač. Pořadové číslo je evidováno v i-uzlu speciálního souboru a říkáme mu *hlavní číslo* (major number) speciálního souboru. Tak dochází k propojení jména speciálního souboru a typu periferie. Označení pořadí periferie (jeden driver může pracovat např. s několika disky SCSI) je další významná hodnota obsahu i-uzlu. Je to *vedlejší číslo* (minor number) speciálního souboru. Hodnoty hlavního a vedlejšího čísla jsou uloženy v i-uzlu na místě odkazů na datové bloky jiných typů souborů (ve výpisu příkazu `ls -l` se objevují odděleny čárkou na místě velikosti obsahu souboru). Jak vyplývá z uvedeného, jsou to dominantní hodnoty tohoto typu i-uzlu. Přestože mají jména speciálních souborů určitý mnemonický význam, jak víme z předchozích částí kapitoly, je jméno souboru pouze část dvojice, která je převáděna jádrem na odpovídající číslo i-uzlu. Hlavní a vedlejší čísla i-uzlů jsou důležitá pro kontakt s periférií. Ve vedlejších číslech je např. kódováno pořadí a umístění sekcí disků. Význam hlavních a vedlejších čísel je také předmětem již zmíněné provozní dokumentace (7). Situaci ukazuje obr. 3.15.

Struktura i-uzlu speciálních souborů je jinak zachována. Typ i-uzlu je speciální soubor. UNIX přitom na této úrovni rozlišuje typ pro speciální soubor znakový a speciální soubor blokový (znak `c` nebo znak `b` při výpisu `ls -l`). Blokové speciální soubory zahrnují všechny typy periférií, které jsou schopny předávat a přijímat data po blocích. Typicky bloková zařízení jsou všechny typy disků. Bloková zařízení jsou ale také některé pásky. Blokové speciální soubory jsou používány pro vytváření a zpřístupňování svazků. Znakové speciální soubory jsou všechny ostatní periferie, které nemohou akceptovat přístup blokový. Přenos dat mezi nimi a uživatelskými procesy je pak uskutečňován po znacích sekvenčně, ale současně je přenášena celá sekvence znaků. Znakové periferie jsou např. terminály (`/dev/tty??`)

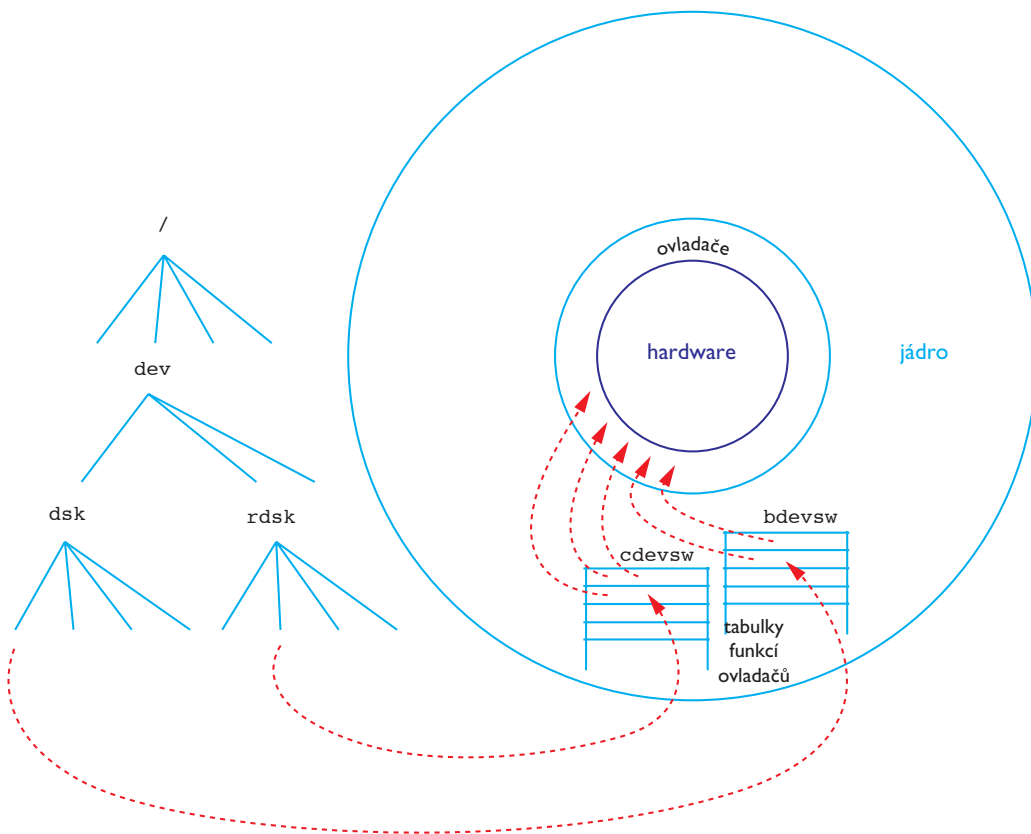
nebo tiskárny (`/dev/lp?`). Přestože typicky znakové periferie nelze ovládat jako bloková zařízení, naopak k periferiím s blokovým přístupem můžeme principálně také přistupovat jako ke znakovým zařízením. Proto každý speciální soubor blokové periferie má také svůj ekvivalent ve znakovém speciálním souboru. Znakový speciální soubor blokového zařízení je nový odkaz na periferii, tj. odkaz na jiný i-uzel, který obsahuje jiné hodnoty hlavního a vedlejšího čísla. Znaková zařízení mají vlastní tabulku funkcí ovladačů v jádru. Celý ovladač např. disku se skládá z funkcí z obsahu tabulky blokových ovladačů a tabulky znakových ovladačů (každá tabulka je pole struktur v jazyce C, jejich jména jsou `bdevsw` pro blokový přístup a `cdevsw` pro znakový přístup, viz obr. 3.15). Znakové speciální soubory jinak blokových zařízení jsou používány pro rychlejší přístup při údržbě svazků, při kopii svazků příkazem `dd` (viz kap. 10) nebo pro jiné speciální případy. Při pohledu na obr. 1.3 vidíme, že funkce ovladače se znakovým přístupem pracují přímo s periferií, kdežto blokový přístup prochází vyrovnávací pamětí (viz následující článek 3.4). To je také důležitý aspekt při práci se speciálními soubory a bude ještě několikrát v knize komentován.

V konvenci jmen speciálních souborů se jména blokových i znakových speciálních souborů konstruují stejným způsobem, ale jméno speciálního souboru má navíc první znak písmeno `r`. Je to zkratka angl. `raw`, kterou překládáme jako přímý, což má souvislost s vyrovnávací pamětí. Proto jsou např. disky rozděleny do podadresářů `dsk` a `rdsk`.

Speciální soubor vzniká pomocí volání jádra `mknod`, jehož formát jsme uvedli v odst. 3.2.1. V parametru `dev` tohoto volání je uvedeno hlavní a vedlejší číslo vznikajícího speciálního souboru. Obsah parametru `dev` je implementačně závislý, a proto je nutné při jeho používání nahlédnout do provozní dokumentace (svazek (2)). Mnohdy to však není potřeba, protože správce systému (`mknod` je privilegované) má k dispozici také příkaz `mknod`, který volání jádra využívá. `mknod` se používá pro vytváření speciálních souborů nebo roury (viz 4.2). Pro vytvoření speciálního souboru se používá ve formátu

**mknod name [cb] major minor**

kde na místě **name** je uvedeno jméno souboru, znak `c` nebo `b` použijeme pro označení typu speciálního souboru a **major** a **minor** jsou hlavní a vedlejší čísla periferie podle tabulky v jádru. Speciální soubor přitom můžeme vytvořit kdekoli jinde než v podstromu adresáře `/dev`, ale není to zvykem. Problém, který nastane při vytváření speciálního souboru, je znalost hlavního a vedlejšího čísla. Příkaz ani volání jádra totiž nezjistí, zda vytvářený speciální soubor koresponduje s tabulkou `bdevsw` nebo `cdevsw` v jádru a jakým způsobem. V systému souborů může existovat řada i-uzlů, jejichž ovladač vůbec není v jádru přítomen a odkaz do jádra není při otevírání speciálního souboru možné zabezpečit. Jádro vrácí pokus o přístup k takové periferii jako nemožný (návrátová hodnota volání jádra `mknod` je `-1`). Přesto speciální soubory mohou existovat a být vytvářeny dokonce i mimo rozsah tabulek funkcí ovladačů. Správce systému proto musí znát konfiguraci ovladačů v jádru. Ta je mu viditelná pomocí speciálních programů v podstromu adresářů, kde je vytvářeno nové jádro v případě změn modulů nebo parametrů jádra, čemuž se budeme věnovat v kap. 10. Používání hlavních a vedlejších čísel při vytváření speciálních souborů je možné nalézt také v provozní dokumentaci, svazku (7), ale vzhledem ke změnám umísťování ovladačů v jádru v nových verzích UNIXu i od téhož výrobce hlavní čísla často ani dokumentace neuvádí. Proto je správce nucen studovat konfiguraci jádra z částí tabulek generace jádra, které konečkonců rozšiřuje vždy při připojování nového ovladače.



**Obr. 3.15 Speciální soubory**

Řešení této nepříjemné situace přinášejí již dnes některé implementace. Např. při startu operačního systému AIX nebo Digital UNIX dochází k porovnání obsahu adresáře `/dev` a tabulek jádra a v případě nesrovnalostí je obsah `/dev` upraven, zůstanou v něm tak pouze speciální soubory, které mají pokračování ovladačem v jádru.

Další informace ohledně periférií, toku v/v (vstupně/výstupních) dat mezi perifériemi a procesy, obsahu funkcí ovladačů a jejich umístění v jádru a další způsoby manipulace s perifériemi obsahuje kap. 6.

## 3.4 Systémová vyrovnávací paměť

Data připojeného svazku, která proces pomocí běžných volání jádra zpřístupňuje pro potřebné manipulace, přenáší moduly jádra. Pro urychlení přenosu dat mezi svazky a procesy používá jádro *systémovou vyrovnávací paměť* (buffer cache, buffer pool). Z pohledu procesu se nic nemění, ale správci systému přináší nový aspekt při sledování a podpoře provozu. Vyrovnávací paměť je umístěna v operační paměti, a to mimo vlastní jádro. Jde o část operační paměti, která má velikost několika stovek diskových bloků. Její velikost je měnitelná změnou parametru jádra (tedy jeho novým sestavením) a je to činnost nutná po ustálení provozovaného informačního systému a všech ostatních aplikací, protože se může stát úzkým místem průchodnosti systému souborů. Jádro ve chvíli startu ještě před vytvořením prvního procesu obsadí odpovídající část operační paměti. Kopie superbloku každého svazku ve chvíli, kdy je svazek připojen, je uložena do vyrovnávací paměti a všechny změny ve struktuře svazku se promítají do této kopie superbloku. I-uzel každého otevřeného souboru je umístěn ve vyrovnávací paměti. Diskové bloky obsahu tohoto i-uzlu jsou při práci umísťovány do vyrovnávací paměti, a to ještě předtím, než proces požádá o přenos dat ze souboru, protože jádro předpokládá sekvenční čtení obsahu souboru. I-uzly i datové bloky souboru dokonce zůstávají ve vyrovnávací paměti bez přepisu na svazek i tehdy, když proces soubor uzavře. Superblok zůstává ve vyrovnávací paměti po celou dobu připojení svazku a přepis se provádí v okamžiku odpojení svazku (provede `umount` před vlastní operací odpojení). Data změněných diskových bloků systémové vyrovnávací paměti jsou aktualizována na svazky teprve při nedostatku volné kapacity vyrovnávací paměti při požadavku přenosu nových dat ze svazku do oblasti zpracování procesem. Jádro tehdy aktivuje algoritmy správy vyrovnávací paměti, které hledají vhodného kopie diskových bloků pro přepis na svazky a jejich uvolnění pro další použití. Jádro proto provádí správu vyrovnávací paměti tak, aby přibližování dat směrem od svazků k procesům bylo optimální. Je pochopitelné, že zvětšením vyrovnávací paměti správce jádra jeho úlohu ulehčí, ale plýtvání operační pamětí je pak také na úkor prostoru přidělovaného procesům. Umístění vyrovnávací paměti ukazuje obr. 3.16.

Vynucený přepis celého obsahu vyrovnávací paměti včetně všech superbloků dosáhne kterýkoliv proces použitím volání jádra `sync`. Má jednoduchý formát

```
void sync(void);
```

a není uveden v POSIXu. Jádro na základě tohoto pokynu spouští algoritmy přepisu celého obsahu vyrovnávací paměti. Pracuje-li systém ve víceuživatelském režimu, je přítomen také démon, který cyklicky vždy každých 30 vteřin volání jádra `sync` provede. Každému uživateli je také dostupný příkaz

### \$ `sync`

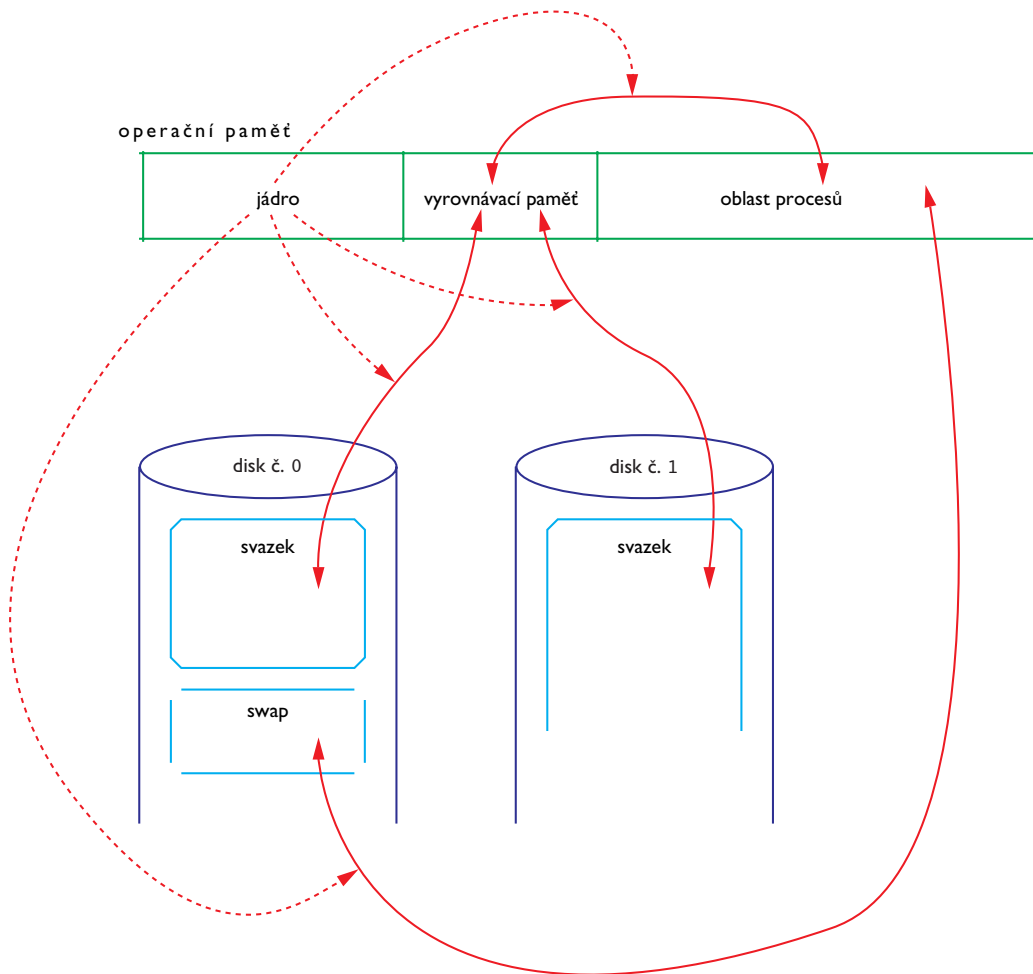
který volání jádra, a tím synchronizaci dat vyrovnávací paměti a všech svazků provede.

Proces má také možnost požádat o synchronizaci dat konkrétního otevřeného souboru. K tomu slouží volání jádra (které POSIX obsahuje)

```
#include <unistd.h>
int fsync(int fildes);
```

v jehož parametru `fildes` proces zadává odkaz do tabulky otevřených souborů a jádro pak provede přepis změněných dat ve vyrovnávací paměti na svazek související s odpovídajícím souborem. Zatímco





Obr. 3.16 Systémová vyrovnávací paměť

pomocí `sync` proces pouze spouští algoritmy synchronizace, instrukce následující za `fsync` je provedena teprve po dokončení přepisu dat vyrovnávací paměti na disk.

Proces má pak ještě jednu možnost synchronizace dat konkrétního souboru. Je to využití příznaku `O_SYNC` v argumentu `oflag` volání jádra `open` při otevření souboru (viz odst. 3.1.1). Všechny změny obsahu a atributů souboru po celou dobu otevření souboru jsou ihned synchronizovány. Znamená to také, že proces pokračuje následující instrukcí za voláním jádra `write` teprve po zajištění průchodu dat přes vyrovnávací paměť na svazek.

Umístění superbloků připojených svazků v systémové vyrovnávací paměti ukazuje na slabé místo ukládání dat. V případě, že dojde k havárii jádra (ať už výpadkem el. proudu nebo z důvodu kolize hardwaru nebo softwaru jádra), je obsah vyrovnávací paměti ztracen a konzistence dat na svazcích je ve zpoždění od posledního **sync**. Konzistenci vnitřní struktury svazku obnovuje program **fsck**, záchranu dat při zhroutěném svazku **fsdb**. Výpadky vyrovnávací paměti jsou také důvodem provádění kopií superbloku v organizaci svazků typu **bsd**.

Při požadavku čtení nebo zápisu dat procesem jádro nejprve zjišťuje, zda je odpovídající blok umístěn ve vyrovnávací paměti, což je možné, protože i-uzel, kde jsou reference datových bloků, je po otevření souboru jádru znám. Je-li blok ve vyrovnávací paměti, procesu jsou nabídnuta jeho data. V opačném případě je zajištěno vyhledání bloku na svazku a provedení jeho kopie. Jádro udržuje obsah systémové vyrovnávací paměti jako dva obousměrně zřetězené seznamy jednak volných a jednak obsazených kopií diskových bloků. Ze seznamu volných kopií čerpá, dokud je neprázdný a v opačném případě spouští algoritmus, který uvolní kopie bloků z obsazeného seznamu metodou nejdéle nepoužitý. Prvek seznamu obsahuje tzv. záhlaví kopie bloku a vlastní data bloku. V záhlaví jsou jednak položky nutné pro připojení k seznamu a dále identifikace svazku (zařízení) a adresu bloku ve svazku. Pro lepší výkon algoritmů vyrovnávací paměti se snaží jádro udržovat seznam všech bloků vyrovnávací paměti v několika klíčovaných frontách. Způsob klíčování je prováděn tak, aby byl zajištěn nejlepší možný způsob zjištění, zda je hledaný blok umístěn ve vyrovnávací paměti nebo ne, a záleží na implementaci. Je pochopitelné, že záhlaví kopie bloku musí také obsahovat položky, jako je příznak modifikace obsahu bloku nebo příznak pro zamknutí obsahu bloku (je možný souběh, tj. přístup dvou procesů k témuž souboru) a algoritmy musí řešení problémů analogických řešit ve svých algoritmech správy vyrovnávací paměti. Správce systému tuto správu systémové vyrovnávací paměti, až na velikost, případně počet klíčovaných front (při regeneraci jádra) neovlivní.

## 3.5 Archivy dat

Pro úschovu dat na externích médiích se sekvenčním přístupem (např. na magnetické pásky) byly v raných dobách UNIXu vytvořeny programy pro zálohování dat svazků. Je to zejména program **tar** (tape archiver) a později vyvinutý **cpio** (copy file archives in and out). Oba programy podle požadavků uživatele čtou obsah a atributy uvedených souborů a vytvářejí z nich sekvenční proud dat, který je zapisován na odpovídající periférii uvedeného speciálního souboru. Sekvenční seznam souborů včetně jejich obsahu je nejpoužívanější způsob zálohování dat v UNIXu. Formát ukládání dat programem **tar** byl dokonce velmi brzy uveden v doporučujících dokumentech (SVID nebo X/OPEN), takže se stal běžným prostředkem přenosu dat na výměnném médiu periferie mezi jednotlivými instalacemi třeba i různých typů UNIXu.

### 3.5.1 Program tar

Formát sekvenčního proudu dat vytvářeného programem **tar** je uveden ve (4) provozní dokumentace a v **<tar.h>**. **tar** vytváří z atributů (včetně jména) každého archivovaného souboru tzv. hlavičku souboru. Za každou hlavičkou následuje obsah souboru. Další archivovaný soubor pak následuje, nejprve hlavička s uvedením atributů a po ní obsah souboru. Teprve celý vytvořený archiv je ukončen značkou konce souboru EOF<sup>7</sup>. Tvar archivu ukazuje obr. 3.17.

Z obrázku je patrné, že archiv není zahájen žádným celkovým soupisem obsahu celého archivu, dokonce nemá ani pojmenování. Rozeznat obsah takového archivu pak znamená čtení postupně každé hlavičky všech archivovaných souborů, tj. procházení celého archivu (které může mnohdy znamenat i několik desítek minut). Přestože odpůrci UNIXu již dvě desetiletí tento formát zesměšňují (možná i právem), je pro svoji jednoduchost a srozumitelnost stále používán nejvíce.

Použití **taru** je jednoduché. Uživatel zadává jednu ze tří možností, vytvoření (**c**reate) archivu **dat** (volba **c**), obnovení (**e**xtract) **dat** z archivu (volba **x**) nebo výpis (**t**able) obsahu archivu (**t**). Např.

#### \$ **tar c soubor**

je vytvoření archivu, který bude obsahovat jeden **soubor** jako sekvenci bytů atributů **souboru** a jeho obsahu. Archiv bude vytvořen a zapsán na periférii, která je stanovena jako implicitní správcem systému, obvykle magnetická páska. Pro obnovení archivovaného souboru zadáváme příkaz

#### \$ **tar x soubor**

Původní **soubor** je **souborem** z archivu přepsán (jak je v UNIXu zvykem, bez varování), pokud jsme mezitím nezměnili adresář (nebo operační systém). **soubor** ale může být zadáván také se zápisem cesty k němu. Jde-li tedy o cestu absolutní, tj. jméno souboru začíná znakem /, je jméno v archivu evidováno v této podobě a při obnovení je soubor obnoven přesně do místa stromu adresářů, odkud byl archivován. Zní to dobře, ale praktické výsledky mohou být katastrofální, protože obnovit potřebujeme obvykle pouze část archivu, a to navíc pro porovnání s verzí, kterou máme na svazku, a ta je archivem přepsána. Absolutní jména souboru se proto používají pouze při vytváření instalačních archivů nebo při globální úschově dat svazků a obyčejný uživatel je nepoužije.

Konečně příklad výpisu archivu na implicitní periférii lze pořídit pomocí

#### \$ **tar x**

Na standardní výstup je vypsán seznam souborů podle obsahu jednotlivých hlaviček archivu.

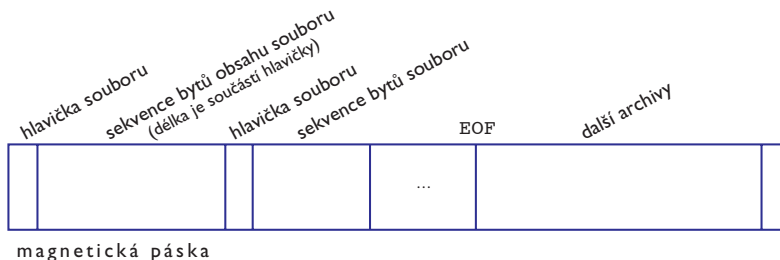
Příkazový řádek při vytváření nebo obnově (konečně i při výpisu archivu) může pokračovat seznamem dalších jmen souborů pro archivaci nebo obnovu. Archiv pak obsahuje více souborů a při obnově si můžeme vybrat pouze některé soubory, a to jejich vyjmenováním. Jméno musí být uvedeno v archivované podobě, soubor může být archivován např. s relativním odkazem nebo absolutní cestou, jak jsme již uvedli. Nejsme-li si jisti, nejprve archiv prolistujeme volbou **t**. **tar** pracuje pro archiv celého podstromu. Patří-li jméno souboru adresáři, **tar** archivuje všechny soubory, které jsou v něm evidovány, a pokud jsou některé opět adresářem, archivuje také jejich obsah atd. do libovolné hloubky vnoření. Vytvoření archivu celého podstromu daného adresářem je tedy jednoduché:

#### \$ **tar c adresář**

Obnovení je analogické a týká se opět celého podstromu. Běžně se také používá

#### \$ **tar c \***

což je archiv obsahu všech souborů pracovního adresáře. Znak **\*** je ale řídicím znakem pro shell, nikoliv pro **tar**. Příkazový řádek je analyzován přijímajícím procesem **sh**, který **\*** nahradí seznamem všech jmen souborů pracovního adresáře (archiv je tedy proveden i pro všechny případné podadresáře atd.). To je dobré si promyslet, protože při obnově archivu **\*** neznamena všechny soubory archivu, ale všechny



Obr. 3.17 Obsah archivu tar sekvenčního magnetického média

soubory podle jmen pracovního adresáře, a to je rozdíl. Také proto při obnově celého archivu **tar** nabízí variantu

**\$ tar x**

Důležitá volba příkazu **tar** je změna periferie archivu. Chceme-li potlačit archivaci na implicitní periferii a označit pro archivaci jinou, používáme k tomu volbu **f** (file). Argument bezprostředně následující za volbou je pak jméno speciálního souboru požadované archivní periferie, např.

**\$ tar cf /dev/mt2 soubor**

provede vytvoření archivu **souboru** na zařízení speciálního souboru 3. magnetické pásky. Obnova nebo výpis archivu bude analogický. Speciální soubor přitom může být jakákoliv periferie, která je schopna ukládat sekvenčně data. Lze použít i znakový speciální soubor diskových periférií (disket, výměnných disků, sekcí pevných disků atd.). Konečně lze použít pro archiv i obyčejný soubor, např. příkazem

**\$ tar cf archiv soubor**

vzniká v pracovním adresáři soubor se jménem **archiv**, který bude obsahovat **soubor** pracovního adresáře. Použijeme-li na místě **souboru adresář**, vznikne **archiv** celého podstromu. Obyčejný soubor **archiv** bývá často vytvářen s příponou **.tar**, protože metoda archivace do obyčejného souboru slouží ke snadnému přenosu celého podstromu adresářů sítí (viz kap. 7) a v Internetu je to jedna z metod distribuce zdarma šířeného programového vybavení, protože po přenosu archivu znamená instalace obnovení archivu v cílovém uzlu. Uživatel Internetu tak mnemonicky snadno rozpozná, který soubor je určen pro přenos a instalaci balíku dat.

Označení archivu může dále být standardní výstup. Takové použití slouží pro přesměrování buď do souboru nebo na vstup procesu, který formátu archivu rozumí, tj. nejčastěji opět procesu **tar**. Příkaz ekvivalentní naposledy uvedenému je

**\$ tar cf - soubor > archiv**

Znak **-** je označení standardního výstupu, ale **-** může znamenat i standardní vstup. Každý popis příkazu **tar** uvádí možnost kopie podstromu pomocí dvou spojených procesů **tar** takto

```
$ ( cd odkud; tar cf - . ) | ( cd kam; tar xf - )
```

**odkud** a **kam** jsou adresáře, oba musí existovat. Příkazem je obsah **odkud** okopírován do **kam** včetně všech podadresářů. Kulaté závorky znamenají interpretaci novým procesem shellu, což je nutné, protože každý z procesů **tar** musí mít nastaven jiný pracovní adresář.

V příkazovém řádku **taru** lze používat tzv. doplňující volby. Jednou z nich je uvedená volba **f**, další je např. **v** (verbose, užvaněnost), kterou používáme pro podrobný výpis právě probíhající akce programu (nejčastěji používaná ve spojení s volbou **t**) kdy je každý přenášený soubor uváděn s výpisem jeho atributů. Výpis připomíná výpis příkazu **ls -l**. Důležitá je volba **L**, **tar** sleduje symbolické odkazy a do archivu ukládá soubory, na které je odkazováno, jinak je uložen pouze symbolický odkaz. Stejně tak je zajímavá volba **o**. Pomocí ní totiž není dodržen při obnově vlastník a skupina evidovaná v hlavičce každého souboru archivu, ale soubory jsou vytvářeny s vlastnictvím pracujícího uživatele (tj. vlastníka a skupiny procesu **tar**). Volba **m** mění datum a čas souborů při obnově na aktuální oproti uvedenému v archivu.

Mezi hlavní volby dokumentace ještě zahrnuje **r**. Jde o připojení dalších souborů k již vytvořenému archivu, tj. odstranění EOF, připojení další sekvence dat a opět ukončení znakem EOF. Hlavní volba **u** rovněž připojuje k již existujícímu archivu, testuje ale dříve přítomnost téhož souboru v archivu. Připojení souboru provede pouze v případě, že v archivu prozatím není nebo byl oproti dřívější archivaci modifikován. Volba **u** implikuje **r**.

Připojování k archivu je složitější a nebezpečnější operace, než se na první pohled zdá. Na obr. 3.17 je v části za archivem dat uveden text „další archiv“. Rozumí se tím, že magnetická páska (nebo jiná periferie se sekvenčním ukládáním dat, ale ne obyčejný soubor) může obsahovat více archivů. V takovém případě je ale rozšiřování předcházejících archivů nebezpečné, protože **tar** nemůže vsunout připojovaná data mezi konec a začátek dvou archivů, a proto začátek následujícího archivu přepíše, a tím znemožní jeho další čtení (data na pásce obvykle zůstanou a lze je číst obtížně, jinými prostředky než programem **tar**, např. pomocí **dd** a analyzovat jejich obsah podle formátu v <**tar.h**>. Ukládáním více archivů v různých formátech za sebou na sekvenční magnetické médium se budeme zabývat v kap. 6, kde se seznámíme s programem **mt**, který umožní s magnetickou páskou manipulovat po jednotlivých archivech.

### 3.5.2 Program **cpio**

Výsledný formát archivu programu **cpio** je obdobou programu **tar** (definice jsou uvedeny v <**cpio.h**>). I zde jde o vytvoření sekvenčního toku bytů ze skupiny souborů pro archivaci. Na rozdíl od svého předchůdce (**taru**) má ale několik nových vlastností. Když pomineme formát příkazového řádku, je konstruován pro archivaci vybrané množiny souborů z různých částí hierarchické struktury adresářů. **tar** naopak vždy respektuje celý podstrom dat. **cpio** je výkonnější, vyrovná se chybnými místy archivu, **tar** při objevení nesrovnalosti v uložených datech končí (obvykle s chybovou zprávou **Directory checksum error**). Přestože přenositelnost archivů vytvořených v **cpio** nebyla od jeho prvních verzí jednoznačná, v současné době je čitelnost archivů pořízených v jiných systémech zaručena. **cpio** je i přes své nesporné výhody používán méně než **tar**. Možná je to i z důvodu méně jasného příkazového řádku při jeho používání.

**cpio** uživatel používá ve třech základních tvarech. Volbou **-o** vytváří archiv. Z archivu data vybírá volbou **-i**. Použitím třetího tvaru (volba **-p**) nepracuje uživatel s archivem, ale pouze kopíruje soubory do určitého adresáře. První dva tvary pracují s archivem, třetí je pomocný. Při vytváření archivu požaduje **cpio** na standardním vstupu seznam souborů k archivaci. Každé jméno souboru je umístěno na samostatném řádku, může být zadáno relativně i absolutně. Běžně lze používat výpis programu **ls**, např.

```
$ ls | cpio -o > /dev/mt0
```

Archiv je vytvářen na standardní výstup, a proto uživatel musí určit sekvenční periférii speciálním souborem. Podobně jako u **taru**, i zde můžeme archiv ukládat do obyčejného souboru. Archivovány jsou pouze soubory a adresáře, nikoliv podadresáře a jejich obsah. **cpio** bývá nejčastěji používáno ve spojení s příkazem **find**, který dokáže vyhledat v zadaných podstromech soubory na základě určitých atributů (atributem může být holé jméno souboru nebo také datum poslední změny v rozsahu např. několika posledních dnů atd.) a jména těchto souborů vypíše na standardní výstup. Např.

```
$ find /usr2 /usr3 -name '*' -mtime 3 -print | cpio -o > /dev/mt0
```

je archivace všech souborů podstromu **/usr2** a **/usr3**, jejichž obsah byl změněn v posledních 3 dnech.

Data se z dříve vytvořeného archivu obnovují použitím volby **-i**. **cpio** očekává archiv na standardním vstupu. Soubory ukládá do pracovního adresáře, pokud nebyly archivovány s absolutní cestou. V případě, že je pro obnovu potřeba vytvořit určitý chybějící adresář, musí být v příkazu uvedena ještě volba **-d**. **cpio** na rozdíl od běžného zvyku v UNIXu respektuje existující soubory. Obnovu provádí pouze v případě, že soubor stejného jména v adresáři není obsažen. Přepis existujících souborů na svazcích lze vynutit použitím volby **-u**. Např.

```
$ cpio -idu < /dev/mt0
```

Výpis obsahu archivu je součástí použití **-i**, a to ve spojení s volbou **-t** :

```
$ cpio -it < /dev/mt0
```

Jak lze očekávat, žádná obnova v tomto případě není provedena. Program **cpio** na rozdíl od programu **tar** dokáže akceptovat expanzní znaky jmen souborů pro určení pouze části obnovy z archivu. Lze je použít v dalších parametrech příkazu, které určují seznam požadovaných souborů pro obnovu, např.

```
$ cpio -i '*.html' '*.c' < /dev/mt0
```

obnoví všechny souboru archivu, jejichž jména končí na **.html** nebo **.c** (expanzní znaky musí být v příkazovém řádku zbaveny speciálního významu, protože by je interpretoval shell a **cpio** by obdržel již seznam jmen souborů, nikoliv regulární výrazy). Pokud není uživatel privilegovaný, obnova z archivu probíhá s nastavováním vlastníka a skupiny používaného uživatele. Pro privilegovaného uživatele je vlastník i skupina zachován. Přístupová práva jsou přenesena z archivu. Pro užívanou obnovu může uživatel použít volbu **-v**.

**cpio** je používán také pro pouhou kopii určených souborů do jiného místa daného adresářem. Je to možné pomocí volby **-p**, např.

```
$ find . -name '*' -print | cpio -pd $HOME/save
```

kopíruje obsah podstromu pracovního adresáře do adresáře **save**, který je podadresářem domovského adresáře uživatele. Adresář v příkazovém řádku je povinným argumentem.

**cpio** podporuje přenositelnost archivu volbou **-c** (hlavičky souborů jsou archivovány v ASCII), ale také volbou

**-H**, za kterou může následovat řetězec **odc**, **crc**, **tar** nebo **ustar**, které připouští SVID. **cpio** tak zahrnuje možnost kompatibility také např. na program **tar**. Vývoj je ale v POSIXu definován jiným způsobem, a to definicí programu **pax**.

### 3.5.3 Archivy programu pax

Uvedený výklad archivních programů **tar** a **cpio** je plně ve shodě s SVID. POSIX ale žádný z nich nedefinuje v uvedené podobě. Součástí normy POSIX je totiž příkaz **pax** (portable archive interchange), který oba programy nahrazuje a sdružuje, **pax** není stále běžně implementován; co však není, brzy bude (viz IRIX 6.2).

V samotné definici programu **pax** se mluví o poskytnutí archivu v různých tvarech. Ve standardu je definováno použití **ustar** nebo **cpio** jako součást volby **-x**, (**ustar** je rozšířený **tar**, používán v progresivních systémech). **pax** podporuje implicitně formát vlastní, který je uveden a je komentován jako „pokojný kompromis mezi zastánci historických programů **tar** a **cpio**“. Způsob ovládání vychází opět ze základních tvarů pro vytvoření archivu (volba **-w**), jeho čtení (**-r**), získání jeho seznamu (pokud není ani **-r** ani **-w** uvedena) a jako doplněk kopie skupiny souborů (současně **-r** i **-w**).

Použití je velmi podobné manipulaci s již uvedenými programy a uživatel se jistě bude dobře orientovat v provozní dokumentaci, pokud je účastník systému, který **pax** vlastní.

### 3.5.4 Další způsoby archivu (dump a restor)

Správce systému obvykle potřebuje archivní programy úrovně archivace celého svazku, nikoliv skupin určených souborů. UNIX od svého vzniku tento způsob archivu podporoval, přestože v průběhu jeho největšího rozšíření v 80. letech programy **dump** a **restor** (ne, nechybí mi tam **e**) většina komerčních systémů vypustila. V dnešních systémech se znova objevují, bohužel pod různými jmény a s různým formátem příkazového řádku (přestože SVID definuje sadu programů se jmény pro takovou archivaci, jako jsou **backup**, **fdisk** atd., jsou velmi málo implementovány a používány)<sup>8</sup>. Společná však zůstává metoda archivu obsahu odpojeného svazku na archivní médium. Svazek je archivován a obnoven jako celek včetně jeho vnitřní organizace (obsahu i-uzlů atd.). Přenositelnost takového archivu je tedy prakticky nemožná, protože je archiv silně závislý na použitém typu svazku. Vzhledem k tomu, že se jedná prozatím o nejednotnou manipulaci se systémem souborů, a také proto, že jde o přístup pouze správce systému, budeme se odpovídajícími programy zabývat v kap. 10.

<sup>1</sup> Terminologie je vágní. Často se úplné jméno souboru označuje jako cesta (path) k souboru. Je to abstrakce, adresář je totiž z pohledu systému souborem. Cesta je proto seznam všech souborů, jejichž spojení přivádí proces na správné místo ve stromu adresářů.

<sup>2</sup> POSIX definuje strukturu `DIR` pouze o jednom členu

```
char d_name[ {NAME_MAX} ] ;
```

i-uzel považuje za implementačně závislou entitu, přestože při volání jádra `stat` má struktura `buf` definovanou položku `st_ino`; je ovšem slovně komentována jako sériové číslo souboru.

<sup>3</sup> Zda funkce `readdir` čte položky tečka nebo tečka-tečka adresáře, POSIX ani SVID3 nedefinují a ponechávají na konkrétních implementacích.

<sup>4</sup> POSIX nepodporuje symbolické odkazy.

<sup>5</sup> POSIX neuvádí symbolický odkaz

<sup>6</sup> Každý proveditelný soubor obsahuje program v binárním tvaru, který je schopen být prováděn procesorem, ale za podpory operačního systému, jádro si musí vystačit samo; jádro je tzv. samostatně proveditelný program, standalone program.

<sup>7</sup> Některé verze systému končí archiv dvěma `EOF`.

<sup>8</sup> POSIX definici prostředí správy operačního systému teprve připravuje.



## 4 KOMUNIKACE MEZI PROCESY

Velmi často dochází k situacím, kdy činnost jednoho procesu závisí na činnosti procesu jiného. I samotná koncepce návrhu programového prostředí v UNIXu [KernPlau76] vychází z procesu jako jednotky výpočtu, vstupy jednoho procesu jsou výstupy procesu jiného v programovém systému aplikace. A pro spolupráci několika procesů je nutno definovat rozhraní jejich komunikace.

Způsoby komunikace procesů tak, jak je uvedla první veřejná verze UNIXu v r. 1979 (UNIX Version 7), byly uvedeny v kap. 2. Jejich používání jsme přitom předváděli v prostředí odpovídajícím současným obecným dokumentům SVID a POSIX. Byly uvedeny, protože úzce souvisí se vznikem a zařazením procesu do stromové struktury procesů. Jedná se o *signály* (čl. 2.2) a *rouru* (odst. 2.5.1). V původní koncepci byla přitom pouze roura zjevně navržena pro spojení dvou procesů a jednosměrný přenos dat mezi nimi.

Signály byly navrženy pro synchronizaci procesů při výskytu určité neobvyklé události v systému a pro samotný proces to znamenalo obvykle pokyn k ukončení, k čemuž skutečně došlo, pokud se proces sám dříve nerozhodl událost přežít. Signály souvisí také s existenční závislostí na rodiči procesu. Signál zaslaný vedoucímu skupiny procesů znamenal zánik celé skupiny. Teprve rozšíření UNIXu přineslo tlak na vývoj systému tak, že byly signály použity pro obecnou synchronizaci procesů. Byly zavedeny signály SIGUSR1, SIGUSR2 a další (z původního počtu 14 jich dnes SVID definuje celkem 28), jejichž vyslání v mnoha případech už neznamenal ukončení cílového procesu v případě, že tento signál proces neočekával, ale např. jeho ignorování (viz Příloha B). Signály jsou dnes stále dominantní v synchronizaci procesů, ať už systémových nebo aplikačních. Problémy, které při jejich používání programátorům nastávají, vychází z jejich původní koncepce. Je např. obtížné zjistit obecně PID libovolného běžícího procesu. Proces má prostředky pro evidenci PID svých dětí (v návratové hodnotě `fork`), může zjistit své PID (`getpid`), svého rodiče (`getppid`) nebo vedoucího své skupiny (`getpgrp`), ale další prostředky nemá. Seznam procesů evidovaných jádrem včetně jejich PID lze pořádit příkazem `ps`, který pracuje buďto přímo s pamětí jádra (speciální soubor `/dev/kmem`) nebo se svazkem `/proc`. Přitom není definováno žádné volání jádra, které by `ps` používal. V případě, že jej chtějí jiné procesy napodobit, musí mít přístup k uvedeným systémovým zdrojům, a ty podléhají privilegovanému přístupu. Přestože takovým způsobem lze skutečně PID procesů a jejich atributy získat, proces hledající svůj protějšek komunikace se v těchto informacích může orientovat jen obtížně, protože nemá k dispozici žádný rozumný aparát selekce v seznamu procesů (nehledě na to, že jde o poměrně kostrbatý přístup k informacím). Procesy proto používají signály především v rámci skupiny procesů a při snaze synchronizovat se s procesem mimo tuto skupinu zveřejňují své PID na smluveném místě v systému. Takové místo může být pochopitelně i soubor, ale problémy nastávají s jeho údržbou při nestandardním ukončení procesů nebo celého operačního systému, a proto jsou k výměně PID procesů dnes používány běžně např. i fronty zpráv.

Pro spojení dvou procesů za účelem jednosměrného toku dat byla v původním UNIXu navržena a implementována roura (`pipe`). Její princip, použití a programování byl uveden v odst. 2.3.3 a v odst. 2.5.1. Přestože jde o elegantní způsob spojení dvou procesů, princip návrhu přináší také několik nevýhod. První je opět v omezení na skupinu procesů. Roura je totiž vytvářena voláním jádra `pipe`, které obsadí 2 pozice tabulky otevřených souborů procesu. Takto vzniklou rouru dědí každé dítě, a to se může na rouru připojit pro čtení i zápis, stejně jako se na ni může připojit rodič, ale nikdo jiný, protože

roura je součástí dědictví, které nelze exportovat do prostředí jiného než dětského procesu. Obecně proto nelze zajistit spojení dvou libovolných procesů. Snaha odstranit tuto nevýhodu vedla k vytvoření tzv. pojmenované roury (named pipe), která je aktivována pomocí otevření souboru (pomocí `open`), jehož i-uzel má typ roury. Spojení jednosměrného kanálu se jménem souboru tak umožnilo přístup k rouře libovolným procesům. Druhá nevýhoda původní roury je také v její jednosměrnosti. Proces KONZUMENT pouze přijímá data a PRODUCENT je pouze zapisuje. Opačný tok dat není možný, nehledě na vzájemnou synchronizaci toku dat. Implementace obousměrné roury byla řešena pomocí technologie STREAMS na cestě hledání principů implementace síťových protokolů a dnes je možné používat obyčejnou rouru jako obousměrný tok dat. Konečně třetí nevýhoda roury jsou chybějící mechanismy zajištění struktury přenášených dat, tj. protokolární komunikace. Procesy sice mohou takovou strukturu vytvářet v rámci vlastní interpretace přenášených dat, ale obecně je tok dat rourou, a to pojmenovanou či nepojmenovanou, pouze sekvence bytů.

Původní implementace UNIXu také žádným způsobem neumožňovaly procesům pracovat s pamětí jejich datové oblasti, která by jim byla společná, tzv. sdílená paměť. Procesy mohly data pro sdílení ukládat do souboru, ve kterém se mohly sice efektivně pohybovat pomocí volání jádra `lseek`, ale tento způsob je zatížen režii systému souborů a pro rychlé výpočty (např. při grafických operacích) se proto nehodí. Navíc původní implementace neřešila zamykání souborů nebo jejich částí. Uvedené nedostatky při používání komunikace procesů pouze pomocí signálů a roury vedly k rozšíření UNIXu o tři další způsoby komunikace, obecně označované zkratkou IPC (Interprocess Communication).

IPC rozšiřuje komunikaci procesů o tři nové způsoby: *fronty zpráv* (messages), *sdílenou paměť* (shared memory) a *semafor* (semaphores). Fronty zpráv využívají procesy pomocí volání jádra `msgget`, `msgsnd`, `msgrcv` a `msgctl`. Tato komunikace procesů umožňuje pomocí záhlaví zpráv dávat přenášeným datům strukturu a programátor tak získává možnost definovat komunikační protokol. Sdílená paměť je možnost pracovat nad daty společnými několika různým procesům. Mechanismus volání jádra `shmget`, `shmat`, `shmdt` a `shmctl` umožňuje sdílenou paměť vytvořit, připojit se k ní a pracovat s ní jako s běžně adresovanou datovou oblastí procesu. Konečně získávat exkluzivní přístup obecně k určitému výpočetnímu zdroji, synchronizovat jeho používání při několikanásobném přístupu různých procesů a zamezovat tak uváznutí v kritických sekcích (deadlock) řeší implementace Dijkstrových semaforů. Proces s nimi manipuluje pomocí volání jádra `semget`, `semop` a `semctl`. Využívání všech tří způsobů komunikace uvedeme v průběhu kapitoly.

IPC znamená obecnou komunikaci více procesů. K tomu, aby se dva libovolné procesy lokálního operačního systému (různého PID, a to v různých instancích svého běhu) dokázaly shodnout na identifikaci prostoru, přes který spolu budou komunikovat, je při vytváření komunikačního prostoru IPC (tj. fronta zpráv, sdílená paměť nebo semafor) používána celočíselná identifikace, tzv. klíč IPC. Klíč je používán jednak při vytváření prostoru komunikace, ale také jej používá každý proces, který se potřebuje na prostor připojit; jak vytvoření, tak připojení zajišťují volání jádra `msgget`, `shmget` a `semget`. Klíč IPC v nich používaný je typu `key_t`, má rozsah 32 bitů a používá ji každý proces, který potřebuje s prostorem pracovat. Přestože rozsah identifikace je příslibem malého procenta kolizí, kolize se tím samy o sobě neřeší. Kolizí zde rozumíme shodu klíčů dvou různých komunikací víceprocesových aplikací. Pokud totiž programátor aplikace použije identifikaci např. 15, není zaručeno, že operační systém sám tuto identifikaci změní v případě, že taková již byla použita v jiné skupině procesů (jiné aplikaci). Dochází ke shodě identifikací a ke kolizi, jejíž důsledky závisí na činnosti jednotlivých skupin

procesů. Pro řešení této situace, tj. jednoznačné stanovení celočíselné identifikace prostoru komunikace neboli klíče, je v UNIXu používána funkce `ftok` (*file to key*). Funkce umí na základě jména existujícího souboru vytvořit jedinečný klíč pro konkrétní instalaci v konkrétním spuštění aplikace. Pokud v téže instalaci bude spuštěna aplikace jiná a při identifikaci použije jiné jméno souboru ve funkci `ftok`, je zajištěno, že hodnota identifikace prostoru (klíče) bude jiná. Funkce je součástí SVID (ale ne POSIXu) a má formát

```
#include <sys/types.h>
#include <sys/ipc.h>

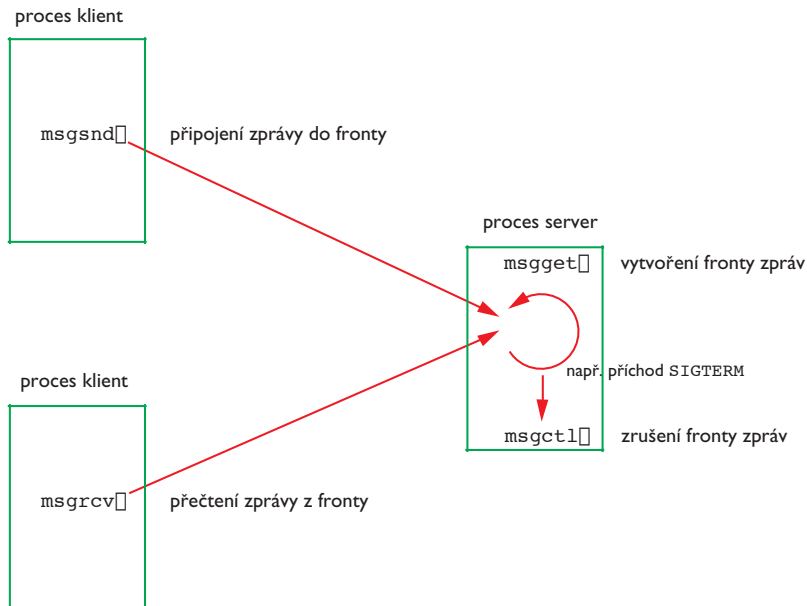
key_t ftok(const char *path, int id);
```

Jméno souboru na místě `path` označuje existující soubor. Je to proto, že obvyklá implementace `ftok` je založena na využití čísla i-uzlu daného souboru (pozor na různá jména téhož i-uzlu, generovaný klíč bude pak tentýž). Pro generaci klíče, který se objeví v návratové hodnotě, `ftok` dále může používat hlavní a vedlejší číslo sekce disku svazku se souborem `path`, a tím získá jedinečnou celočíselnou hodnotu v rámci instalace systému. Programátor může dále používat číslování prostorů komunikace v parametru `id` v případě, že skupina procesů komunikuje více různými způsoby. Uvedené výchozí hodnoty pro výpočet klíče při mechanickém skládání převyšují rozsah 32 bitů (hlavní a vedlejší číslo i-uzlu po 8 bitech, `id` 16 bitů, samotné číslo i-uzlu 32 bitů), proto je v současných implementacích ignorováno např. hlavní číslo, i-uzel se nepředpokládá vyšší než 16 bitů atd. Důležité je však zajištění výsledku, kdy na základě jména souboru `path` a číselné identifikace `id` bude vždy vytvořen jedinečný klíč k označení prostoru komunikace. Při programování je běžný postup, kdy proces, který je odpovědný za vytvoření prostoru komunikace, vytváří soubor `path` (v případě jeho dřívější existence např. hlásí kolizi) a používá jej pro vytvoření klíče a následně prostoru komunikace IPC. Ostatní procesy pak podle souboru vytvářejí klíč a připojují se. Příklady jsou uvedeny v dalších částech kapitoly.

Jádro registruje všechny aktivity IPC. Uživatel má právo seznámit se s těmi aktivitami, které mu jsou dostupné pro čtení (privilegovaný čte všechny), a to pomocí příkazu **ipcs**. Příkaz na standardní výstup vypisuje tabulku existujících prostorů komunikace s jejich atributy. Pokud uživatel potřebuje některý prostor odstranit, může používat příkaz **ipcrm**, má-li potřebná oprávnění. **ipcrm** používá uvedená volání jádra, **ipcs** pak obsah tabulek jádra (ve speciálním souboru `/dev/kmem` nebo informace svazku `/proc`). Podrobnosti potřebné k používání obou příkazů uvedeme v závěru kapitoly (odst. 4.8).

Při využívání technologií IPC je uplatňována tzv. architektura *klient-server* (client-server). Serverem rozumíme automatizovaného správce a poskytovatele určitého výpočetního zdroje, klientem naopak mechanismus, který služby serveru využívá. Pod pojmem architektura klient-server pak rozumíme komunikaci mezi nimi, která má za cíl klient uspokojit. V pojmech UNIXu a IPC je serverem proces, který spravuje prostor komunikace (vytváří jej, nastavuje přístupová práva, naplňuje informacemi, uzavírá a ruší jej atd.). Klient je každý proces, který se na prostor připojuje a využívá jej ke své další činnosti (čte nebo zapisuje informace). Použijeme-li jako příklad fronty zpráv, server bude procesem, který frontu zpráv mimo jiné vytvoří a nakonec zruší. Klient je každý proces, který frontu zpráv využije pouze pro čtení nebo zápis zprávy. Použitá volání jádra mají typický charakter podle obr. 4.1.

Z obrázku je patrné, že komunikace se účastní obecně více procesů v roli klientů, ale pouze jeden proces jako server. Architektura klient-server je tedy také zabezpečení řízeného přístupu k místu spolupráce více procesů.



Obr. 4.1 Architektura klient-server při využití fronty zpráv

Komunikace procesů při využití architektury klient-server je dominantní v síťové komunikaci (viz kap.7). Základní princip spojení dvou operačních systémů UNIX vychází z dorozumění dvou procesů, z nichž každý je evidován jiným jádrem. Příklad je uveden na obr. 4.2. Uživatel, který požaduje přihlášení ve vzdáleném systému, zadává příkaz **telnet**. Vzniká odpovídající proces (téhož jména) klientu, který se pomocí volání jádra pro síťové spojení procesů domlouvá s procesem server (se jménem **telnetd**) ve vzdáleném operačním systému. Na tomto triviálním obrázku odhlížíme od komunikačních protokolů, síťových adres a jiných neméně podstatných částí provozu síťového spojení, které však zabezpečuje jádro a nastavuje správce systému. Zde nás zajímá síťová aplikace **telnet**, která je programována pomocí mechanismu síťové komunikace procesů, tj. odpovídajících volání jádra, které je v principu analogické prostředkům komunikace procesů IPC.

Pro síť tedy existují zvláštní volání jádra, která procesům umožňují komunikovat, přestože běží v různých operačních systémech typu UNIX. Používané jsou dvě základní skupiny síťových volání jádra: Berkeley sockets a knihovna TLI firmy AT&T. Budeme se jimi zabývat v kap.7. Berkeley sockets (i TLI) byly přitom koncipovány na základě právě uvedené úvahy, proto jejich komunikační možnosti mohou procesy využívat pro spojení procesů v různých operačních systémech, ale také pro spojení a přenos dat mezi dvěma procesy téhož UNIXu. Jde o rozlišení spojení typu INTERNET nebo UNIX. Spojení typu UNIX je místní komunikace a může být zpřístupněno např. pomocí souboru, jehož i-uzel je typu `s` (socket), viz 3.2.1.

Pro pochopení smyslu a významu obsahu kapitoly je tedy důležité si uvědomovat každý proces jako nositele aktivity činnosti v operačním systému, který může ovlivňovat aktivity jiných spolupracujících procesů. Zpětně může akceptovat reakci operačního systému a spolupracujících procesů. K tomu může používat uváděné prostředky komunikace mezi procesy, jejichž koncepce zvláště v podobě IPC je charakterizována architekturou klient-server.

Proces přitom pro komunikaci vždy používá volání jádra, které komunikaci zajišťuje, viz obr. 4.3.

## 4.1 Signály

Signál je celočíselná informace, jejíž výskyt v evidenčních strukturách procesu v jádru může znamenat okamžité ukončení procesu. Signál je přitom generován jádrem na základě výskytu neobvyklé události (např. odkaz procesu mimo jeho adresový prostor) nebo na základě požadavku obecně libovolného procesu. Takový požadavek zadává proces ve tvaru volání jádra `kill` (viz 2.2). Naopak každý proces příjemce signálu může pamatovat na možnost příchodu některého signálu, a tak pomocí volání jádra `signal` (viz také 2.2) definovat funkci svého programu, která je procesem provedena v okamžiku příchodu signálu. Tak je předdefinována implicitní situace zachování operačního systému vůči procesu a další činnost procesu je věcí obsahu obslužné funkce programu.

Seznam signálů, které může proces obdržet, je uveden v Příloze B včetně jejich celočíselných hodnot. Jejich textové ekvivalenty nebo přímo číselné hodnoty používá programátor v argumentu volání jádra `kill` nebo `signal`. Jádro přitom vyřídí požadavek zaslání signálu, pokud je `kill` oprávněné, tzn. že vlastníkem procesu (reálným nebo efektivním), kterému je signál určen, i procesu, který signál posílá, je tentýž uživatel, nebo je vlastníkem procesu s `kill` privilegovaný uživatel. Konečně jádro vyřizuje signál také tehdy, je-li hodnota signálu `SIGCONT` a identifikace sezení (více viz kap. 5) obou procesů je shodná. Volání jádra pro vyslání signálu má formát

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

kde hodnota `sig` je jedna z možností podle Přílohy B. Pokud je na místě `sig` uvedena 0, proces tím pouze testuje, zda použití `kill` neskončí s chybou (např. existuje-li proces, kterému je signál posílán, zda nebudou porušena práva přístupu atp.), ale samotný signál není vyslán. Parametr `pid` je PID procesu, kterému je signál určen. Proces přitom může na místě `pid` použít PID konkrétního procesu nebo také:

- 1, signál bude poslán všem procesům v systému (až na vybrané systémové procesy – o které se jedná, je věcí implementace, vždy ale jde o proces **init**, **swapper**, případně **vhand** atd.),

- 0, signál bude poslán všem procesům skupiny, do které proces patří (s výjimkou vybraných systémových procesů),

- zápornou hodnotu, ale ne –1, pak je signál poslán všem procesům, jejichž identifikace skupiny je shodná s absolutní hodnotou `pid`.

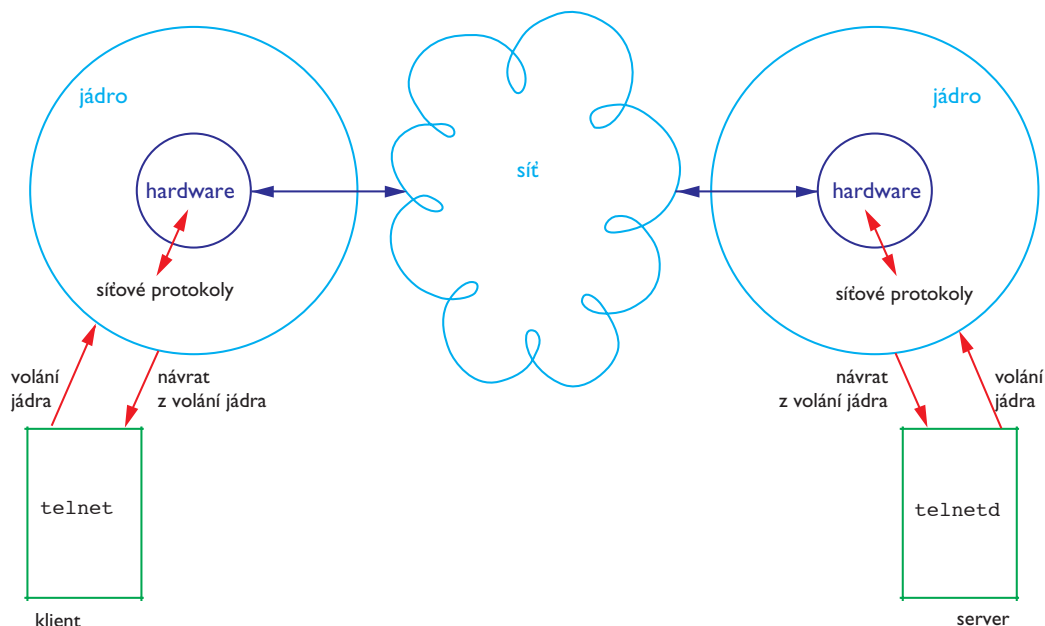
Proces, kterému je signál určen, může být na jeho příchod připraven. Používá k tomu volání jádra `signal` ve formátu

```
#include <signal.h>
```

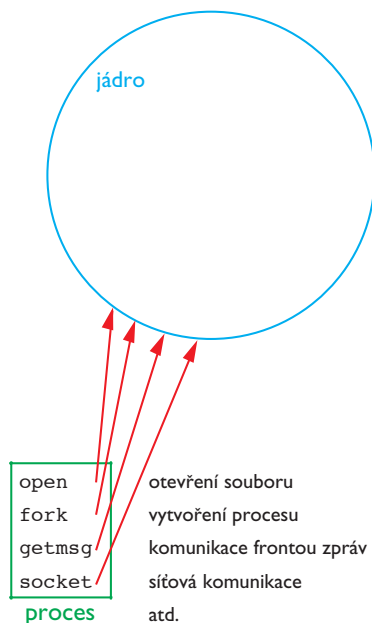
```
void (*signal(int sig, void (*disp)(int)))(int);
```

kde `sig` je signál, jehož příchod proces očekává a `disp` způsob reakce při jeho příchodu. `disp` je definován jako ukazatel na funkci. Tento externí symbol musí být pokryt buďto programátorem, který v programu definuje tělo takové funkce a programuje reakci na příchod signálu, nebo může být použita jedna z makrodefinic `SIGIGN` (ignoruj příchod signálu) nebo `SIGDFL` (ponechej implicitní nastavení, vyřídí jádro), viz Příloha B. Pokud proces použije ve volání jádra `signal` makrodefinici `SIGDFL`, je na rozhodnutí jádra, jaký bude další osud procesu. Podle konkrétního signálu jej může jádro ukončit, a to s příznakem abnormální situace (současně s ukončením je obsah paměti procesu zapsán do souboru se jménem `core`). V případě, že jde o normální ukončení, může proces pouze pozastavit nebo může signál ponechat bez povšimnutí (ignorování). V Příloze B je u každého signálu uvedena také reakce jádra. Signál, kterému nelze nastavit jinou reakci než `SIGDFL`, je např. č. 9, `SIGKILL`, který je takto rezervován pro možnost ukončit proces za jakékoliv situace. Příklad ošetření příchodu signálu uživatelskou funkcí jsme uvedli v čl. 2.2. Při využití takové možnosti jádro v okamžiku příchodu signálu nejprve obnoví výchozí stav procesu pro příjem signálu (tj. jakoby bylo použito `SIGDFL`), a teprve pak předá řízení obslužné funkci programátora.

Současné doporučující dokumenty a samotné implementace verzí UNIXu doplňují uvedený mechanismus předávání signálů. Vzhledem k tomu, že signálů může být procesu posláno více v rychlém



Obr. 4.2 Architektura klient-server v příkladu síťového spojení telnet



Obr. 4.3 Proces a jádro, komunikace mezi procesy

časovém sledu a nastává tak nebezpečí uváznutí procesu v kritických sekcích, může proces příchod signálu blokovat, tj. odložit jeho zpracování. Blokování procesu je prováděno pomocí tzv. *masky signálů* (signal mask), která je součástí dědictví procesu od svého rodiče. Masky signálů popisuje množinu blokovanych signálů, které se po příchodu dostávají do stavu doručený a nevyřízený.

Proces může pro konstrukci množiny signálů použít funkce `sigemptyset`, kterým vyprázdní množinu signálů. Naopak funkce `sigfillset` zahrne do množiny všechny signály (nelze však blokovat `SIGKILL` a `SIGCONT`). Nad množinou signálů dále mohou pracovat funkce `sigaddset` pro přidání signálu do množiny, `sigdelset` pro vyjmutí a `sigismember` pro testování, zda daný signál je v množině zahrnut či nikoliv. Funkce mají tvar

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(const sigset_t *set, int signo);
```

kde `signo` je konkrétní signál a `set` množina signálů, nad kterou je prováděna operace. Datový typ `sigset_t` je implementačně závislý (bývá jím pole čtyř 32 bitových prvků). Proces pak pracuje se svou maskou signálů pomocí volání jádra

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

Operace `how` umožňuje prověřit nebo pozměnit stav masky signálů. Je-li použito na jejím místě `SIG_BLOCK`, jsou signály podle masky `set` přidány do množiny blokováných signálů.

`SIG_UNBLOCK` znamená naopak signály podle `set` z masky vyjmout. `SIG_SETMASK` znamená náhradu současné masky signálů za definovanou hodnotu v `set`. Parametr `oset` použije jádro pro uložení předchozího stavu masky signálů, pokud `oset` není `NULL`. Současnou masku signálů v `oset` obdržíme, použijeme-li na místě `set` hodnotu `NULL`. `how` v tomto případě nemá žádný význam.

K tomu, aby byl proces schopen rozpoznat, zda signál, který blokuje, mu již byl doručen, používá volání jádra

```
int sigpending(sigset_t *set);
```

Nevyřízené signály jsou uloženy do parametru `set`, který může být prozkoumán pomocí `sigismember`.

POSIX nedefinuje volání jádra `signal`. Pro nastavení reakce doručených signálů definuje volání jádra `sigaction`, které je i součástí SVID. Jde o návaznost na rozšíření UNIXu ve smyslu blokování signálů. Volání má tvar

```
int sigaction(int sig, const struct sigaction *act, struct sigaction *oact);
```

`sig` je signál a struktura `sigaction` popisuje akci v kontextu s příchodem signálu. Má členy

```
void (*sa_handler)();  
sigset_t sa_mask;  
int sa_flags;
```

Funkce `sa_handler` má přitom tentýž význam jako parametr `disp` u `signal`. Maska `sa_mask` určuje všechny signály, které budou blokovány v průběhu interpretace funkce `sa_handler`.

`sa_flags` má význam doplňujících údajů vztahujících se např. k některým zvláštním situacím konkrétních signálů.

Původní UNIX definoval volání jádra `pause`, které je také součástí POSIXu i SVID. Znamená zablokování procesu do příchodu nějakého signálu. Má jednoduchý formát

```
int pause(void);
```

Dokumenty dnes definují volání jádra

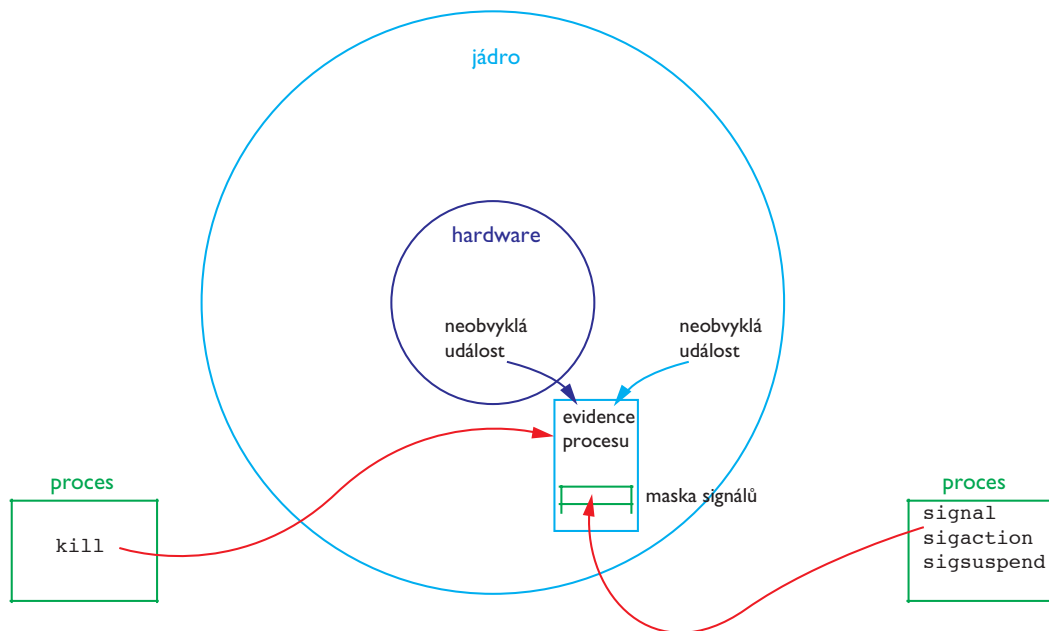
```
int sigsuspend(const sigset_t *sigmask);
```

jehož použití znamená, že jádro volajícímu procesu nahradí současnou masku signálů za `sigmask` a blokuje proces do příchodu některého signálu podle `sigmask`. Pokud má proces příchod signálu ošetřen tak, že neskončí, po ošetření doručeného signálu je maska signálů obnovena na původní hodnotu.

Doplňující je i volání jádra

```
unsigned int alarm(unsigned int seconds);
```





Obr. 4.4 Cesty signálu k procesu

kdy si proces nastaví příchod signálu SIGALRM za počet vteřin stanovený v seconds.

Přehled odesílání, doručení a blokování signálů ukazuje obr. 4.4.

Práci s maskou signálů ilustruje následující fragment programu.

```
#include    <signal.h>
main(void)
{
    sigset_t maska;
    ...
    sigemptyset(&maska);
    sigaddset(&maska, SIGINT);
    /* blokování signálu přerušení z klávesnice */
    sigprocmask(SIG_BLOCK, &maska, NULL);
    /* vstup do kritické sekce */
    ...
    /* výstup z kritické sekce */
}
```

```
sigprocmask(SIG_UNBLOCK, &maska, NULL);  
...  
}
```

Všechny doposud uvedené metody používání signálů pro synchronizaci práce procesů nepracují v kontextu skutečného času příchodu signálů. Důležitý je pouze příchod signálu, ale nikoliv pořadí, ve kterém přišel vzhledem k ostatním signálům. POSIX a mnohé implementace hovoří v tomto pojetí o reálném čase příchodu signálů. Je definováno volání jádra `sigqueue`, kterým je signál posílán tak, aby mohlo být akceptováno pořadí jeho příchodu, a volání jádra `sigwaitinfo`, které tento příchod umí ošetřit. Signál procesu `pid` je zaslán do fronty ve formátu

```
int sigqueue(pid_t pid, int signo, const union sigval value);
```

V parametru `value` přitom lze navíc procesu poslat další informace (obvykle ve formě celočíselné hodnoty a ukazatele). Svou frontu příšlých a nevyřízených signálů proces zpřístupňuje ve formátu

```
int sigwaitinfo(const sigset_t *set, siginfo_t *info);
```

kde v `set` definuje signál, na který proces čeká a který je z fronty vyjmut v případě, že byl již doručen a není ještě vyřízený. V případě, že je fronta prázdná, je proces zablokován do příchodu některého ze signálů podle `set`, a to voláním jádra `sigqueue`. `info` je ukazatelem na strukturu, která je naplněna jednak číslem příšlého signálu a jednak informacemi z fronty podle `value` v `sigqueue`. Vzhledem k tomu, že proces zablokovaný pomocí `sigwaitinfo` lze odblokovat pouze doručením některého z požadovaných signálů nebo signálem, který má smrtící účinky, programátor může také používat volání jádra `sigtimedwait`, jehož účinek je tentýž jako `sigwaitinfo`, ale v rozšiřujícím parametru může programátor určit časový interval, po který bude proces na příchod signálu do fronty čekat.

## 4.2 Roura

Jak jsme uvedli v odstavcích 2.3.3 a 2.5.1, roura slouží jako komunikační prostředek procesů, obvykle v příbuzenském vztahu rodiče a dítěte. V základním použití jde o jednosměrný tok dat mezi procesem PRODUCENT a procesem KONZUMENT. Proces, který zřizuje komunikaci, používá volání jádra `pipe` a jádro vytváří tok dat, přičemž každému ze zúčastněných procesů přiděluje odpovídající konec roury pro zápis nebo pro čtení. V klasické implementaci je roura provedena jako alokace nového i-uzlu ve strukturách jádra, který není spojen s žádným jménem v systému souborů. Odkazy na data jsou přitom evidovány v systémové vyrovnávací paměti s příznakem bez přepisu na odpovídající svazek<sup>1</sup>. Připojené deskriptory kanálů pro čtení a zápis vytvořené roury předá jádro procesu, který `pipe` provedl. Oba deskriptory jsou přitom od sebe vzdáleny velikostí nejvýše několika diskových bloků, které jsou zřetězeny v kruhovém seznamu. Velikost roury je dána konstantou `PIPE_BUF` (v `<limits.h>`) nebo `<unistd.h>`), jejíž hodnota není ale nikdy menší než 4096 bytů. Rouru využívají procesy pomocí volání jádra `read` nebo `write` stejným způsobem jako u zápisu nebo čtení obsahu obyčejného souboru a jádro synchronizuje přístup procesů k rouře podle jejího zaplnění.

Omezení velikosti roury je limitující pro zápisovou nebo čtecí operaci procesu. V případě, že se totiž proces pokusí do roury zapsat více dat, než je tato schopna přijmout, `write` vrací počet skutečně zapsaných bytů. Nejde přitom o kolizi, ale o systémové limity. Proto mají procesy v rámci volání jádra

`fcntl` možnost použít nastavení `O_NDELAY`, tj. čtení nebo zápis bez čekání na dokončení operace přenosu všech požadovaných dat. Proces pak může v návratové hodnotě `read` nebo `write` testovat skutečný počet přenesených bytů a operaci přenosu po čase opakovat bez toho, že by byl proces zablokován do ukončení operace přenosu. Příznak `O_NDELAY` mohou procesy používat také u volání jádra `open` při otvírání souboru. Setkáme se s ním při otvírání dalších komunikačních prostředků mezi procesy, kdy musí programátor ošetřit čekání na přenos dat mezi procesem samotným a komunikačním mechanismem, a to jak v místním systému, tak při programování sítí. Proto je také při programování aplikací v procesech respektována konstanta `PIPE_BUF`.

Proces, který zapisuje data do roury, je také upozorňován jádrem pomocí signálu `SIGPIPE` na skutečnost, kdy na druhé straně vytvořenou rouru nikdo nečte (žádny proces rouru neotevřel pro čtení). Signál ovšem proces může ošetřit při stanovení časového intervalu, v průběhu něhož by mělo ke čtení dojít (tzv. `timeout`). Procesy v UNIXu rouru vytvářejí a předávají procesům, jak bylo uvedeno v odst. 2.5.1, čili nejprve je vytvořen proces, který data z roury čte a on vytváří svůj dětský proces, který bude data zapisovat. Sekvenčním zpracováním tak dojde k přirozenému zamezení vzniku situace signálu `SIGPIPE`.

Každý UNIX dnes nabízí (a SVID definuje) pro pohodlnější programování roury funkce

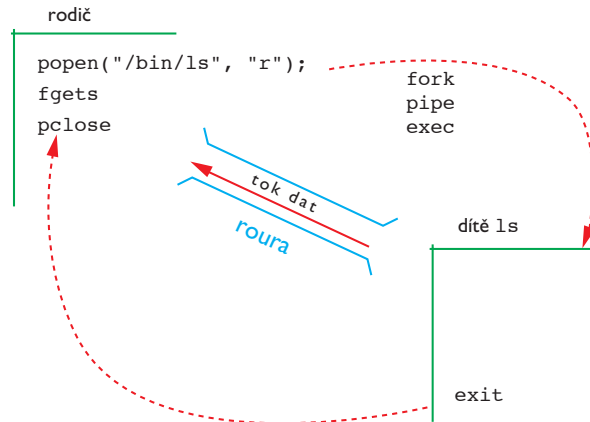
```
#include <stdio.h>

FILE *popen(const char *command, const char *type);
int pclose(FILE *strm);
```

Proces pomocí `popen` vytváří proces dětský, který realizuje program daný parametrem `command` podobně, jak by učinil shell s `command` na svém příkazovém řádku. Roura je pro proces rodiče reprezentována v/v proudem v návratové hodnotě `popen` (viz čl. 3.1), a to její čtecí konec, pokud je v `type` použito `r`. Rodič tak může číst standardní výstup dítěte programu `command`. Použije-li proces na místě `type` text `w`, dětský proces bude namísto ze standardního vstupu číst z roury, kterou zapisuje rodič. Rodič tak může pohodlně vytvářet dítě, které pro něj zpracuje potřebná data. Pomocí `pclose` uzavírá rodič v/v proud. Touto funkcí čeká na dokončení procesu dítěte, v její návratové hodnotě získává návratový status ukončeného dítěte. Situaci ukazuje obr. 4.5.

a ilustruje následující fragment příkladu:

```
...
main(void)
FILE *fin;
{
...
if((fin=popen("/bin/ls", "r"))==NULL)exit(1);
while(fgets(file, MAXLINE, fin)!=NULL)
{
...
}
exit(pclose(fin));
}
```



Obr. 4.5 Funkce popen a pclose

Pro účely obecné komunikace procesů datovým kanálem typu roura byla zavedena *pojmenovaná roura* (named pipe). Jde o spojení roury se jménem v systému souborů. Pojmenovaná roura vzniká přidělením jména nově alokovaného i-uzlu typu roura (textové označení **p**, viz odst. 3.2.1). Takový soubor je pak možné otevírat, číst, zapisovat do něj a uzavírat běžným voláním jádra (**open**, **close**, **read**, **write**), ale proces přitom nevyužívá datovou část odpovídajícího svazku, ale datovou konstrukci podporovanou jádrem tak, jako by se jednalo o obyčejnou (nepojmenovanou) rouru. Vytvoření takové konstrukce je spojeno s otevřením a zápisem nebo čtením pojmenované roury. Vzhledem k tomu, že pojmenovaná roura není vytvářena voláním jádra **pipe**, není manipulace s ní podmíněna dědictvím od jádra získaných deskriptorů pro čtení a zápis. S pojmenovanou rourou tak mohou manipulovat obecně libovolné procesy, které znají její jméno a mají přístupová práva odpovídající jejich požadavkům na přenos dat. Procesů může také být více, než je PRODUCENT a KONZUMENT u obyčejné roury, ale synchronizace jejich zápisu a čtení musí být zajištěna ještě jinými prostředky (např. semaforem). V případě více procesů pracujících s jednou pojmenovanou rourou je obvyklé, že procesy spolupracují technologií klient – server, jak bylo popsáno v úvodu kapitoly.

U pojmenované roury také platí všechny souvislosti dané limitující velikostí roury nebo signálu SIGPIPE, jak jsme uvedli v úvodu tohoto článku, protože mechanismus zajištění toku dat kanálem pojmenované roury v jádru je tentýž jako u obyčejné roury.

Pojmenovaná roura vzniká voláním jádra

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *path, mode_t mode);
```

kde **path** je jméno souboru pojmenované roury daného adresáře a **mode** je vyjádření přístupových práv stejně jako při vytváření obyčejného souboru u volání jádra **open** nebo **creat** při bitovém součinu

nastavené masky přístupových práv (volání jádra `umask`), viz odst. 3.1.1. `mkfifo` obvykle používá proces serveru programového systému. Po vytvoření roury ji pak otevírá pro čtení, zatímco klienty používají existující rouru ve svazku pro zápis, ale postup využití pojmenované roury může být prakticky libovolný.

Správce systému má přitom k dispozici také příkaz

**mkfifo path ...**

pomocí kterého může pojmenovanou rouru `path` vytvořit.

SVID uvádí i možnost vytvářet pojmenovanou rouru pomocí volání jádra `mknod` (viz 3.2.1), kde na místě `mode` v bitovém součtu s přístupovými právy uvádíme také `S_IFIFO`. Vzhledem k tomu, že `mknod` je implementačně závislé volání jádra (používané především pro vznik speciálních souborů periferií), POSIX je neuvádí. Jinak pouze privilegované `mknod` je v použití pro vznik pojmenované roury v SVID uvolněno také obyčejnému uživateli (a jeho procesům), stejně jako odpovídající jinak privilegovaný příkaz

**mknod name p**

kdy na místě `name` je uvedeno jméno souboru pojmenované roury (viz také čl. 3.3). Rozdíl mezi obyčejnou a pojmenovanou rourou ukazuje obr. 4.6.

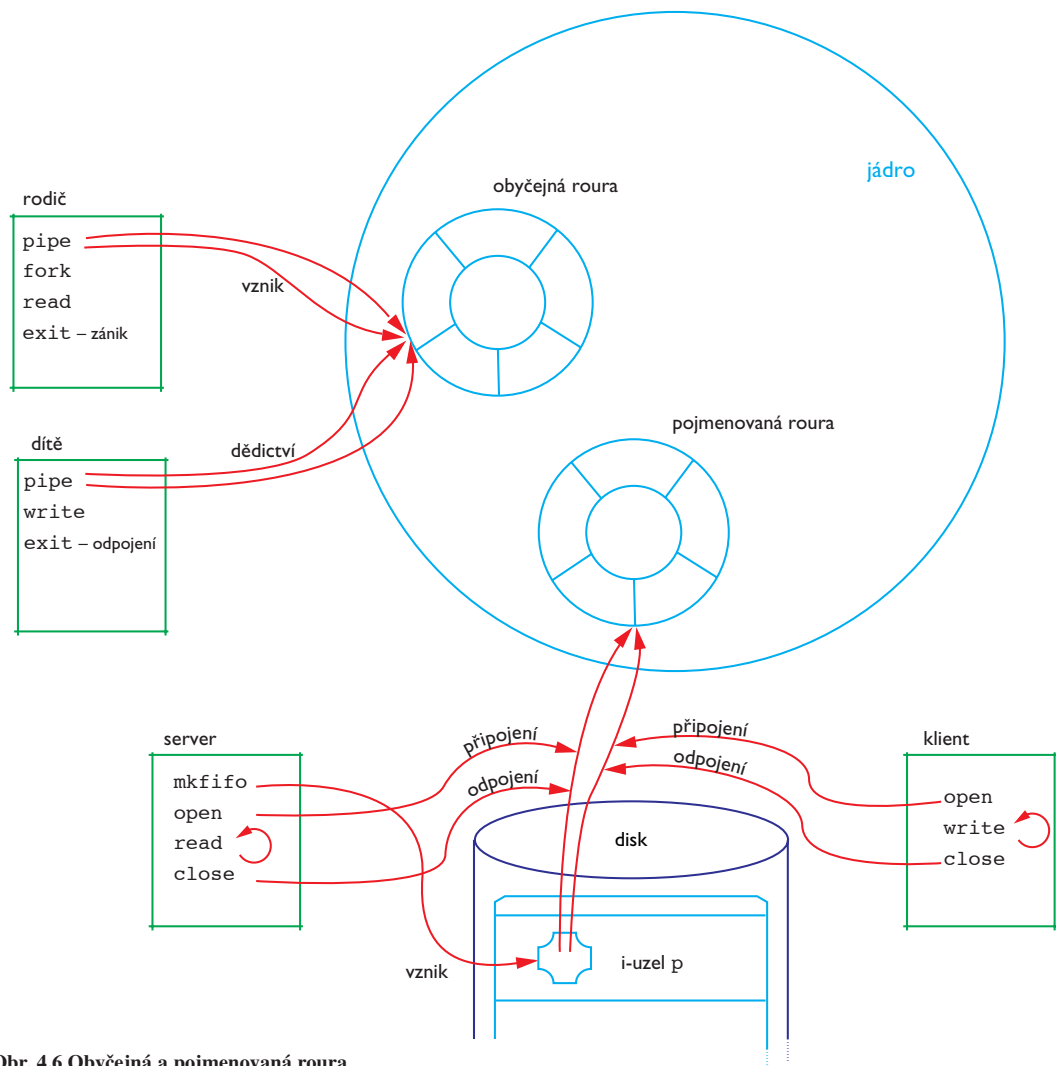
O rouře, ať už obyčejné nebo pojmenované, jsme doposud hovořili jako o kanálu jednosměrného toku dat. Současná dokumentace UNIX SYSTEM V, SVID a systémy jim odpovídající uvádí ovšem u volání jádra `pipe`, že jeho použitím vzniká kanál plně duplexní. Znamená to, že vytvořená roura je použitelná v obou směrech, které jsou vzájemně nezávislé.

Obousměrná roura je podporována v systémech provozně odpovídajících systémům firmy AT&T (tj. SYSTEM V a jeho definice v SVID), kde byla implementována nová technologie přístupu procesů k periferiím, tzv. PROUDY (STREAMS). V principu jde o možnost vkládání programových modulů jádra mezi proces a jeho přístup k ovladači periferie. Modul jádra je řízeně vložen mezi volání jádra `read` nebo `write` a ovladač samotným procesem a také je samotným procesem ovládán (pomocí dalších volání jádra jako např. `putmsg`, `getmsg` atd.). Technologii PROUDŮ se budeme věnovat podrobně v kap. 6, zde se o ní zmiňujeme, protože z ní vychází nová implementace roury, která umožňuje obousměrný tok dat.

Obyčejná obousměrná roura vzniká na základě volání jádra `pipe`, kdy proces od jádra obdrží dva odkazy do své tabulky otevřených kanálů, `files[0]` a `files[1]`, oba pro čtení i zápis. První kanál roury je směrem od `files[0]` k `files[1]` (0 slouží pro zápis a 1 pro čtení dat roury) a druhý kanál je směrem od `files[1]` k `files[0]` (1 pro zápis a 0 pro čtení). Oba deskriptory jsou děděny z procesu rodiče na dítě, tedy manipulace s nimi je stejná jako u jednosměrné roury. Použití je plně kompatibilní se způsobem původním, jenom procesy programované v minulosti nevyužívají druhý kanál roury.

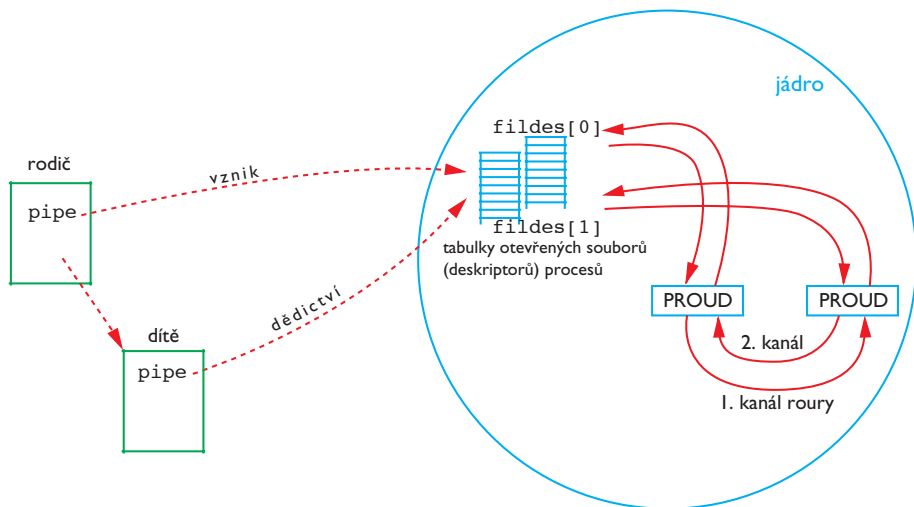
Obousměrná roura vzniká v jádru v okamžiku aktivace vznikem a spojením dvou PROUDŮ proti sobě, viz obr. 4.7.

Oba PROUDY jsou reprezentovány vždy odpovídajícím deskriptorem. Při vzniku roury přitom neobsahují žádný modul filtrace dat. Data jsou jednoduše předávána z výstupu jednoho PROUDU na vstup druhého a opačně. Obecně PROUDY však umožňují pomocí volání jádra `ioctl` zadávat vkládání



Obr. 4.6 Obyčejná a pojmenovaná roura

modulů pro úpravu procházejících dat PROUDEM při identifikaci odpovídajícím deskriptorem. Proces proto může využít této vlastnosti a na rozdíl od původní implementace vytvářet protokol dat roury, viz obr. 4.8. Použitý filtrační modul však musí být součástí jádra.



Obr. 4.7 Obousměrná roura na principu PROUDŮ

Při práci s obyčejnou rourou implementace pomocí PROUDŮ usnadňuje a zobecňuje práci procesu s obousměrnou rourou tak, že lze kterýkoliv z obou deskriptorů získaných voláním jádra `pipe` pojmenovat (pojmenovat lze obecně libovolný PROUD). Pomocí funkce

```
int fattach(int fildes, const char *path);
```

spojí proces deskriptor PROUDU (jeden konec roury) `fildes` se jménem souboru `path` (`path` přitom již musí existovat). Od této chvíle je `path` při použití volání jádra `stat` rozeznáváno jako jméno souboru typu roura a přístupem k němu (voláním jádra `open`) jiný proces získává komunikaci s procesem, který rouru vytvořil. Procesy tak mohou dynamicky pracovat s obyčejnými nebo takto pojmenovanými rourami na principu PROUDŮ a zajišťovat si tak předávání dat různými kanály. Funkcí `fattach` lze navíc přidělit existujícímu deskriptoru roury více jmen systému souborů. Spojení deskriptoru PROUDU a jména souboru zaniká použitím funkce

```
int fdetach(const char *path);
```

A konečně je možné ještě používat funkci

```
int isastream(int fildes);
```

pro testování, zda je `fildes` deskriptorem PROUDU (v případě návratové hodnoty 1) či ne (0). Obvykle tuto funkci používají dětské procesy, které dědí tabulku deskriptorů od svých rodičů.

Vyzkoušejte! V případě nejasností vyplývajících z příliš rozsáhlé provozní dokumentace PROUDŮ vyčkejte na kap. 6, kde budou PROUDY vysvětleny podrobně, včetně jejich použití v systémových aplikacích pro místní systém nebo v implementaci síťových protokolů, pro které tuto technologii v UNIXu D. Ritchie začátkem 80. let vynalezl a popsal (viz [Ritch84]).

Klasická pojmenovaná roura, která vzniká pomocí volání jádra `mkfifo`, při otevírání a přenosu dat zůstává zajištěna stejným mechanismem jako u původní implementace. Rozdíl použití takové roury a pojmenovaného PROUDU je na obr. 4.9.

Obousměrná roura našla své uplatnění i v aktivaci uživatelem. Uživatel KornShellu např. používá znaky zvláštního významu `|&` pro spojení dvou procesů obousměrnou rourou. Formule

```
$ PRODUCENT | KONZUMENT
```

pak nabývá tvaru

```
$ PRODUCENT_KONZUMENT |& KONZUMENT_PRODUCENT
```

pokud oba spojené procesy obousměrné rouře rozumí.

### 4.3 Soubory

Komunikace procesů může probíhat také pomocí souborů. Přestože jde o nejméně efektivní a nejméně spolehlivý způsob přenosu dat a synchronizace mezi procesy, UNIX ji v počátcích využíval hojně a dodnes v něm přetrvává.

Existence souboru může znamenat zámek k systémovému zdroji. Provedení volání jádra je operace atomická. I vytvoření souboru pomocí `open` nebo `creat` znamená, že není možné, aby současný požadavek na vytvoření téhož souboru uvážnul v kritické sekci. Jádro totiž takové požadavky frontuje, zpracovává sekvenčně a na dobu odezvy z periferie disku vytváří pro vznikající soubor vlastní interní zámek.

Mnohé demony (např. pro obsluhu více tisků současně nebo exkluzivního přístupu k sériové lince atd.) vytvářely pro synchronizaci zpracování požadavků od jiných procesů zámky vznikem souboru se jménem např. `LOCK` v adresáři odpovídajícího pracovního adresáře aplikace. Proces, který přicházel s požadavkem na systémový zdroj, zjišťoval existenci souboru `LOCK` (voláním jádra `access`) a jeho existenci respektoval. Po uplynutí určitého časového intervalu test opakoval a teprve pokud `LOCK` na odpovídajícím místě nenašel, pracoval se systémovým zdrojem.

Tato metoda přinášela zjevně nevýhodu bezohledného procesu, který použije systémový zdroj, aniž testuje existenci souboru `LOCK`. Nevýhoda je ovšem obecná a souvisí více s organizací bezpečného chodu systému. Správce systému totiž musí zajistit pomocí přístupových práv k systémovému zdroji manipulaci pouze vybraným procesům, které jsou prověřené, tj. mají nastavený `s-bit`, ale vždy `LOCK` respektují. Další nevýhoda je uvážnutí programového systému využívajícího `LOCK` havárií procesu, který má systémový zdroj v držení (nebo havárií operačního systému). `LOCK` zůstane přítomen, přestože systémový zdroj je volný. Obvykle je potom nutné provádět procesy, které uvedou programový systém do výchozího stavu, tj. zruší `LOCK` a kontrolují konzistenci systémového zdroje.

Systémovým zdrojem může ovšem často být i samotný obsah souboru. Procesy tak mohou sdílet obsah souboru pro předávání dat.

Je to jednak sekvenční metoda, tj. obsah souboru je výstupem procesu, který pro další zpracování pobídne (např. signálem) jiný proces k další manipulaci s obsahem souboru. Je také obvyklé, že na obsah souboru dohlíží rodič dětí, které si obsahem souboru data předávají. Je to běžné např. při překladač zdrojového textu kompilátorem do strojového kódu. Kompilátor je řídicí proces, který startuje postupně procesy pro zajištění jednotlivých průchodů kompilátoru (tj. lexikální analýzu, syntaktickou analýzu



atd.). Jejich mezivýsledky jsou přitom ukládány do obsahu souboru smlouveného jména a nová část kompilátoru (nový průchod) je spuštěn vždy jako nový proces, dítě řídicího procesu. Tento způsob předávání dat pak ale nevyžaduje zvláštní zamykání samotného souboru, protože synchronizace procesů je prováděna buďto signálem nebo rodičem skupiny procesů, která se uchází o systémový zdroj.

Druhou metodou je metoda současného přístupu více procesů k obsahu jednoho souboru, tj. sdílení jeho obsahu v čase. Přístup k takovému souboru musí být ovšem opět nějakým způsobem synchronizován. Jak bylo řečeno v úvodu kapitoly, některý z procesů operující nad obsahem souboru musí mít funkci serveru, který zajistí např. vytvoření souboru nebo jeho zrušení v případě zastavení výpočtu nebo celého operačního systému. Vlastní manipulaci s daty pak provádí procesy, které nazýváme klienty. Vedle toho, že jádro poskytuje volání jádra `lseek` pro rychlý přesun v obsahu souboru, také víme, že lze obsah souboru nebo jeho část zamykat voláním jádra `fcntl` (viz odst. 3.1.4). UNIX má pro pohodlnější manipulaci také funkci `lockf`. Procesy přitom mohou testovat přístup k datům souboru také použitím `fcntl` nebo `lockf`, nebo mohou používat příznak `O_NDELAY` při otevírání souboru.

Pro účely vzniku dočasných souborů pro zpracování, ať už postupně nebo současně různými procesy, byla v UNIXu zavedena (a dodnes se hojně používá) oblast veřejného přístupu začínající adresářem `/tmp` a `/usr/tmp`. V těchto adresářích může libovolný proces vytvářet své dočasné soubory (nic nebrání vytvářet celé podadresáře) a využívat je pro předávání dat jiným procesům. Pro jednoznačnou identifikaci souboru i v rámci běhu několika instancí téhož programu je při vytváření jména souboru používáno PID běžícího procesu tak, že číselná návratová hodnota volání jádra `getpid` je převedena na znakovou podobu, která je základem jména souboru. Každý UNIX ale vlastní funkci

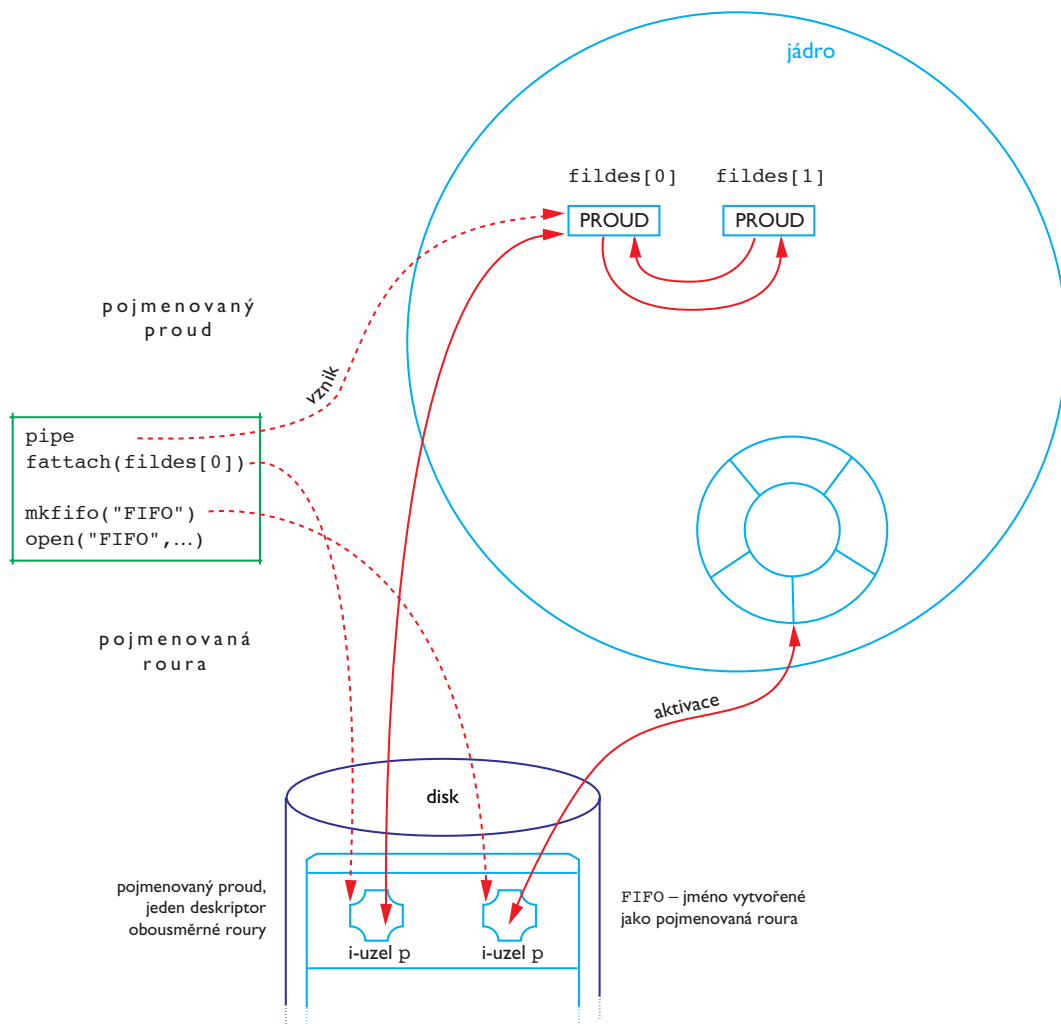
```
#include <stdio.h>
FILE *tmpfile(void);
```

kteřá vytváří dočasný soubor a otevírá jej jako v/v proud, který proces získá v návratové hodnotě funkce. Dočasný soubor je přitom spolehlivě zrušen při jeho uzavření (funkcí `fopen`) nebo při ukončení procesu, který funkci `tmpfile` použil. V případě snahy ovlivnit jméno vznikajícího dočasného souboru může proces použít funkce `tmpnam`, `tempnam` a `mktemp`. Ty jsou v `tmpfile` použity v implicitním nastavení.

Za existenci dočasných souborů ve veřejných adresářích `/tmp` a `/usr/tmp` mají zodpovědnost procesy, které je vytvořily. Znamená to, že po ukončení práce s dočasným souborem by měly být zrušeny. Proces také nesmí zapomínat na příchod signálů a zajistit zrušení dočasných souborů v obsluhovaných funkcích, zvláště jedná-li se o systémové procesy. Správce systému při pravidelných údržbách oblasti dočasných souborů prohlíží a čistí. Vzhledem k tomu, že jsou umístěny na systémových svazcích, musí také sledovat volnou kapacitu pro potřeby dočasných souborů a v případě omezené volné kapacity požadovaný prostor poskytovat (např. připojením nového svazku nebo symbolickým odkazem na adresář jiného svazku, viz kap. 10).

Oblast `/tmp` nebo `/usr/tmp` bývá využívána procesy zejména při sekvenčním zpracování toku dat více procesy, kdy se z různých důvodů nehodí použití roury. Je používána také při současném čtení nebo zápisu téhož souboru více procesy, kdy soubor slouží pro předávání dat mezi nimi. Současné zpracování obsahu souboru více procesy najednou je ale běžné také při práci s datovými soubory aplikací, které jsou uloženy v částech uživatelských dat. V případě, kdy nejsou procesy v příbuzenském vztahu, je





Obr. 4.9 Pojmenovaný PROUD a pojmenovaná roura

Každý zúčastněný proces tak posílá nebo odebírá zprávu jako posloupnost bytů, která je určena jejím typem. Např.

```
main(void)
{
    int id;
    struct { long mtype; char mtext[1024]; } zprava;
    id=msgget(ftok("/usr/local/pool/fronta", 0), 0777|IPC_CREAT);
    zprava.mtype=1L;
    strcpy(zprava.mtext, "Zítřa si dáme do nosu");
    msgsnd(id, &zprava, sizeof(zprava.mtext), IPC_NOWAIT);
}
```

je vytvoření (IPC\_CREAT) fronty zpráv na základě generace klíče podle existujícího souboru /usr/local/pool/fronta a zaslání zprávy typu 1 s textem Zítřa si dáme do nosu. Použitá volání jádra mají formát

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgflg);
```

kde v key používáme jednoznačnou číselnou identifikaci fronty zpráv a v msgflg při vytváření fronty stanovujeme její přístupová práva. Použitý logický součet s IPC\_CREAT znamená požadavek vytvoření nové fronty zpráv. key přitom nesmí být klíčem některé již existující fronty. Novou frontu zpráv je také možné vytvořit bez zadání klíče. Na místě key lze použít konstantu IPC\_PRIVATE. Jádro pak vyhledá volnou pozici v tabulce registrace zpráv a tu procesu přiřadí. Odkaz do tabulky registrace fronty zpráv jádrem získá proces v návratové hodnotě volání jádra. Tuto hodnotu pak proces používá při manipulaci s frontou u ostatních volání jádra. Jde o tzv. deskriptor fronty zpráv (je to obdoba např. tabulky otevřených souborů procesu). V příkladu je to u msgsnd, které má obecně formát

```
int msgsnd(int msqid, void *msgp, size_t msgsz, int msgflg);
```

Další parametry mají význam ukazatele na strukturu s typem a obsahem zprávy (msgp), délky textové části zprávy (msgsz) a způsobu chování jádra v případě, že dojde k překročení hranice délky jedné zprávy nebo maximálního počtu všech zpráv v systému (msgflg, proces může být buďto zablokován a čekat na uvolnění potřebné kapacity, nebo může pokračovat a použít msgsnd později, viz konstanta IPC\_NOWAIT v dokumentaci). V případě úspěchu vrací msgsnd hodnotu 0.

Po spuštění procesu podle uvedeného programu můžeme použít příkaz pro výpis existujících front zpráv v systému

```
$ ipc -q
IPC status from /dev/kmem as of Sun Nov 3 12:09:21 1996
T ID KEY MODE OWNER GROUP
Message Queues:
q 100 0x00010ecf --rw-rw-rw- skoc user
```

Uvedená fronta s deskriptorem 100 a klíčem 0x00010ecf bude součástí výpisu, který může obsahovat pochopitelně i jiné právě používané fronty zpráv.

Proces, který se bude k frontě zpráv vytvořené v příkladu připojovat, použije následující úryvek:

```
int cid;
struct { long mtype; char mtext[1024]; } czprava
...
cid=msgget(ftok("/usr/local/pool/fronta", 0), 0777);
msgrcv(cid, &czprava, 1024, 1L, IPC_NOWAIT);
...
```

Voláním `msgget` žádá připojení (nepoužil ani `IPC_CREAT` ani `IPC_PRIVATE`) k frontě, která musí existovat a v parametru `msgflg` vyjadřuje proces požadovaná přístupová práva k frontě (není-li možné je garantovat, `msgget` selže a vrací chybu). V návratové hodnotě v případě úspěchu proces získá deskriptor odpovídající fronty zpráv, která je v příkladu použita u volání jádra `msgrcv`, kdy proces žádá o předání první zprávy typu 1. Pokud taková zpráva ve frontě byla některým procesem zaslána, je procesu předána. Obecný formát čtení z fronty zpráv je

```
int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,
           int msgflg);
```

Parametry přitom mají význam deskriptoru fronty zpráv (`msqid`), ukazatele na strukturu, do které jádro vloží obsah požadované zprávy (`msgp`), největšího požadovaného počtu přenášených znaků zprávy (`msgsz`), typu požadované zprávy (`msgtyp`, je-li zde 0, vyzvedne se první zpráva fronty, je-li záporná, pak ta, jejíž typ je menší nebo roven parametru, v příkladu používáme 1, vyzvedne se proto první zpráva právě typu 1) a způsobu ošetření při chybějící požadované zprávě (`msgflg`, pokud proces nepoužije `IPC_NOWAIT`, je zablokován do příchodu vyhovující zprávy) nebo jak má jádro postupovat v případě delší zprávy, než je požadovaná velikost (použitím `MSG_NOERROR` je zpráva zkrácena). `msgrcv` při úspěchu vrací skutečný počet přenesených znaků zprávy.

Uvedený příklad má však závažný nedostatek. Žádný z fragmentů (každý z nich bude využit jiným procesem) neobsahuje zrušení fronty zpráv. Správně by se o tuto akci měl zajímat proces, který ji vytvořil. Pokud jsme programovali a zkoušeli uvedeným způsobem, fronta zpráv zůstane evidována jádrem až do zastavení operačního systému nebo do použití volání jádra `msgctl` procesu, jehož efektivním vlastníkem musí být ten, který zprávu vytvořil (nebo privilegovaný uživatel). Použití v našem příkladě lze programovat takto

```
...
msgctl(id, IPC_RMID, 0);
...
```

Kdy je použit deskriptor zprávy `id` (návratová hodnota `msgget`) s příkazem `IPC_RMID`. Volání jádra `msgctl` používá také standardní program `ipcrm`, pomocí kterého může uživatel frontu zpráv odstranit na úrovni příkazového řádku shellu, v našem případě:

```
$ ipcrm -q 100
```

(zruší frontu zpráv s deskriptorem 100).

Volání jádra `msgctl` však také slouží ke zjišťování (nebo nastavování) atributů fronty. Formát

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

totiž umožňuje na místě `cmd` zadávat také konstanty `IPC_STAT` pro zjištění nebo `IPC_SET` pro nastavení atributů fronty zpráv s deskriptorem `msqid`. `buf` je přitom struktura, která bude jádrem použita pro sdělení nebo nastavení atributů fronty. Struktura `msqid_ds` je definována v `<sys/msg.h>` a obsahuje položky:

```
struct ipc_perm msg_perm;      /* přístupová práva */
unsigned long msg_qnum;        /* počet zpráv v současné chvíli
                                ve frontě */
unsigned long msg_qbytes;      /* největší možný počet znaků ve frontě
                                */
pid_t msg_lspid;               /* PID procesu, jenž poslední použil
                                msgsnd */
pid_t msg_lrpid;               /* PID procesu, jenž poslední použil
                                msgrcv */
time_t msg_stime;              /* čas posledního provedení msgsnd */
time_t msg_rtime;              /* čas posledního provedení msgrcv */
time_t msg_ctime;              /* čas poslední modifikace atributů */
```

Na základě použití `msgctl` s příkazem `IPC_STAT` tak může libovolný proces o dané frontě získat uvedené informace. Struktura `msg_perm` eviduje přístupová práva k frontě zpráv včetně jejího vlastníka a skupiny, k nimž se přístupová práva vztahují. Obecná struktura pro všechny formy komunikace IPC (fronty zpráv, sdílená paměť, semaforey) `ipc_perm` je definována v souboru `<sys/ipc.h>` a obsahuje tyto složky

```
uid_t uid;                     /* identifikace vlastníka */
gid_t gid;                     /* identifikace vlastnické skupiny */
uid_t cuid;                    /* identifikace vytvářejícího uživatele */
gid_t cgid;                    /* identifikace vytvářející skupiny */
mode_t mode;                   /* práva pro čtení a zápis */
```

Slovo přístupových práv je čitelné v kontextu definic konstant, které jsme uvedli v odst. 3.1.1 u volání jádra systému souborů `open` a `creat`. Slovo přístupových práv `mode` u front zpráv však obsahuje pouze příznaky pro čtení a zápis vlastníka `uid`, skupiny uživatelů `gid` a ostatních uživatelů. Vlastník i skupina jsou registrováni v číselné podobě, která je převáděna ve výpisech programů na textová jména podle tabulek souborů `/etc/passwd` a `/etc/group`. V uvedeném výpisu příkazu **ipcs -q** tohoto článku je dobře vidět interpretace struktury `ipc_perm` pro fronty zpráv. Slovo přístupových práv při vytváření fronty je zapisováno v argumentu `msgflg` volání jádra `msgget`. V příkladu jsme použili osmičkovou hodnotu `0777`, která by správně (z důvodů přenositelnosti) měla být zapsána ve formě `S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH` (za využití hlavičkového souboru `<sys/stat.h>`).

Pomocí příkazu `IPC_SET` u volání jádra `msgctl` může proces s efektivním vlastníkem shodným s majitelem nebo tvůrcem fronty zpráv (nebo privilegovaným uživatelem) změnit pouze tyto složky struktury `msqid_ds` argumentu `buf`: vlastníka fronty zpráv (`msg_perm.uid`), skupinu

(`msg_perm.gid`), slovo přístupových práv (`msg_perm.mode`). Privilegovaný uživatel operačního systému může navíc změnit hranici maximální délky jedné fronty zpráv (`msg_qbytes`).

Jádro eviduje fronty zpráv ve svých vnitřních strukturách podle obr. 4.10.

Organizace vychází z tabulky seznamu všech vytvořených zpráv, `msgget` vrací odkaz na její položky. Každá fronta zpráv má odkaz na začátek zřetěženého seznamu hlaviček jednotlivých zpráv. Každá hlavička obsahuje odkaz do třetí části podle obrázku, do prostoru textových částí zpráv. Takové schéma umožňuje efektivní správu front zpráv. Jádro vlastní algoritmy pro rychlý průchod frontami zpráv a přidělování nebo uvolňování zpráv nebo celých front zpráv.

Po zpřístupnění fronty zpráv mohou procesy střídavě používat `msgsnd` a `msgrcv` a předávat si tak zprávy různého typu. Procesů, které se mohou na frontu zpráv připojit, může být libovolné množství; je zvykem používat architekturu klient – server (viz úvod kapitoly), kdy jeden ze zúčastněných procesů spravuje frontu zpráv a ostatní procesy ji využívají. Dále je zvykem, že fronta zpráv přežívá klienty, kdežto server při svém zániku frontu zpráv odstraňuje.

Standard POSIX definuje pro práci s frontami zpráv jiné rozhraní. Na rozdíl od SVID uvedená volání jádra začínající textem `msg` nezná, a přestože je mechanismus předávání zpráv mezi procesy principiálně stejný, jeho ovládání je následující.

Základní definice konstant a používaných struktur je evidován v hlavičkovém souboru `<mqueue.h>`, volání jádra, která procesy používají, nesou pojmenování `mq_open` pro zpřístupnění fronty zpráv (vytvoření nebo připojení k ní), `mq_close` pro odpojení od fronty zpráv, `mq_unlink` pro zrušení fronty zpráv, `mq_send` a `mq_receive` pro vkládání a čtení front zprávy. Pomocí `mq_notify` žádá proces o oznámení v případě, že do prázdné fronty přišla zpráva, na kterou čeká (příchod zprávy do fronty je procesu oznámen signálem), a pro získání atributů fronty zpráv, případně pro změnu vybraných atributů slouží `mq_getattr` a `mq_setattr`.

Práce je obdobná jako v SVID. V parametrech `mq_open` používáme přímo jméno fronty (které se doporučuje používat ve tvaru jména souboru) namísto číselného klíče a způsob otevření nebo vytvoření fronty zpráv pomocí konstant známých z volání jádra `open` a `creat` u systému souborů. Návratová hodnota je v případě úspěchu deskriptor fronty zpráv, který proces používá při manipulaci s takto otevřenou frontou. Odpojení od fronty zpráv, přestože proces i fronta může dál existovat, je rozšíření oproti SVID a provede se pomocí `mq_close` (s uvedením deskriptoru fronty), zatímco `mq_unlink` ruší celou frontu zpráv včetně jejího obsahu (uvádíme jméno fronty). Parametry `mq_send` a `mq_receive` jsou ve stejném významu jako v SVID, rovněž tak `mq_getattr` a `mq_setattr`. Při používání doporučujeme používat provozní dokumentaci, přestože současné systémy UNIX funkce uvedených jmen nepodporují (prozatím...).

## 4.5 Sdílená paměť

Jádro může poskytnout procesům jako prostor jejich komunikace přístup do téže části operační paměti, tzv. sdílenou paměť (shared memory). Společnou paměť registruje jádro podobně jako prostor pro předávání zpráv z minulého článku. Velmi podobný je také způsob ovládání. Proces může sdílenou paměť vytvořit nebo zpřístupnit pomocí volání jádra

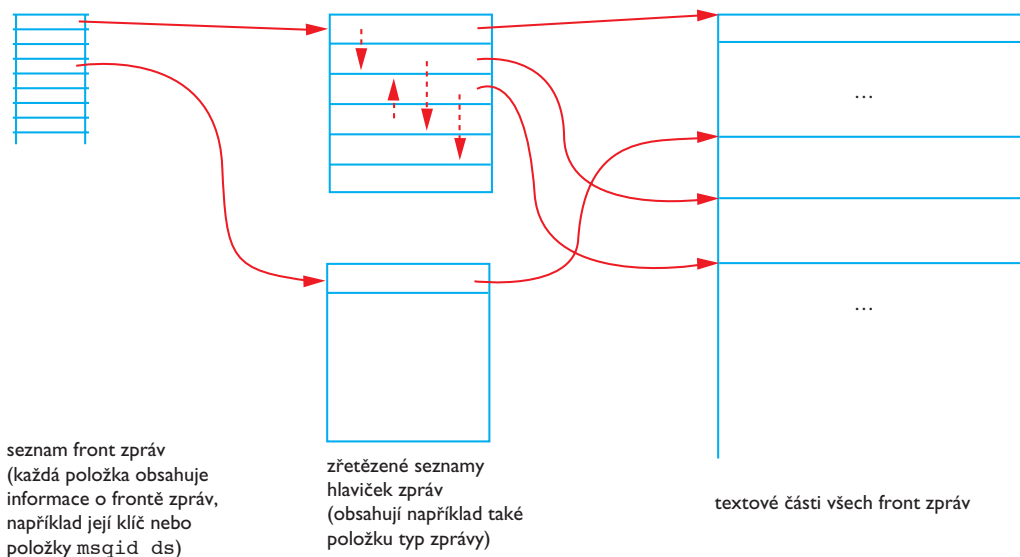
```
#include <sys/types.h>
```

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, int size, int shmflg);
```

Klíč `key` znamená opět jednoznačnou identifikaci prostoru komunikace, zde sdílené paměti. Argumentem `size` vyjadřujeme, o jak velkou paměť v bytech máme zájem. Proces může žádat připojení pouze na část již vytvořené sdílené paměti (pak uvádí menší hodnotu, než je velikost vytvořené paměti). Zda bude sdílená paměť nově vytvářena nebo zda jde pouze o její připojení k adresovému prostoru procesu, rozhoduje třetí argument `shmflg`, kde zadáváme přístupová práva, a `IPC_CREAT`, pokud sdílenou paměť vytváříme. Stejně jako u front zpráv i zde můžeme použít při vytváření na místě prvního argumentu `IPC_PRIVATE` a jádro nám poskytne nepoužitou položku tabulky sdílených pamětí (jde o použití, kdy na klíči nezáleží, protože prostor komunikace bude používán např. v rámci rodiny procesů – víte jak?). Rovněž tak zadávání přístupových práv podléhá stejné konvenci jako u front zpráv (při vytváření je naplněno slovo přístupových práv, u připojování je slovo testováno na oprávněnost požadavku přístupu efektivního vlastníka a skupiny procesu vzhledem k údajům v attributech sdílené paměti, viz dále). V návratové hodnotě `shmget` získá proces identifikaci sdílené paměti, její tzv. deskriptor.

Po připojení procesy využívají sdílenou paměť tak, že provedou spojení adresy jejich datového prostoru se sdílenou pamětí. Takových připojení může být současně více v rámci jednoho procesu nebo z různých procesů. Po spojení adresy datové oblasti procesu a sdílené paměti proces používá sdílenou paměť



Obr. 4.10 Evidence fronty zpráv v jádru



přímo, tj. stejným způsobem, jako používá ostatní data svého adresového procesu. Spojení datové oblasti a sdílené paměti dosáhne proces pomocí

```
void *shmat(int shmid, void *shmaddr, int shmflg);
```

kde `shmid` je deskriptor sdílené paměti získaný pomocí `shmget` a `shmaddr` je odkaz na oblast sdílené paměti. Je důležité, že v návratové hodnotě proces získá adresu požadované oblasti sdílené paměti. Přiřazuje ji proto do proměnné typu ukazatel, `shmflg` určuje, zda má být paměť dostupná explicitně pouze pro čtení (`SHM_RDONLY`), nebo také určuje význam `shmaddr`, protože sdílená paměť může být uvažována po tzv. regionech.

```
main(void)
```

```
{
int id;
char *pamet;
id=shmget(ftok("/usr/local/pool/pamet", 0), 1024, 0777|IPC_CREAT);
pamet=shmat(id, 0, 0);
strcpy(pamet, "Zítra se bude tančit všude");
}
```

V příkladu alokujeme (vytváříme) sdílenou paměť o velikosti 1024 bytů a spojujeme její začátek s ukazatelem `pamet`. Do paměti pak ukládáme text `Zítra se...` Proces, který takto interpretuje vytvoření a zápis do sdílené paměti, skončí, aniž sdílenou paměť jakkoliv dále spravuje, což je nezodpovědné (sdílenou paměť a její obsah to ovšem nijak nepoškodí). I zde se dá použít příkaz

```
$ ipcs -m
```

```
IPC status from /dev/kmem as of Wed Nov 6 13:18:32 1996
T      ID      KEY      MODE      OWNER      GROUP
Shared Memory:
m      201      0x00010ee4  --rw-rw-rw- skoc      user
```

a použitím

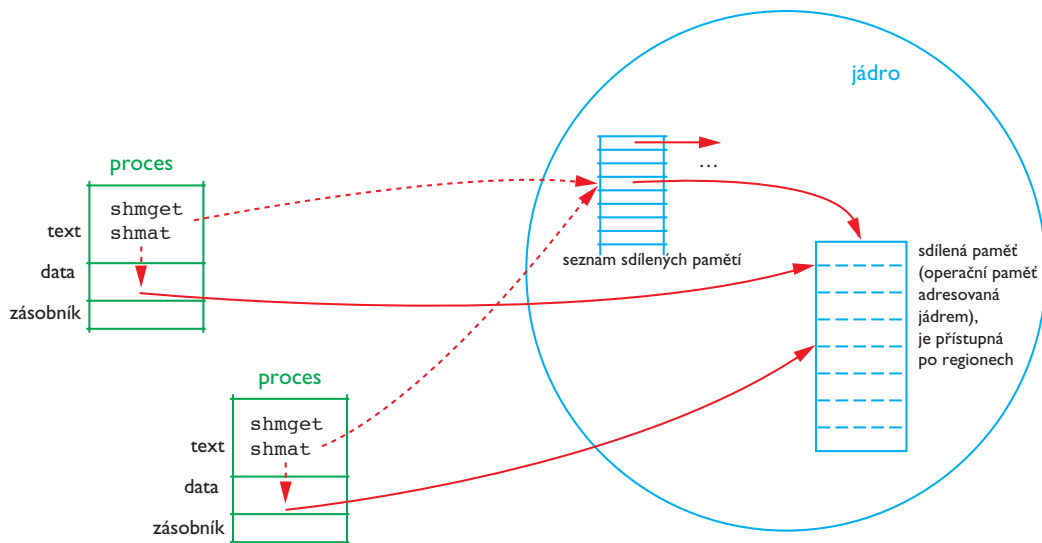
```
$ ipcrm -m 201
```

sdílenou paměť (s deskriptorem 201) zrušit. Úryvek programu, který by data z téže paměti (ještě před jejím zrušením) vytisknul na standardní výstup je:

```
int cid;
char *cpamet;
...
cid=shmget(ftok("/usr/local/pool/pamet", 0), 512, 0777);
cpamet=shmat(cid, 0, 0);
printf("%s\n", cpamet);
...
```

Povšimněte si, že velikost sdílené paměti, o kterou proces žádá, je menší, než jak byla paměť dříve vytvořena. Po spojení pomocí `shmat` může proces pak pracovat přímo se sdílenou pamětí.

V použití `shmat` na místě `shmaddr` jsme v příkladu použili 0, kdy ve spojení s hodnotou 0 na místě `shmflg` tak jde významově o připojení na vytvořenou paměť od jejího začátku. 0 v `shmflg` znamená



Obr. 4.11 Sdílená paměť

použití hodnoty `shmat` jako vzdálenosti v bytech od začátku sdílené paměti. Proces si tak může pomocí `shmat` spojit ukazatele s různými částmi sdílené paměti. Proces ale také může uvažovat připojenou sdílenou paměť v regionech (segmentech). Velikost regionu sdílené paměti je implicitně dána hodnotou `SHMLBA`. V bitovém součinu s obsahem `shmflg` pak používáme `SHM_RND` a hodnota v `shmat` určuje začátek regionu, protože místo, které jádro spojí s ukazatelem, je zaokrouhleno podle vztahu  $(shmat - (shmat \text{ modulo } SHMLBA))$ . V dokumentaci se termín region používá i v případě, že nepoužijeme `SHM_RND`. Region v tom případě znamená spojení s prvním regionem, tj. se začátkem sdílené paměti. Sdílenou paměť tak lze zakreslit schématem obr. 4.11.

Proces může zrušit spojení proměnné ukazatele se sdílenou pamětí pomocí

```
int shmdt(void *shmat);
```

kde na místě `shmat` zadává adresu získanou v návratové hodnotě `shmat`. Pokud by např. náš úryvek programu pro čtení obsahu sdílené paměti potřeboval použít proměnnou `cpamat` pro spojení na jiný region, voláním

```
shmdt(cpamat);
```

původní spojení zruší, ale obsah sdílené paměti zůstane zachován.

Obsah sdílené paměti zůstane zachován, dokud některý z oprávněných procesů (efektivní uživatel procesu, který sdílenou paměť vytvořil nebo privilegovaný uživatel) nepoužije volání jádra

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

kde na místě `cmd` použije `IPC_RMID` pro deskriptor sdílené paměti `shmid`. Jde o přesnou analogii `msgctl` z předchozího článku. Proces ve funkci serveru v našem příkladu tedy použije

```
shmctl(id, IPC_RMID, 0);
```

Na obsahu třetího parametru v tomto případě nezáleží. Záleží na něm však v případě použití jiných hodnot na místě `cmd`, protože podobně jako u fronty zpráv může za existence sdílené paměti zjistit její atributy (`IPC_STAT`) a některé změnit (`IPC_SET`). `buf` tehdy ukazuje na strukturu `shmid_ds` definovanou v `<sys/shm.h>`, která obsahuje složky

```
struct ipc_perm shm_perm;      /* přístupová práva */
int shm_segsz;                 /* velikost regionu v bytech */
pid_t shm_lpid;                /* PID procesu, jenž poslední použil
                                shmat nebo shmdt */
pid_t shm_cpid;                /* PID procesu, který sdílenou paměť
                                vytvořil */
unsigned long shm_nattch;       /* počet současných spojení */
time_t shm_atime;              /* čas posledního provedení shmat */
time_t shm_dtime;              /* čas posledního provedení shmdt */
time_t shm_ctime;              /* čas poslední modifikace atributů */
```

jejichž význam je analogický frontám zpráv včetně struktury přístupových práv (`shm_perm`), jejíž definice je i zde podle `<sys/ipc.h>`.

POSIX i zde nepoužívá uvedená jména pro práci se sdílenou pamětí, sdílenou paměť ovšem definuje. Rutiny `shm_open` pro vytvoření nebo připojení ke sdílené paměti a `shm_unlink` pro její zrušení jsou součástí rutin práce se správou operační paměti, jejíž definice a definice konstant s nimi souvisejících je uvedena v `<sys/mman.h>`. Parametr `shm_open` je jméno sdílené paměti (i zde se doporučuje používat úplné jméno souboru), způsob přístupu (čtení, zápis, vytvoření atd.) a přístupová práva. Návrhová hodnota `shm_open` je přidělený deskriptor z tabulky otevřených souborů (!), přes který pak probíhá přenos dat pomocí rutin (volání jádra) práce s otevřenými soubory (`read`, `write`). Volání jádra `shm_unlink` má jediný parametr, jméno sdílené paměti. Mapování na regiony je zobrazeno rutinami `mmap` a `munmap` (které ale zná i SVID), pomocí kterých dochází ke spojení proměnné ukazatel s pamětí, která je registrována v tabulce deskriptorů otevřených souborů. Tak lze mapovat na přímo adresovatelnou oblast procesu jak sdílenou paměť, tak i obsah diskových souborů. Paměť pak lze zamykat a odemykat pomocí `mlock` a `munlock`. Není bez zajímavosti, že je definována také rutina `msync`. Jako synchronizace odkazované operační paměti a odpovídající diskové.

## 4.6 Semaforey

Povšimněte si, že ani systém pro práci s frontami zpráv, ani systém vyrovnávacích pamětí neobsahuje jako součást synchronizaci mezi procesy. U fronty zpráv může proces požadovat zablokování do příchodu zprávy určitého typu, a tak lze jistě synchronizace docílit. U sdílené paměti, pokud neomezíme přístupová práva, musí být synchronizace vždy provedena nějakým dodatečným mechanismem. Vzpomeňme si na pojmenovanou rouru, i zde jsme hovořili o zajištění synchronizace procesů nad oblastí

komunikace „jinými prostředky“. Jiné prostředky mohou být signály nebo zamykací soubory. Obě metody jsme v kapitole již uvedli. Ale tvůrci IPC zavedli a implementovali ještě další metodu pro synchronizaci procesů nad výpočetním zdrojem. Jde o využití semaforů (semaphores), obecně definované metody synchronizace v computer science.

Implementace semaforů v UNIXu (a definice v SVID) vycházejí z Dekkerových algoritmů publikovaných E.W. Dijkstrou v r. 1968. Semafor je celočíselná proměnná, jejíž obsah určuje možnost využití výpočetního zdroje. Pokud je hodnota semaforu 1 a více, lze přes semafor projít a využít zdroje, je-li 0 a méně, semafor je uzavřen a zdroj nelze použít. Procesy při synchronizaci používají atomické operace P a V, které nastavují hodnotu semaforu. P (od holandského Passeren, procházet) sníží hodnotu semaforu o 1. Při snaze projít přes semafor používáme tuto operaci a výsledná hodnota semaforu je jednak určující pro získání zdroje a jednak při uzavření semaforu může v absolutním vyjádření vyjadřovat počet procesů, které se ucházejí o zdroj. Pouze jeden proces může používat zdroj, ostatní procesy, které použily operaci P, systém řadí do fronty odpovídajícího semaforu. V (Vrijngeven, uvolňovat) naopak hodnotu semaforu o 1 zvýší a použije ji proces, který opouští výpočetní zdroj. Součástí operace V je také následné uvolnění jednoho z fronty čekajících procesů, kterému je zdroj přidělen, pokud není fronta prázdná. Inicializační hodnota semaforu může být libovolná. Je-li kladná, určuje počet procesů, které mohou současně zdroj využít. Význam hodnoty semaforu, vztažený k číselné ose, ukazuje obr. 4.12<sup>2</sup>.

Důležitá je podmínka atomičnosti operací P a V. Při provádění všech instrukcí, které operaci zajišťují, nesmí být přerušena jinou operací P nebo V.

Implementace v UNIXu dále rozšiřují obecně popsaný princip tak, že je možné provádět atomicky několik operací P a V nad několika semaforey současně (nad tzv. množinou semaforů). Operace přitom mohou navíc snižovat nebo zvyšovat hodnoty semaforu o více než 1. Jádro přitom provede buďto všechny požadované operace, nebo žádnou, pokud je některá z operací odmítnuta.

Voláním jádra

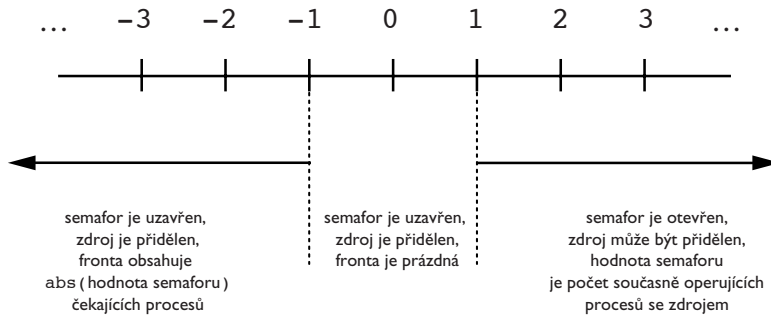
```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsem, int semflg);
```

v jeho návratové hodnotě získá proces deskriptor množiny semaforů (viz obr. 4.13). Podobně jako u zpráv nebo sdílené paměti je použit klíč obvykle jednoznačně generovaný pomocí funkce `ftok`. Tak si různé procesy vzájemně zajistí přístup k těžce množině semaforů. Hodnotou `nsem` při vytváření proces stanovuje počet semaforů v množině, při využívání již existující množiny určuje počet používaných semaforů (semaforey jsou zpřístupňovány sekvenčně v poli struktur, viz dál). Argument `semflg` i zde stanovuje jednak vznikající přístupová práva, jednak požadovaný přístup. I zde používáme bitového součtu `semflg` s `IPC_CREAT` při vytváření nebo `IPC_PRIVATE` na místě `key`. V případě úspěchu volání jádra vzniká komunikační oblast procesů ve formě množiny semaforů. Např.

```
id=semget(ftok("/usr/local/pool/semafor", 0), 3, 0777|IPC_CREAT);
```

vytváří množinu o 3 semaforech s přístupovými právy provádění operací nad semaforey pro všechny (vznik množiny semaforů můžeme prověřit příkazem `ipcs -s`, zrušit jej `ipcrm -s`, stejným způsobem jako u zpráv nebo sdílené paměti). Každý semafor množiny je struktura obsahující



Obr. 4.12 Hodnoty semaforu

```

ushort semval;    /* hodnota semaforu */
pid_t sempid;     /* PID procesu poslední operace */
ushort semncnt;   /* počet procesů čekajících na zvýšení současné
                  hodnoty semaforu */
ushort semzcnt;   /* počet procesů čekajících na nulovou hodnotu
                  semaforu */

```

a operace P a V nad množinou semaforů zadává proces pomocí volání jádra

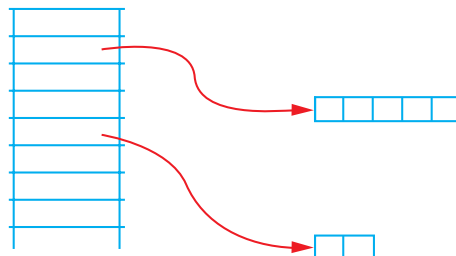
```
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

kde `semid` je použitá návratová hodnota `semget` (přidělený deskriptor), `sops` je ukazatel na pole operací nad semaforey, `nsops` je počet prvků tohoto pole. Prvek pole `sops`, struktura `sembuf`, obsahuje tyto položky

```
ushort sem_num; /* číslo semaforu */
```

deskriptory množin semaforů

množiny semaforů



Obr. 4.13 Množiny semaforů

```
short sem_op;    /* operace nad semaforem */
short sem_flg;   /* příznaky pro operaci nad semaforem */
(sem_num je index do pole semaforů). Operaci P nebo V vyjadřujeme pomocí číselné hodnoty
sem_op.
```

Je-li kladná, jádro ji použije ke zvýšení hodnoty semaforu a odblokuje všechny procesy, které čekají na zvýšení hodnoty semaforu (V).

Je-li 0, jádro kontroluje hodnotu semaforu, pokud není nulová, zvýší počet procesů čekajících na nulovou hodnotu semaforu a proces zablokuje.

Je-li záporná a její absolutní hodnota rovna hodnotě semaforu nebo menší, jádro přičte hodnotu `sem_op` (hodnota semaforu je snížena, P). Je-li pak hodnota semaforu 0, jádro aktivuje všechny zablokováné procesy čekající na nulovou hodnotu semaforu.

Je-li záporná a její absolutní hodnota je větší nebo rovna hodnotě semaforu, jádro proces zablokuje a zvýší o ni hodnotu semaforu.

Obsah `sem_flg` se nejčastěji používá s hodnotou `SEM_UNDO`. Je to užitečné nastavení v jádru, kterým proces požaduje zajištění vrácení poslední provedené operace nad semaforem v případě, že by náhle ukončil činnost (toto nastavení tedy obvykle nepoužívá v případě, že bude končit vědomě). Tím je zajištěno neuváznutí ostatních čekajících procesů. Dále lze použít `IPC_NOWAIT`, kdy proces může pokračovat bez uplatnění obsahu operace nad semaforem v případě, že by došlo k jeho zablokování.

Následující příklad obsahuje zdrojový text programu, kdy proces vytvoří semafor (množina obsahuje pouze 1 prvek), inicializuje jej na hodnotu 1 a čeká 10 minut, poté semafor zruší.

```
main(void)
{
    int sid;
    short semval[1];

    sid=semget(ftok("/usr/local/pool/semafor", 0), 1, 0777|IPC_CREAT);
    /* inicilizace výchozí hodnoty semaforu na 1 */
    semval[0]=1;
    semctl(sid, 1, SETALL, semval);
    sleep(600);
    /* po 10 minutách čekání semafor zrušíme */
    semctl(sid, 1, IPC_RMID, 0);
}
```

Dále si uvedme fragment programu, který využívá takto vytvořený semafor a používá na něj operaci P. Při volání jádra `semop` bude proces zablokován, pokud je hodnota semaforu záporná nebo nula. Při prvním provedení následujícího kódu proces přes operaci P projde, příští průchod je ale možný teprve po operaci V :

```
int csid;
struct sembuf semafor;
...
```

```

csid=semget(ftok("/usr/local/pool/semafor", 0), 1, 0777);
semafor.sem_num=0;
semafor.sem_op=-1; /* hodnota určuje operaci P */
semafor.sem_flg=SEM_UNDO; /* skončí-li proces, semafor je uvolněn */
/* provedení operace P */
semop(csid, &semafor, 1);
...

```

V příkladu jsme byli nuceni použít volání jádra `semctl` pro inicializaci a zrušení semaforu. `semctl` je určeno pro globální práci s celou množinou semaforů (její zrušení, zjištění nebo nastavení jejich atributů včetně její inicializace na výchozí hodnoty semaforů) a má formát

```

int  semctl(int semid, int semnum, int cmd,
            union semnum{ int val; struct semid_ds *buf;
            ushort *array;} arg);

```

Kde na místě `cmd` může být označen požadovaný příkaz se semaforem zadaný `semid` a `semnum`. Unie `arg` může být použita pro nastavení nebo zjištění hodnoty semaforu (jako `val` nebo `array`) a dalších hodnot. Také jsou čitelné (a měnitelné) hodnoty v `buf`, struktura `semid_ds` totiž obsahuje položky

```

struct ipc_perm sem_perm /* přístupová práva a vlastnictví */
ushort sem_nsems; /* počet semaforů v množině */
time_t sem_otime; /* čas naposledy provedené operace P nebo V */
time_t sem_ctime; /* čas poslední změny pomocí semctl */

```

Slovo přístupových práv množiny semaforů a manipulace s ním jsou stejné jako u fronty zpráv nebo sdílené paměti.

POSIX nezná volání jádra uvedených jmen. Práce se semaforey je také jiná. Semaforey jsou uvažovány ve své základní definici podle Dijkstry (viz úvod článku). Oproti SVID jsou manipulace vztaženy vždy k jednomu semaforu (není zaveden pojem množina semaforů). Stejně tak hodnota semaforu je zvyšována nebo snižována vždy pouze o hodnotu 1 (operace P a V).

Podle POSIXu může semafor být *bezejmenný* (unnamed) nebo *pojmenovaný* (named semaphore). Po jeho zpřístupnění je pro operace a jiné manipulace s ním dostupný přes přidělený deskriptor tabulky otevřených souborů (může to tak být, říká se v POSIXu, semafor pak je součástí dědictví mezi procesy). Bezejmenný semafor je vytvořen pomocí volání jádra `sem_init`. V parametrech stanovujeme identifikaci semaforu (typ `sem_t`) a jeho inicializační hodnotu. Pokud není návratová hodnota záporná, semafor je vytvořen a další volání jádra pro práci s ním se odkazují pomocí definice `sem_t`. Pojmenovaný semafor je vytvářen pomocí `sem_open`, v parametru zadáváme jeho jméno jako textový řetězec (jméno souboru typu semafor) nebo `sem_open` existující semafor otevírá (jako klient). V obou případech je návratová hodnota typu `sem_t` a přístup k semaforu je prováděn stejnými rutinami jako u bezejmenného semaforu. Jsou to `sem_wait` (operace P se zablokováním), `sem_trywait` (operace P v případě možného průchodu semaforem) a `sem_post` (operace V). Proces také může získat aktuální hodnotu semaforu pomocí `sem_getvalue`. Pojmenovaný semafor opouští proces pomocí `sem_close` (uzavírá `sem_t`) a zruší pojmenovaný semafor pomocí `sem_unlink` s odkazem na jméno semaforu. Bezejmenný semafor zaniká pomocí volání jádra `sem_destroy`.

Každý, kdo pozorně četl, dokáže formalizovat implementaci uvedených rutin standardu POSIX pro semafore v UNIXu pomocí používaných volání jádra a volání jádra systémů souborů. Přesto je dnešní systémy většinou neobsahují. Nejsou totiž stále součástí SVID a jsou příliš nové.

### 4.7 Komunikace procesů v síti

Podpora operačního systému UNIX pro zajištění služeb síťového propojení počítačů je principiálně postavena na síťové komunikaci procesů. Bazální dokumenty z počátku vývoje síťového prostředí pro UNIX (viz [LeffFabr86]) zahrnují spojování operačních systémů do problematiky IPC. Skutečně, aktivita procesu, který požaduje např. přenos obsahu souboru mezi různými počítači, se musí projevit v činnosti procesu jiného, který provede zpřístupnění souboru ve vzdáleném počítači. V úvodu kapitoly jsme uvedli zjednodušený obr. 4.2, charakteristický pro síťovou komunikaci procesů. Provoz sítí je principiálně zabezpečován aktivací procesů ve vzdálených uzlech, které provedou zpřístupnění požadovaného výpočetního zdroje. Návrh a implementace sítí v UNIXu tedy vycházela z rozšíření potřebných komunikačních prostředků procesů. Jako Berkeley sockets je označována jedna z prvních implementací skupiny volání jádra, která pracovala pro síťové spojení procesů, ale může být také využita pro spojování procesů v rámci téhož operačního systému.

Síťová komunikace procesů má své speciality, které ovlivňují nejenom implementaci, ale také správu a používání sítí. I dnes musí uživatel vnímat určitou topologii sítě a v rámci ní se pohybovat a těžko se v blízké budoucnosti na tom něco změnit. Základní princip uživatele, procesu, i jádra je vzájemné spojení dvou procesů. Spojení je možné, pokud jsou procesy v síti navzájem identifikovatelné. K identifikaci prostoru pro komunikaci zde nemůžeme použít dynamicky se měnící PID procesu ani převod jména souboru na celočíselnou identifikaci. Jak poznáme v kap. 7, uzel v síti je jednoznačně určen svou adresou IP (Internet Protocol address). Proces, který navazuje spojení, proto používá tuto adresu IP. V rámci vlastního uzlu se pak ohlašuje číselnou identifikací nazývanou číslo portu (port number), což je smluvená celočíselná hodnota, na základě které proces trvale naslouchající na síťovém spojení (např. síťový superserver **inetd**) vytvoří proces odpovídajícího požadavku na spojení (síťové služby mají pevně přidělená čísla portů, např. **telnet** má přidělenou hodnotu 23, ftp 21 atd.). Vytvořený proces je server komunikace procesů, který zpřístupňuje vzdálenému klientu požadovaný výpočetní zdroj. Informace nutné k provedení spojení jsou sdruženy a označeny termínem asociace (místní proces, místní uzel, vzdálený proces, vzdálený uzel a způsob komunikace tj. přenosový protokol). Pro síťovou komunikaci procesů byla zavedena buďto nová volání jádra, jako je tomu u Berkeley sockets (**socket**, **bind**, **connect**, **listen**, **accept**, **send**, **recv**, **shutdown** atd.), nebo bylo rozšířeno použití **ioctl**, **read** a **write** a dalších z nově zavedených technologií (např. **getmsg**, **putmsg** a **poll** ve STREAMS).

Vzhledem k tomu, že komunikace procesů v sítích, jak zde naznačujeme, je oproti místní IPC výrazně složitější a vyžaduje také znalostní aparát obecných pravidel a terminologie počítačových sítí, výklad a prozkoumání prostředků komunikace procesů v různých operačních systémech provedeme v rámci specializované kap. 7. Přípravnou bude také kap. 6, kde se seznámíme s mechanismy podporujícími implementaci síťového spojení procesů, jako je technologie PROU

DŮ (STREAMS), která je dominantní v síťovém rozhraní knihovny síťového spojení TLI (Transport Layer Interface) v UNIX SYSTEM V.



## 4.8 Správa IPC

V průběhu kapitoly jsme v uvedených příkladech několikrát použili příkaz **ipcs** a **ipcrm**. S jejich pomocí lze zobrazovat a odstraňovat existující fronty zpráv, sdílené paměti a semaforey. Jde o dodatečnou možnost práce s prostředky IPC, protože každý prostředek komunikace má správně být ve správě procesu serveru. Jak již bylo v kapitole několikrát zmíněno, používají se v případech neobvyklých stavů systému komunikujících procesů nebo v etapě ladění.

Příkaz **ipcs** slouží k výpisu jádrem evidovaných vytvořených a zatím nezrušených prostředků IPC. Jeho základní použití je bez parametrů, kdy vypisuje postupně všechny fronty zpráv, sdílené paměti a množiny semaforů, které jsou právě evidovány jádrem. Např.

### \$ ipcs

```
IPC status from /dev/kmem as of Sun Nov 3 12:09:21 1996
T      ID      KEY                MODE                OWNER      GROUP
Message Queues:
q      100     0x00010ecf  --rw-rw-rw-  skoc      user
Shared Memory:
m      201     0x00010ee4  --rw-rw-rw-  skoc      user
m      0       0x53637444  --rw-r--r--  root      sys
Semaphores:
s      5520    0x0f127f42  --ra-ra-ra-  root      sys
```

V záhlaví výpisu jsou uvedeny jednotlivé atributy IPC. T je typ IPC (q je fronta zpráv, m sdílená paměť, s semafor), ID je deskriptor přidělený jádrem, KEY klíč. MODE jsou přístupová práva a příznak, první znak položky je identifikace očekávání příchodu dat (znakem S), druhý znak očekávání čtení dat (R). Je-li na obou místech znak – (naš případ), žádný proces na vznik situace nečeká. Zbývajících 9 znaků jsou přístupová práva pro vlastníka, skupinu a ostatní (znaky rw) analogicky s přístupovými právy v i-užlu souboru. Je-li uvedeno –, právo je odepráno, a znamená možnost změny (podle angl. *alter*).

Výpis příkazu můžeme omezit postupně pouze na fronty zpráv (volba **-q**), sdílené paměti (**-m**) a semaforey (**-s**). V rámci příkazu lze použít také další volby, které mají význam podrobnějšího výpisu, rozšířeného např. o velikosti alokované paměti nebo o procesy poslední manipulace s IPC atd. Všechny příkazem dostupné informace lze požadovat volbou **-a**. Záhlaví u výpisu pak obsahuje pochopitelně také další texty, které je dobré konzultovat s provozní dokumentací.

Požaduje-li oprávněný uživatel (tj. vlastník IPC nebo superuživatel **root**) zrušení prostředku komunikace IPC, může toho docílit příkazem **ipcrm**. Jde o okamžité zrušení, kdy se zúčastněné procesy v případě snahy o další komunikaci mohou dostat do stavu neobvyklého, pokud však mají ošetřeny všechny chybové návraty z odpovídajících volání jádra, jenom ukončí svou činnost. Nejjednodušší cesta správného nastavení všech IPC je pochopitelně nový start operačního systému, protože jádro všechny IPC zruší svým ukončením. Při novém startu jsou tabulky inicializovány jako prázdné a jsou naplněny (a inicializovány) servery odpovídajících systémů procesů. Přesto lze použít **ipcrm** s některou z voleb **-q** (týká se front zpráv) **-m** (sdílené paměti) a **-s** (semaforey) a deskriptor IPC (pod textem záhlaví ID ve výpisu **ipcs**). Je také možné v příkazovém řádku namísto deskriptoru použít k identifikaci klíč IPC (KEY). Volba se pak používá s velkými písmeny **-Q**, **-M** a **-S**. Příklady jsme uvedli v průběhu kapitoly.

<sup>1</sup>Jádro je vytvořeno (vygenerováno) s nastavením tzv. `pipedev`, zařízení pro rouru, což je speciální soubor svazku, který bude používán při vytváření každé obyčejné roury. Obvykle je zařízení pro rouru shodné s kořenovým (`rootdev`) svazkem systému.

<sup>2</sup>Binární semafor je zúžením. Semafor nabývá hodnot buď 1 nebo 0, volný nebo obsazený. Semafor pouze blokuje, fronta není udržována.

# 5 UŽIVATEL

Každý operační systém je vystavěn pro potřeby uživatelů, tedy lidí, kteří vyžadují služby zpracování dat. Jejich komunikace se strojem je dnes především interaktivní. Člověk u obrazovky používá příkazy pro další činnost stroje na základě předchozích odpovědí. Člověk vede se strojem určitý rozhovor, jehož obsah odpovídá používané aplikaci a bázi dat, kterou stroj i člověk disponují. UNIX je zaměřen právě na takovou činnost počítačů. Podporuje interaktivní komunikaci uživatelů a používaných aplikací, a to několika najednou. K tomu, aby rozeznával účastníky a jimi vyžadované aplikace, které spolu souvisejí v určité relaci (sezení, session) uživatele a stroje, musí uživatele a jejich aktivity (procesy) evidovat a odlišovat. Chce-li člověk vstoupit do relace se strojem, pak musí prokázat svou identifikaci, a pokud je taková identifikace operačním systémem registrována a je oprávněná, relaci s člověkem otevírá. To je obecný přístup uživatele k výpočetnímu systému.

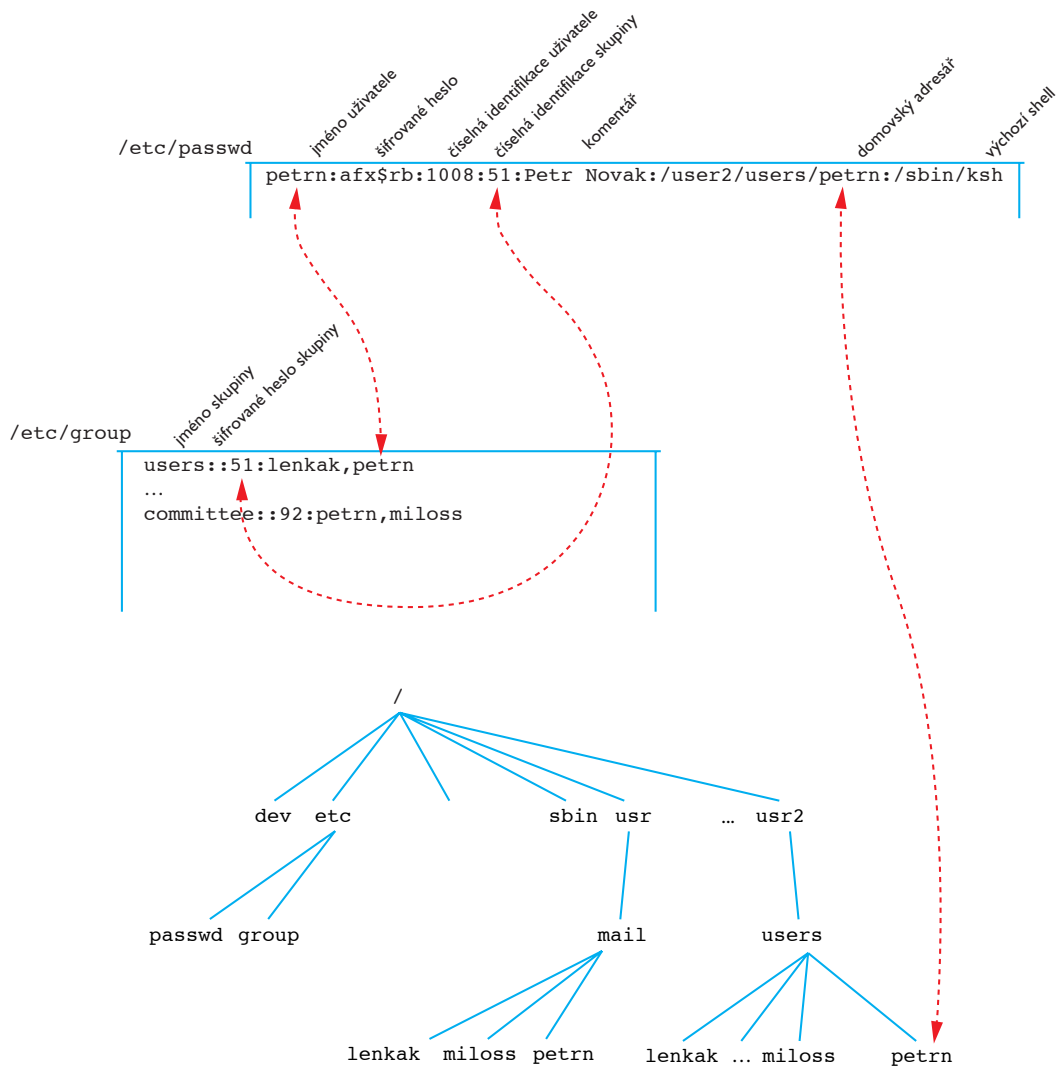
Uvedený způsob komunikace se strojem používá i programátor, který aplikace pro ostatní uživatele vytváří. Musí navíc znát prostředí práce uživatele, do kterého aplikaci programuje, kdy prostředí poskytované operačním systémem (pokud je aplikace dobře programovaná, uživatel nemusí prostředí operačního systému příliš znát, více mu slouží komfortní prostředí vlastní aplikace). Přestože programátor musí dobře znát rozhraní práce procesů v operačním systému, nemusí se zajímat o celkovou funkci a provoz operačního systému. Ve skutečnosti je neovlivní, stejně jako obyčejný uživatel.

Za účelem vlastního bezproblémového chodu operačního systému je ustanoven zvláštní uživatel s neomezenými právy přístupu k výpočetnímu systému. Jeho přístup je v UNIXu implementován prostřednictvím uživatele se jmennou identifikací `root`. Práce v rámci sezení takového uživatele umožňuje hluboké zásahy do chodu operačního systému. `root` má tedy vysoké pravomoci, ale také velkou zodpovědnost. Chybný příkaz totiž může paralyzovat práci ostatních uživatelů, programátorů nebo dalších uživatelů. Organizačně je obvykle ustanovena jedna osoba, která má oprávnění přístupu k takovému sezení; říkáme jí správce operačního systému. Přístup každého uživatele lze kontrolovat heslem, kterým se musí uživatel prokázat při požadavku navázání sezení. Znalost hesla uživatele `root` pak znamená neomezený přístup ke všem systémovým zdrojům.

## 5.1 Registrace uživatelů

Ve správě uživatele `root` jsou pro registraci uživatelů tabulky v obsahu souborů `/etc/passwd` a `/etc/group`. Klíčová je tabulka `/etc/passwd`, obsahuje jména uživatelů a jim přiřazené číselné hodnoty. Každý registrovaný uživatel má v souboru vyhrazen jeden řádek. Na řádku jsou atributy uživatele odděleny znakem `:`. Registrovat nového uživatele v systému (a umožnit mu tak vstup a práci v něm) znamená především připojit další řádek jeho evidence k tomuto souboru (třeba obyčejným textovým editorem). Obsah jednotlivých položek (tj. atributů uživatele) řádku souboru v příkladu ukazuje obr. 5.1.

V tabulkách obou souborů na obrázku jsou uvedeny všechny atributy, které potřebuje operační systém znát pro registraci uživatele. Některé mutace UNIXu evidují další informace o uživateli v tabulkách jiných souborů, které však mají pouze doprovodný charakter nebo souvisí se speciální ochranou UNIXu a probereme je v kap. 9.



Obr. 5.1 Atributy uživatele v `/etc/passwd` a `/etc/group`

*Jméno uživatele* (name of user, **petrn**) je první atribut v souboru. Jednoznačně koresponduje s rovněž unikátní *číselnou identifikací uživatele* (1008), která má zkratku UID (numerical user identification). Se jménem pracuje operační systém při komunikaci s uživatelem, UID používá ve svých interních strukturách (v jádru, i-uzlech atd.). Změna jména nebo UID v souboru **/etc/passwd** může být fatální pro samotného uživatele, pokud nedojde současně ke změně UID (případně jmen) všude, kde je záznam o uživateli pevný (i-uzly, seznamy uživatelů). To rovněž platí i o *jménu skupiny* (name of group, **users**) a její *číselné identifikaci skupiny* se zkratkou GID (numerical group identification), která je uváděna jako čtvrtá položka v **/etc/passwd** (51) a která odkazuje do souboru **/etc/group**, který eviduje všechny skupiny uživatelů v operačním systému. I zde je každá skupina evidována na samostatném řádku. Znak **:** je oddělovací pro atributy skupiny. Jméno skupiny je definováno jako první atribut, GID jako třetí, čtvrtý je seznam znakem **,** oddělených uživatelů, kteří do skupiny patří. Druhá položka je vyhrazena pro heslo skupiny. Položka je však většinou nepoužívaná a je prázdná, protože krytí přístupu k datům určité skupiny je zajišťováno jinými prostředky (ACL, viz kap. 9). Všimněte si, že náš uživatel **petrn** je evidován ve dvou skupinách (**users** a **commitee**). Přitom v **/etc/passwd** je uvedena pouze skupina jedna (**users**). Taková skupina je pro uživatele výchozí. Její GID je používáno při práci uživatele např. při vytváření nových souborů v evidenci v i-uzlu nebo při kontrole přístupu k datům či jiným výpočetním zdrojům. Změnu GID na jiné podle evidence v **/etc/group** umožňuje příkaz **newgrp**, který vyžaduje v případě existence heslo pro změnu skupiny. Dnes je používán málo, přestože jej SVID i POSIX (!) obsahuje. Práce se skupinami uživatelů dnes totiž probíhá především pod řízením ACL (**A**ccess **C**ontrol **L**ist, viz kap. 9).

*Šifrované heslo* (encrypted password, **afx\$rb**) uživatele (druhá položka v **/etc/passwd**) je ekvivalent textového řetězce, kterým je kryt přístup uživatele k práci v operačním systému (k sezení). Bez znalosti hesla nemůže uživatel zahájit sezení (viz 2.2). Při pokusu o vytvoření sezení proces **login** přijímá od uživatele z klávesnice nešifrované heslo (bez opisu na obrazovku), rozšifruje heslo z druhé položky **/etc/passwd** a v případě shody sezení zahájí, v opačném případě sezení zahájit odmítne. Přestože správce systému může uživatele vytvořit editací souboru **/etc/passwd**, položku šifrovaného hesla nechává prázdnou. Registrace je bez hesla po dobu, než uživatel v průběhu svého sezení použije příkaz **passwd**, kterým heslo vytvoří a naplní tak odpovídající položku v **/etc/passwd**. Příkaz slouží i pro změnu stávajícího hesla. Privilegovaný uživatel jej může použít pro jiného uživatele, protože doba od vytvoření záznamu v **/etc/passwd** po zavedení hesla by měla být co nejkratší. Znalost a změna hesla je věcí uživatele. Pouze on zná heslo své registrace. Zapomene-li jej, správce systému pod sezením uživatele **root** může heslo zrušit (vyprázdněním položky v **/etc/passwd** např. editorem), ale ne rozluštit. Zapomene-li správce systému heslo uživatele **root**, není definována standardní cesta k přístupu k jeho sezení kromě reinstalace celého operačního systému. Ochranu dat a související bezpečnost provozu operačního systému, která začíná v UNIXu oddělením uživatelů s exkluzivními právy ke svým datům a krytí jejich sezení heslem, budeme diskutovat především v kap. 9, kde se seznámíme s kritickými místy bezpečnosti UNIXu a metodami, které je zeslabují nebo zcela odstraňují. Původní algoritmus šifrování hesel v UNIXu byl v počátcích systému vytvořen velmi dobře a časem byl zdokonalován. V 80. letech bylo dokonce výnosem vlády Spojených států zakázáno prodávat ho jako součást UNIXu mimo území USA. Je to tentýž algoritmus, kterým může libovolný uživatel šifrovat obsah obyčejných souborů příkazem **crypt** a **encrypt** (viz 5.3) nebo vnitřním příkazem X editorů **ed** nebo **ex**, kde ovšem textový klíč před zahájením šifrování uživatel zadává. Klíč šifrování hesel není

v komerční instalaci operačního systému zjištělný, a proto správce nemůže hesla jednoduše rozšifrovat. Položka hesla v `/etc/passwd` je ale i tak předmětem zájmu procesů, které usilují o nelegální průnik do operačního systému. Proto jsou hesla často přesouvána do jiného místa v systému souborů, např. do souboru `/etc/shadow`, který je dostupný pouze uživateli `root`. Soubor `/etc/passwd` nebo `/etc/group` nelze takto zůžit v přístupových právech, protože obsahují bijekci jména uživatele a UID nebo jména skupiny a GID, které čtou i mnohé procesy obyčejných uživatelů. Na místě položky šifrovaného hesla v `/etc/passwd` někdy správce systému vkládá text, např. `nologin`. Určuje tak tzv. pseudouzivatele, který slouží jako vlastník určitých systémových zdrojů, ale nikoliv jako běžný uživatel s možností přihlásit se.

Položka *komentář* je v SVID definována jako free field (volná položka) a jde skutečně o volný text, který obvykle charakterizuje uživatelské zařazení. Podle SVID je možné dále komentář strukturovat, ale tato struktura je pak věcí implementace. Nejrozšířenější je tzv. čárková podoba komentáře (tzv. GECOS), kterou používaly systémy v Bell Laboratories a převzaly zejména systémy BSD. Komentář je rozdělen na texty oddělené znakem čárka (,) a texty mají obvykle po řadě význam celého jména uživatele, pracovního umístění, telefonního čísla do práce a telefonního čísla domů.

*Domovským adresářem* (home directory nebo initial-working-directory, `/usr2/users/petrn`) začíná oblast uživatelských dat ve stromu adresářů systému souborů. Domovský adresář je ve vlastnictví uživatele a skupiny, která je mu přiřazena jako implicitní. Pouze uživatel má právo rozšiřovat podstrom tohoto adresáře a ukládat nebo modifikovat data jeho obyčejných souborů. Členům jeho skupiny a ostatním je obvykle dovoleno data číst (viz maska vytváření souborů v čl. 2.2).

*Výchozí shell* (program to use as command interpreter) je položka `/etc/passwd`, která obsahuje úplné jméno souboru s programem, jenž použije proces **login** k výměně svého textového segmentu (voláním jádra `exec`), aby tak zahájil uživatelské sezení. Může to být Bourne shell (`/sbin/sh`), C-shell (`/usr/sbin/csh`), KornShell (`/sbin/ksh`) nebo jiný používaný příkazový interpret uživatelského sezení nebo jakýkoliv jiný interaktivní aplikační program, který uživateli zprostředkuje sezení. Je-li položka prázdná, je použit Bourne shell.

Obě tabulky v `/etc/passwd` a `/etc/group`, jak již bylo řečeno, mohou být rozšiřovány nebo modifikovány obyčejným editorem. Správce systému přitom nesmí zanedbat některé důležité aspekty. UID musí být jedinečné, GID musí být evidováno, domovský adresář musí existovat, mít odpovídající přístupová práva a musí být ve vlastnictví právě vytvořeného uživatele, výchozí shell musí v systému existovat.

Automatizovaná údržba registrace uživatelů je v UNIXu podporována také několika systémovými příkazy. Jejich jména bývají různá a často lze také použít grafické prostředí X pracovní plochy privilegovaného uživatele. SVID definuje příkaz **useradd**, kterým privilegovaný uživatel registruje nového uživatele. Program realizuje vznik domovského adresáře a kontroluje odpovídající vazby v tabulkách registrace uživatele. Příkaz má řadu voleb, které mnohdy souvisí také s bezpečností (viz kap. 9). Příkaz **userdel** naopak registraci uživatele ruší. Volebou **-r** lze vynutit také zrušení domovského adresáře včetně jeho obsahu (v případě, že data uživatele zůstanou na disku, jejich vlastnictví je přisuzováno podle údajů tabulky `/etc/default/userdel`). Konečně pro změnu atributů uživatele je k dispozici příkaz **usermod**. Pro údržbu skupin, do kterých jsou uživatelé sdružováni podle tabulky `/etc/group`, SVID definuje obdobné příkazy se jmény **groupadd**, **groupdel** a **groupmod**.

Konečně pro kontrolu konzistence tabulek uživatelů a skupin SVID nabízí programy **pwck** pro kontrolu uživatelů a **grpck** pro kontrolu skupin. Oba programy v případě zjištěných nedostatků tyto komentují na standardní výstup. Konečně i obyčejný uživatel může požádat o výpis všech registrovaných uživatelů příkazem **listusers**. Při použití vhodné volby může výpis zúžit např. pouze na uživatele určité skupiny takto:

```
$ listusers -g commitee,sys
```

je žádost o seznam všech uživatelů skupin se jmennou identifikací **commitee** a **sys**. Pro získávání informací registrovaných uživatelů je také v SVID doporučován a běžně implementován příkaz **logins**, pomocí kterého lze také zjistit případné nesrovnalosti v tabulce uživatelů a skupin. Např.

```
$ logins -p
```

vypíše seznam všech uživatelů, jejichž vstup do systému není chráněn heslem. Všechny (zajímavé) možnosti tohoto příkazu nalezneme v odpovídající části provozní dokumentace.

Mnohdy užitečný příkaz je **finger**, pomocí kterého lze získat informace o registraci uživatele. Byl vytvořen při implementaci síťových služeb (umožňuje zadávat i uživatele vzdálených systémů), a proto jej ani SVID ani POSIX zatím nezahrnují. **finger** má jednoduché použití. V argumentu zadáváme jméno uživatele, na standardní výstup získáme obsažné informace, např.:

```
$ finger petr
```

```
Login name: petr                               In real life: Petr Novák
Directory: /usr2/users/petr                     Shell: /bin/ksh
Last login Jan 13 09:40:37 on tty2
Project: Otevíráme novou pobočku v Brně.
Plan:
Poštu čtu dvakrát denně, dopoledne
a v průběhu odpoledních hodin; také o víkendu.
```

Vypisuje informace podle obsahu souborů s registrací (příkaz má volby) a informaci, zda je uživatel přihlášen nebo kdy byl naposledy přihlášen. Používá-li systém čárkovou strukturu komentáře v **/etc/passwd**, (GECOS) jsou informace podle ní strukturovány ve výpisu. Uživatel sám může výpis doplnit obsahem souborů **.project** a **.plan** svého domovského adresáře, jak je vidět z příkladu.

Tabulky registrace uživatelů a skupin jsou veřejně přístupné pro čtení. Libovolný proces může otevírat soubory **/etc/passwd** a **/etc/group** a hledat odpovídající atributy uživatele nebo skupiny na základě znalosti jejich jmenné nebo číselné identifikace a mnohé procesy tyto informace pro korektní splnění své funkce nutně potřebují znát. SVID programátorům takových procesů nabízí několik funkcí, které orientaci v uživateli a skupinách usnadňují. Definuje hlavičkové soubory **<pwd.h>** a **<grp.h>**, ve kterých jsou definovány struktury **passwd** a **group** s položkami odpovídajícími atributům uvedených tabulek v **/etc/passwd** a **/etc/group** (**passwd** neobsahuje položku pro komentář, **group** položku šifrovaného hesla skupiny). Funkce **getpwnam** ve svém argumentu přijímá jmennou identifikaci uživatele, **getpwuid** číselnou identifikaci a obě vrací ukazatel na strukturu **passwd**. Dále může proces použít funkce pro postupné čtení souboru s atributy uživatelů, **getpwent** čte a vrací ukazatel na následujícího uživatele, **setpwent** nastavuje čtení na prvního uživatele a **endpwent** uzavírá takovéto cyklické čtení. Obecně lze číst z libovolného souboru stejného formátu

jako `/etc/passwd` pomocí funkce `fgetpwent`, která v parametru obsahuje deskriptor otevřeného souboru (pomocí funkce `fopen`). Jmennou identifikaci uživatele, který proces spustil (pokud proces vznikl v rámci sezení, tzn. je spojen s určitým terminálem), proces získá pomocí funkce `getlogin`, a to odkazem na pole z její návratové hodnoty. Stejně tak jsou definovány adekvátní funkce pro práci se skupinami funkce `getgrnam`, `getgrgid` a `getgrent`, vrací ukazatel na strukturu `group` podle zadaného jména nebo GID v parametru anebo následující skupinu v pořadí ze souboru `/etc/group`. Existují i funkce `setgrent`, `endgrent` a `fgetgrent`, jejichž použití pro seznam skupin je analogické jako při čtení seznamu uživatelů.

Přestože se POSIX zatím nevěnuje definicím příkazů správy operačního systému, pro evidenci uživatelů uvádí sekci tzv. systémových databází (System Databases), ve které je definována databáze přístupových uživatelů a přístupových skupin. Věta databáze uživatelů je prakticky shodná s definicí v SVID. Struktura se jménem `passwd` pouze neobsahuje položku šifrovaného hesla a rovněž komentář, struktura `group` i zde pouze neobsahuje šifrované heslo skupiny. POSIX definuje funkce `getpwuid` a `getpwnam`, `getgrgid` a `getgrnam` shodné syntaxe i obsahu jako SVID. Pro získání jména uživatele je i zde definována funkce

```
char *getlogin(void);  
a jí odpovídající příkaz login.
```

Význačný uživatel je `root`. Je to uživatel s číselnou identifikací 0, která je směrodatná. Proces tohoto EUID má neomezená práva. Znamená to, že je mu umožněno měnit prakticky libovolná systémová nebo uživatelská data. Pouze uživatel s `UID=0` má takové pravomoci. Nejčastěji je v operačním systému jen jeden takový uživatel se jménem `root`. Správci systému však nic nebrání založit uživatele shodných UID, a proto i privilegovaných uživatelů může být prakticky více. To se využívá např. při některých jednoúčelových systémových akcích, kdy na místě výchozího shellu v `/etc/passwd` u takového uživatele je uvedena cesta k programu, který vykoná určitou systémovou akci a sezení se po ukončení této akce ukončí. Jde např. o možnost zastavení systému osobou, která zná heslo uživatele, jehož výchozí shell je program **shutdown** (podrobněji viz kap. 10). Podobně jsou registrováni uživatelé, jejichž sezení je nedostupné (na místě šifrovaného hesla je uveden text např. `nologin` nebo `*` nebo jiný text, který nelze očekávat jako výsledek šifrovacího algoritmu) a vystupují jako vlastníci určitých systémových zdrojů, protože jsou účastníky stejné skupiny jako `root`. Uživatelé registrovaní za jiným účelem, než je běžné interaktivní sezení, jsou v literatuře označováni jako *pseudouživatelé* a v každé instalaci UNIXu je jich několik. Obvykle jsou evidováni k zajištění určité systémové nebo síťové akce.

## 5.2 Identifikace

Uživatel vstupuje do operačního systému tak, že prokáže určitou identifikaci. Proces **getty** z klávesnice přijímá jméno uživatele a **login** prověřuje uživatelem zapsané heslo. V případě shody s údaji registrace uživatele proces **login** (všimněte si, že musí mít práva privilegovaného uživatele) nastavuje UID, GID, domovský adresář a mění sám sebe na proces řízený programem výchozího shellu. Nastavením těchto atributů procesu výchozího shellu se uživatel stává účastníkem výpočetního systému. Proces, který získal jeho atributy v průběhu přihlašování, je v jeho vlastnictví (má nastaveno jeho UID) a je rodičem dalších procesů, které na pokyn uživatele vytváří. Podle zákonitostí uvedených v kap. 2 každý dětský proces a tím celý podstrom procesů v rámci uživatelova sezení je ve vlastnictví uživatele.



Oblast dat, v níž tyto procesy operují, je dána výchozím domovským adresářem uživatele na základě nastavení jeho vlastnictví a přístupových práv ve smyslu modifikace této datové základny výhradně vlastníkem uživatelem, pokud tento neřekne jinak. S tím souvisí také děděná maska vytváření nových souborů, která znamená ukládání nových dat v totéž významu. Kromě privilegovaných procesů (ve vlastnictví uživatele `root`) pak ostatní procesy v operačním systému mohou přistupovat k datům nebo procesům jiného uživatele pouze takovým způsobem, jaký uživatel stanoví (implicitně je např. povoleno čtení všech dat uživatele). Rovněž tak jiná data, procesy nebo periferie jsou pro uživatele dostupná takovým způsobem, jakým stanoví vlastníci uživatel. Tento princip vlastnictví procesů, dat a ostatních výpočetních zdrojů v kontextu s obsahem tabulek registrace uživatelů je obecný pro práci v operačním systému UNIX.

Pro praktický provoz operačního systému v uvedeném kontextu potřebuje běžící operační systém identifikovat vlastnictví a podle toho s ním pracovat. K tomu slouží dále uváděné identifikace.

UID (user identification) je číselná identifikace uživatele podle registrace v `/etc/passwd` a procesům, datům a jiným výpočetním zdrojům je přidělována při vzniku podle vlastníka procesu provádějícího operaci. Při vzniku elementu v systému souborů je vkládána jako jeden z atributů `i-uzlu`. Vzniklý proces ji má přiřazenu v odpovídající struktuře jádra `user` (viz odst. 2.4.4).

GID (group identification) je analogická číselná identifikace skupiny podle registrace v `/etc/group`.

EUID (effective user identification) je efektivní číselná identifikace uživatele. Je to identifikace, která je nastavena procesu pro další provádění. Je to opět identifikace některého z uživatelů podle `/etc/passwd`, která umožňuje procesu chovat se ve smyslu přístupových práv tohoto uživatele.

UNIX používá EUID např. pro zpřístupnění běžně nedostupných systémových služeb i obyčejnému uživateli, a to při práci konkrétního programu. Pokud není řečeno jinak, je EUID shodné s UID (které je v tomto kontextu označováno jako reálné UID). Vlastník programu však může nastavením tzv. `s-bitu` (viz čl. 2.2) ve slově přístupových práv souboru s programem umožnit na dobu provádění používat jako EUID identifikaci vlastníka programu, kterým se pak proces prokazuje v přístupu k výpočetním zdrojům. EUID má tak při běhu procesu větší důležitost než UID. Změnu EUID proces (pokud je oprávněn) provádí voláním jádra `setuid`, viz čl. 2.2.

EGID (effective group identification) je analogie EUID pro skupinu. I zde, pokud není řečeno jinak (nastavením `s-bitu` nebo voláním jádra `setgid`), je EGID shodné s GID.

V rámci identifikace přihlášeného uživatele je zaveden také termín identifikace sezení, SID (session identification). Je to celočíselná kladná hodnota, která je shodná s PID vedoucího skupiny procesů daného sezení (PGID). V kap. 2 jsme se zmínili, že proces, který řídí sezení uživatele (obvykle nějaký shell), je vedoucím skupiny procesů, které vznikají jako jeho potomci. SID je nastaveno v okamžiku přihlášení na toto PGID (`process group identification`). Přestože každý další dětský proces může dosáhnout osamostatnění (pomocí `setpgrp`), SID zůstává totéž. Různé verze UNIXu nakládaly s takto prezentovaným faktem různě a různě SID interpretovaly a používaly. Jednoznačnost přineslo vydání standardu POSIX, který SVID respektuje, přestože i nadále podporuje předchozí volání jádra. POSIX sdružuje procesy do skupin. Příslušnost k určité skupině získá proces v návratové hodnotě volání jádra

```
#include <sys/types.h>
pid_t getpgrp(void);
```

kteřé nikdy neskončí s chybou. Návratová hodnota obsahuje PGID vedoucího skupiny, do které proces patří. Změnu příslušnosti ke skupině může proces vykonat pomocí volání jádra

```
int setpgid(pid_t pid, pid_t pgid);
```

kde procesu s identifikací `pid` nastavujeme skupinu `pgid`. Pokud skupina s `pgid` neexistuje, je vytvořena a proces s `pid` se stává jejím vedoucím. Pokud je na místě `pid` 0, nastavení se vztahuje k volajícímu procesu. Pokud je `pgid` 0, je použito PID volajícího procesu. Volání jádra `setpgrp` POSIX nezná. SID je nastaveno pomocí volání jádra (které je privilegované)

```
pid_t setsid(void);
```

Volající proces tak je nastaven na vedoucího nové skupiny procesů (pokud jím již není) a je vytvořeno nové sezení, které nemá nastaveno řídicí terminál. SID i PGID je nastaveno podle PID. SID je součástí struktury jádra `proc` procesu a v SVID (ale ne v POSIXu) je lze získat voláním jádra

```
pid_t getsid(pid_t pid);
```

kde je-li v argumentu použita 0, v návratové hodnotě získáme SID volajícího procesu.

Sezení každého přihlášeného uživatele je registrováno ve strukturách jádra. Každé sezení má svou identifikaci SID a je charakterizováno skupinou procesů, které jsou součástí stromové struktury, jejímž kořenem je výchozí shell.

Příkazem **who** lze získat seznam všech přihlášených uživatelů. **who** je jeho doplňkem, jehož výpis obsahuje seznam všech právě běžících procesů každého přihlášeného uživatele. Oba příkazy lze pocho-pitelně zkoumat nejlépe s provozní dokumentací v ruce. Uveďme si příklad:

```
$ who -Tu
```

```
lenkak      +      tty2   Jan 10 10:29      old    4829
root        -      tty0   Jan 11 11:01      .        5079
petrn       +      tty1   Jan 11 11:19      00:07  5459
```

Přihlášení jsou uživatelé se jmennou identifikací `lenkak`, `root` a `petrn`. K terminálu uživatele s označením `+` má pomocí komunikačních prostředků přístup každý jiný přihlášený uživatel. Uživatelé pracují na terminálech podle speciálních souborů `tty[0-2]`. V další položce výpisu každého řádku je uvedeno datum a čas přihlášení. Následuje označení aktivity uživatele, dále je uveden počet minut, kdy uživatel neaktivoval žádný proces jeho sezení. Znak `.` znamená aktivitu a `old` je sezení, které za posledních 24 hodin nebylo využito. Poslední položka je PID výchozího shellu. Pokud v příkazovém řádku použijete volbu **-H**, první řádek výpisu je hlavička pro jednotlivé položky výpisu.

Podle SVID má **who** další možnosti práce s výpisem (POSIX definuje pouze **-u**, **-T**, a **-m**), např. je možné požadovat výpis nepřihlášených uživatelů pomocí **-l**, výpis pouze seznamu uživatelů a jejich celkový počet **-q** a **-p** pro zobrazení všech ostatních aktivních procesů výchozích shellů, které jsou aktivní a byly vytvořeny procesem **init** jako proces komunikace s uživatelem (spawned by init), tj. např. právě přihlašovaných uživatelů, a **-d** takové procesy výchozích shellů, které skončily a nebyly znova vytvořeny procesem **init** (tzv. respawn). Obecný formát výpisu příkazu je definován jako

```
name [state] line time [idle] [pid] [comment] [exit]
```

ze kterého vyplývá formát výpisu bez použití jakékoli volby (`name line time`, tj. jméno uživatele, terminál a čas přihlášení), který je také shodný s použitím volby **-s** (v POSIXu **-m**). **who** je program,

který poskytuje také informace cenné pro správce systému, jako je čas startu systému (volba **-b**), čas změny systémových hodin (**-t**) a současnou nastavenou úroveň operačního systému (**-r**, viz kap. 10).

Volbou **-a** získáme výpis všech možných informací, které je **who** schopen zobrazit. **who** nepoužívá při své práci žádné volání jádra, aby získal informace uložené ve strukturách jádra. Systém je sám aktivní ve smyslu poskytování těchto informací a udržuje aktuální data v souborech `utmp` a `wtmp`. Formát jejich obsahu je podle dokumentů závislý na implementaci, rovněž tak jejich umístění. Původně umístěny v adresáři `/etc`, jsou dnes obsahem adresáře `/var/adm` a další informace při jejich zpracování uvedeme ve čl. 5.4.

Seznam procesů pro aktivního uživatele získáváme pomocí příkazu **ps**, jehož použití jsme uvedli v odst. 2.4.4, a to jeho volbou **-U**, za kterou následuje jméno nebo UID uživatele (analogicky pro skupinu volbou **-G**). Procesy spojené s určitým terminálem pak vypíšeme volbou **-t**, za kterou následuje jméno speciálního souboru (line z výpisu **who**). Za každou z uvedených voleb může být uvedena více než jedna identifikace oddělená mezerou.

Často je také používán již zmíněný **whodo** (POSIX jej nezná), kdy se z výpisu dovídáme seznam procesů všech sezení, nebo jen vybraného (jméno uživatele je pak argumentem) aktuálního sezení, např.

```
$ whodo -h
tty2      lenkak      old
          tty2      4829      0:00 ksh
tty0      root        11:01
          tty0      5079      0:00 csh
tty1      petrnr      11:19
          tty1      5459      0:00 ksh
          tty1      5551      0:00 whodo
```

Příkaz je realizován scénářem, který využívá výpisu příkazů **who** a **ps**. V našem příkladu jsme volbou **-h** potlačili výpis hlavičky, která podle SVID obsahuje současný čas, jak dlouho je systém zaveden, počet přihlášených uživatelů a průměrný počet procesů ve frontě připravených k běhu za posledních 1, 5 a 15 minut. U každého uživatele je uveden seznam procesů, jejich PID, čas a jméno procesu z výpisu **ps**.

Pro získání informací o seznamu přihlášených uživatelů se také používá program **finger**, který jsme uvedli při zjišťování registrace určitého uživatele (čl. 5.1). Pokud je v příkazovém řádku **finger** uveden beze jmen uživatelů, na standardní výstup je zobrazen seznam právě přihlášených uživatelů.

Pro uživatele jsou mnohdy důležité informace vlastní identifikace v době sezení. Např.

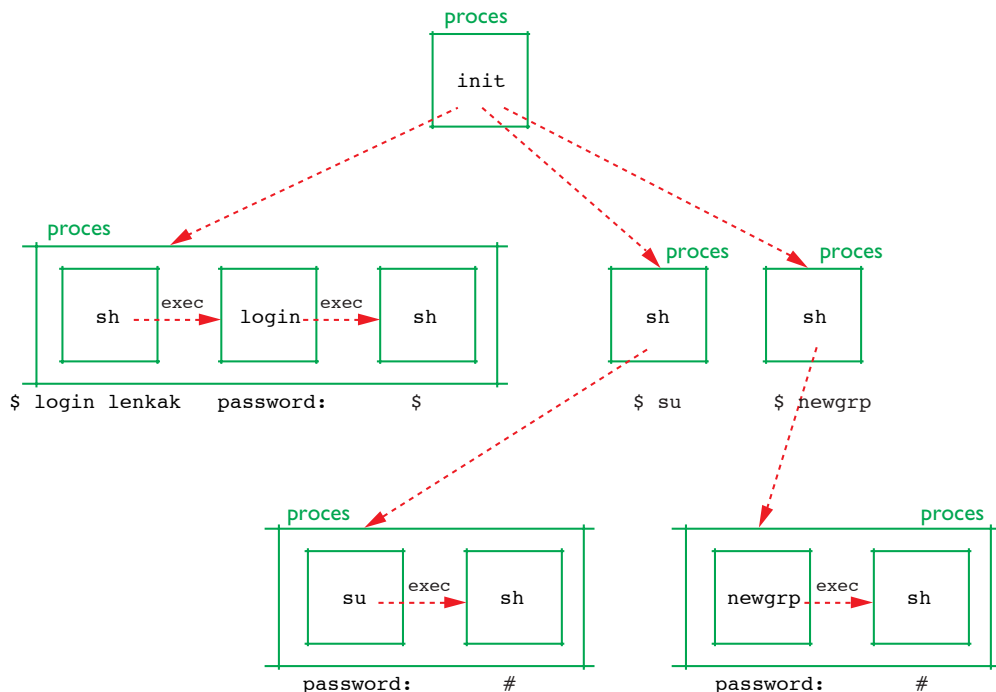
```
$ who am I
petrn      tty1 Jan 11 11:19
```

sdělí identifikaci `name`, `line`, a `time` (zvláště `line` nás zajímá často). Při psaní scénářů je také užitečný např. příkaz **logname**, který vrací na standardní výstup jméno uživatele (`name`). SVID i POSIX definují také (v různém formátu) příkaz **id**. Jde o získání jmenové a číselné identifikace samotného uživatele a podle SVID i jmen spuštěných procesů. POSIX navíc umožňuje zadávat jméno uživatele v parametru a získávat tak identifikace ostatních uživatelů. Příkaz rozlišuje reálnou a efektivní identifikaci uživatele, pokud se liší. Oba dokumenty také (zde shodně) definují příkaz **tty**, kterým uživatel získává jméno speciálního souboru řídicího terminálu (`line`). Přístup a práce uživatele s prostředím

uživatelská terminál si podrobněji probereme v kap. 6, kde uvedeme i další důležité příkazy terminálu uživatelská sezení (např. **stty** nebo **tabs**).

V rámci sezení může uživatel používat příkazy změny své identifikace nebo proměny na jiného uživatele či skupinu. Šifrovaná hesla uživatelů správce systému obecně nezná a zavádí a mění si je uživatel, který k tomu používá příkaz **passwd**. Program požaduje zápis hesla z klávesnice, a protože jej neopíše na terminál, musí heslo uživatel opakovat pro srovnání. Privilegovaný uživatel může použít v příkazu argument, kterým určí uživatele, kterému mění heslo na nové. Rovněž tak může privilegovaný uživatel **root** editovat soubor **/etc/passwd** a na řádku odpovídajícího uživatele vypustit položku šifrovaného hesla. Tak se stává sezení uživatele nechráněné heslem a může je využít kdokoliv, což je nebezpečné. Daleko nebezpečnější je heslem nechráněné sezení uživatele **root**, protože instalace systému je tak veřejně dostupná komukoliv pro jakékoliv změny a zásahy (viz následující čl. 5.3).

Uživatel může změnit své sezení na sezení jiného uživatele příkazem **login**. Jde o vyvolání programu, který řídí proces **login** v průběhu přihlašování uživatele. Jak jsme uvedli v čl. 2.2, je to proces, který porovnává heslo z klávesnice s položkou šifrovaného hesla v souboru **/etc/passwd**. Můžeme si



Obr. 5.2 Sezení a změna uživatele

všimnout, že v případě zadání nesprávného hesla nabízí v průběhu přihlašování proces **login** znovu možnost vstupu do systému výpisem

**login:**

podobně jako to provedl **getty** při svém startu. Teprve po určitém počtu neúspěchů nebo po dlouhé nečinnosti uživatele proces **login** skončí a **init** znova vytváří **getty**. Příkaz **login**, pokud v argumentu obsahuje jméno uživatele, znamená výpis

**password:**

což je žádost o heslo stejně jako v době přihlašování, a pokud zadání hesla selže, objeví se text **login:**, protože proces **login** nabízí novou možnost změny uživatele. Proces **login** pracuje tak, že nahradí proces shellu, ze kterého je použit. Shell při interpretaci příkazu **login** nepoužije volání jádra **fork** pro vytvoření nového procesu, ale pouze **exec**, a tím promění sám sebe na **login** (totéž jako při použití **\$ exec login**). Při práci s výchozím shellem dojde tedy ke skutečnému přihlášení na jiného uživatele. Může se ale stát, že uživatel pracuje interaktivně s dětským shellem výchozího shellu (viz odst. 2.5.3). Tehdy dochází k proměně pouze dětského shellu, čehož si uživatel nemusí být vědom. Některé implementace proto tuto možnost odmítají a **login** přihlášení pro jiný než výchozí shell neprovede. Situaci ilustruje obr. 5.2.

**login** znamená přihlášení, a proto je prostředí uživatele nově nastaveno. Je přiděleno odpovídající UID, GID, domovský adresář atd. Na rozdíl od **login**, proces **su** znamená propůjčení práv bez změny sezení (viz obr. 5.2). Příkaz **su** může mít argument, ve kterém stanovujeme jméno uživatele, pod jehož právy vzniklý proces pracuje. Není-li uživatel uveden, uvažuje se uživatel **root**. **su** vzniká jako dětský proces interaktivního shellu, a teprve v případě úspěchu (shody odpovídajícího hesla) se promění na shell podle položky odpovídajícího uživatele v **/etc/passwd**. Odpovídající UID, GID a prostředí nového uživatele je také nastaveno, ale předchozí sezení včetně všech nadřazených procesů zůstává zachováno a ukončením práce nového shellu (příkazem **exit**) v předchozím sezení pokračujeme (viz obr. 5.2). Pokud jej implementace obsahuje, pracuje obdobně i **newgrp**, který vzniká jako dětský proces, mění nastavení GID a jako poslední akci provádí volání jádra **exec** a promění se tak na proces shellu podle **/etc/passwd** (viz obr. 5.2).

POSIX ani SVID příkaz **login** neobsahují, naopak málo používaný **newgrp** oba dokumenty zahrnují. **su** je neodmyslitelnou součástí UNIXu, POSIX jej ale zatím neobsahuje.

Na základě znalosti identifikace registrovaných uživatelů můžeme využívat komunikační prostředky. Pokud je uživatel přihlášen, můžeme s ním promluvit, pokud přihlášen není, můžeme mu poslat poštu. Pod vágním výrazem promluvit nebo rozhovor rozumíme využití např. programu **write**, který navazuje spojení s jiným přihlášeným uživatelem a zobrazuje námi zapisovaný text na obrazovce jeho terminálu. Oslovený uživatel může náš vstup na jeho obrazovku oplatit stejným příkazem. Pak jím zapisovaný text se naopak objevuje na naší obrazovce. Přenos textu při použití **write** je po řádcích. Uživatelé proto musí být vzájemně ohleduplní, protože každý dokáže psát na klávesnici různě rychle. Aby byl rozhovor přehlednější a přenášený ihned po znacích, byl naprogramován a je používán **talk**, (není v SVID, ale v POSIXu ano) který je obrazovkově orientovaný. Každý uživatel má přitom určenou plochu pro svůj text. Pokud příkazy neznáte, vyzkoušejte je! Parametrem příkazů je **name** nebo **line**. Pokud jste nedůtklivý uživatel, můžete použít příkaz **mesg** s parametrem **n** (**no**) a každý, kdo se pokusí s vámi navázat rozhovor, bude odmítnut (parametr **y** možnost rozhovoru opět nastaví). Příkazy pracují

za podpory možnosti zápisu do speciálního souboru terminálu. Po přihlášení je speciální soubor použitého terminálu ve vlastnictví přihlášeného uživatele a přístupová práva k tomuto souboru jsou nastavena obvykle s právem zápisu kohokoliv, např.

```
$ ls -l /dev/tty1
crw--w--w- 1 petrnt tty 4, 1 Nov 14 10:57 /dev/tty1
```

Přístupová práva se pro skupinu a ostatní změnil

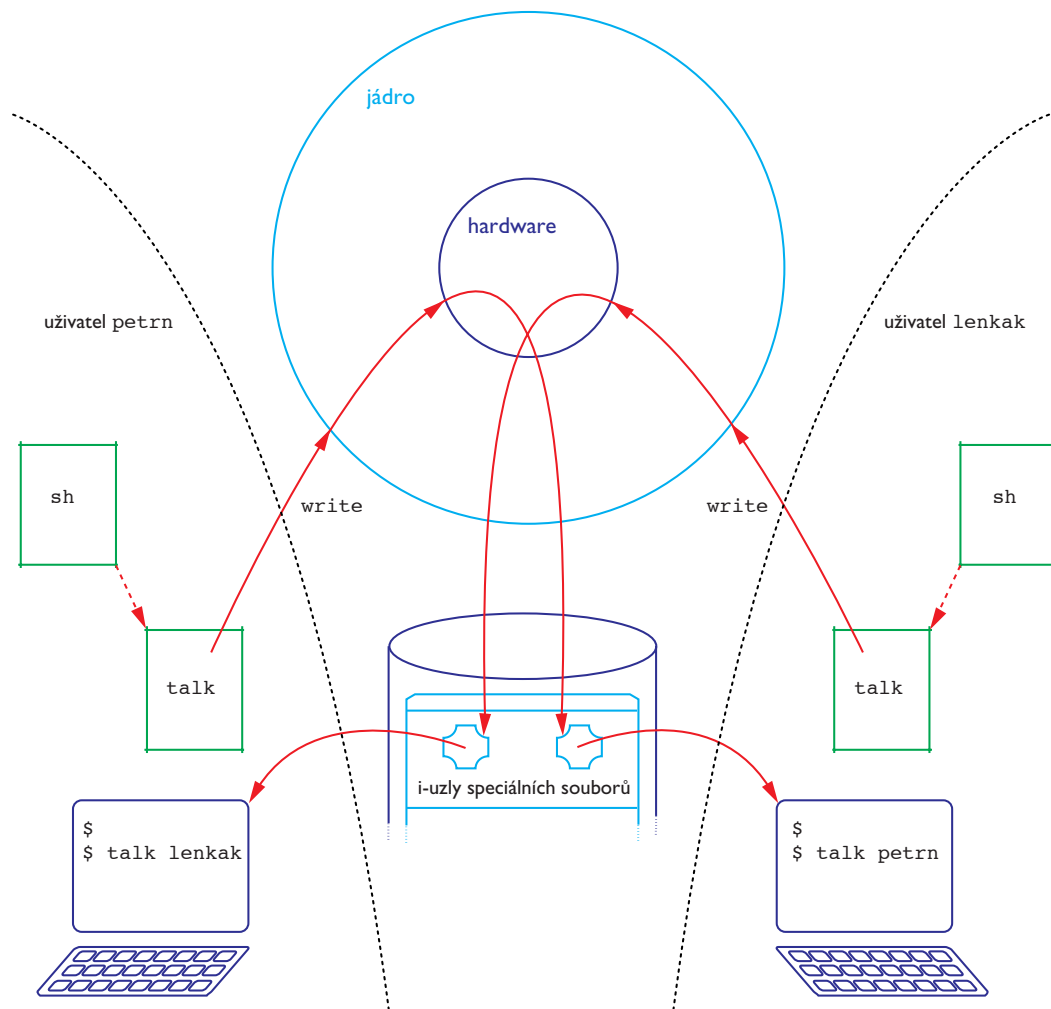
```
$ mesg n
$ ls -l /dev/tty1
crw----- 1 petrnt tty 4, 1 Nov 14 10:57 /dev/tty1
```

Uvedený příklad je také na obr. 5.3.

Uživatel s oprávněním EUID=0 (zejména **root**) může používat program **wall**, kterým zasílá zprávu všem přihlášeným uživatelům, a to bez ohledu nastavení **mesg**. Program je používán při havarijních stavech a běžně při zastavování systému pro varování přihlášených uživatelů.

Přihlášenému nebo nepřihlášenému uživateli lze poslat poštu. *Elektronická pošta* (electronic mail, email) je fenomén, který dosáhl svého největšího rozšíření teprve s příchodem rozsáhlých počítačových sítí (viz opět kap. 7), ale UNIX jej obsahoval od prvních verzí, protože koncepce registrace uživatelů a i jejich současné práce na tomtéž počítači ke vzájemnému zanechávání zpráv na smluveném místě je přirozená. Každý uživatel má v UNIXu přiřazenu tzv. poštovní schránku (mailbox), která je souborem v adresáři **/usr/mail** (nebo lépe **/var/mail**) jména shodného se jménem uživatele (jméno souboru poštovní schránky má shell nastaveno v proměnné **MAIL**). Schránka je ve vlastnictví uživatele, ostatní uživatelé k ní mají přístup pouze prostřednictvím poštovních programů, které zajišťují posílání pošty. Systém doručování a čtení pošty je rozdělen na tři části: uživatelskou (user agent), systémovou pro přenos pošty (transport agent) a systémovou pro převzetí pošty (delivery agent). Uživatelská část znamená poštovní program, pomocí kterého uživatel poštu čte nebo odesílá. Uživatelských poštovních programů je dnes celá řada. V UNIXu je původní program pro čtení i posílání pošty **mail** (jeho rozšířená verze v SVID a jediná v POSIXu má název **mailx**), jehož použití je prostinké, ale i možnosti jsou chudí. Při identifikaci adresáta uživatel uvádí jeho jmennou identifikaci (v **mail** na příkazovém řádku). Systémová část se stará o požadavek přenesení pošty do poštovní schránky a dnes je chápána především jako síťová aplikace, jejíž výsledkem je také doručování pošty v rámci téhož uzlu. Nejpoužívanější systém předávání pošty je prostřednictvím síťové aplikace **sendmail**, jejíž princip si uvedeme v kap. 7, kde se také zmíníme o dalších možnostech správy přijímání a odesílání elektronické pošty.

**News** je informační služba podobná aplikaci elektronické pošty, ale slouží pro uživatele jednostranně, tj. pouze pro čtení. Podobně jako elektronická pošta se uplatnila především při rozvoji sítí. Jde o distribuci zpráv současně na různá místa ve veřejných sítích a jejich zpřístupnění prakticky každému uživateli sítě. Síťová podoba **News** má název **Usenet** a její princip opět probereme v rámci popisu práce síťových aplikací v kap. 7. SVID uvádí v tomto kontextu příkaz **news**, pomocí kterého uživatel může číst novinky zveřejňované v rámci místního uzlu. Ve zjednodušeném podání jde o čtení souborů adresáře pro veřejné zprávy **/var/news** (dříve **/usr/news**). Zda jsou to zprávy pro uživatele nové nebo ne, je identifikováno v souboru **.news\_time** domovského adresáře, ve kterém je uloženo datum a čas posledního použití **news**.



Obr. 5.3 Rozhovor dvou uživatelů

Zajímavou a frekventovanou možností uživatele je spouštět procesy v jeho vlastnictví, aniž je uživatel přihlášen. Jde o tzv. opožděné spuštění programu v zadaném čase. Uživatel k tomu používá příkaz **at** nebo **batch**. Systém přijímá takové požadavky a ke spuštění požadovaného procesu (který může být pochopitelně rodičem dalších procesů) používá démon **cron**. Uživatel dále může pro požadavek cyklického opakování akcí používat příkaz **crontab**.

V argumentu příkazu **at** zadává uživatel datum a čas, kdy má být daný proces, skupina procesů nebo sekvence procesů provedena. Formát data a času je poměrně široký a je dobré seznámit se s ním v uživatelské dokumentaci. Ze standardního vstupu (nebo ze souboru použitím volby **-f**) pak **at** čte seznam příkazů, které má v daném čase provést. Např.

```
$ at 0912am tomorrow+1
mail petr@lenkak <ukoly.pozitri
^d
```

je požadavek na zaslání pošty uživateli **petr@** a **lenkak** pozítří dopoledne v 9 hodin 12 minut. Příkaz ukládá požadavek do adresáře `/var/spool/cron/atjobs` a jeho zařazení do `/etc/cron.d/queuedefs`. Současně také posílá smluvený signál démonu **cron**, který se po jeho přijetí odblokuje a prohlédne nový požadavek. **cron** je nastaven na probuzení podle naposledy zadaného požadavku. Pokud má nově registrovaný požadavek čas, který předchází, přestaví se **cron** nyní na něj. Démon startuje odpovídající proces (nebo skupinu procesů) v danou dobu tak, že nastavuje identifikaci procesu podle uživatele, který požadavek zadal. Pro správce systému jsou dále důležité soubory `/etc/cron.d/at.allow` a `/etc/cron.d/at.deny`, které obsahují seznamy jmen uživatelů (na každém řádku jedno jméno), kterým je používání příkazu **at** dovoleno (`at.allow`) nebo zakázáno (`at.deny`). **at** hledává nejprve seznam v `at.allow`. Pokud tento soubor není přítomen, hledá `at.deny`. Povolení všem uživatelům umožní pouze existence souboru `at.deny`, který je prázdný.

Příkaz **batch** používá uživatel bez jakýchkoliv parametrů. Ze standardního vstupu zadává seznam příkazů, které se mají provést ihned po zadání. Seznam příkazů je však připojen k frontě všech takto zadaných požadavků a příkazy budou provedeny, až na ně přijde řada, protože fronta je prováděna sekvenčně vždy s čekáním na dokončení posledního zadaného příkazu.

V dokumentaci se mluví o frontách požadavků na provedení v určitém čase i v kontextu s příkazem **at**. Obecně se definují fronty s označením **a** (pro příkaz **at**), **b** (pro příkaz **batch**) a **c** (pro příkaz **crontab**), které jsou vedeny v adresáři `/etc/cron.d/queuedefs`. Uživatel může pomocí příkazu **at** s použitím volby **-l** požadavky své fronty vypisovat, **-r** požadavek z fronty vyjmout (u privilegovaného uživatele se takto pracuje s požadavky všech uživatelů dané fronty).

Démon **cron** pracuje pro požadavky podle uvedených front. Fronta **c** pak přináší požadavkům podle příkazu **crontab**. Jde o trvalé zavedení požadavku na provedení procesu v identifikaci uživatele vždy po uplynutí určitého časového intervalu. Časový interval je určen na standardním vstupu včetně příkazu, který se má provádět. Každý vstupní řádek je zavedení nového požadavku. Požadavek má formát o šesti položkách oddělených mezerou nebo tabulátorem, přitom po řadě znamenají minutu (v rozmezí 0-59), hodinu (0-23), den v měsíci (1-31), měsíc v roce (1-12) a den v týdnu (0-6, 0 je neděle). Každá položka může být výčtem hodnot oddělených čárkou nebo může být položka obsazena znakem **\***, který znamená všechny hodnoty, např.

```
$ crontab
0,20,40 * * * 0 /usr/local/bin/testy
^d
```

je připojení dalšího požadavku k tabulce zadávajícího uživatele tak, že odpovídající program **testy** bude aktivován každých dvacet minut každou neděli. Každý uživatel má ve vlastnictví tabulku udržo-



vanou pomocí příkazu **cron** (je uložena v adresáři `/var/spool/cron/crontabs`), kterou může prohlížet volbou **-l**, rušit požadavek pomocí **-r** a editovat ji pomocí volby **-e**. Privilegovaný uživatel může použít v parametru jméno uživatele a pracovat tak s tabulkou libovolného uživatele. Přestože jsou tabulky textové a dobře čitelné, nedoporučuje se je editovat jinak než příkazem **cron**, který provede odpovídající zápis do fronty **c** a komunikuje s démonem **cron**.

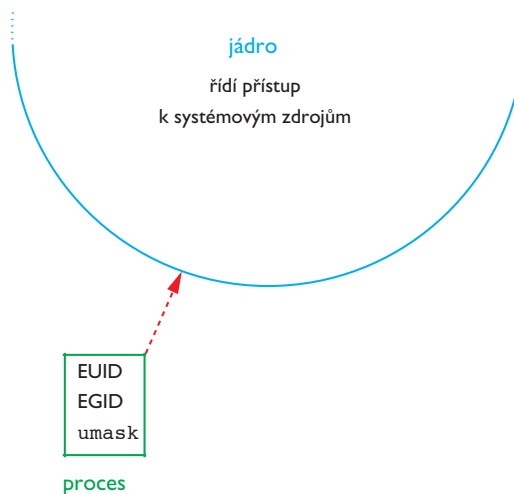
### 5.3 Přístupová práva

Aniž by v době, kdy UNIX vznikl, byly požadavky na bezpečnost operačního systému definovány státními dokumenty, jako je tomu dnes (viz kap. 9), přístup k výpočetním zdrojům byl navržen tak, aby obyčejný uživatel nemohl provádět zásahy do chodu výpočetního systému, ale pouze jej používal. Rovněž tak musí mít zajištěnu ochranu dat před jejich zničením.

V UNIXu jsou uživatelé registrováni jmennou a číselnou identifikací. Uživatel se jménem `root` má číselnou identifikaci 0 a je jediný privilegovaný s neomezenými právy (v jeho moci je zásah do všech systémových zdrojů tak silný, že systém může být prakticky okamžitě zničen). Ostatní uživatelé jsou neprivilegovaní, obecně s možnostmi pouze systém používat. Uživatelé jsou sdružováni do skupin. Jsou-li uživatelé účastníky téže skupiny, jsou si bližší z pohledu přístupu k obsahu souborů a ostatním výpočetním zdrojům.

Jakékoliv nestandardní využívání systémových zdrojů obyčejnému uživateli povoluje `root` prostřednictvím

nastavení `s`-bitu, jehož platnost je vztažena buď ke skupině, nebo ke všem uživatelům, zavedením uživatele do skupiny uživatelů, ve které je `root`, změnou přístupových práv souborů.



Obr. 5.4 Přístupová práva

Přístup k systémovým zdrojům je obecně vždy dán přístupovými právy pro čtení (`read`), zápis (`write`) a provádění (`execute`), jejichž interpretace je závislá na konkrétním systémovém zdroji. Přístupová práva jsou dále vztažena k vlastníkovi systémového zdroje (`UID`), vlastnické skupině systémového zdroje (`GID`) a ostatním uživatelům. Tyto atributy systémového zdroje jsou jeho součástí a obecně jsou stanoveny metody (volání jádra), jak získat jejich hodnoty. Viz obr. 5.4.

Typické a nepoužívanější je to u systémového zdroje jako souboru. Atributy přístupu a vlastnictví jsou uloženy v i-uzlu a jsou zjistitelné voláním jádra `stat`. Soubor vzniká na základě požadavku některého z procesů, který má nastaveno `EUID` a `EGID` a podle nich je souboru při jeho vzniku přiřazeno vlastnictví. Slovo přístupových práv je naplněno v i-uzlu podle masky vytváření souborů, kterou proces nastavuje pomocí volání jádra `umask`, pokud mu nevyhovuje implicitní přiřazení. U souborů je typické nastavení umožňující všem soubor číst a vlastníkovi přepisovat. To je úzus UNIXu, který od jeho vzniku stanovuje politiku zveřejňování všech informací bez možnosti zásahu bez výhradního svolení supervizora operačního systému. I odsud se odvodilo označení otevřenost operačního systému.

Na přístupová práva k systémovým zdrojům také nejčastěji správce systému zaměřuje svoji pozornost, pokud určitá systémová aplikace nepracuje správně pro všechny uživatele. Obvykle totiž jde o nedostatečné zpřístupnění systémového zdroje. Na straně druhé musí být zamezeno neoprávněné manipulaci obyčejného uživatele. Obvyklý způsob je práce zprostředkovatele, který přebírá požadavky od obyčejných uživatelů, zkoumá je a aplikuje je nad systémovým zdrojem, ke kterému má výlučný přístup. Zprostředkovatelem bývá proces `démon` a z tohoto důvodu jich při běžném chodu systému bývá poměrně dost.

Proměnu uživatele na jiného (`login`), změnu skupiny (`newgrp`) nebo dokonce práci shellu v privilegovaném režimu (`su`) jsme si ukázali v předchozím čl. 5.2. Jak zde bylo řečeno, i přes ochranu sezení každého uživatele heslem (`password`) může privilegovaný uživatel číst a měnit data každého dalšího uživatele systému, přestože mu nejsou k dispozici nástroje dešifrování hesla. Má totiž neomezený přístup ke všem systémovým zdrojům. Aby i tak mohl uživatel ochránit svoje data před zneužitím, s výjimkou změny přístupových práv jeho souborů (příkazem `chmod` a voláním jádra `chmod`), může obsah souboru šifrovat (encoding). Používá k tomu příkaz `crypt`, který pracuje jako filtr. Při šifrování čte ze standardního vstupu text, který na standardní výstup vypisuje šifrovaný. Pokud nebylo součástí příkazového řádku šifrovací heslo (podobné jako heslo sezení), vyžaduje jeho zápis na klávesnici, což je bezpečnější způsob, protože příkazové řádky jsou čitelné veřejně např. příkazem `ps`. Pokud přesměrujeme zašifrovaný obsah souboru na vstup příkazu `crypt`, rozšifrujeme jej a opět může být heslo součástí příkazového řádku nebo bude požadováno (bez opisu) z klávesnice. Každý manuál v UNIXu uvádí příklad

```
$ crypt heslo < čitelný > šifrovaný
```

pro šifrování a

```
$ crypt heslo < šifrovaný | lp
```

pro tisk původního obsahu šifrovaného souboru na tiskárnu. Bez znalosti hesla je možnost dešifrovat obsah souboru malá, přestože záleží na šifrovacím mechanismu a schopný matematik si poradí. Většina současných implementací bez zavedení zvláštní úrovně ochrany systému (viz kap. 9) používá šifrovací algoritmus zavedený v počátcích UNIXu, který využívá principu tzv. Enigma encryption machine<sup>1</sup>. Je to způsob šifrování pomocí záměny znaků průchodem přes jeden nebo více záměnných kotoučů, na jejichž

obvodu jsou rozmístěny znaky, jejichž vzájemná korespondence a korespondence jednotlivých kotoučů pak vytváří výsledek šifrování. I přes svou jednoduchost jde o algoritmus dostatečně účinný a vláda Spojených států dokonce zakázala jeho vývoz jako součástí komerčního UNIXu. V současně dodávaných systémech je však obvykle přítomen, přestože jeho definici (včetně příkazu **crypt**) SVID ani POSIX neobsahuje. SVID pouze definuje funkce **crypt**, **setkey** a **encrypt**, které lze pro šifrování používat, a přestože formát argumentů těchto funkcí vzhledem k jejich rozměru je přesně dán a omezuje tak používaný šifrovací mechanismus, způsob šifrování není striktně předepsán. Šifrování obsahu právě editovaného souboru může uživatel při použití standardních editorů **ed**, **ex** a **vi** zapínat vnitřním příkazem těchto editorů **X** (je následně požádán o heslo), kdy je využíván tentýž algoritmus, a později lze rozšifrovat editovaný text také použitím **crypt**.

## 5.4 Účtování

Uživatel **root** má oprávnění ovlivňovat i jiným uživatelům způsob jejich sezení. Vedle jeho práva je registrovat, měnit jejich registraci nebo ji i zrušit, může v okamžiku jejich přihlašování vynutit provedení akcí, které samotný uživatel nemůže odmítnout.

Obsah textového souboru `/etc/motd` je např. vypisován na obrazovku terminálu po přihlášení, a tak může správce sdělovat všem uživatelům aktuální pokyny pro jejich práci nebo jiná upozornění. O obsah souboru se zajímá v době přihlašování proces **login** a před svou proměnou na proces shellu jej opisuje na obrazovku terminálu.

Privilegovaný uživatel může také obohatit text, který se objevuje na obrazovce aktivního terminálu před přihlášením uživatele. Obecně je na obrazovku vypisován text končící na **login:**, kterému předchází označení operačního systému (jako uzlu v síti), případně označení výrobce nebo uvítací text. Textový řetězec **login:** je uveden jako součást nastavení terminálu v `/etc/gettydefs`, podle kterého pracuje přihlašovací proces **getty** a odkud přebírá nastavení charakteristik terminálu (soubor bude popsán v kap 6). Může být vyměněn za text jiný, přestože je to krajně nezvyklé, protože **login:** je jeden z typických průvodních znaků UNIXu. V `/etc/gettydefs` tomuto textu může předcházet další text, ale obecně se využívá obsah souboru `/etc/issue`, který **getty** vypisuje na terminál po nastavení charakteristik terminálu, ale před textem v `/etc/gettydefs`.

Jak jsme uvedli v čl. 2.5.3, každý uživatel může mít ve svém domovském adresáři uveden soubor **.profile**, který shell výchozího sezení interpretuje jako scénář před vlastním sezením přihlašujícího se uživatele. Výchozí atributy svého sezení tak může uživatel ovlivňovat změnami jeho obsahu. Správce systému jako uživatel **root** může navíc měnit obsah scénáře se jménem `/etc/profile`, který interpretuje výchozí shell každého sezení v okamžiku přihlašování před místním souborem **.profile**. Často v něm správce systému definuje, nastavuje a exportuje proměnné shellu, které souvisí s prací všech uživatelů. Uživatelé si je nemusí pak definovat místně, přestože akce scénáře `/etc/profile` mohou následně eliminovat, pokud jim nevyhoví. Zabránit interpretaci scénáře `/etc/profile` však nemohou. Na rozdíl od procesu **getty** již jde o nastavení práce procesu shellu a jeho potomků. Typ terminálu, který je zde běžně interaktivně nastavován se souhlasem uživatele, souvisí již jen s obrazovkově orientovanými operacemi, případně s definicí správného rozložení kláves (viz opět kap. 6).

Vstup uživatele do systému, frekvenci těchto vstupů a následnou práci jeho procesů lze sledovat.

Informace o právě přihlášených uživateli jsou evidovány v souboru `/var/adm/utmp` (dříve v `/etc/utmp`). Tuto tabulku všech sezení rozšiřuje vždy proces **login** v případě úspěšného přihlášení a takto zavedený záznam z ní odstraní proces **init** v okamžiku odhlášení.

Informace o každém přihlášení (i neúspěšném) a odhlášení jsou ukládány vždy jako další záznam uvedenými procesy do souboru `/var/adm/wtmp`. Vzhledem k tomu, že soubor se stále rozrůstá, správce by měl jeho velikost pravidelně kontrolovat.

Konečně pro každého uživatele je evidován také záznam jeho posledního přihlášení (datum a čas) ve `/var/adm/lastlog`.

Formát souborů `utmp`, `wtmp` i `lastlog` je implementačně závislý, ale SVID stanovuje funkce pro práci s jejich záznamy, které pak procesy využívají. Odpovídající definice nalezne čtenář v `<utmp.h>`. Přenositelné funkce jsou vedeny v provozní dokumentaci pod heslem `getut`, kde jsou uvedeny funkce `getutent`, `getutid`, `getutline`, `setutent`, `endutent` a `utmpname` pro čtení a manipulaci se záznamy uvedených souborů a také funkce `pututline`, která do souboru přidává další záznam.

Funkce pracují podle SVID se strukturou záznamu

```
struct utmp
{
    char ut_user[];    /* jméno uživatele */
    char ut_id[];      /* identifikace podle /etc/inittab */
    char ut_line;      /* jméno speciálního souboru */
    pid_t ut_pid;      /* PID */
    short ut_type;     /* typ záznamu */
};
```

Praktické implementace však běžně používají ve struktuře další položku

```
time_t ut_time ; /* datum a čas vytvoření záznamu */
```

Důležitá je položka typu záznamu `ut_type`, která rozlišuje jeho význam a která má definovány možné hodnoty `EMPTY`, `RUN_LVL`, `BOOT_TIME`, `OLD_TIME`, `NEW_TIME`, `INIT_PROCESS`, `LOGIN_PROCESS`, `USER_PROCESS`, `DEAD_PROCESS` a `ACCOUNTING`. Interpretace záznamu pak pokrývá široké možnosti podle typu sledované události. Se strukturou a uvedenými funkcemi pracují procesy, které soubory evidence o aktivitách uživatelů doplňují nebo čtou, např. již uvedené **login**, **init**, **who**, ale třeba i **last**, který vypisuje informace o naposledy provedeném přihlášení uživatele, nebo také **fwtmp**, který obsah `wtmp` opisuje v textové podobě na obrazovku. Kontrolu `wtmp` může správce systému provádět pomocí **wtmpfix**, atd., viz provozní dokumentace.

Správce systému může dále rozšiřovat sledování aktivit uživatelů při zapínání systému *účtování* (accounting). Systém účtování v UNIXu je evidence informací o spotřebovaných výpočetních zdrojích jednotlivých procesů. SVID definuje privilegované volání jádra

```
#include <sys/types.h>
int acct(const char *path);
```

pomocí kterého zapínáme účtovací mechanismus jádra. Od okamžiku úspěšného provedení `acct` jádro pro každý ukončený proces (voláním jádra `exit` nebo signálem) připojuje k souboru se jménem `path` informace podle následující struktury (její definice je v `<sys/acct.h>`)

```

struct acct
{
    char  ac_flag;      /* účtovací příznak */
    char  ac_stat;      /* návratový status procesu */
    uid_t ac_uid;       /* UID */
    gid_t ac_gid;       /* GID */
    dev_t ac_tty;       /* řídicí terminál */
    time_t ac_btime;    /* čas startu procesu */
    comp_t ac_utime;     /* spotřebovaný čas v uživatelském režimu */
    comp_t ac_stime;     /* spotřebovaný čas jádra */
    comp_t ac_etime;     /* celkový čas existence procesu */
    comp_t ac_mem;       /* použití operační paměti */
    comp_t ac_io;        /* počet přenesených znaků vstupu/výstupu */
    comp_t ac_rw;        /* počet zapsaných či přečtených diskových
                        bloků */
    char  ac_comm[8];   /* jméno příkazu */
}

```

Volání jádra používá standardní příkaz **accton**, který jako `path` používá existující soubor `/var/adm/pacct`, pokud není v (jediném) argumentu příkazu uvedeno jméno jiné. Příkazem zapínáme účtování, které je po instalaci obvykle vypnuto (viz kap. 10). Příkazů pro práci s účtovacími informacemi je více. Jsou uloženy v adresáři `/usr/lib/acct` (někdy v `/usr/sbin/acct`) a mají funkci transformační, tj. z informací nasbíraných jádrem vytvářet data snadněji čitelná.

Vedle souboru `/var/adm/pacct` můžeme považovat za podklad pro účtování také obsah souboru `/var/adm/wtmp`, který je plněn procesy **login** a **init**, a dále soubory `/var/adm/acct/nite/diskacct`, který je produktem programu **ddisk** (je startován obvykle pomocí **cron** a shromažďuje informace používání disků), a `/var/adm/fee` jako výsledek činnosti programu **chargefee**, kterým stanovujeme účtovací poplatky jednotlivým uživatelům.

Podle informací v uvedených souborech vytvoří startovaný scénář **runacct** periodicky jednou denně (opět obvykle pomocí **cron**), podklady pro shromáždění účtovacích informací a přípravu dat pro tisk účtování daného dne. Příkaz lze přitom parametrizovat dnem v měsíci, a proto nemusí být nutně startován denně. Rovněž tak dalším parametrem stanovujeme typ generovaných dat nebo typ akce scénáře (není-li argument uveden, je čten ze souboru `/var/adm/acct/nite/statefile`). Typickým dítětem scénáře **runacct** je např. program **lastlogin**, který doplní informace souboru `/var/adm/acct/sum/loginlog`, kde jsou uvedeny informace naposledy provedeného přihlášení uživatele.

S daty vytvořenými scénářem **runacct** pak pracují programy **acctmerg**, který shromáždí informace v adresáři `/var/adm/acct/sum` (soubory `taacct` a `cms`), a **prdaily**, který připraví tiskový výstup účtování (tamtéž). Konečně výsledky **acctmerg** do textové podoby zpráv převádí program **monacct** spouštěný, jak už jeho jméno naznačuje, pro výsledky účtování jednou měsíčně, a který vytváří soubory v adresáři `/var/adm/acct/fiscal`.

Správce systému má k dispozici ještě doplňující příkazy, jako je **ckpacct** pro prošetření správnosti informací nasbíraných jádrem, **startup** a **shutup** pro spuštění a zastavení celého systému účtování atd.

Zvládnout celý systém účtování od jeho jednoduchého zapnutí až po tiskové výstupy a zprávy pro platící uživatele není snadný úkol. Vyžaduje trpělivost a pročtení provozní dokumentace (případně čtení souboru se scénářem **runacct**), ale výsledek stojí za to.

Některé implementace UNIXu dnes používají také jednodušší systém účtování podle principů navržených v systémech BSD. Přestože tento způsob účtování není součástí doporučujících dokumentů, uveďme si stručně jeho práci. Základní účtovací data poskytovaná jádrem jsou uložena v souboru `/var/adm/acct` a příkazem **sa** jsou převáděna a spravována v souborech **savacct** a **usracct** adresáře `/var/adm`. Účtování zapíná a vypíná správce systému příkazem **accton**. Vedle základního **sa** je pro získání textových informací účtování k dispozici ještě příkaz **ac**, který vypisuje celkový čas spotřebovaný přihlášenými uživateli, a tiskový **pac**, který účtovací data připravuje jako podklady pro tisk.

Správce systému (jako přihlášený uživatel **root**) má ještě další možnosti sledování uživatelů v jejich využívání systémových zdrojů. Pomocí příkazů **sa1**, **sa2** a **sar** v kombinaci s již popsaným démonem **cron** může sledovat vytížení systémových zdrojů sběrnou metodou v zadaných časových intervalech a také může pomocí dále uvedených příkazů sledovat obsazení disků, a to také na úrovni jednotlivých uživatelů. Příkazy **sa1** ... souvisí s celkovým sledováním obsazení tabulek při provozu jádra a budeme se jim prakticky věnovat především v čl. 10.3, v části o přizpůsobení jádra provozním požadavkům.

Obsazení systému souborů po jednotlivých adresářích pro zadaný podstrom vyhodnocuje příkaz **du**. Jeho výstupem je seznam všech podadresářů s uvedením počtu bloků, který podstrom tohoto adresáře zabírá. Pokud použijeme volbu **-s**, uvádí výpis pouze celkový součet celého podstromu. Naopak při **-a** program vypisuje velikost v blocích o každém souboru. Správce také ocení volbu **-r**, protože při jejím použití jsou nedostupné adresáře a jiné kolize při čtení komentovány. Implicitně totiž **du** v těchto případech mlčí.

Příkaz **diskusg** (bývá součástí účtování v rámci příkazu **do disk**) prochází zadaný diskový svazek a na standardní výstup vypisuje informace, kdy na každém řádku je uvedena číselná i jmenná identifikace uživatele podle `/etc/passwd` a celkový počet využitých diskových bloků jeho souborů. Pomocí takto získané tabulky má správce přehled o rozdělení využitého diskového prostoru mezi uživatele. Nápomocný je mu i následně použitelný příkaz **acctdisk**, který prohlíží záznamy využívání diskového prostoru a vytváří celkové statistiky.

Příkaz, kterým rovněž získáme využití disků uživateli, je **quot**, jehož povinný argument je označení svazku (speciálním souborem nebo spojovacím adresářem). Na výstupu získáváme na každém řádku počet využitých diskových bloků a odpovídající jméno uživatele. Příkaz je součástí systému kontroly a omezování diskové kapacity poskytované uživatelům. Příkazy, které může správce využívat, jsou **edquota** pro editaci tabulek omezení jednotlivých uživatelů, **quotaon** a **quotaoff** pro zapínání a vypínání sledování využitého diskového prostoru. Kontrolu tabulek při porovnání s aktuálním stavem disků lze provádět pomocí **quotacheck** a konečně **quotarep** je příkaz pro výpis zprávy o stavu dosahovaných limitů uživatelů na daném svazku. Spustit celý systém kontroly disků není bez zvládnutí

provozní dokumentace možný, protože SVID ani POSIX jej neobsahují a systém tak není zcela jednotný ve všech implementacích, přestože je prakticky v každém (s výjimkou LINUXu) obsažen.

Limity registrovaných uživatelů je téma, které dříve nebo později bude zajímat správce systému, a při jejich nasazení pak i následně každého omezeného uživatele. Jde však o běžnou a nutnou praxi chodu víceuživatelského systému, kde správce ručí za správný provoz všem uživatelům, a to demokraticky při respektování nutných průvodních nepříjemností. Správce může omezovat uživatele nejen v diskovém prostoru, ale např. i v počtu spustitelných procesů a velikosti využitelné operační paměti, přestože to dynamicky umožňuje jenom několik implementací (např. AIX). Pevná nastavení systémových kvót vychází z kapacity tabulek struktur jádra pro evidenci současně otevřených souborů připojitelných svazků, současně běžících procesů (celkově nebo pro jednoho uživatele) atd. a souvisí se sledováním jejich využitelnosti (příkazy **sa1** ...). Případné změny je pak dosaženo výměnou jádra (tzv. regenerací, tj. sestavením jádra podle nových podkladů). Problematiku provozu jádra v kontextu požadavků využívání výpočetního systému budeme zkoumat opět v kap. 10, kde si také uvedeme příklady možností dynamické změny limitů pro uživatele, tj. bez výměny jádra. SVID i POSIX definují limity systému (tj. jádra) v souboru `<limits.h>`, ve kterém se může libovolný uživatel s kapacitou struktur jádra seznámit i v konkrétní instalaci.

<sup>1</sup> Jde o německý šifrovací mechanismus Arthura Scherbiuse, který jej vynalezl v tomto století, a používal se zejména v průběhu druhé světové války.



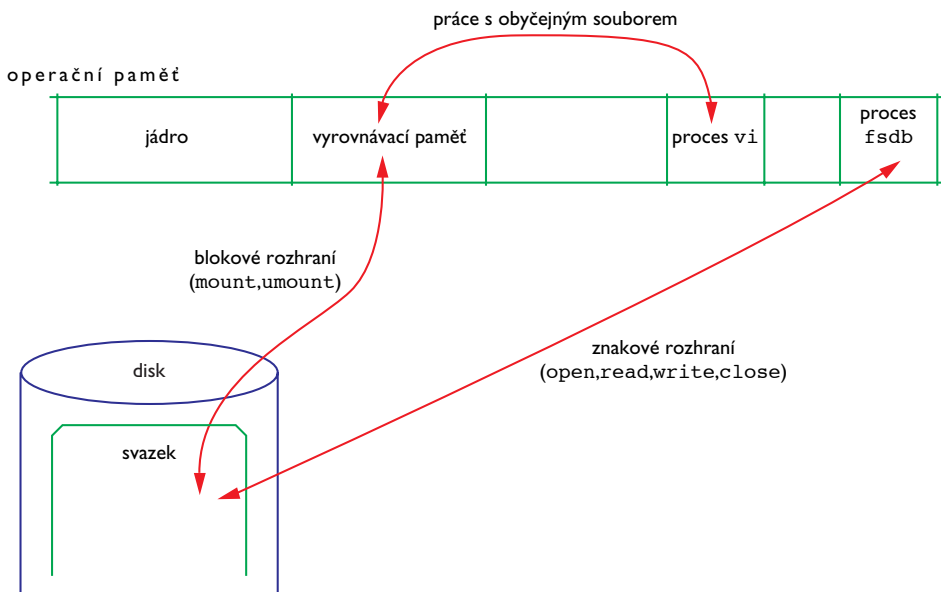


## 6 PERIFERIE

*Periferie* (device) výpočetního systému je zařízení, které centrální jednotce (CPU, Central Processor Unit, tj. procesoru) předává vstupní data ke zpracování nebo naopak od ní přijímá výstupní data pro jejich zobrazení nebo delší uchování. Dnes typickým představitelem periférií je terminál, disk, síťové rozhraní, tiskárna. Periferie jsou používány pro komunikaci uživatelů s výpočetním systémem.

Odkaz na periferie je v operačním systému UNIX realizován pomocí speciálních souborů, jejichž popis jsme z pohledu systému souborů uvedli již v odst. 3.3. Podle obr. 3.15 je také zřejmé, jakým způsobem je speciální soubor spojen s ovladačem v jádru. Principem obsluhy v/v (vstupně/výstupních, i/o input/output) zařízení jádrem a možnostmi, které má správce systému k dispozici, se budeme věnovat v této kapitole.

Při práci s periferií, a tedy se speciálním souborem, používá proces volání jádra jako při práci s obyčejným souborem. Jde trochu o paradox, protože disk je také periferie a přístup k ní znamená také aktivaci rutin ovladače. Tehdy však proces neotevřít speciální soubor odpovídajícího svazku disku, protože jádro systém souborů procesům zpřístupňuje na základě připojení svazků voláním jádra `mount`, které je privilegovaným procesem na jádru požadováno v průběhu startu operačního systému. Jde o komfort práce se soubory a adresáři, který je popsán v kap. 3. Přesto i disky musí mít své speciální



Obr. 6.1 Přístupové cesty k disku

soubory a pokud má proces potřebná přístupová práva, může s každým svazkem zacházet pomocí speciálního souboru přímo. Doporučuje se přitom mít svazek odpojen (volání jádra `umount` dříve připojený svazek odpojí). Proces v další práci musí znát strukturu uložení dat na disku, tj. způsob implementace svazku. Přestože myšlenka práce se sekcemi disků tímto způsobem se zdá netypická a nesnadná, musíme si uvědomit, že takto musí postupovat všechny systémové procesy, které svazky testují, případně v nich provádějí opravné zásahy (jedná se např. o **fsck** nebo **fsdb**). Správce systému také může používat sekci disku nikoliv za účelem připojitelného svazku, ale např. jako zařízení pro sekvenční úschovu dat, např. při archivaci pomocí programu **tar** nebo **cpio** na místě archivního zařízení (přenášet soubory na výměnných discích je dnes běžná praxe). Pak proces řízený takovým programem přistupuje k diskové periférii jako k sekvenčnímu znakovému souboru a nikoliv po blocích podle organizace dat odpovídajícího typu svazku. Svazek je organizace dat vytvořená v sekci disku programem **mkfs** a jádro tuto organizaci přijímá, svými vnitřními algoritmy akceptuje a usnadňuje tak procesům (potažmo uživatelům) snadnější práci. Jde tedy o vyšší vrstvu, která ale nemusí být při práci s periférií využita.<sup>1</sup> O paradoxu jsme mluvili v tomto kontextu, protože pro přístup k perifériím musíme mít nejprve viditelný adresář `/dev` výchozího svazku operačního systému, který je implementován pomocí organizace svazku.

Práce jádra se speciálním souborem periférie vychází z hlavního čísla i-uzlu speciálního souboru a aktivace tomuto číslu odpovídající funkce ovladače. V argumentu funkce pak jádro používá vedlejší číslo téhož i-uzlu. Podle toho, zda se jedná o blokový nebo znakový speciální soubor, jádro funkci vybírá z pole struktur **bdevsw** nebo **cdevsw** (jak je uvedeno na obr. 3.15). Hlavní číslo je přitom indexem do jednoho z nich. Odkazem tedy jádro získá seznam textových řetězců, které mají význam jmén funkcí ovladače periférie. Pořadí ve struktuře udává význam jednotlivých funkcí, které pak jádro používá podle požadované manipulace. Např. pro čtení z periférie je uvedena jedna funkce, pro zápis další atd. Pokud odpovídající manipulace s periférií přitom nedává smysl (na tiskárnu např. můžeme pouze zapisovat), na místě jména funkce je prázdný řetězec a jádro manipulaci s periférií při použití volání jádra procesem odmítne. Detail pole struktur např. **cdevsw** z obr. 3.15 pak ukazuje obr. 6.2.

Na obrázku jsou již také uvedena jména funkcí, které dohromady vytvářejí ovladač. Ovladač je tedy skupina funkcí, která je sestavena s jádrem. Připojení další jádru neznámé periférie tedy znamená nově sestavit jádro s rozšířením pole **cdevsw**, případně i **bdevsw**. Poskytnutím odpovídajících funkcí při sestavování jádra (generací jádra) se budeme zabývat v kap. 10. Každá funkce ovladače může být běžně použita z různých procesů pro různé speciální soubory téhož typu periférie.

Použije-li proces podle našeho příkladu volání jádra `open` pro periférii `xx`, jádro aktivuje funkci `xx_open`, která připraví soukromé struktury pro práci se zařízením. Speciální soubor může ale otevřít více procesů současně, inicializace struktur je přitom provedena pouze při prvním otevření, což si zajišťuje ovladač, který si takto může evidovat násobný přístup k periférii a může jej omezovat.

Při použití volání jádra `close` je funkce `xx_close` jádrem aktivována, pouze jde-li o poslední proces, který dříve periférii otevřel, protože tato funkce ovladače uvolní soukromé struktury pro práci s periférií a zruší tak spojení s hardwarem.

Vlastní přenos dat zajišťují funkce `xx_read` a `xx_write` při používání volání jádra `read` a `write` pro čtení dat z periférie a zápis dat na periférii. Ovladač přitom často používá pro přenos dat soukromou vyrovnávací paměť, kterou odděluje proces od periférie, její zaplnění přitom přivodí zablokování

procesu. Typicky terminálový ovladač (viz čl. 6.1) svoji vyrovnávací paměť používá např. i pro řídicí operace nad tokem v/v dat. Manipulace s periferií při přenosu dat je silně závislá na hardwaru, který se může výrazně lišit, což je věcí programátora ovladače.

Volání jádra `ioctl` je obecně v UNIXu určeno pro řídicí operace nad periferiemi, a přestože má jednotný formát

```
#include <sys/types.h>
```

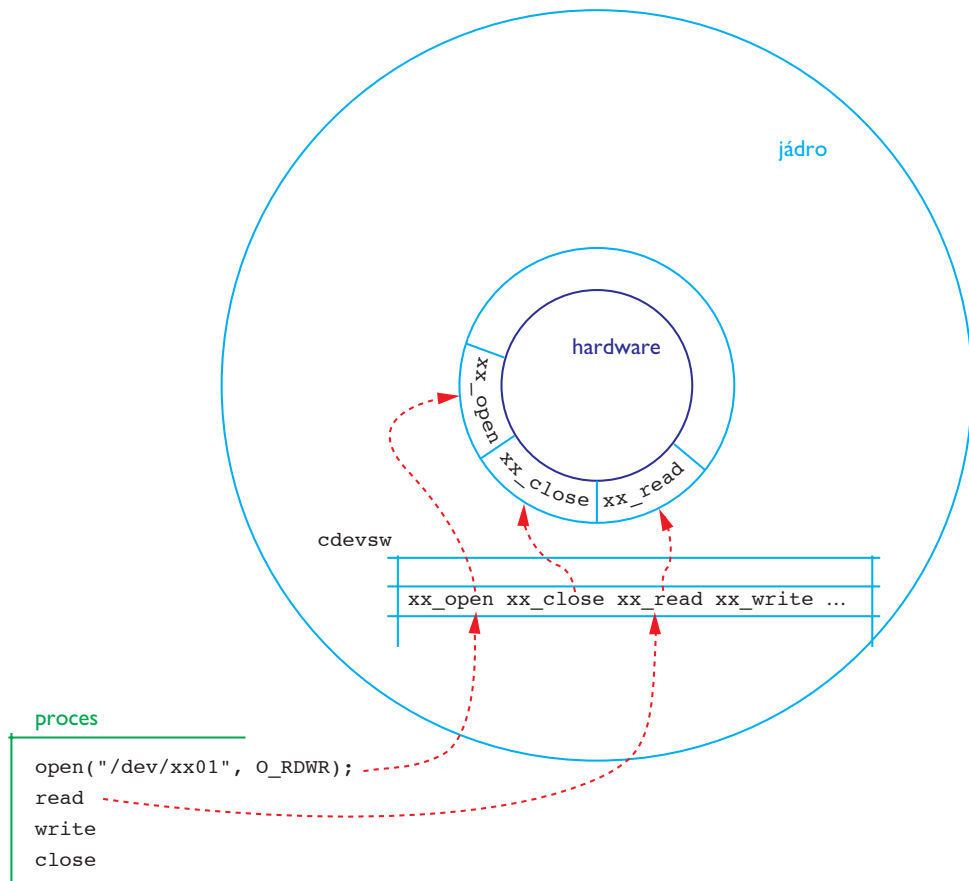
```
int ioctl(int fildes, int request, ... /* arg */);
```

požadované řídicí operace `request` s možnými argumenty `arg` nad deskriptorem otevřeného speciálního souboru `fildes`, jeho smysl je dán typem a možnostmi periferie. Praktické použití `ioctl` si uvedeme v dalších částech kapitoly. Ovladač pro jeho zabezpečení používá funkci `xx_ioctl`. Při konkretizaci volání jádra `ioctl` se velmi často používá `arg` jako ukazatel na strukturu. Funkce `xx_ioctl` ovladače tak přejímá řídicí data a charakteristiky pro nastavení periferie z datového prostoru procesu, nebo naopak sděluje nastavení periferie naplněním této struktury.

Blokové ovladače mají doplněnu také ještě funkci `xx_strategy`, která je aktivována jádrem při přenosu dat po blocích mezi systémovou vyrovnávací pamětí a svazkem. Jde o přístup k periférii pro optimální přenos celých diskových bloků, což má souvislost se systémem souborů podle kap. 3. Každopádně je funkce `xx_strategy` aktivována tehdy, je-li otevřen blokový speciální soubor periferie a data pro čtení a zápis procházejí systémovou vyrovnávací pamětí. Takový soubor je otevřen v okamžiku použití `mount` a svazek je tak virtualizován přes vyrovnávací paměť. Použijeme-li ale v průběhu připojení svazku další otevření téhož speciálního souboru (např. procesem `fsdb`), svazek je virtualizován dvakrát – pro jádro a pro proces – a může dojít ke kolizi.

Aktivace funkce `xx_intr` není závislá na použití některého volání jádra. Je to funkce, kterou jádro používá při výskytu přerušení, tj. v okamžiku, kdy naopak periferie aktivuje ovladač. Jádro podle typu a adresy přerušení rozezná konkrétní přerušující periférii a dokáže tak funkci ovladače parametrizovat konkrétní periferií. Funkce je podle nastavených priorit přerušovacího systému maskována a v takto daném pořadí provedena jádrem. Jaký je výsledek práce funkce, závisí na jejím obsahu. Jádro je takto upozorněno např. na dokončení v/v operace a oživuje proces, který je nad periferií zablokovaný. K vážné kolizi ve strukturách jádra by došlo v případě, že by zablokovaný proces mezitím skončil činností (dokážete promyslet proč?). Proto jádro odmítá odstranit proces zablokovaný v operaci jádra nad periférií třeba i při použití signálu `SIGKILL` privilegovaného procesu. Nastavení priority přerušení a samotné umístění přerušovacích vektorů je věcí strojově závislé části, ale při generaci systému jsou tyto informace často čitelné (a měnitelné) v některém ze základních hlavičkových souborů parametrů jádra v podstromu adresářů jeho generace. Takový soubor je v průběhu generace nového jádra kompilován a sestaven s ostatními moduly jádra.

Pojetí periferií jako speciálních souborů je výhodné, protože řada operací nad periférií (např. volání jádra `stat` nebo `lseek`) se vyřídí v jádru v obecných modulech systému souborů, aniž je potřeba aktivovat ovladač. Samotný ovladač je tak obvykle pouze několikastránkový program. Je napsán v jazyce C a jádro poskytuje programátorům knihovnu funkcí pro spojení s periférií např. pro přenos bytu z určité adresy atd. Programátor ovladače tak dokáže periférii připojovat k různým typům UNIXu, a to bez výrazné změny, ale vždy musí mít k dispozici dokumentaci generace jádra a připojování periferií k jádru, která by měla být součástí každého prodáváného UNIXu. Z uvedeného také vyplývá, že hlavní



Obr. 6.2 Ovladač

a vedlejší čísla speciálních souborů jsou dána implementací a chováním periferií. Obsazení tabulky `cdevsw` a `bdevsw`, resp. její volné pozice, jsou vždy předmětem zájmu programátora doplňujícího ovladače a nejsou obecným příkazem zjistitelné. Je nutné hledat v části generace jádra, kde jsou tabulky ve zdrojových textech generovány, nebo v provozní dokumentaci, viz také kap. 10.

Nahlédneme-li do adresáře `/dev`, nalezneme v něm v rozporu s tím, co zde bylo řečeno, i speciální soubory např. operační paměti (`/dev/mem` a `/dev/kmem`). Jádro takto poskytuje rozhraní speciálních souborů k hardwaru, které není typickou periferií. Sjednocuje se tak ale přístup k hardwaru, přestože jádro zajišťuje funkce ovladače takového rozhraní prostřednictvím modulů práce s operační pamětí.

## 6.1 Terminál

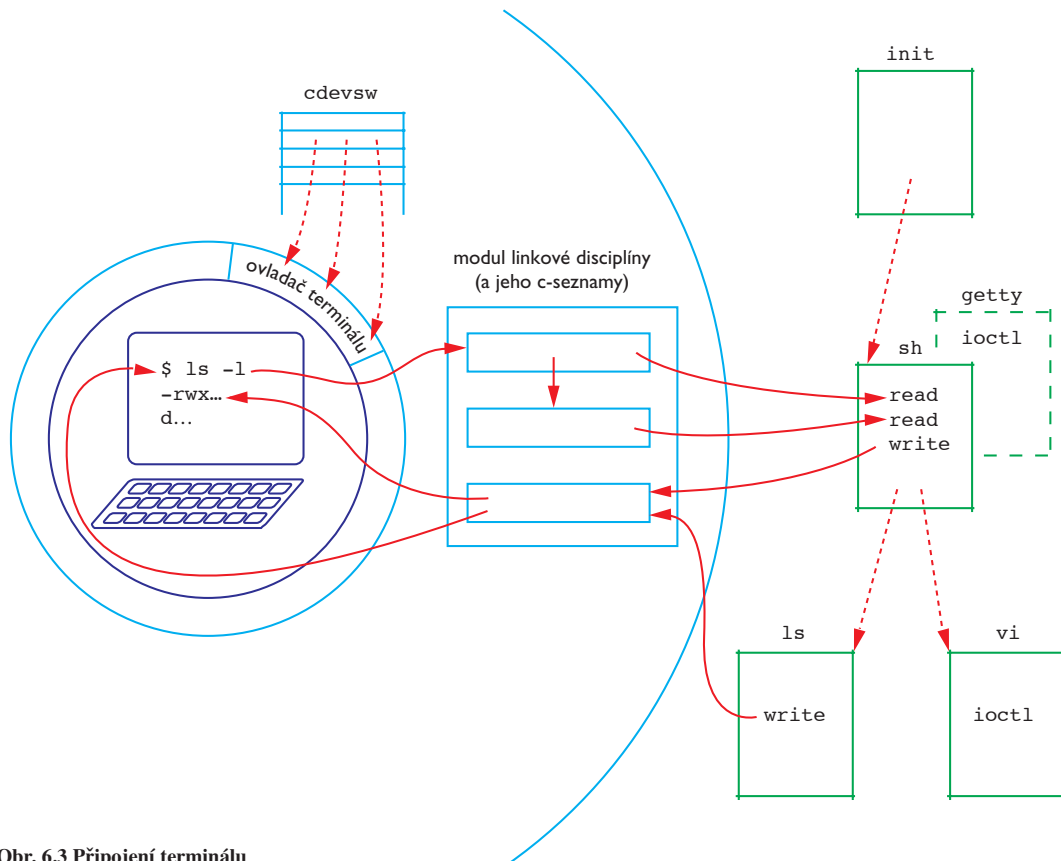
Terminál je znakové zařízení, jehož ovladač je součástí jádra tak, jak bylo řečeno v úvodu kapitoly. Přestože jde o sekvenční zpracování toku bytů mezi periferií a procesem, UNIX jej podporuje současně pro čtení i zápis. Říkáme, že jde o plně duplexní sériový přenos bytů. Proces komunikuje s terminálem pomocí běžných volání jádra, ale jde o poněkud výjimečnou periferii, protože většina uživatelských procesů má interaktivní charakter, jak vyplývá z povahy UNIXu. Každý interaktivní proces uživatelského sezení proto musí být spojen s nějakým terminálem – říkáme mu *řídící terminál* (controlling terminal). V běžném zpracování je řídící terminál součástí struktury `user` (viz kap. 2) procesu, tedy je zde evidováno hlavní a vedlejší číslo konkrétního zařízení. Proces řídící terminál získává jako součást svého dědictví rodiče, a přestože je terminál procesu dostupný pomocí tabulky deskriptorů otevřených souborů (položky 0, 1 a 2, postupně pro čtení, zápis a chybový zápis), protože tyto mohou být přesměrovány, má proces k dispozici ještě speciální soubor `/dev/tty`, který jádro vždy spojuje s odkazem na řídící terminál podle `user`. Řídící terminál však není pro proces existenčně důležitý výpočetní zdroj. Pokud proces řídící terminál nemá, otevřením speciálního souboru konkrétního terminálu jej získá, musí přitom ale být vedoucím skupiny procesů (předchází volání jádra `setpgrp`). Takovou akci provádí každý proces `getty` v okamžiku svého startu procesem `init`, který mu v parametru zadává speciální soubor jeho terminálu. `getty`, jak jsme ukázali v kap. 2, je prvním vedoucím skupiny procesů uživatelského sezení. Všechny jeho děti tak dědí jím nastavený řídící terminál. Proces naopak dosáhne nezávislosti na řídícím terminálu (odpojení od řídícího terminálu) založením vlastní skupiny procesů (pomocí `setpgrp` nebo `setsid` podle POSIXu), což je aktuální u démonů, pokud jsou spouštěny z příkazového řádku (a to nelze nikdy vyloučit). Z uvedeného také vyplývá, že volání jádra `open` přiřazuje řídící terminál, ale `close` jej nezavírá. Odpovídající funkci pro uzavření jádro používá v případě ukončení posledního procesu, který měl terminál nastavený jako řídící (obvykle konec práce shellu), nebo při `setpgrp` jediného takového procesu, např. při

```
$ exec hangup příkazový_řádek
```

příkazu pro shell sezení.

S otevřením řídícího terminálu je přiřazena procesu pro komunikaci se zařízením také *linková disciplína* (line discipline). Jedná se o způsob zpracování znaků zapisovaných na klávesnici a jejich předávání přijímajícímu procesu a řízení výstupu dat z procesu na obrazovku terminálu. Proces totiž může pracovat s terminálem (nebo se zařízením, které předává data počítači prostřednictvím sériového rozhraní, jako např. RS-232) jednak po řádcích a jednak po znacích. Linková disciplína je nastavena v okamžiku jejího přiřazení fyzickému terminálu na vstup dat po řádcích. Jde o tzv. *kanonický režim* (canonical mode). Proces jej však může (pomocí volání jádra `ioctl`) změnit na tzv. *přímý režim* (non-canonical mode). Kanonický (tedy řádný) režim je typický pro řádkovou komunikaci v shellu, a proto jej linková disciplína podporuje implicitně. Moduly jádra linkové disciplíny pracují s vyrovnávacími paměťmi, kterým říkáme *c-seznamy* (c-lists). Ty implementují fronty znaků vstupu a výstupu na terminál, jak ukazuje obr. 6.3.

C-seznam je organizace toku dat mezi periferií a procesem do sekvencí po několika znacích, tzv. c-bloků. Organizaci, tj. alokaci a zařazení dalšího c-bloku toku dat nebo uvolnění c-bloku v případě jeho vyprázdnění přenosem dat atd., v c-seznamu zajišťuje právě linková disciplína, která je věcí implemen-

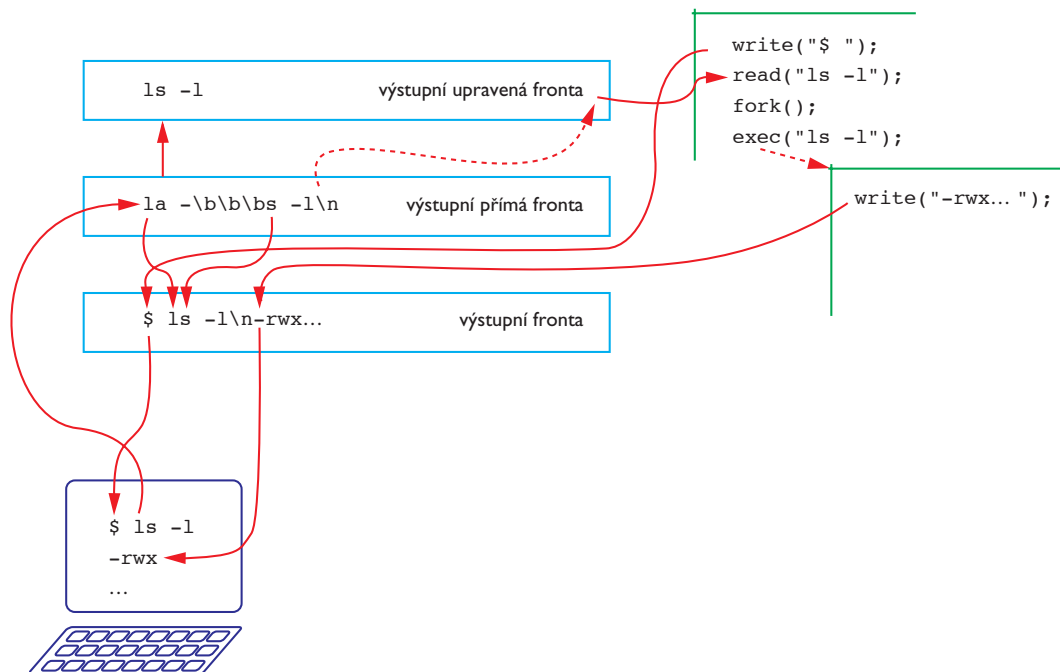


Obr. 6.3 Připojení terminálu

tace a správce ani programátor ji neovlivní. Pomocí c-seznamů jsou realizovány vstupní a výstupní fronty znaků pro zápis dat z procesu na periférii a čtení dat z periférie.

Na obr. 6.3 a 6.4 jsou uvedeny tři c-seznamy, které linková disciplína využívá pro každou periférii nezávisle na počtu procesů, které s ní manipulují. Terminál má tak přiřazeny tři fronty: dvě výstupní pro přenos dat z terminálu do čtoucího procesu a jednu vstupní pro zápis dat na terminál.

Přenos dat z klávesnice do adresového prostoru procesu v kanonickém režimu znamená využívání obou výstupních front linkové disciplíny. Fronta přímá (raw) obsahuje všechny znaky, které uživatel napsal, tzn. včetně všech překlepů a jejich opravných znaků. Výstupní fronta upravená (cooked) pak obsahuje už pouze data, která uživatel považuje za definitivní pro předání čtoucímu procesu. Která to jsou, čili které znaky jsou řídicí pro úpravu dat fronty přímé (např. pro smazání znaku, smazání celého řádku atd.), je definováno implicitně, ale lze je změnit voláním jádra `ioctl`. Linková disciplína provádí při každém stisku klávesy odpovídající zápis znaku do obou výstupních front a při každém stisku smluvené



Obr. 6.4 Vstupní a výstupní fronty linkové disciplíny terminálu

klávesy (obvykle Enter) provede přenos znaků z upravené výstupní fronty do adresového prostoru procesu, který na data čeká (je zablokovan ve volání jádra `read`). Na základě dalších smluvených kláves (např. Backspace) data upravené výstupní fronty upravuje podle smluveného významu. Součástí implicitního kanonického režimu je také opis stisknuté klávesy na obrazovku terminálu (plně duplexní režim musí mít vstup a výstup obecně nezávislý). Znamená to, že znaky přímé výstupní fronty jsou opisovány také do fronty vstupní. Upravená fronta je tedy využívána pro zajištění kanonického režimu. Změnu režimu linkové disciplíny (z kanonického na přímý nebo naopak) lze provádět voláním jádra `ioctl`. Nezávisle na nastaveném režimu lze pomocí `ioctl` zapínat nebo vypínat opis znaků z klávesnice na obrazovku (tj. jejich přenos z fronty výstupní do vstupní).

Vstupní frontu aktivuje linková disciplína v okamžiku, kdy proces použije volání jádra `write`. Plní data z adresového prostoru procesu do vstupní fronty. Přiblíží-li se zaplnění fronty mezní hodnotě její kapacity, linková disciplína aktivuje funkci ovladače a předává jí data, která se pak objevují na obrazovce. Linková disciplína přitom pracuje tak, že současně akceptuje přepis znaků výstupní fronty, pokud tedy v průběhu výpisu textu na obrazovce uživatel píše na klávesnici, znaky se na obrazovce objevují také.

Na obr. 6.4 je ukázka práce linkové disciplíny v kanonickém režimu procesu **sh** a **ls**. Uživatel se na klávesnici přepsal, namísto „ls -“ napsal „la -“, pak použil třikrát klávesu Backspace a tím zrušil v upravené frontě řetězec „a -“ a napsal správně „s -l“. Do vstupní fronty se přenášely všechny znaky výstupní přímé fronty, ale na obrazovce v době, kdy již zapisuje proces **ls** je správně viditelný text „ls -l“. Výstupní fronta bude dále plněna výstupem procesu **ls** („-rwx“ ... je začátek výpisu atributů některého souboru). Znak nového řádku (**\n**) způsobil odeslání textu „ls -l“ procesu **sh** a současně byl přenesen do vstupní fronty a zobrazen na obrazovce.

V režimu přímém čte linková disciplína znaky z terminálu a bez očekávání znaku konce řádku je ihned předává čtoucímu procesu. Přenos znaků lze (opět pomocí **ioctl**) řídit časovým limitem **TIME** (je nastavován v desítkách sekund, tj. hodnota 10 znamená 1s), v průběhu kterého linková disciplína očekává čtení nejméně **MIN** znaků z terminálu. Hodnoty obou konstant lze měnit a podle nich se mění chování linkové disciplíny v přímém režimu. Výchozí nastavení je **MIN** na hodnotu 1 a **TIME** na 0. Znamená to, že čtení (volání jádra **read**) je uspokojeno ihned po přečtení každého znaku. Význam různých hodnot konstant souvisí s chováním konkrétního terminálu, protože alfanumerický terminál běžně generuje po stisku některých kláves několik znaků (např. klávesy šipek nebo funkční klávesy). Taková sekvence má běžně uváděcí znak, např. Esc; proces, který čte uživatelskou klávesnici, přitom musí rozlišit, zda uživatel stisknul klávesu Esc nebo zda šlo třeba o klávesu šipka vpravo. Uživatel totiž těžko dokáže stisknout po Esc v rychlém sledu několik dalších kláves bez prodlevy např. 0.1 vteřiny – každopádně taková činnost uživatele není přirozená. Obrazovkové aplikace proto pracují s časovým limitem **TIME**, v průběhu kterého přenášejí **MIN** znaků, jejichž sekvence je teprve jako celek předána procesu, který čeká v operaci **read**.

Pro jednu periférii pracuje jedna sada v/v front. Periferie může být sdílena obecně více procesy. Jádro neorganizuje obecně zápis dat do vstupní fronty terminálu současně více procesy (pro uživatele v rámci sezení pracuje obecně více procesů), data na obrazovce mohou tak být promíchána v předem nezjistitelném pořadí. Totéž se týká výstupních front, kdy o výstup terminálu soupeří obecně více procesů. Takto pracují operační systémy UNIX, které nepodporují tzv. řízení prací uživatelského sezení. POSIX uvádí řízení prací jako nepovinnou část operačního systému. SVID ji definuje bez takového dovětku a v praxi je dnes známo jen málo současných verzí UNIXu, které by řízení prací neobsahovaly. Řízení prací je způsob synchronizace interaktivních procesů téhož řídicího terminálu. Jinými slovy, současné implementace různých shellů synchronizují přenos dat na obrazovku nebo z klávesnice téhož terminálu mezi procesy téhož sezení. V řízení prací (Job Control, viz čl. 2.5.1) je proces běžící na pozadí při operaci se svým řídicím terminálem jádrem zablokovan až do jeho převedení na popředí (příkazem **fg**). Jádro totiž generuje signál **SIGTTIN** při pokusu o čtení z terminálu nebo **SIGTTOU** při pokusu o zápis na terminál pro každý proces běžící na pozadí. Pokud proces signál nemaskuje, je pozastaven (zablokovan) do příchodu signálu **SIGCONT**; pokud nebyl proces dříve pozastaven, je tento signál implicitně ignorován. Na popředí běží pouze jeden proces sezení v rámci jedné skupiny procesů. Pokud proces ignoruje signál **SIGTTIN** nebo jej maskuje, a pokusí se přesto číst data z řídicího terminálu, volání jádra končí s chybou (**EIO** v **errno**, viz kap. 1). Pro zápis na terminál je situace stejná. Proces běžící na popředí však musí použitím **ioctl** (nastavením **TOSTOP** v **c\_lflag**, viz následující popis **ioctl**) situaci vytvořit což každý shell udělá, jinak je zápis na řídicí terminál úspěšný.

Uvedená situace se vztahuje na členy jedné skupiny procesů, chcete-li podle POSIXu, jednoho sezení. Řídicí terminál může mít přiřazen také sirotek nebo jiná skupina procesů, z níž žádný proces neběží (ani



nemůže) na popředí. Uvedené signály jim nebudou zaslány a při pokusu o čtení z řídicího terminálu skončí s chybou. Při pokusu o zápis rovněž, pokud jim to proces na popředí pomocí `ioctl` dovolí.

Linková disciplína (v přímém i kanonickém režimu) pracuje za podpory sady znaků zvláštního významu, které lze v `ioctl` předefinovat. Jde např. o znak, který, je-li klávesnicí generován, jádro nepředává procesu jako znak, ale posílá procesu signál `SIGINT` (jde o stisk klávesy Delete nebo Ctrl-c). Také je běžné, že terminál pracuje v režimu XON/XOFF, tj. zápis na terminál, pokud je příliš rychlý, může uživatel zastavit stiskem určité klávesy (Ctrl-s) a posléze požadovat jeho pokračování (Ctrl-q). Struktura `termios` zahrnuje pole takových znaků `cc_c` (viz dále) a podle POSIXu má pořadí prvků pole význam definovaný v `<termios.h>`:

kanonický režim	přímý režim	význam
VEOF		znak ukončení vstupu (EOF)
VEOL		znak ukončení řádku (EOL)
VERASE		zrušení předchozího znaku (ERASE)
VINTR	VINTR	přerušování, generace <code>SIGINT</code> (INTR)
VKILL		zrušení celého řádku (KILL)
	VMIN	počet znaků MIN v čase TIME
VQUIT	VQUIT	XON
VSUSP	VSUSP	XOFF
	VTIME	čas TIME počtu znaků MIN
VSTART	VSTART	signál <code>SIGCONT</code>
VSTOP	VSTOP	signál <code>SIGTTIN</code> a <code>SIGTTOU</code>

Na úrovni programátora lze ovládat práci linkové disciplíny voláním jádra `ioctl`. Jeho popis pro použití ve všech uvedených možnostech linkové disciplíny terminálu je v sekci (4) provozní dokumentace pod označením `termio` nebo `termios` (pro POSIX), všechny potřebné definice struktur a konstant pak v souborech `<termio.h>` nebo `<termios.h>` (který podporuje pouze POSIX). Při používání `ioctl` pro práci s terminálem je v dokumentaci definována řada možností, které lze použít na místě `request` (podle formátu `ioctl` ze začátku kapitoly). Pro naplnění obsahu struktury `termio` (na kterou je ukazatel v následujícím argumentu) aktuálními hodnotami práce linkové disciplíny a přenosovými charakteristikami používáme `TCGETA`, naopak při požadavku změn jádru poskytujeme obsah struktury pomocí `TCSETA`. POSIX pracuje pouze se strukturou `termios`. Pro zajištění kompatibility SVID proto uvádí tuto možnost jako alternativní a definuje pro její použití `TCGETS` pro získání hodnot od jádra a `TCSETS` pro jejich nastavení podle `termios`, rovněž uvedené odkazem v následujícím argumentu `ioctl`. Struktura `termios` obsahuje následující položky:

```
tcflag_t   c_iflag;    /* vstupní charakteristiky */
tcflag_t   c_oflag;    /* výstupní charakteristiky */
tcflag_t   c_cflag;    /* řídicí charakteristiky */
tcflag_t   c_lflag;    /* lokální charakteristiky */
cc_t       c_cc[NCCS]; /* řídicí znaky */
```

`c_iflag` definuje práce s klávesnicí, např. zapínání a vypínání přerušování z klávesnice, kontrola parity atd., `c_oflag` definuje způsob zacházení s výstupem na obrazovku, např. zda je znak nového řádku LF nahrazen dvěma znaky CR a LF (je důležité i při konfiguraci sériové tiskárny), zpoždění výpisu po

řádcích apod. Důležitá je položka `c_cflag`, v jejímž obsahu jde o definici přenosových charakteristik práce ovladače s konkrétním terminálem, jako je např. přenosová rychlost, počet bitů na znak, parita atd. Konečně `c_lflag` definuje režim linkové disciplíny, opisování znaků z klávesnice na obrazovku, generování signálů na základě znaků přerušeni nebo pozastavení procesu na popředí atd. Pole znaků `c_cc` je definice znaků zvláštního významu. Následující příklad zachovává současný stav přenosových charakteristik řídicího terminálu procesu, nastavuje však režim linkové disciplíny na přímý, zapíná respektování signálů a vypíná opis znaků:

```
#include <termios.h>

struct termios t;

int main(void)
{
    int fd;
    f=open("/dev/tty", O_RDWR);
    ioctl(fd, TCGETS, &t);
    t.c_lflag|= ~ICANON | ISIG | ~ECHO;
    ioctl(fd, TCSETS, &t);
}
```

Po provedení našeho krátkého programu je změna práce řídicího terminálu provedena pro práci všech procesů, které mají nastavený tentýž terminál jako řídicí terminál. Rekonstrukce stavu např. pro shell, v rámci kterého jsme program spustili, bude pak poněkud obtížná, přestože nikoliv nemožná. Dosáhneme ji buďto pomocí příkazu **stty** nebo pouze prostým odhlášením (zrušením procesu výchozího shellu).

Proces **getty** otevírá řídicí terminál uživatelského sezení. Proces výchozího shellu přihlášeného uživatele je proměněn z **getty** (pomocí volání jádra **exec**, viz kap. 2) a používá tak stejný řídicí terminál jako každý jeho dětský proces. Ukončí-li proto uživatel sezení, proces **init** startuje **getty** znovu a znovu otevírá právě uzavřený řídicí terminál, inicializuje přitom stav linkové disciplíny a ovladače terminálu. Správce systému tuto inicializaci může ovlivnit změnou v `/etc/inittab` a `/etc/gettydefs`, jak ukážeme v kap. 10. **getty** k potřebnému nastavování používá `ioctl` v uvedeném kontextu.

Přestože to není nutné, i uživatel v příkazovém řádku svého shellu může měnit nastavení linkové disciplíny řídicího terminálu svého sezení. Učiní tak pomocí příkazu **stty**, jehož formát je v každém UNIXu podrobně komentován. **stty** pro zajištění uživatelského požadavku používá rovněž volání jádra `ioctl`. Současný způsob práce řídicího terminálu a jeho linkové disciplíny lze pořídit zadáním

```
$ stty -a
```

Změny se provádějí volbami příkazu podle dokumentace a v kontextu uvedeného významu, např.

```
$ stty -echo
```

vypne opisování klávesnice na obrazovku a

```
$ stty echo
```

je opět zapíná. Pro legraci si **stty** vyzkoušejte. SVID i POSIX definují pro pohodlí uživatele v uvedeném kontextu také ještě příkaz **tabs**, kterým lze nastavit pozice kurzoru při používání klávesy tabulátoru.

POSIX nepodporuje volání jádra `ioctl`. Pro práci s periferiemi vždy definuje konkrétně pojmenované funkce, které pak odpovídající UNIX implementuje buďto jako volání jádra, nebo funkcí s použitím `ioctl`. Funkce `tcgetpgrp` ve své návratové hodnotě např. poskytne PID procesu, který pracuje nad řídicím terminálem v popředí. Řídicí terminál je uveden v argumentu funkce v podobě deskriptoru otevřeného souboru. `tcsetpgrp` přesouvá proces uvedený v parametru na popředí také v parametru zadaného řídicího terminálu. Pro zjišťování a nastavování přenosových charakteristik a linkové disciplíny jsou definovány funkce `tcgetattr` a `tcsetattr`, u nichž v parametru zadáváme odkaz na strukturu `termios`. Zvláště jsou definovány funkce pro práci s přenosovou rychlostí (funkce `cfgetospeed`, `cfsetospeed`, `cfgetispeed` a `cfsetispeed`) a dále řídicí funkce zajišťující pozastavení procesu do faktického uskutečnění přenosu znaků (`tcdrain`, `tcflush`), průchod znaku zvláštního významu linkovou disciplínou mimo pořadí (`tcsendbreak`) a funkce pro zajištění protokolu XON/XOFF (`tcflow`). V SVID nebo každé implementaci UNIXu s POSIXem kompatibilní nalezne programátor funkce v manuálu `termios` části volání jádra nebo funkcí systému.

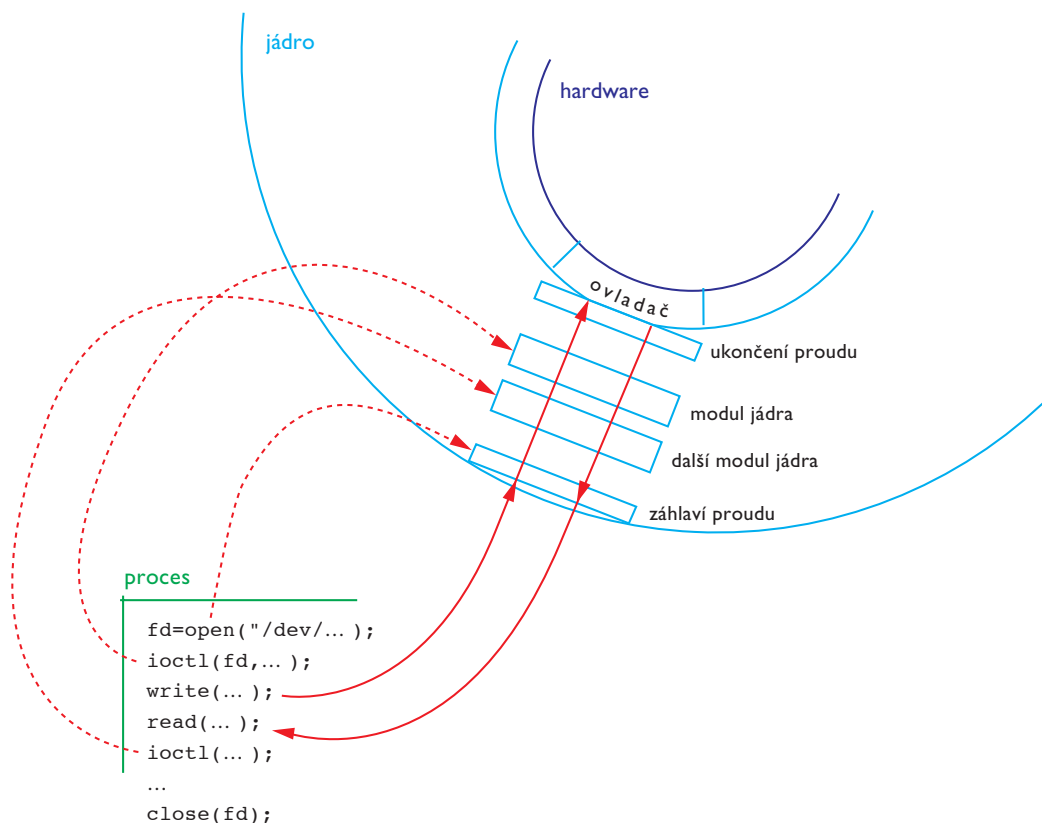
Změnu kanonického režimu na přímý, vypínání opisu znaků z klávesnice na obrazovku používá běžně každá obrazovkově orientovaná aplikace, jako je např. editor **vi**. Pro snazší programování takových aplikací právě při vývoji programu **vi** vznikla knihovna funkcí zvaná CURSES, která umožňuje nejenom nastavovat přímý režim a zpět kanonický atd., tj. práci linkové disciplíny, ale umožňuje pracovat s obrazovkou terminálu jako s plochou pro definici oken o kapacitě znaků počet řádků vynásobený počtem sloupců. Práci v oknech pak podporuje sadou dalších funkcí, které každý programátor uvítá, zvláště je-li jejich použití nezávislé na typu alfanumerického terminálu. V proměnné prostředí prováděného procesu má totiž každý uživatel definovanou proměnnou **TERM** (viz. čl. 2.5.2), podle jejíhož obsahu inicializační funkce knihovny CURSES čtou způsob ovládání obrazovky a generaci znaků z klávesnice z databanky terminálů `terminfo` (v místě `/usr/share/lib/terminfo`). Programátor nalezne popis použití odpovídajících funkcí knihovny CURSES v sekci (3) provozní dokumentace UNIXu nebo v SVID.

Grafické periferie přístupu uživatelů k operačnímu systému využívají rozhraní X-Window System. Jde o část UNIXu, které v této knize věnujeme zvláštní kapitolu. Pracovní plocha (desktop) uživatelského sezení umožňuje spouštět procesy kliknutím myši periferie na odpovídající ikonu, která je s procesem spojena. Takovým procesem je také okno s emulací prostředí alfanumerického terminálu pro zpřístupnění vrstvy textové komunikace pomocí některého z shellů. Zde jde o emulaci terminálu a plnohodnotné zajištění linkové disciplíny, jak bylo v tomto článku uvedeno. Periferie terminálu je i zde virtualizována přes `/dev/tty`, ale fyzicky jde o tzv. *pseudoperiferii*. *Pseudoterminály* jsou využívány pro práci síťových aplikací vzdáleného přihlašování (jako je **telnet** nebo **rlogin**). Jejich implementace je pak realizována pomocí tzv. STREAMS, kterým věnujeme následující čl. 6.2. Pseudoterminály popíšeme v odst. 7.4.3. Vzhledem k tomu, že pro pochopení koncepce X-Window System je nutná také znalost principů sítě a zároveň jde o samostatnou partii systému, prozkoumáme ji v samostatné kap. 8.

## 6.2 PROUDY – STREAMS

Linková disciplína a sériové připojení terminálu k výpočetnímu systému jako hlavní komunikační prostředek vstupu a výstupu dat v operačním systému UNIX je často používána nejenom pro interaktivní přístup uživatele prostřednictvím klávesnice a obrazovky. Chování linkové disciplíny je řízení

duplexního toku znaků vstupu/výstupu sériového rozhraní, a to se využívá např. pro připojení inteligentních strojů, kde jde o omezený objem přenášených dat (např. data měřicího přístroje nebo průmyslového výrobního stroje). V kap. 7 také uvidíme, že modemové spojení dvou výpočetních systémů využívá linkovou disciplínu a sériové rozhraní nikoliv pouze pro vzdálené přihlášení jednoho uživatele (pomocí programu **cu** v podsystému UUCP), ale také lze prostřednictvím dalších prostředků **slip** nebo **ppp** vytvořit emulovaný přenosový protokol běžné sítě. Takovou komunikaci na bázi sériového rozhraní je totiž možné v přímém režimu linkové disciplíny dostatečně zpracovat, jak vyplývá z předchozího článku. Obecné nasazení linkové disciplíny pro zajištění síťových protokolů by bylo sice možné, ale velmi neflexibilní. Síťová komunikace totiž vyžaduje vrstvenou stavbu, kdy lze programové moduly jednotlivých vrstev zaměňovat za jiné, a to nikoliv pouze při vytvoření nové verze téhož protokolu, ale v běžném provozu, kdy např. namísto protokolu stálého spojení (TCP) se využívá stejným síťovým rozhraním spojení datagramové (UDP). Implementaci sítí lze pochopitelně dosáhnout i využíváním



Obr. 6.5 Princip PROUDŮ (STREAMS)

různých pevně vestavěných modulů jádra, které se aktivují podle použitého volání jádra nebo při kombinaci jeho parametrů, což je způsob zvaný Berkeley sockets (schránky). Vývojový tým firmy AT&T pro UNIX SYSTEM V uplatnil při práci na implementaci síťových protokolů návrh D. Ritchieho z r. 1984. Jde o tzv. STREAMS (PROUDY), což je možnost využívání modulů jádra pro zpracování (filtraci) dat, která proces přenáší (duplexně voláním jádra `write` nebo `read`) mezi svým datovým prostorem a ovladačem periferie. Modulů jádra lze přitom takto využít několik. Data jimi procházejí postupně a proces přitom sám řídí pořadí vsouvání modulů mezi něj a periferii. Využití PROUDŮ se neomezuje pouze na zvláštní periferie, jako jsou např. sítě. Je obecně použitelné a dnes je pomocí nich implementována i linková disciplína terminálového rozhraní. PROUDY tak sjednocují v jiných implementacích násobně programované moduly práce s periferiemi a eliminují tak nejednoznačnost. PROUDY jsou podporovány pro znakové rozhraní.

Obr. 6.5 ilustruje výchozí situaci.

Z pohledu procesu jde o rozšíření dosavadní komunikace s periferií. Pokud proces otevírá speciální znakový soubor, který podporuje PROUD, současně s přidělením deskriptoru je pro tento vstup do jádra alokováno tzv. záhlaví (stream head module) a ukončení (stream end module) PROUDU. Záhlaví je modul nejblíže k procesu, ukončení odpovídá spojení na ovladač.<sup>2</sup> Pokud proces neřekne jinak, data prochází záhlavím i ukončením PROUDU bez jakékoliv změny a výsledek je stejný jako bez použití PROUDŮ. Mezi záhlavím a ukončením může proces vkládat další moduly jádra, která data a jejich prostupnost mezi procesem a ovladačem mění. Používá k tomu volání jádra `ioctl`, kdy na místě `request` je uveden příkaz `I_PUSH` a jako další parametr následuje text jména modulu. Jména modulu musí být pochopitelně správná. Která to jsou, je definováno v dokumentaci pro jednotlivé periferie. Např.

```
ioctl(fd, I_PUSH, "ip");
```

Následným použitím `ioctl` s použitím `I_PUSH` vkládá proces do cesty svým datům další modul, a to opět pod záhlaví PROUDU, např.

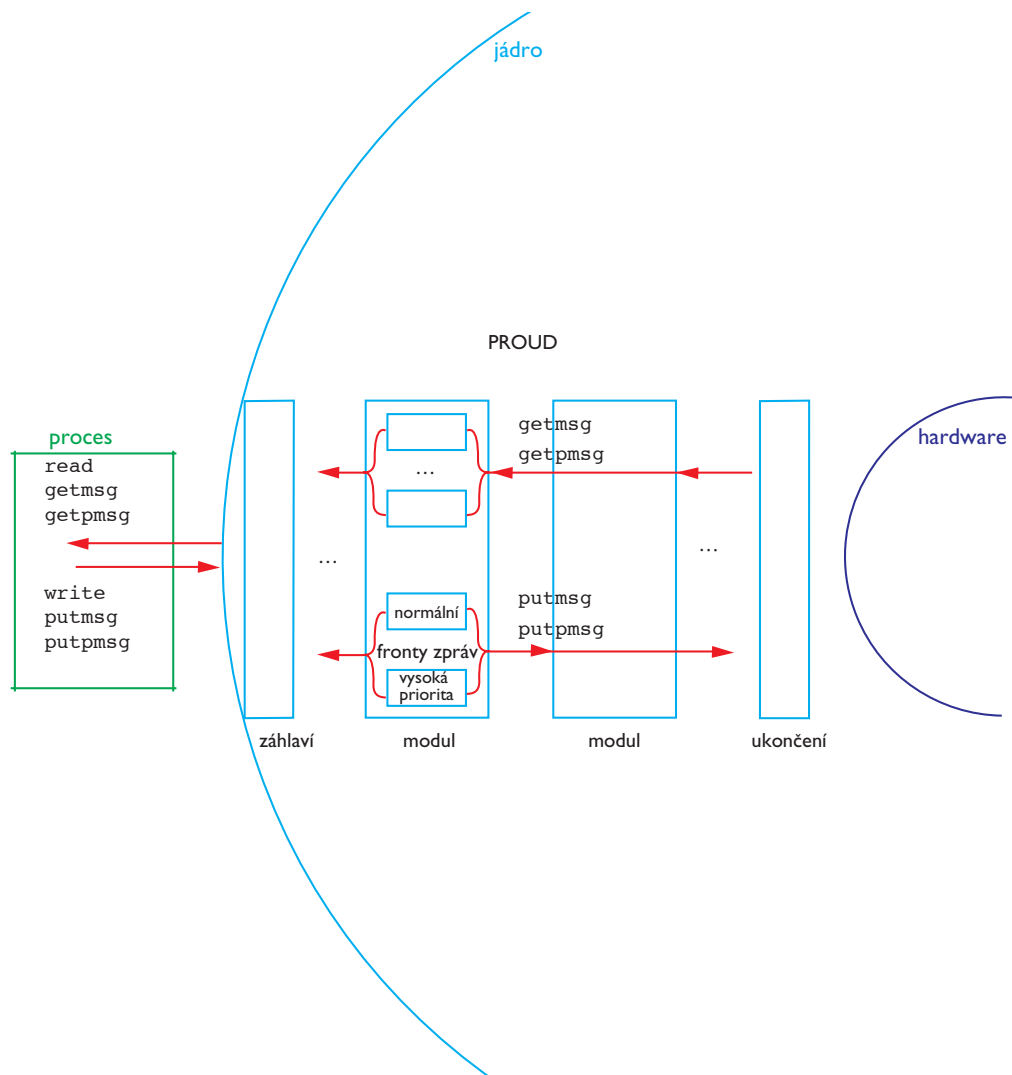
```
ioctl(fd, I_PUSH, "tcp");
```

Data zapsaná pomocí volání jádra `write` nyní budou procházet nejprve záhlavím PROUDU, modulem `tcp`, pak `ip`, odkud jsou předána přes ukončení PROUDU funkcím ovladače. U volání jádra `read` bude cesta dat přesně opačná. Vyjmout z PROUDU lze pouze modul naposledy vložený, a to pomocí `I_POP`:

```
ioctl(fd, I_POP, 0);
```

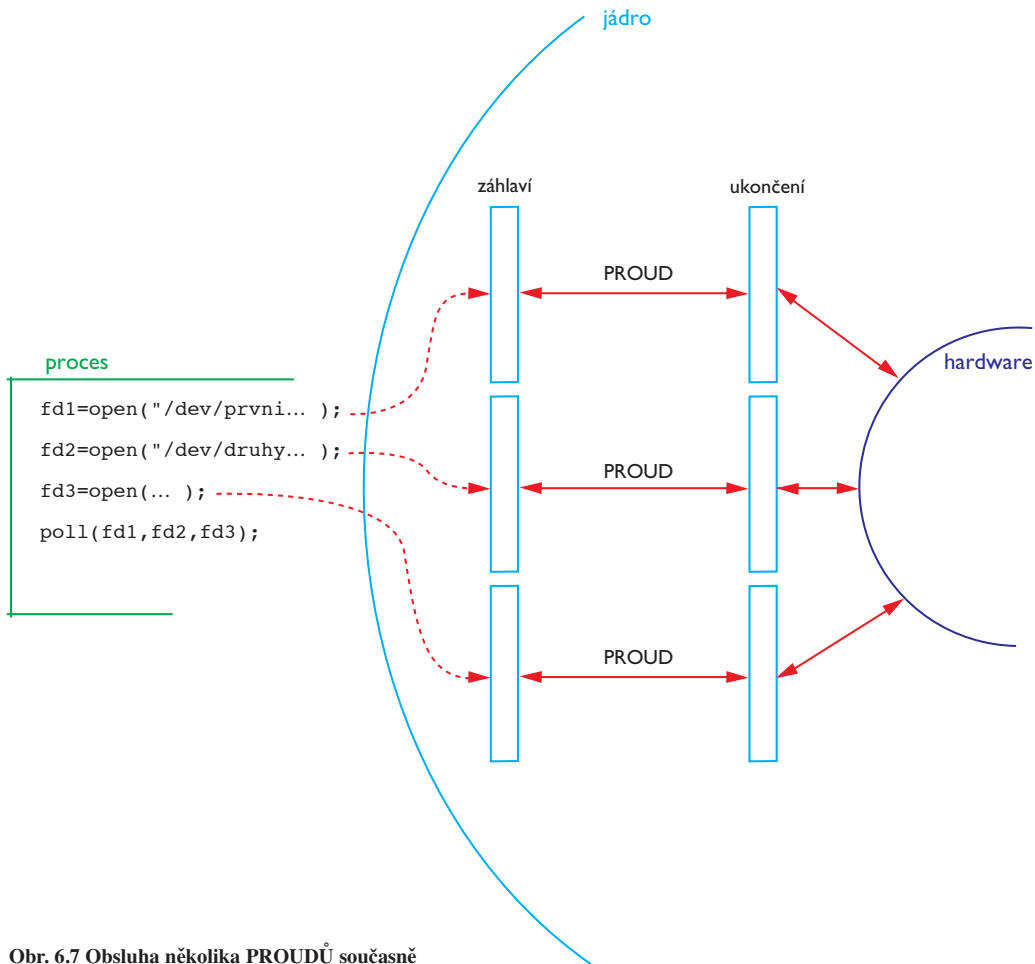
Práce procesu se zásobníkem modulů je tedy používání různých vrstev průchodu dat jádrem, které jsou programovány jako moduly jádra. Počet současně používaných modulů jednoho PROUDU je omezen pouze parametrem jádra a lze jej změnit novou generací jádra (viz kap. 10).

Technologie PROUDŮ je založena na předáváním zpráv mezi jednotlivými moduly. Zpráva prochází modulem PROUDU oběma směry a tvoří tak tok v/v dat procesu. Každá zpráva má svůj typ a obsah. Modul PROUDU přebírá každou zprávu, zkoumá její typ a pokud mu nerozumí, předává zprávu dalšímu modulu v pořadí. Pokud typu zprávy rozumí, provede s daty zprávy odpovídající operaci, jejíž výsledek může např. být nový typ zprávy a změněná data. Proces s daty pracuje pomocí `write` a `read`, ale může používat i další volání jádra, kdy data vyjadřuje jako zprávu,



Obr. 6.6 Zprávy PROUDU

kteřá je vkládána do záhlaví PROUDU. Jde o použití funkcí jádra, kterými si moduly PROUDU mezi sebou předávají data a které jsou s jistým omezením zveřejněny procesům jako volání jádra. Volání jádra `putmsg` a `putpmsg` vkládají (zapisují) zprávu do PROUDU a `getmsg` a `getpmsg` zprávu z PROUDU odebírají (čtou). Typ zprávy je rozlišován jejich prioritou, která může být buď normální, tj. žádná (normal, non-priority), prioritní (priority) nebo s vysokou prioritou (high-priority). Podle priority



Obr. 6.7 Obsluha několika PROUDŮ současně

může být zpráva předávána dalšímu modulu mimo pořadí jejího příchodu. Zprávy jednotlivých skupin podle priorit jsou přitom zařazovány do front a předávány dál sekvenčně podle pořadí jejich příchodu. Priorita u zpráv je vyjádřena číselně. Normální zprávy mají prioritu č. 0. Vysoká priorita je vyjádřena konstantou `RS_HIPRI`. Mezi nimi jsou ostatní typy zpráv (prioritní zprávy). Podle typu zprávy také říkáme, že zpráva patří do určité prioritní skupiny (priority band). Prioritní zprávy jsou vyřizovány přednostně a zprávy s vysokou prioritou jsou předávány zcela mimo pořadí (out of band), přestože i tyto jsou frontovány. PROUDY používají priority zpráv např. pro přenos řídicích znaků klávesnice terminálu při vzdáleném přihlášení prostřednictvím sítě nebo u běžné linkové disciplíny, kdy stisk např. klávesy Delete musí „předběhnout“ všechna data již odeslaná do jádra (viz obr. 6.6).

Volání jádra na úrovni zpráv otevřeného PROUDU mají podobný formát. `getpmsg` a `putpmsg` umožňují citlivější práci s prioritami front, jinak jejich použití znamená totéž co `getmsg` a `putmsg`.

```
#include <stropts.h>
```

```
int putmsg(int fd, const struct strbuf *ctlptr,  
           const struct strbuf *dataptr, int flags);  
int putpmsg(int fd, const struct strbuf *ctlptr,  
            const struct strbuf *dataptr, int band, int flags);  
int getmsg(int fd, const struct strbuf *ctlptr,  
           const struct strbuf *dataptr, int flagsp);  
int getpmsg(int fd, const struct strbuf *ctlptr,  
            const struct strbuf *dataptr, int bandp, int flagsp);
```

Záhlaví PROUDU, daného deskriptorem `fd` (získaného v návratové hodnotě `open`), předáme řídicí `ctlptr` nebo datovou `dataptr` zprávu (nebo obě) pomocí `putmsg`. Hodnota `flags` vyjadřuje prioritu zprávy buď jako `RS_HIPRI` pro vyjádření vysoké priority nebo 0 pro normální zprávu. V případě vkládání zpráv jiného typu používáme `putpmsg`, kde je typ zprávy dán v `band`. `flags` pak má význam bitové masky vzájemně se vylučujících hodnot `MSG_HIPRI` a `MSG_BAND` a podle jejich použití lze do PROUDU vkládat také prioritní zprávy nebo zprávy vysoké priority. Datová struktura `strbuf` má položky

```
int maxlen; /* max. délka dat v buf;  
            využívá se pouze u getmsg a getpmsg */  
int len;    /* délka předávaných dat */  
char *buf;  /* ukazatel na přenášena data */
```

Jejich použití je zřejmé. Podobně volání jádra `getmsg`, příp. `getpmsg`, přijímá zprávu ze záhlaví PROUDU. Proces se přitom může zajímat o prioritní nebo vysoceprioritní typy zpráv a ošetřit tak data zvláštního posílání. Také se může pouze zajímat o přítomnost zprávy určitého typu v PROUDU.

Proces má tedy další možnosti ovládání toku dat, přestože sémantika jednotlivých modulů PROUDU je uvedeným mechanismem ovlivnitelná pouze na úrovni určení různého typu zprávy, což je dobře. Programovat modul jádra, který je pak využitelný libovolným procesem pro práci s danou periferií, znamená zvládnout také zařazení modulu do jádra. Metody, jak se toho dosahuje, nejsou obecné (stejně jako nejsou obecné metody generace jádra, viz kap. 10) a ani je neobsahuje SVID, přestože součástí dokumentace každého průmyslového UNIXu je kniha (obvykle několikasetstránková) s návodem, jak má programátor postupovat. POSIX PROUDY neuvádí vůbec, protože jde o technologii implementace. SVID definuje PROUDY tak, jak zde bylo popsáno.

Uvedené zpracování dat při průchodu jádrem je mnohdy nutné využívat multiplexovaně, čímž zde rozumíme použití PROUDU pro jeden proces a více periferií (např. při použití několika síťových adaptérů při tzv. směrování, routing) nebo pro více procesů a jednu periferii (grafický terminál je používán pro vstup i výstup několika nezávislými procesy) nebo v různých dalších kombinacích.

Obsluhu několika toků dat v různých PROUDECH zajistí procesu volání jádra `poll`, viz obr. 6.7.

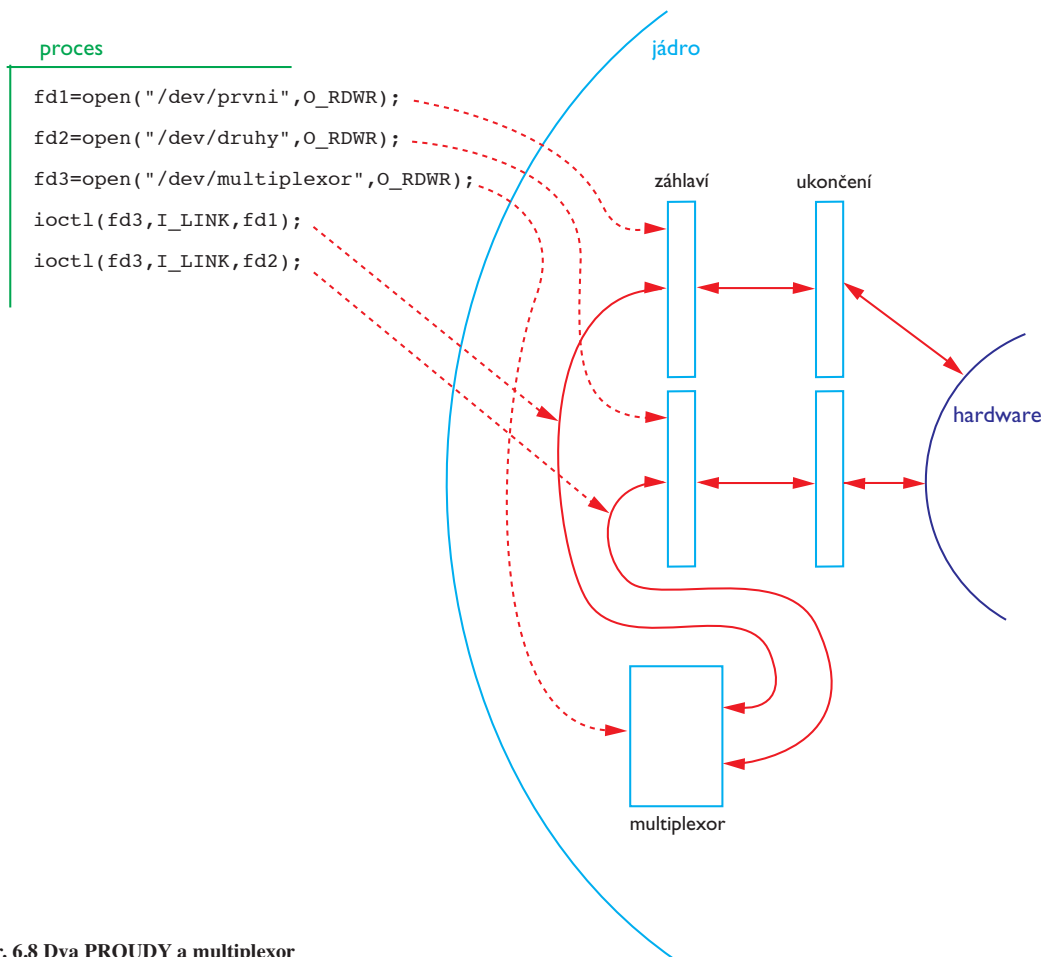


Proces, který zpracovává údaje PROUDŮ potřebuje informace o připravenosti PROUDU pro přenos dat, případně k jakému typu (prioritě) se přenášena data vztahují. Volání jádra má formát

```
#include <poll.h>
```

```
int poll(struct pollfd fds[], unsigned long nfds, int timeout);
```

Odkazy na PROUDY jsou uloženy v poli `fds`. Následující argument `nfds` uvádí jejich počet, tj. velikost pole `fds`. `timeout` je časový interval, po který bude proces blokován v očekávání události nad některým z PROUDŮ. Pokud je 0, jádro zjistí stav PROUDŮ, předá je procesu, který ihned pokračuje. Je-li -1, proces je zablokován do chvíle, kdy požadované stavy PROUDŮ nastanou. Proces může



Obr. 6.8 Dva PROUDY a multiplexor

přítom být v tomto blokování přerušen signálem. Položky pole `fds` jsou struktury `pollfd`, jejíž členy jsou

```
int fd;           /* deskriptor otevřeného PROUDU */
short events;     /* očekávané události */
short revents;    /* události, které nastaly */
```

Proměnné `events` i `revents` jsou přítom bitová pole, která mohou zahrnovat informaci o možnosti čtení normálních dat (`POLLRDNORM`), prioritních dat (`POLLRDBAND`) nebo dat s vysokou prioritou (`POLLPRI`), analogicky o možnosti zápisu dat (`POLLWRNORM`, `POLLWRBAND`, data s vysokou prioritou lze zapsat vždy) nebo některé další manipulace s PROUDY. Potvrzení požadavku v `revents` pak znamená, že odpovídající volání jádra pro přenos dat mezi procesem a PROUDEM nezablokuje proces v jádru.

Pokud proces zajímá pouze jeden PROUD, nemusí k jeho testování používat `poll`, ale `ioctl`, které všechny potřebné informace o PROUDU od jádra získá. Pomocí `ioctl` může také proces požadovat od jádra zaslání signálu `SIGPOLL`, pokud nastane v odkazovaném PROUDU událost, kterou si určí. Např. zadáním

```
ioctl(fd, I_SETSIG, S_HIPRI);
```

žádá proces jádro o zaslání signálu `SIGPOLL` v okamžiku, kdy se v PROUDU daného deskriptorem `fd` objeví data mimo rozsah (zpráva s vysokou prioritou).

Další možnosti a podrobnosti, týkající se např. zjišťování konzistence PROUDŮ nebo vysílání testovacích dat PROUDEM, lze objevit v dokumentaci PROUDŮ v části provozní dokumentace, která se vztahuje k hardwaru u všech systémů odpovídajících SVID.

Multiplexování (multiplexing) na úrovni spojování PROUDŮ v jádru dosahujeme voláním jádra `ioctl`. Na obr. 6.8 je výchozí situace po otevření dvou PROUDŮ (deskriptory `fd1` a `fd2`) a modulu jádra, který dokáže tok dat mezi procesem a více periferiemi multiplexovat (deskriptor `fd3`). Na obr. 6.8 je takový modul označen jako multiplexor. Přitom jde o modul jádra, který je do něj začleněn jako běžný ovladač, tzn. je také aktivován otevřením speciálního souboru. Takovému modulu jádra říkáme pseudozařízení (pseudo-device). Obr. 6.8. dále ukazuje účinek volání jádra `ioctl`, kdy deskriptor multiplexoru spojíme s deskriptorem PROUDU `fd1` a následně opět pomocí `ioctl` s deskriptorem `fd2` druhého PROUDU. `fd3` tak ukazuje na multiplexor, který data distribuuje mezi oběma PROUDY.

SVID doporučuje mít zásobník modulů každého multiplexovaného PROUDU vytvořený před navázáním na multiplexor. Rovněž tak je doporučováno odkazovat po připojení multiplexoru na PROUD pouze odkazem na deskriptor multiplexoru. Jádro by mělo přístup přes původní deskriptory odmítat. Proces je může dokonce uzavřít a uvolnit tak pozici tabulky deskriptorů. Jak je na obr. 6.8 uvedeno, `ioctl` používáme s parametrem `I_LINK` na místě `request` pro připojení dalšího PROUDU. Pro odpojení PROUDU od multiplexoru pak používáme na místě `request` hodnotu `I_UNLINK`. Při odpojování PROUDU od multiplexoru je nutné použít návratovou hodnotu `ioctl` z okamžiku připojení, protože původní deskriptor PROUDU již nemá žádný význam, takže správně bychom např. na obr. 6.8 měli psát

```
fd1=open("/dev/prvni", O_RDWR);
...
```

```
fd3=open("/dev/multiplexor", O_RDWR);
stream1=ioctl(fd3, I_LINK, fd1);
close(fd1);
```

```
...
ioctl(fd3, I_UNLINK, stream1);
```

Pro odpojení všech dříve připojených PROUDŮ od multiplexoru lze také použít na místě `arg` hodnotu `MUXID_ALL`, např.

```
ioctl(fd3, I_UNLINK, MUXID_ALL);
```

Pravděpodobně není nijak obtížné konstruovat případ sdílení jednoho PROUDU více deskriptory (obr. 6.9).

a čtenář si může sekvenci volání jádra z pilnosti naprogramovat za domácí úkol. Kombinovat přitom různé způsoby spojování modulů jádra, které plní funkci multiplexování s PROUDY, je možné a jedná se již o obsahovou stránku toku dat mezi periferiemi a procesy. Z obr. 6.9 se vnucuje myšlenka trvalejšího spojení multiplexoru a PROUDU. `ioctl` disponuje parametrem `P_LINK`, pomocí kterého vznikne spojení PROUDU a multiplexoru, a to trvá až do použití `ioctl` s hodnotou `P_UNLINK`, rovněž na místě `request` (`MUXID_ALL` lze také použít). Lze tak připravit periferii (pod jménem multiplexoru), která plní tu funkci, kterou správce systému nastaví.

Praktický příklad použití PROUDŮ v síťové komunikaci je knihovna TLI (viz čl. 7.3.2). S pseudozařizováními se setkáme také při emulaci terminálů na síťovém rozhraní. Konečně vzpomeňme také současné implementace roury a pojmenované roury v kap. 4, kdy je technologie PROUDŮ používána ve svém vedlejším efektu.

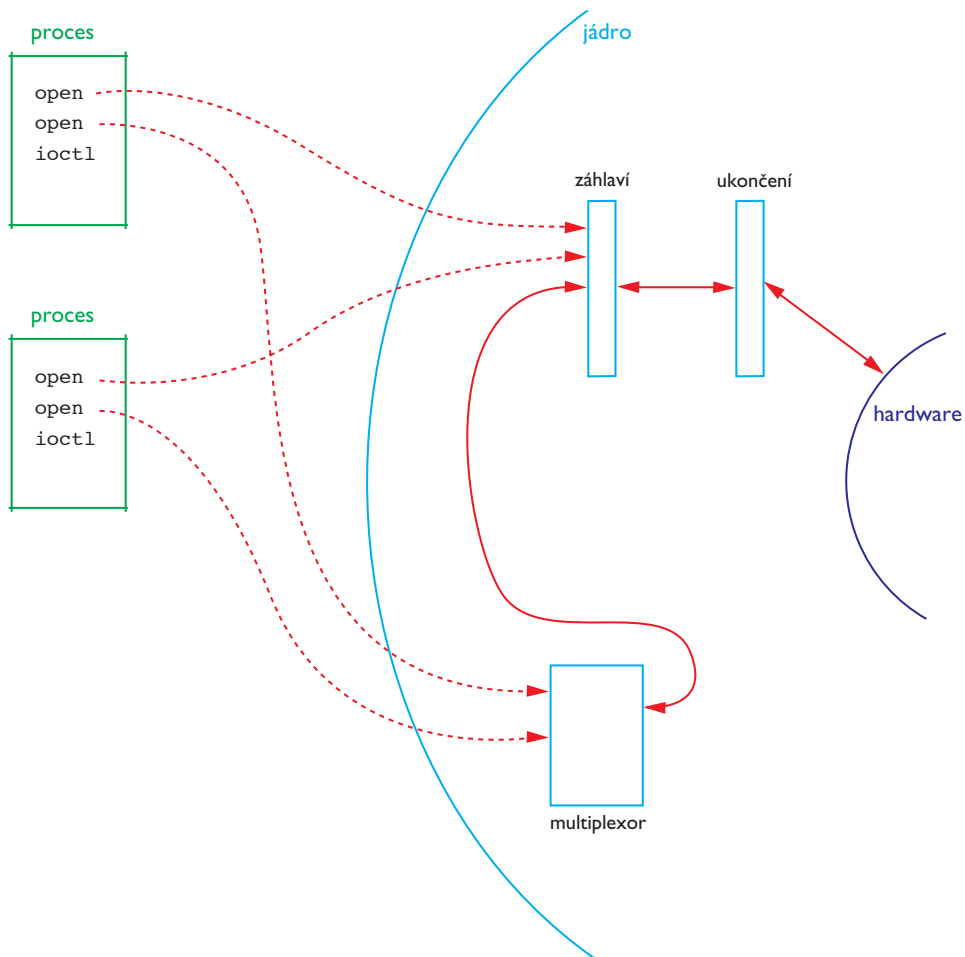
## 6.3 Tiskárna

Přestože je tisk méně a méně častý požadavek, stále je uživateli vyžadován a zdá se, že přežije rok 2000. Víceuživatelský operační systém UNIX podporuje uživatele frontováním jejich požadavků na tisk, pokud tiskárna jako pomalá výstupní periferie právě tiskne. Uživatelé používají k tisku obsahu souboru na sdílenou tiskárnu příkaz **lp**, který zařazuje požadavek na tisk do fronty odpovídající tiskárny. Tiskovou frontu čte démon **lp sched**, který její požadavky sekvenčně zapisuje na tiskárnu. Uživatel může příkazem **cancel** svůj požadavek na tisk z fronty vyjmout. Privilegovaný uživatel může vyjmout z fronty libovolný požadavek. Informativní příkaz je pak **lpstat**, který na obrazovku vypisuje obsah tiskové fronty. Fronty pro tisk jsou udržovány v adresáři `/var/spool/lp`. Uvedené příkazy definuje SVID, POSIX definuje pouze příkaz tisku **lp**.

Uživatelský proces **lp** převezme z parametru jméno souboru pro tisk, případně (použitím volby **-d**) označení tiskárny (systém může mít připojeno více tiskáren). Např.

```
$ lp -dlaser seznam
```

je požadavek tisku obsahu souboru `seznam` na tiskárnu se jménem `laser`<sup>3</sup>. Proces **lp** vytvoří kopii tištěného souboru ve `/var/spool/lp/request/laser/d4-561` (`d` označuje datovou část požadavku na tisk a číselná hodnota `4-561` jeho pořadí) a soubor `/var/spool/lp/request/laser/r4-561`, který je řídicí (obsahuje např. seznam souborů pro tisk, je-li jich více, počet kopií atd., tj. hodnoty voleb příkazu **lp**). Jméno adresáře pro tiskové úlohy



Obr. 6.9 PROUD, multiplexor a několik deskriptorů

(v příkladu **laser**) je specifické pro každou tiskárnu. Proces **lp** pak dále k binárnímu souboru `/var/spool/lp/outputq` připojí záznam a démonu **lp sched** pošle pojmenovanou rourou `/var/spool/lp/FIFO` informace o tiskovém požadavku (odkazem na řídicí soubor). Tím práce **lp** končí. Tiskový démon **lp sched** přebírá požadavky z pojmenované roury a postupně pro jednotlivé tiskové úlohy vytváří dětský proces, který se postará o samotný tisk. Jako rodič přitom čeká na dokončení práce dítěte, než převezme z roury další požadavek. Při hlubším studiu odhalíme existenci postupně dvou nově vzniklých dětí **lp sched**, z nichž první zpracovává řídicí soubor (`r4-561`) a jím vytvořené další dítě **lp sched** tiskne data, ovšem prostřednictvím opět

nově vytvořeného dalšího procesu, kterému říkáme proces rozhraní tiskárny (jde o filtr tištěných dat – často je to např. scénář pro shell, který je tiskárně přiřazen a umístěn v adresáři `/var/spool/lp/interface`). Teprve po ukončení druhého dítěte **lpsched** první dítě smaže řídicí soubor požadavku tisku (`r4-561`) a všechny datové soubory (pro nás pouze jeden, `d4-561`).

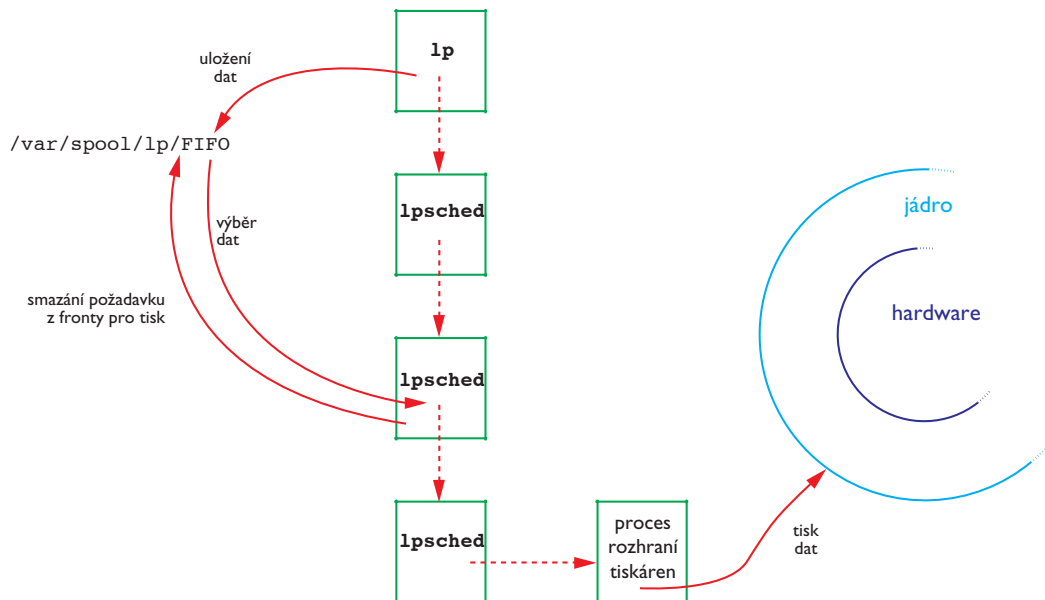
Systém tisku na jednu nebo více tiskáren obsluhovaných jedním jádrem (jedním operačním systémem) je v systémech kompatibilních se SYSTEM V zabezpečován uvedeným démonem **lpsched**. Správce systému má ve většině implementací také k dispozici několik příkazů pro nastavování a údržbu tiskových front. Je to zejména **accept** pro ohlášení obsluhy nové fronty démonu tisku a **reject** naopak o odpojení uživatele od této fronty. Příkaz **disable** odpojí frontu od tiskárny (např. je-li potřeba vyjmout zmačkaný papír) a **enable** tiskárnu k frontě opět připojí. Nastavování typů tiskárny, spojování tiskárny s frontami na tisk atd. poskytuje příkaz **lpadmin**. Konečně je možné také přesouvat požadavky z jedné fronty do druhé pomocí **lpmove**. **lpusers** je příkaz pro stanovení úrovní přístupu k tiskárně pro různé uživatele. Viz obr. 6.10.

Uvedený způsob práce procesů pro tisk je běžně používaný a je také rozšířen pro síťové služby (jako tiskový server), jak si uvedeme v čl. 7.4.3. Hojně je používán i způsob obsluhy tiskových front převzatý ze systémů BSD, který je možná pro účely síťového tiskového serveru i více rozšířen. Uživatel zařadí obsah souboru pro tisk do fronty pomocí příkazu **lpr**, tisk požadavků z fronty realizuje démon **lpd**. **lpq** je program prohlízející obsah tiskových front. Uživatel může také použít **lprm** k vyjmutí požadavku z fronty. Správu tiskového podsystemu podporuje příkaz **lpc**. Typy tiskáren a jejich vlastnosti (jako je počet řádků, počet sloupců, přenosová rychlost atd.) jsou ohlašovány v souboru `/etc/printcap` (je to obdoba dříve používaného `/etc/termcap` knihovnou CURSES pro popis různých typů terminálů).

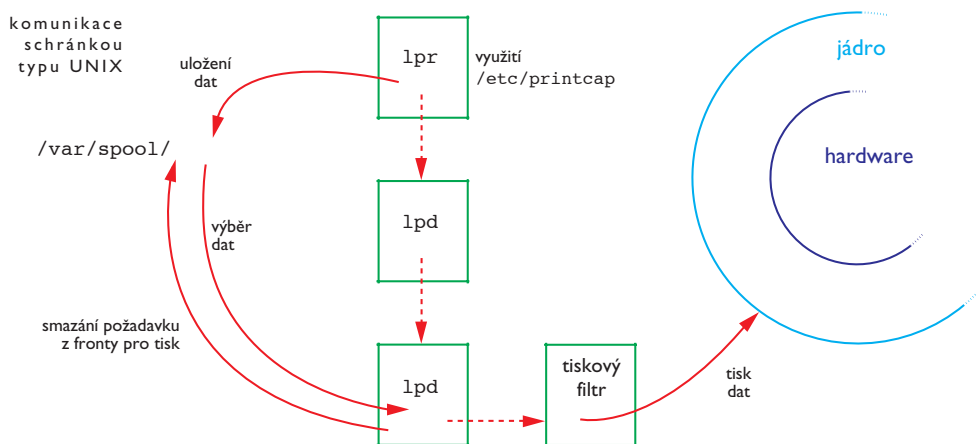
V souboru `/etc/printcap` je uveden i adresář tiskové fronty každé tiskárny, který začíná oblastí `/usr/spool`. Zde ukládané soubory mají identifikační první znak `c` pro řídicí soubor tiskové úlohy a `d` pro soubory s kopií obsahu souborů pro vlastní tisk. Po uložení dat pro tisk posílá **lpr** zprávu prostřednictvím schránky typu UNIX v IPC démonu **lpd**, což je oznámení požadavku tisku ve frontě, **lpr** pak končí svoji činnost. Podobně jako démon **lpsched** i **lpd** vytváří pro zajištění každé tiskové úlohy nové dítě **lpd**, které převezme řízení tiskové úlohy podle řídicího souboru. Rodič přitom i zde čeká na dokončení dítěte, a teprve pak pokračuje ve čtení dalšího požadavku ze schránky. Dítě **lpd**, které řídí tisk podle obsahu řídicího souboru, může tištěná data ještě filtrovat dalším procesem (řízeným např. programem **pr**), který případně vytvoří (informace o takovém filtru je zavedena podle volby příkazu **lpr** v obsahu řídicího souboru). Pro spojení mezi dítětem **lpd** a programem filtru, tj. pro předání tiskových dat, se používá obyčejná roura. Situaci ukazuje obr. 6.11.

Z pohledu přístupu k perifériím u řízení přístupu k tiskárnám zde však není žádný nový prvek oproti běžnému chování ovladače v jádru, zpřístupňovaného speciálním souborem. Z periferie tiskárny nelze pouze data číst, ale to je naopak zjednodušující vlastnost.

Uvedené způsoby zajištění obsluhy front na tiskárnu nejsou v UNIXu jediné, přestože jsou nejvíce používané. Např. AIX používá pro podsystem obsluhy tisku tzv. obsluhu front (Queuing Facility), což je obecná podpora mechanismu front. Pro tiskárny je využíván příkazy **qprt** (tisk) a **enq** (zrušení tisku), démon má jméno **qdaemon**. I tak ale AIX podporuje i oba dříve uvedené systémy tisku pro SYSTEM V i BSD.



Obr. 6.10 Přístup uživatele k tiskárně



Obr. 6.11 Tiskárna v systémech BSD

## 6.4 Ostatní periferie

Rozdělení periferií na znakové a blokové určuje jejich používání. Disk může být používán připojováním jeho sekcí jako svazků k systému souborů nebo pro archiv dat se sekvenčním přístupem (např. programem **tar**). Vzhledem k tomu, že diskový přístup je tentýž pro rychlé vnitřní disky i pro disketové mechaniky nebo výměnné magnetooptické disky atd., je věcí správce systému, jakým způsobem bude periferie používat. I magnetická páska, přestože je používána zejména sekvenčně (opět programem **tar** nebo **cpio**), může být přístupná po blocích, protože podsystém vstupu a výstupu, jak jsme jej v této kapitole popsali, to umožňuje.

U magnetických pásek ovšem můžeme uvažovat o jejich zániku daleko vážněji, než jsme to naznačili u tiskáren. Výměnné velkokapacitní disky umožňují pohodlnější a hlavně rychlý přístup k archivům dat nebo jejich přenos mezi počítači. Pro zálohování dat se pak stále častěji používá systém zdvojeného zápisu dat (tzv. mirroring) na provozní disková média (viz kap. 3) a archivy jsou vypalovány na CD, nebo jsou používány kapacitně výkonnější mechaniky DVD). Přesto nelze vývoj magnetických pásek považovat za uzavřený, jak naznačují snahy tento typ periferie posílit jeho vyšší přístupovou rychlostí (např. v technologii DLT, digital linear tape ).

Pro archivy dat jsou také používány zásobníky disků (nebo diskových polí) na bázi magnetooptické technologie, tzv. jukebox WORM – write-once-read-many (jednou zapsáno, mnohokrát čteno), kdy jsou nosná média vybavena výrazně rychlejším přístupem pro čtení než pro zápis; jejich kapacita je dnes přitom počítána ve stovkách GB. Jejich využití pro archivní účely je také typické pro vyvíjené technologie obsluhy síťových datových svazků, jak uvidíme v kap. 11.

Ještě dále se ubírá tzv. systém hierarchického zálohování. Jedná se totiž o kombinace disků, magnetooptických disků a magnetických pásek. Jeho ovládání je podmíněno přítomností speciálního softwaru, který organizuje přístup k datům. Méně používaná data jsou totiž postupně znevýhodňována tak, že jsou automatizovaně přesouvána na pomalejší (ale kapacitně výkonnější) média (magnetooptické disky, pásky). Takové technologie jsou dnes nabízeny prakticky každým významným výrobcem a jejich použití je především v ostrém provozu s nutností dobrého zabezpečení nepřetržitě používané (síťové) datové základny.

<sup>1</sup> Výrobci aplikací někdy nevyužívají organizace svazku. Např. datové části databáze jsou umístovány v sekcích znakových speciálních souborů podle organizace procesů realizujících chod databáze. Organizace dat sekce disku je navržena a implementována výrobcem aplikace. Lze tak dosáhnout vyššího výkonu aplikace. Data pak nejsou viditelná připojením sekce k systému souborů UNIXu. Podobně jsou někdy implementovány systémy souborů jiných operačních systémů, které lze připojit jen obtížně nebo vůbec v kontextu hierarchické struktury systému souborů UNIXu. Přístup k datům takových sekcí disků pak opět zpřístupňují zvláštní procesy, které obsahům znakových speciálních souborů těchto sekcí rozumí.

<sup>2</sup> Otevření každého znakového speciálního souboru nemusí znamenat přiřazení PROUDU. V tabulce `cdevsw` seznamu funkcí ovladače musí být aktivace PROUDŮ nastavena. Pokud není, průchod dat jádrem je zajištěn pouze tak, jak bylo popsáno dosud.

<sup>3</sup> Uživatel může ještě před tiskem použít filtr **pr** pro úpravu tištěného textu. **pr** rozdělí obsah souboru pro tisk na stránky a každou obohatí o záhlaví a patu informacemi čísla stránek, jména souboru, data, času úpravy atd. Náš příklad lze pak psát v koloně

\$ **pr seznam | lpr -dlaser**

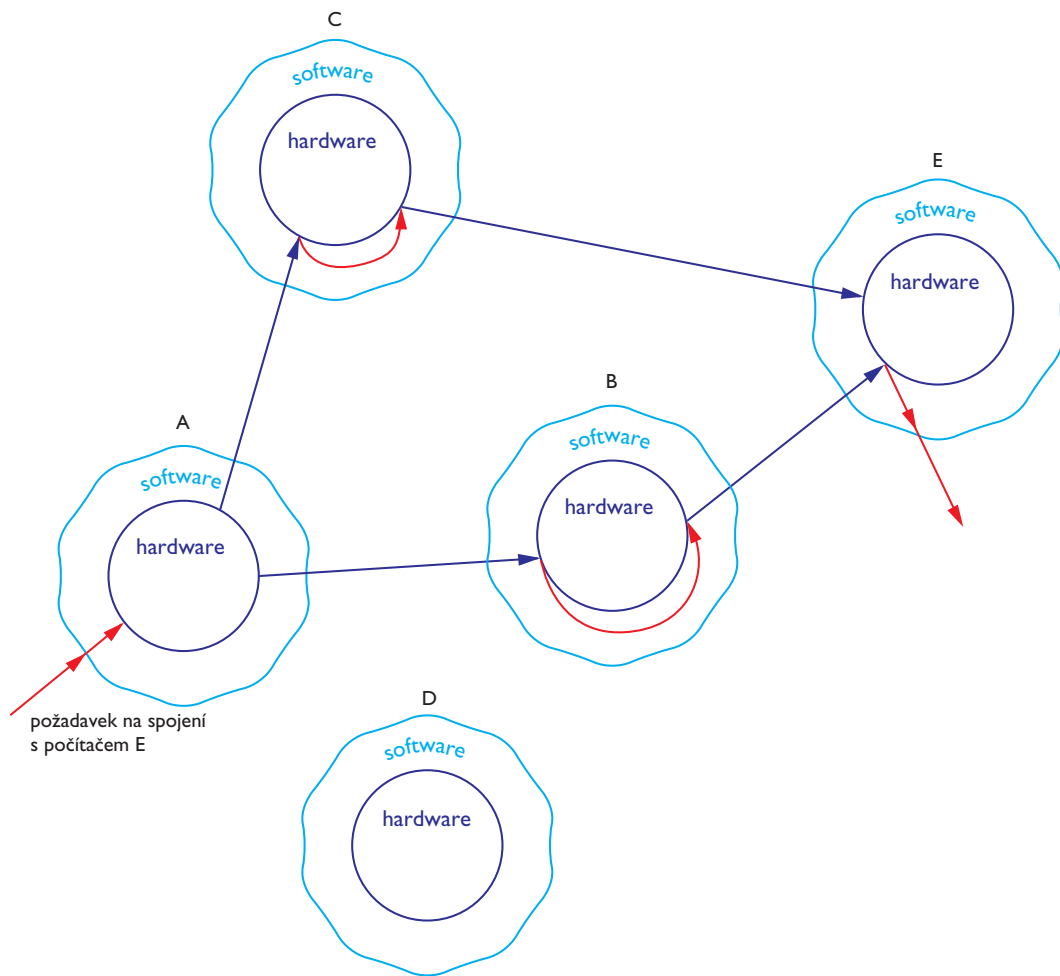




## 7 SÍTĚ

Při využívání několika nezávislých výpočetních systémů se brzy objeví požadavek přístupu k datům (a jejich přenosu) mezi jednotlivými operačními systémy navzájem. Přenos dat na výměnných médiích je sice možný a používaný, ale pro zpracování dat různých výpočetních systémů současně se časem stává organizačně neúnosný. V průběhu posledního desetiletí, s nárůstem využívání většího počtu počítačů je stále více patrná potřeba elektronického propojování počítačů mezi sebou. Sítové služby požadované uživatelem přitom diktovaly vývoj technického řešení sítí. Uživatel požadoval nejprve zpřístupnění práce ve vzdáleném systému (remote system), tj. možnost vzdáleného přihlášení (remote login) neboli podpory emulace terminálu (alfanumerického nebo grafického) ve vzdáleném operačním systému. Ihned ovšem přibyla potřeba data ze *vzdáleného* (remote) systému přenášet do *místního* (local) a naopak. Postupně se objevovaly další možnosti a požadavky, především posílání elektronické pošty (electronic mail, email) uživatelům registrovaným ve vzdálených systémech, využívání vzdálených tiskáren, kreslicích a jiných zařízení, spouštění úloh ve vzdálených systémech (remote execution) atd. Současný trend podpory práce uživatele v sítovém spojení je tzv. distribuované zpracování dat, tj. podpora přímého sdílení dat souborů v různých systémech, v UNIXu nazývaná NFS (Network File System) nebo RFS (Remote File Sharing). Od možnosti připojit vzdáleným systémem nabízenou část diskového prostoru jako svazek je již jen krůček k zobecnění tohoto principu na libovolnou periferii a k nabídce a využití libovolného výpočetního zdroje prostřednictvím sítě. Dnešní vývoj pak dále nasvědčuje pojmání sítě jako množiny výpočetních zdrojů, uživateli poskytované po přihlášení do libovolného systému, který je součástí takové sítě. Při takto využívaných výpočetních zdrojích celé sítě začínáme hovořit o sítových operačních systémech, které uvažují hardware množiny počítačů jako celek a jako celek jej uživateli zpřístupňují. Architektura dnešních v praxi používaných operačních systémů však většinou nevyhovuje takovému pojetí, přestože je termínem sítový označován každý operační systém, který podporuje třeba jen některé sítové služby. UNIX v tomto kontextu stojí na hranici svého uplatnění, protože jeho architektura sice umožňuje poskytovat prakticky všechny dnes známé sítové služby, ale síť jako celek nechápe. Pokračováním UNIXu (opět v kontextu standardu POSIX) je dnes např. operační systém Plan 9, jehož stručný popis si uvedeme v kap. 11 a který je koncipován od svého vzniku jako sítový.

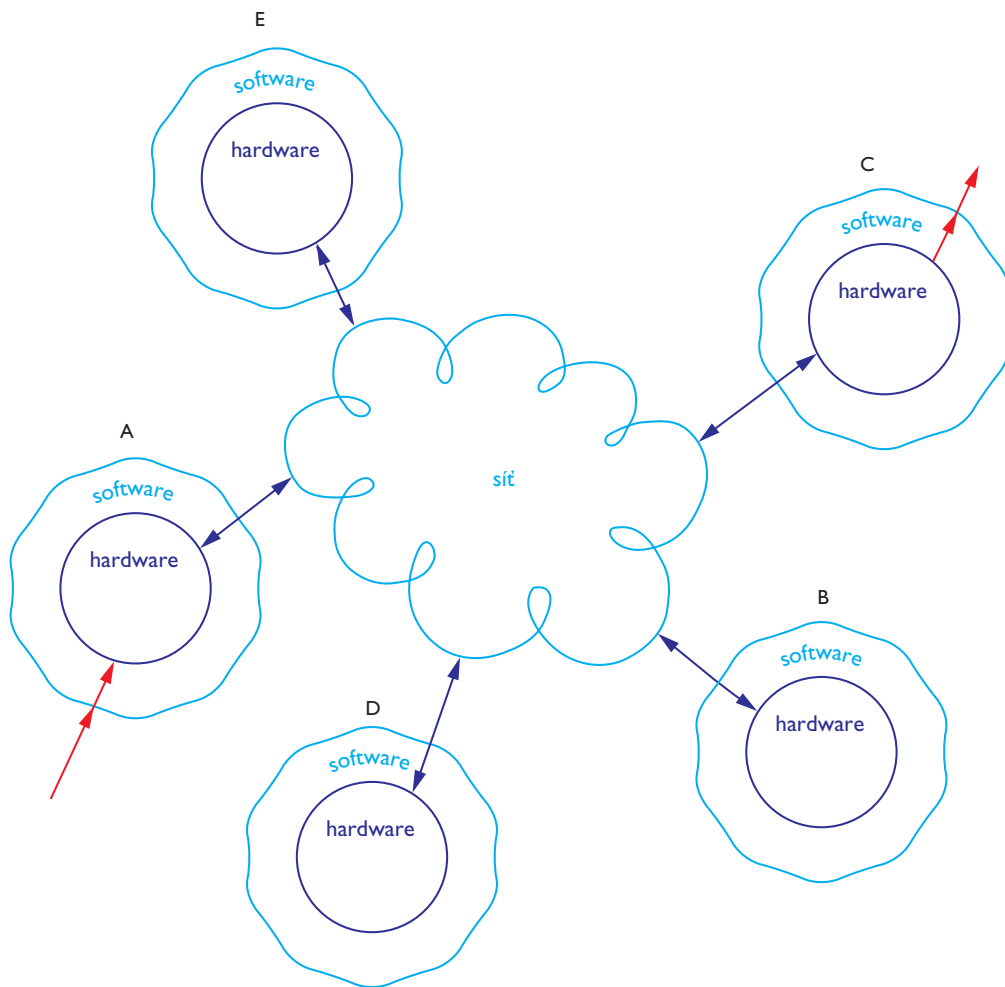
Vytvořit sítové spojení mezi několika počítači představuje nejprve propojit je prostřednictvím za tímto účelem pracující hardware, který musí být na každém z počítačů obsluhován sítovým softwarem. Technické řešení spojení několika počítačů v rámci určité lokality, jako je např. budova nebo areál firmy, může být principiálně různé, ale vždy realizuje rychlý přenos dat mezi počítači navzájem. Z pohledu našeho výkladu tedy pravděpodobně půjde o periferii, nad kterou lze používat volání jádra `read` a `write` jako čtení a zápis dat na vzdáleném počítači. Hardwarem může být např. (u nás nejvíce používaný) Ethernet. Ethernet je periferie, která fyzicky znamená elektronickou desku připojenou na přenosový hardware vnitřního toku dat počítače (např. sběrnice), stejně jako je tomu u disků nebo operační paměti. Ethernet přitom podobně jako jiné typy sítového hardwaru umožňuje používat kabelové spojení typu zúčastněných počítačů za sebou (tzv. lineární spojení) nebo navzájem mezi sebou (hvězdicové spojení). Lineární spojení znamená, že kabelové spojení vede jako jedna linka od počítače k počítači a každý počítač sítě má tak fyzicky svého souseda. Hvězda je spojení všech počítačů do jednoho bodu, aktivního sítového prvku, takže fyzicky sousední počítač je zde ten, který je právě vhodný. Z pohledu



Obr. 7.1 Síť několika počítačů

síťového softwaru je však tato struktura nedůležitá. Software totiž vymezuje ovládání hardwaru na úrovni ovladačů a v dalším abstrahuje síť jako volně adresovatelné počítače, se kterými lze navázat spojení a přenášet data, jak je nakresleno na obr. 7.1.

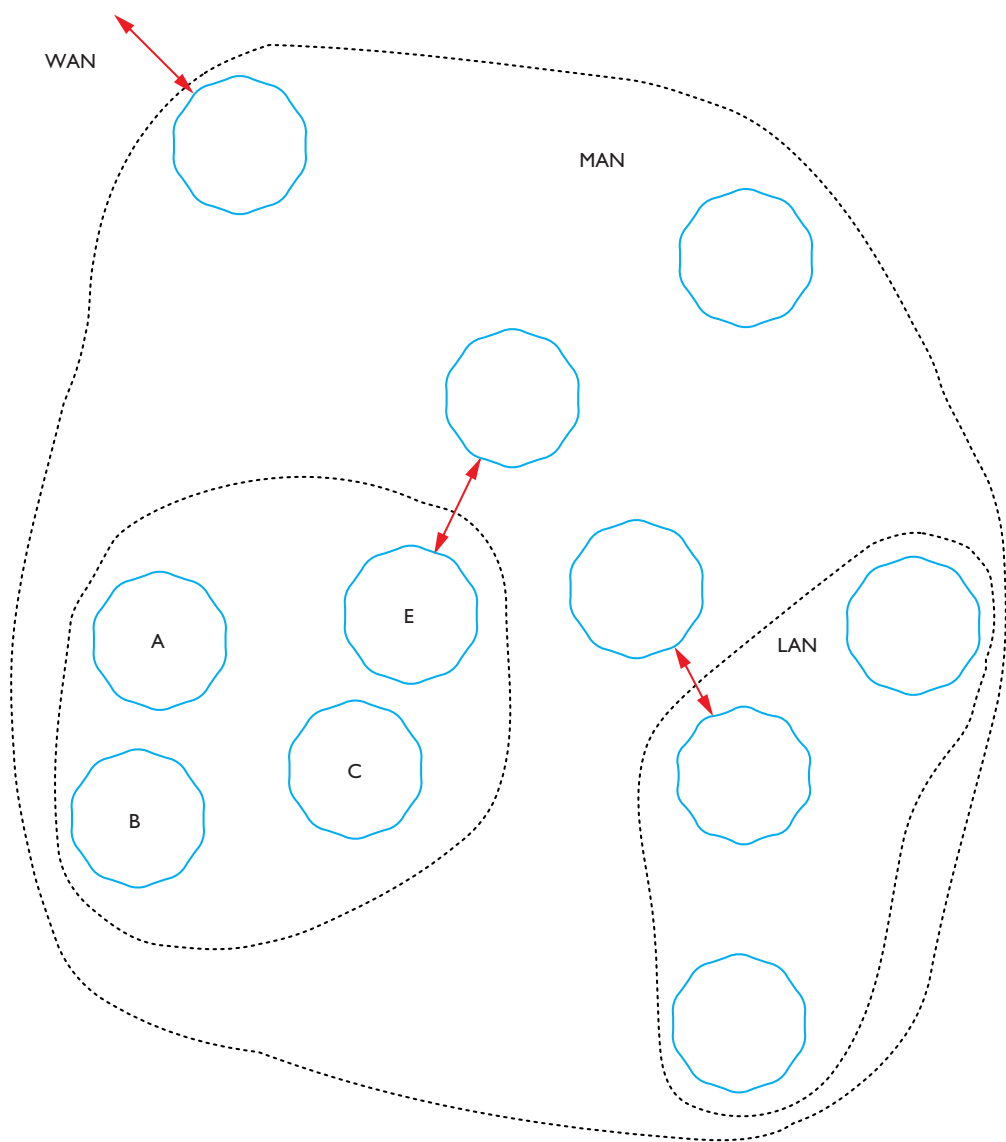
Na obrázku jsou uvedeny dvě možné cesty realizace požadavku na spojení počítače A s počítačem E. Je-li E nedostupný přímo (hardwarovou konfigurací), může cesta procházet uzlem B nebo C. Přestože adresace uzlů zůstává nezměněna, přenos dat může probíhat různými cestami přes uzly sítě. To je zajištěno síťovým softwarem každého počítače, který vždy dokáže prozkoumat data zapsaná některým



Obr. 7.2 Počítač jako účastník sítě

z počítačů do sítě, zjistit, zda mu patří a převzít si je nebo je odeslat známou cestou dalšímu počítači v pořadí. Sítový software se přitom může orientovat podle pevně zadaných cest sítě nebo může používat dynamicky se měnící vytížení sítě a podle toho posílání dat přizpůsobuje. Pro aplikaci na určitém počítači je pak síť viditelná podle obr. 7.2.

Počítači, který je součástí sítě, říkáme *uzel* nebo angl. *host* (hostitel). Uvedený způsob spojování několika počítačů mezi sebou nazýváme místní (nebo lokální) počítačová síť (LAN, Local Area Network) a slouží pro potřeby určitého okruhu navzájem komunikujících uživatelů jednotlivých uzlů.

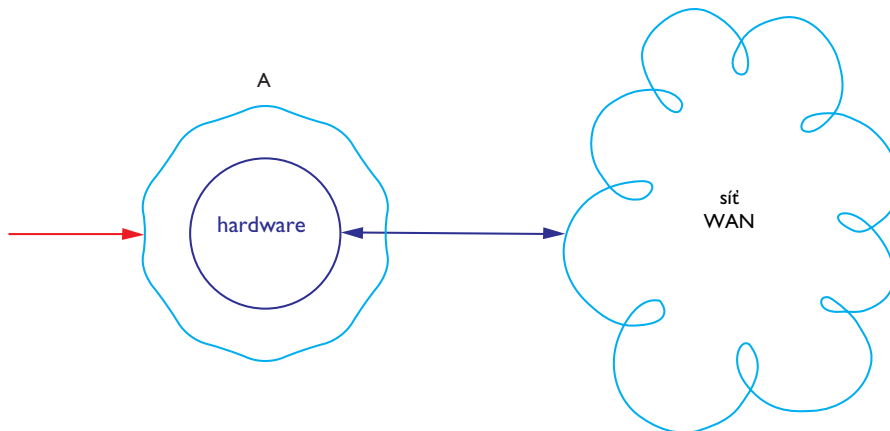


Obr. 7.3 Internetworking

Dnes známá technologie spojování místních sítí mezi sebou se nazývá internetworking nebo internet (odtud výraz současné celosvětové síť Internet). Vznikají tak rozsáhlejší síť celých měst nebo regionů, kterým říkáme metropolitní síť (MAN, Metropolitan Area Network), jejichž technické řešení je dnes obvykle na bázi optického vlákna. Technologie internetworking vychází z funkce některého z uzlů každé LAN jako spojovacího článku s metropolitní sítí, viz obr. 7.3.

Uzel E na obr. 7.3 je takový spojovací článek. MAN je opět určitá síť několika počítačů, které pracují pro vzájemné propojení jednotlivých LAN. Spojovací počítač tedy realizuje propojení dvou sítí, LAN a MAN. Podle způsobu, který spojovací počítač používá, je toto připojení označováno termínem *směrovač* (router) nebo *brána* (gateway). Funkce je stejná, ale brána je spojení dvou sítí, jejichž síťový software je jinak koncipován, např. spojení typu TCP/IP a BITNET, kdežto směrovač je spojení dvou sítí téhož síťového softwaru (např. pouze TCP/IP). Brána tedy musí např. převádět data do jiných formátů, transformovat označování dat, měnit způsoby adresace uzlů atd., kdežto směrovač pouze převádí data z jedné sítě do druhé<sup>1</sup>. Na úrovni hardwaru nebo jeho přímého ovládání je možné ještě používat spojování sítí, kterému říkáme *opakovač* (repeater), kdy jde o kopie toku elektrické informace. Např. je-li LAN technicky rozložena do několika hvězd nebo několika lineárních spojení (tzv. segmentů), do jedné sítě ji sjednocuje opakovač (ale také směrovač). Na úrovni přímého ovládání hardwaru je také používán způsob nazývaný *most* (bridge), kde jde o kopírování *rámců* (frames) mezi sítěmi. V případě opakovačů i mostů jde více o úroveň technické realizace než o softwarové řešení, pro které je pak spojení dvou sítí transparentní a celek se chová jako např. jedna LAN.

Funkci směrovače plní vybraný počítač sítě, který může pracovat i pro jiné účely. Software realizující směrovač je přitom instalován jako ovládání dvou (nebo více) síťových hardwarů. Každý pracuje pro jednu ze spojovaných sítí. Každý další uzel sítě LAN má nastaven jako směrovač počítač, kterým vede cesta mimo známé adresy ostatních uzlů LAN. Přestože směrovačů může být v síti více (a MAN jich více mít musí), pro LAN panuje obecná snaha realizovat připojení vždy jednou cestou.



Obr. 7.4 Uzel a síť

Na obr. 7.3 je také uveden termín WAN, kterým pokračuje další spojení. Rozsáhlé počítačové sítě (Wide Area Network) je označení pro spojení různých sítí MAN i LAN navzájem mezi sebou. Technicky se může takové propojení realizovat vysokorychlostním modelem nebo satelitním spojením nebo i optickým kabelem. Softwarově jde ale rovněž o totéž: určení směrovače nebo brány, kterým spojení pokračuje dál mimo známé adresace uzlů LAN i MAN. Znamená to, že uzel A lokální síť z obr. 7.2 může při realizaci potřebných spojovacích členů oslovit adresací kterýkoliv uzel MAN nebo WAN. Síť se pro něj tak stává zcela transparentní, jak ukazuje obr. 7.4.

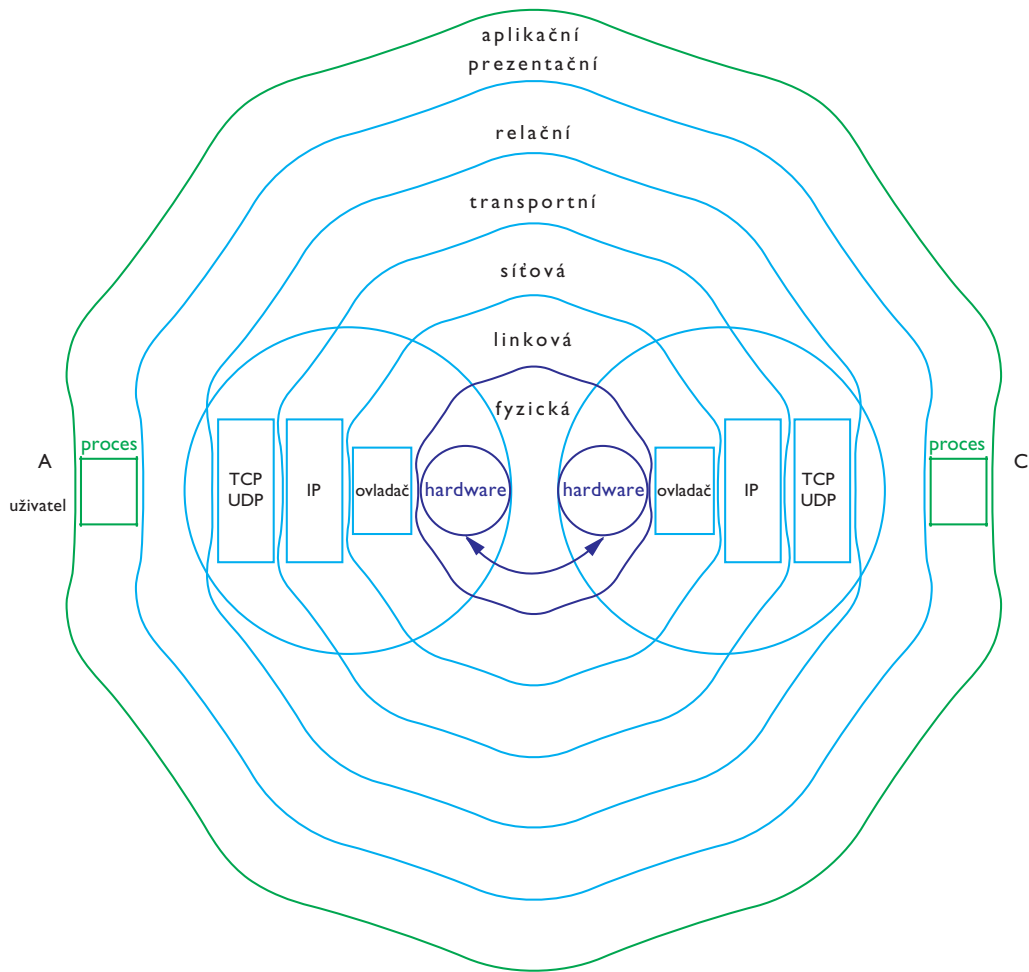
Počítač, přestože je připojen k síti, ji nemusí využívat neustále. Teprve na základě požadavku uživatele dochází k aktivaci síťového softwaru a následně hardwaru uzlu. Při požadavku síťového přenosu dat tak musí nejprve proběhnout oslovení protilehlého uzlu. Říkáme, že dochází k realizaci *spojení* (connection). Je-li spojení *ustanoveno* (established), proběhne přenos dat, po kterém je spojení zase *ukončeno* (terminated), a uzel se stává opět pasivním účastníkem sítě. Jemnější terminologie pak ještě rozlišuje výraz pro *přenos se stálým spojením* (connection-oriented, např. u realizace vzdáleného terminálu), tj. kdy je nutné udržovat trvalý oboustranný tok dat mezi uzly, a přenos *bez stálého spojení* (connectionless, např. u sdílení disků), kdy nejde o tok dat, ale pouze o občasné předávání zpráv. V obou případech je však princip stále zachován.

vrstva č.	pojmenování
7	aplikační (Application)
6	prezentační (Presentation)
5	relační (Session)
4	transportní (Transport)
3	síťová (Network)
2	linková (Data Link)
1	fyzická (Physical)

Obr. 7.5 Model OSI v sedmi vrstvách

Síťové spojení má vrstvenou architekturu. Podobně jako je vrstveně budován celý operační systém (viz kap. 1), také implementaci přenosu dat sítí je doporučováno řešit ve vrstvách. Návod, jak vrstvy rozdělit a přidělit jim odpovídající funkce, řeší standard ISO (International Standards Organization) pro počítačové sítě OSI (Open System Interconnection), viz [ISOOSI], a dnes je tímto dokumentem každé síťové spojení poměřováno. Model OSI má 7 vrstev, jak ukazuje obr. 7.5.

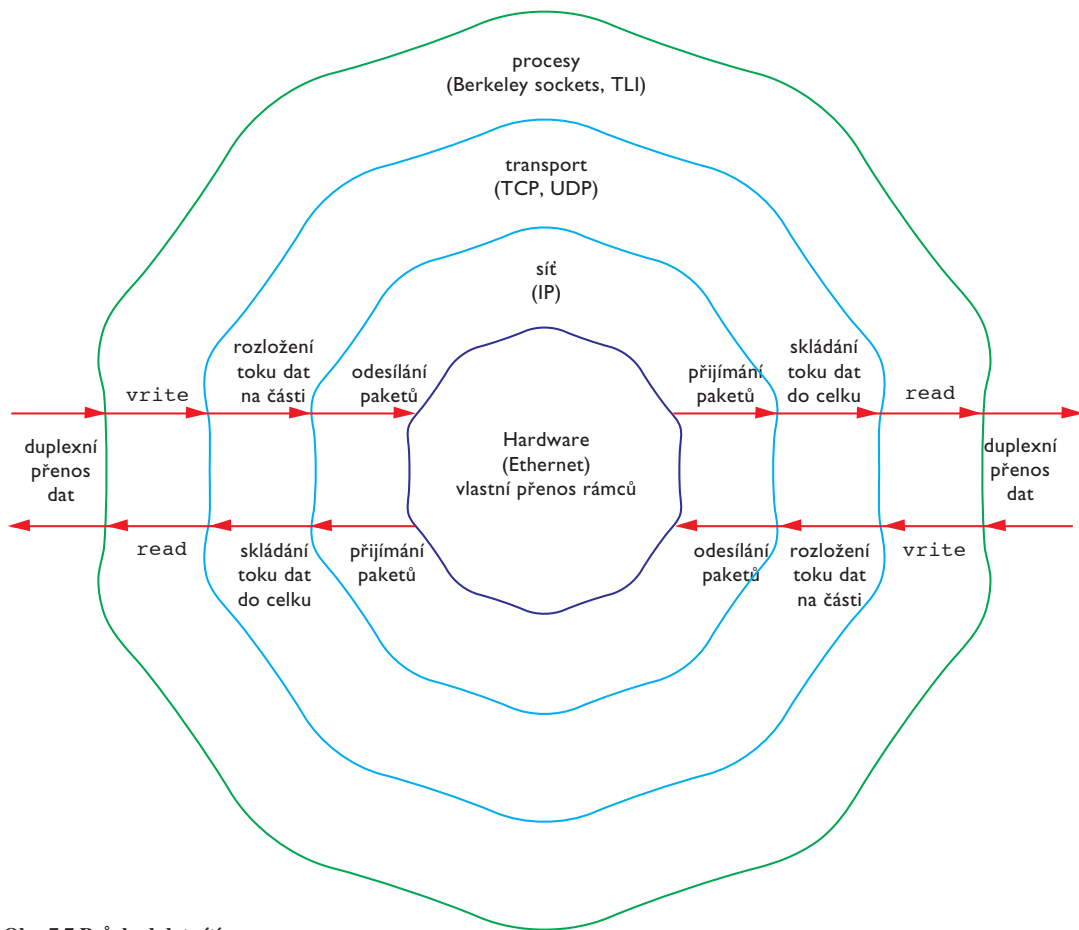
Jednotlivé vrstvy znamenají postupné vzdalování se od hardwaru výpočetního systému. Vrstva *fyzická* je hardware, *linková* jeho přímé ovládání. Vrstvy *síťová* a *transportní* jsou věnovány přenosovým protokolům. Síťová rozeznává síť, tj. zajišťuje přístup uzlů k síti ve smyslu obr. 7.4. Transportní zajišťuje



Obr. 7.6 Síťové vrstvy OSI v UNIXu

přenos toku dat, a to ve srozumitelných formátech. *Relační* vrstva je věnována programovací úrovni. Jedná se o síťové programovací jazyky a jejich rozhraní na přenosové protokoly. Vrstva *prezentační* provádí optimalizaci a kódování dat, *aplikační* je zpřístupnění sítě uživatelům v síťových aplikacích. Na obr. 7.6 přikládáme model OSI struktury operačního systému UNIX.

Nejnižší 4 vrstvy jsou součástí jádra. Jako příklad jsme uvedli skupinu nejpoužívanějších přenosových protokolů TCP/IP, ale (vzpomeneme-li na PROUDY z kap. 6) používat lze i protokoly jiné. Pro relační vrstvu je v prvním přiblížení zřejmá skupina volání jádra pro síť (Berkeley sockets nebo knihovna TLI využívající volání jádra pro ovládání PROUDŮ), ale patří sem např. i programovací jazyk RPC (Remote



Obr. 7.7 Průchod dat sítě

Precedure Calls), případně další programovací jazyky (např. JAVA). Vrstva prezentační a aplikační je v UNIXu úroveň procesů, které mohou plnit různou funkci (síťové demony nebo instance aplikací, jako je **telnet**, **ftp** atp.). V textu kapitoly se budeme jednotlivým částem UNIXu v odpovídajících vrstvách věnovat podrobně.

Pro správnou představu o síti je důležité vidět průchod dat uživatele sítě mezi dvěma uzly jako postupné klesání dat do jednotlivých vrstev a v cílovém uzlu opět vystupování dat od vrstvy hardwarové až po aplikační, jak ukazujeme na obr. 7.7.

Na obrázku využíváme zjednodušeného síťového modelu o čtyřech vrstvách, kterým se v UNIXu síť lépe ukazuje (vrstvy jsou pojmenovány *procesová*, *transportní*, *síťová* a *linková*)<sup>2</sup>. Rozhraní mezi



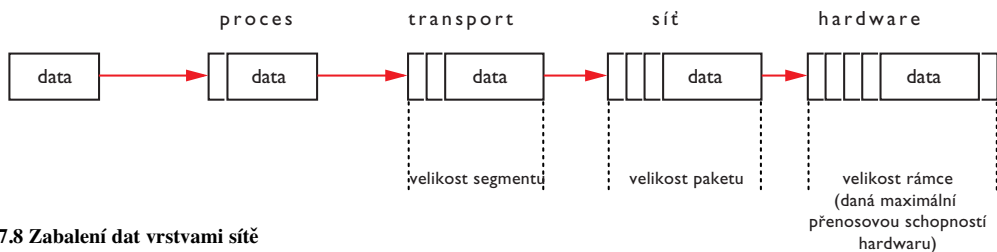
procesem a jádrem (volání jádra) je rozhraní procesové a transportní vrstvy. Sítová vrstva je reprezentována protokolem IP, který byl vyvinut zvláště pro spojování uzlů a sítí. Proces určité sítové aplikace zapisuje data do jádra jako sekvenci bytů a opačně z kanálu (deskriptoru z tabulky procesu) přiděleného sítovému spojení data ze sítě čte. Jádro požadavek zápisu sekvence dat do sítě rozdělí na části, které postupně odesílá. Za souvislost dat odpovídá např. protokol TCP, protože od sítové vrstvy mohou data přicházet v jiném pořadí, než byla odeslána. Velikost části dat, která může cestovat sítí, je obvykle dána typem hardwaru v linkové vrstvě (např. Ethernet přenáší data po 1500 slabikách, síť token ring přibližně 4500, ale u sériových linek je to třeba jen 128 bytů), která data přijímá od sítové vrstvy a sítová vrstva se postará o takovou odpovídající velikost dat. Část dat na úrovni transportní vrstvy nazýváme *segment* nebo *datagram*. Data, která sítová vrstva (protokolu IP) předává hardwaru, nazýváme *paket*, a data, která pak fyzicky cestují sítí, *rámec* (frame). Principiálně jsou však data vždy rozložena na části (v tzv. fragmentaci) a po přenosu zpětně zkompletována. Proces ve výsledku pracuje s nepřerušovaným tokem dat a o fragmentaci se nestará. Dolní (jádrem počínaje) vrstvy sítového přenosu části dat (segmenty, pakety atd.) nutně označují, tj. k vlastním datům připojují, potřebná označení. Říkáme, že přenosové protokoly data zabalí nebo zapouzdří (od angl. encapsulation). Jak ukazuje obr. 7.8, data jsou v jednotlivých vrstvách obohacována hlavičkami, případně ukončením (na obr. 7.8 u hardwaru).

Na obr. 7.8 je situace zjednodušena, protože velikost paketu a segmentu se může lišit. Pokud sítová vrstva obdrží od transportní větší datagram, než je schopna najednou zabalit do paketu, vytvoří z něj paketů několik.

Data jsou tedy rozložena, zabalena, přenesena a v cílovém uzlu v jednotlivých vrstvách zpětně rozbalena a složena. Proces je pak čte a i on může část dat považovat za své interní informace, podle nichž pak data uživateli prezentuje do aplikační vrstvy. Je také zřejmé, že moduly jádra skupiny protokolů TCP/IP jsou používány pro více sítových spojení současně, jak již bylo uvedeno v kap. 6 u PROUDŮ (multiple-xování).

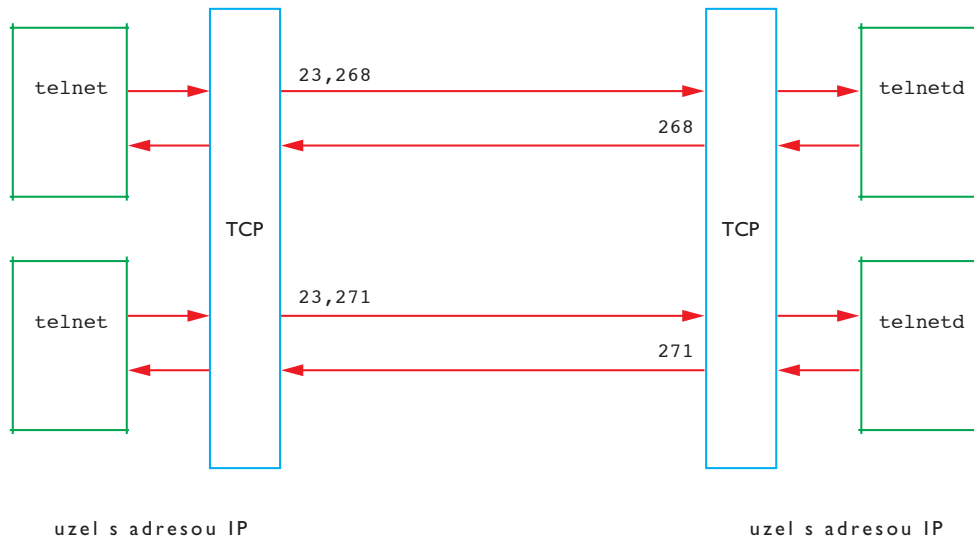
Data v síti TCP/IP jsou směrována principem daným protokolem IP. Sítová vrstva využívá přidělení adresy IP každému uzlu v síti. Jednotlivé síť pak mají odlišení sítovou adresou IP. Prakticky je adresa uzlu i síť sdružena do jednoho (dnes) 32 bitového celého čísla a má označení pouze adresa IP (Internet Protocol address). Použije-li proto aplikační vrstva označení uzlu adresou IP, je uzel jednoznačně rozpoznatelný v rámci třeba celosvětové sítě TCP/IP (což je např. Internet). Adresa IP je proto přidělována uzlu uvážlivě. Pokud místní síť neuvažuje o připojení na větší sítové lokality, může správce sítě přidělit adresy IP uzlům podle svého rozmaru, jinak však správce musí respektovat označování větších sítí a o přidělení sítové části adresy IP žádat odpovídající správce větší sítě. Sítová vrstva každého uzlu má tak za úkol rozpoznat, zda je příchozí paket určen pro její uzel, a pokud ne, který uzel je zodpovědný za její odeslání (tj. směrovač) a jemu paket přesměrovat.

Z uvedeného vyplývá, že implementace sítového přenosu v UNIXu je založena na komunikaci dvou procesů, které běží v různých operačních systémech (uzlech), jak jsme také uvedli u popisu obr. 4.2. K tomu, aby bylo možné procesy jednoznačně identifikovat v rámci celé sítě, byl pro sítové spojení zaveden termín *port*. Číslo portu je abstraktní celočíselná 16-bitová dekadická hodnota, která označuje typ sítové aplikace. Číslo portů zpracovává (tj. přiděluje a rozlišuje) transportní vrstva. Použije-li např. uživatel sítovou aplikaci **telnet** pro přihlášení ve vzdáleném uzlu, je odpovídající proces vytvořen se smluveným číslem portu 23 a v uzlu, který je pro uživatele cílový, je aktivován proces, který spojení podle čísla portu zajišťuje (v našem případě proces **telnetd**). Identifikace procesu **telnetd** je tedy



Obr. 7.8 Zabalení dat vrstvami sítě

podle adresy IP a čísla portu 23. Číslo portů rozlišujeme na rezervovaná, dočasná (efemérní) a volná. Rezervované číslo portu je např. 23, protože je jím určena aplikace **telnet**. Přesněji řečeno, je jím určen démon **telnetd**, který se rozběhne v cílovém uzlu. Aplikace **telnet**, která síťové spojení vyvolá, se jím prokazuje pro svou identifikaci. Výchozí uzel (vrstva transportní, TCP) při vytváření spojení také dynamicky přidělí vznikajícímu spojení další, tzv. dočasné číslo portu, kterým se bude naopak prokazovat proces **telnetd** při zasílání dat zpět do výchozího uzlu. Použije-li další uživatel téhož výchozího uzlu ještě v průběhu takového spojení aplikaci **telnet** do stejného cílového uzlu, prokazuje se tímž rezervovaným číslem portu (23), ale výchozí systém přidělí novému spojení jiné dočasné číslo portu, kterým se bude prokazovat nový proces **telnetd** z cílového uzlu. Situaci ukazuje obr. 7.9.



Obr. 7.9 Číslo portu v příkladu dvou spojení telnet do téhož uzlu

Do cílového bodu transportu daného adresou IP protokol TCP posílá při navazování spojení data s označením 23 (rezervované číslo portu) a upozorněním přijímat data s označením 268 (dočasně přidělené číslo portu pro toto spojení). Cílový uzel podle 23 startuje **telnetd** a předává mu číslo portu 268 a adresu IP uzlu, ve kterém na něj čeká proces **telnet**. Spojení pak probíhá za znalosti obou adres IP a portu 268 na obou stranách spojení. V síti TCP/IP je tak spojení jednoznačně určeno tzv. *asociací*, tj. pěticí hodnot určující spojení dvou procesů v síti:

{protokol, místní adresa, port, vzdálená adresa, port}

např. {tcp, 147.229.112.10, 23, 147.229.17.10, 268} pro příklad z obr. 7.9 (adresy IP jsou uvedeny v tzv. tečkové podobě a jejich význam bude objasněn v následujícím čl. 7.1).

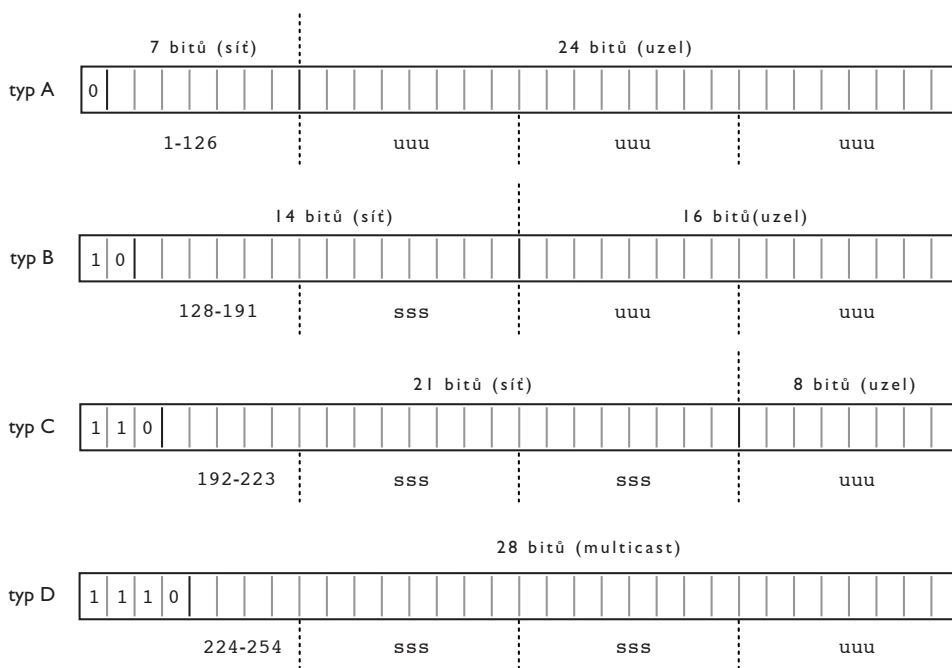
Rezervovaná čísla portů jsou definována v rozsahu 1 – 255. Některé systémy rezervují čísla portů až do 1023. V rozsahu nad 255 jsou ale i tak používány porty dočasné (transportní vrstva pak rezervované porty v oblasti nad 255 obchází podle definic v systémových tabulkách). Všechny známé porty síťových aplikací typu Internet jsou definovány v oblasti do 255. Efemérní čísla portů systém přiřazuje v rozmezí od 1024 do 5000. Hodnoty nad 5000 jsou volné a mohou je používat procesy v rámci dalších síťových aktivit.

Síťové spojení zajištěné jádrem používá procesová vrstva v technologii klient - server. Uvedli jsme její princip v kap. 4, kde jsme se také zmínili o jejím použití v síťovém spojení. Jak také vyplývá z právě uvedeného, síť je v UNIXu realizována komunikací procesů, které běží v různých uzlech. Znamená to, že proces, který aktivuje spojení, je klientem žádajícím o síťovou službu (**telnet**), kterou v cílovém uzlu zajistí proces server (**telnetd**). Ten s klientem komunikuje a plní jeho požadavky. Princip aktivace odpovídajícího procesu serveru je dán systémem rezervovaných portů, pro které je vždy odpovídající server startován. Typy serverů tak vyjadřují síťové služby, které uzel do sítě nabízí a které mohou být pro dva uzly sítě symetrické nebo nikoliv (správci systému nemůže nikdo zabránit odpojení služby **telnetd**, což znemožní používat uzel pro přihlašování ze vzdálených výpočetních systémů).

O startu prací na sítích v UNIXu můžeme hovořit od první poloviny 80. let, a to jednak v systémech BSD a jednak na půdě samotné AT&T. Součástí týmu pro vývoj systémů BSD byla skupina odborníků s označením BBN (Bolt, Beraneck, Newman), která iniciovala práce na implementaci síťového protokolu IP a přenosového protokolu TCP. Vrstva procesová se pak projevila rozšířením o nová volání jádra, souhrnně nazývaná Berkeley sockets (schránky typu Berkeley). Protokol IP byl přijat jako závazný v kontextu s výsledky práce projektu DARPA (Defense Advanced Research Project Agency) vlády Spojených států. DARPA řešil propojování heterogenních sítí již od druhé poloviny 60. let (tehdy pod označením pouze ARPA). Výsledkem byl právě protokol IP (Internet Protocol, protokol pro propojování sítí), který vrstvu síťovou řeší, jak bylo uvedeno výše. Vzhledem k současnému stavu celosvětové sítě (zejména z důvodů brzkého vyčerpání rozsahu adresace uzlů) je dnes prezentován návrh protokolu IP nové generace (s označením IPng), jehož hlavní rysy uvedeme v následujícím čl. 7.1. Začátkem 80. let D. Ritchie uvedl písemný dokument [Ritch84] o rozšíření jádra UNIXu ve smyslu PROUDŮ, jak jsme uvedli v kap. 6, což znamenalo obecný návrh implementace síťových protokolů, který tehdy skupina BBN nepoužila. Použil jej ale vývojový tým komerčního UNIXu firmy AT&T a výsledek jejich práce z pohledu procesové vrstvy nazval TLI (Transport Layer Interface). Je to varianta pro používání různých protokolů jak síťové, tak transportní vrstvy v jádru, kdy je zásobník PROUDŮ použit podle potřebného síťového spojení, samozřejmě za přítomnosti modulů jádra, které protokoly realizují. Tento způsob

implementace sítí v UNIXu také více odpovídal již zmíněnému sedmivrstvému modelu OSI, který je registrován jako standard ISO. Oba způsoby, TLI i Berkeley sockets, jsou dnes používány prakticky ve všech prodáváných verzích UNIXu.

Standardem pro síťové spojení je model OSI. POSIX jej cituje jako základní způsob definice propojování operačních systémů, stejně jako další dokumenty síťových standardů ISO od IEEE (Institute for Electrical and Electronic Engineers) nebo CCITT (Consultative Committee for International Telephony and Telegraphy). POSIX sám síťové protokoly, služby nebo volání jádra nedefinuje. SVID popisuje TLI, a to velmi podrobně. V tomto textu se proto u rozhraní procesů budeme na tento dokument často odvolávat. Prezentační vrstva je v SVID definována v podobě RPC (Remote Procedure Calls) a XDR (External Data Representation), kde je uvedena úplná definice gramatik obou prostředků. Aplikační vrstva je zde zastoupena také a budeme ji v průběhu textu uvádět. Správa sítí je z pohledu standardů popelkou a správcům systémů tak uchystá nejedno nemilé překvapení ve formátu systémových tabulek



sss, uuu v rozsahu 0-255

v uuu od konce 0 rezervována jako identifikace sítě

ve všech uuu 255 rezervováno jako adresa broadcast

v sss adresa sítě 127 rezervována pro loopback

Obr. 7.10 Typy adres IP

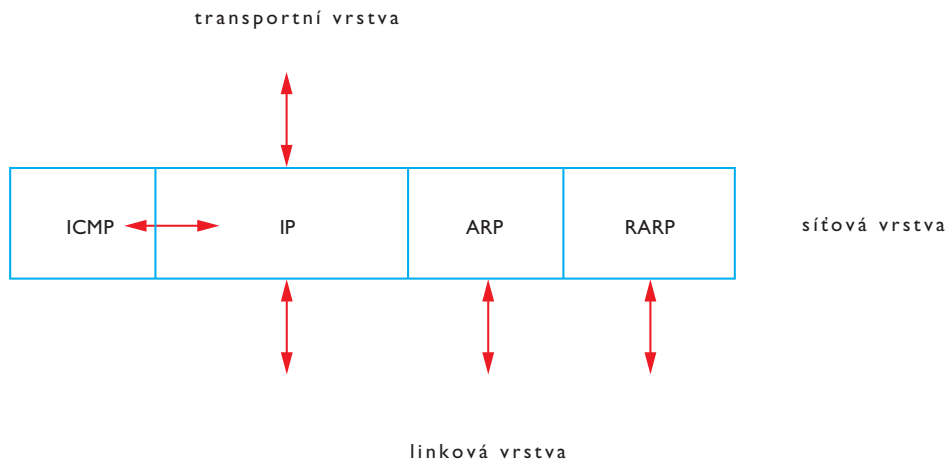
nebo používání příkazů pro nastavování vrstev sítě. Proto se budeme správou zabývat především obecně a uvedeme jen ty informace, které jsou ustáleny.

První verze UNIXu na počátku 80. let obsahovaly možnost spojení dvou uzlů pomocí sériového rozhraní. UUCP je označení sady programů, které dodnes v systému umí realizovat spojení typu peer to peer (komunikace právě dvou protilehlých uzlů), dnes zejména používané k připojování uzlů nebo celých LAN pomocí pevné nebo vytáčené telefonní linky. Protokoly PPP (Point to Point Protocol) nebo SLIP (Serial Line Interface Protocol) jsou nadstavbou síťové vrstvy nad UUCP vlastně mimo jádro a podrobně ji uvedeme v čl. 7.5.

Většina přenosových protokolů, o kterých budeme v této kapitole hovořit, vznikla v rámci vývoje celosvětové sítě Internet. Internet, kterému věnujeme samostatný článek a který je postaven na bázi sítě TCP/IP, obsahuje mimo jiné také informace cenné pro každého správce místní sítě na bázi UNIXu. Základní informace technického charakteru vydává rada Internetu prostřednictvím tzv. dokumentů RFC (Requests For Comments, žádost o komentáře). Dokumenty s označením RFC jsou číslovány a jejich počet neustále narůstá. Obsah dokumentů přitom může být jak pracovní poznámka k vývoji některé současné problematiky v Internetu, tak i obsažný popis práce některého protokolu používaného různými vrstvami v síti. Jsou obsažnější než registrované standardy a také správci sítí více studované, protože jde většinou o popis již praxí prověřeného softwaru. Jedná se o základní zdroj informací síťových protokolů UNIXu. Jejich umístění v Internetu je dnes na adrese <http://info.internet.isi.edu:80/in-notes/rfc>

## 7.1 Vrstva síťová, protokol IP

Jak již bylo uvedeno, *Internet Protocol* (dále jen IP) je nalezené řešení projektu DARPA pro propojování sítí. Je realizací síťové vrstvy tak, že uzly sítě mají přidělenou adresu, která je jedinečná z pohledu všech připojených uzlů nebo sítí. Každý uzel celé sítě IP tak může kdykoliv jednoznačně adresovat kterýkoliv



Obr. 7.11 Struktura IP

jiný uzel. V současné době to není jediný způsob spojování sítí mezi sebou. Zejména v celosvětové síti Internet je však používán nejvíce a každý UNIX jej doporučuje.

Identifikace uzlu je adresou IP. Je to celočíselná hodnota o velikosti 32 bitů. V ní je kódováno číselné označení sítě, ve které se uzel nachází, a dále je v ní kódován samotný uzel v rámci této sítě.

Ve 32 bitech označení adresy mohou být síť a uzel kódovány různým způsobem. Mluvíme tak o různých typech adres IP. Jak ukazuje obr. 7.10, rozlišujeme typy adres A, B, C a D.

Rozlišení typu adresy je podle prvních několika bitů. Zbytek do 32 bitů adresy je rozdělen pro identifikaci sítě (A 7 bitů, B 14 bitů, C 21 bitů) a identifikaci uzlu v síti (24, 16 a 8 bitů). Výjimkou je adresa typu D, která je rezervována pro zvláštní skupiny uzlů určitých aplikací (např. videokonference) a běžně se v sítích nepoužívá. Na obrázku je schematicky také uveden rozsah jednotlivých bytů 32 bitového čísla IP. Je zde i rozsah prvního bytu pro jednotlivé typy adres, jak vyplývá ze způsobu rozdělení rozsahu 32 bitového čísla. Zápisem hodnot jednotlivých bytů v desítkové reprezentaci dostáváme používaný tzv. tečkový formát IP. Tento způsob zápisu je nejsrozumitelnější, protože je z něj jasné patrný typ adresy a označení uzlu v adrese (na obr. 7.10 uvedeno uuu pro uzel, sss pro síť). Rozsah jednotlivých adres je pak následující:

- A umožňuje adresovat 126 sítí, v každé síti vždy přibližně 16 miliónů uzlů,
- B 16256 sítí, v každé přibližně 65000 uzlů,
- C přibližně 2 milióny sítí, každá po 254 uzlech.

Pokud bude správce počítačové sítě plánovat připojení k jiným sítím, musí respektovat IP těchto sítí. Dnes je to respektování adres sítě Internet, která je celosvětovým sdružením uživatelů sítí typu TCP/IP. Přestože jde pouze o dobrovolnou iniciativu, jsou adresy IP vždy respektovány; vznikla organizační struktura přidělování adres, která vychází z centra údržby sítě sítí Internet (NIC, Network Information Center) přes *poskytovatele Internetu* (Internet provider) v každé zemi, kteří mají oprávnění adresy IP přidělovat. Poskytovatel zájemcům přiděluje síťovou část adresy IP a správce sítě pak používá část pro identifikaci uzlů k přidělování adres IP jednotlivým počítačům jeho sítě. Např. správce může obdržet adresu (typu C) 194.1.1. Poslední byte je pak variabilní hodnotou a správce jí identifikuje jednotlivé uzly své sítě. 194.1.1.1 je správná adresa, 194.1.1.2 také atd. V tomto případě má správce možnost spojení až 254 uzlů v jedné síti. Adresa s hodnotou uzlu 0 je totiž rezervována pro označení samotné sítě (194.1.1.0 je identifikace sítě 194.1.1, což je v praxi totéž) a 255 pro tzv. adresu broadcast. To je adresa pro oslovení (rozeslání zprávy) všech uzlů dané sítě, což IP může učinit. Takže 165.34.255.255 je adresa pro oslovení všech uzlů sítě 165.34 (typ B). Rezervována je také adresa 127. Uzel s označením 127.0.0.1 je adresa používaná pro referenci na sebe samu a je používaná pro testování síťových služeb uzlu (tzv. loopback address). Stejně tak je rezervovaná adresa 0, která je používaná pro tzv. *implicitní směrování* (default routing), tj. adresu, kterou uzel používá pro zasílání paketů, které nedokáže jednoznačně směrovat.

Po přidělení síťové části adresy IP nemusí být správce sítě vždy spokojen. Jeho síť totiž může být také složena z několika sítí, připojuje tedy k Internetu také síť sítí. K tomu účelu je součástí IP tzv. *síťová maska* (netmask). Jde o možnost rozdělit přidělenou síťovou adresu na několik podsítí. Síťová maska je odvozena z adresy sítě. Na místě identifikace uzlu můžeme stanovit počet bitů, které identifikují podsít. Např. síťová maska adresy typu C bez rozdělení na podsítě je 255.255.255.0. Použijeme-li způsob rozdělení na dvě podsítě, maskujeme horní bit posledního byte adresy, tj. síťová maska je

255.255.255.128. Dělíme-li síť na 4 podsítě, maskujeme 2 bity, poslední byte je tedy 192 atd. Takto vzniklé adresy sítí pak používáme pro jednotlivé připojované sítě. Problém spojení takových podsítí je pak prakticky ve správném směrování (routing) v uzlu, který jednotlivé sítě fyzicky spojuje. Takový uzel je směrovač (router). Každý jeho síťový adaptér má přiřazenu síťovou adresu IP a pomocí odpovídajícího nastavení jsou adaptéry mezi sebou propojeny. Propojení se provádí příkazem **route**, který používá volání jádra `ioctl` pro oznámení modulu síťové vrstvy v jádru jeho chování. Příkaz používá tzv. statické směrování (static routing), protože vždy při startu operačního systému (resp. jeho části pro podporu sítě) jsou cesty paketů IP pevně určeny. Lze však používat také tzv. dynamické směrování (dynamic routing). Jeho představitelem je např. démon **routed**, kdy je směrování paketů určováno podle vytížení sítě. Směrovače je důležité znát při propojování sítí do větších celků a o jejich nastavování bude v této kapitole a v dalších kapitolách ještě mnohé řečeno. Na tomto místě je důležité si uvědomit jemnější vnímání adres IP. Adresa IP určuje uzel, avšak uzel může mít více adres IP. Adresa IP je totiž vždy vztažena k síťovému rozhraní, kterých může mít uzel více.<sup>3</sup> Síťové rozhraní přiděluje správce systému adresu IP pomocí příkazu **ifconfig**. V parametrech příkazu zadáváme označení linkové vrstvy (hardwaru, tedy jméno speciálního souboru), adresu IP uzlu, síťovou masku a adresu broadcast, např.

```
$ ifconfig ec0 194.1.1.2 netmask 255.255.255.128 broadcast 194.1.1.127
```

Moduly jádra, které realizují práci IP, ukazuje obr. 7.11.

Jak ukazuje obrázek, síťová vrstva při použití IP má několik modulů, které jsou také označovány termínem protokol.

Protokol ICMP (Internet Control Message Protocol) je používaný síťovou vrstvou k dorozumívání uzlu a směrovače nebo ke zpracování chyb síťové vrstvy. Aktivuje jej např. program **ping**, který používáme ke zjištění, zda je určitý uzel zadaný adresou IP dostupný. Při jeho použití např. zápisem

```
$ ping 194.1.10.1
```

```
PING 194.1.10.1 (194.1.10.1): 56 data bytes
64 bytes from 194.1.10.1: icmp_seq=0 time=11 ms
64 bytes from 194.1.10.1: icmp_seq=1 time=10 ms
64 bytes from 194.1.10.1: icmp_seq=2 time=10 ms
64 bytes from 194.1.10.1: icmp_seq=3 time=10 ms
```

```
^C
```

```
-----194.1.10.1 PING Statistics-----
```

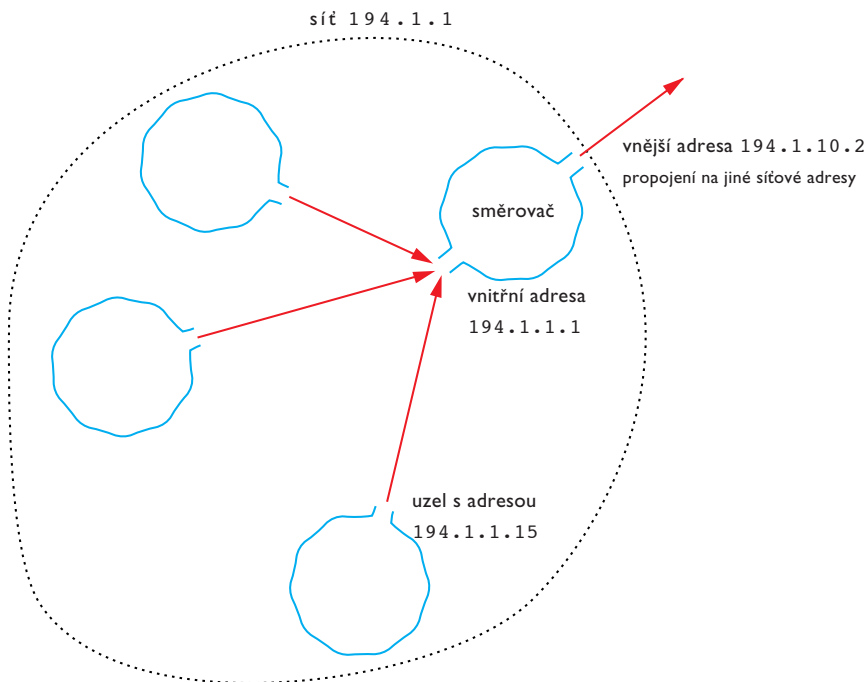
```
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 10/10/11 ms
```

```
$
```

je až do přerušení prověřována cesta paketů sítí mezi naším uzlem a uzlem s adresou použitou v příkazu. Výpis programu na prvním řádku nejprve uvádí komentář se jménem programu a adresu IP, o kterou se zajímáme. V závorce je tato adresa opakována, protože uživatel může v příkazovém řádku uvádět jméno uzlu (např. podle DNS) a nikoliv adresu IP. **ping** takto vyjádří jím zjištěné přiřazení jména uzlu a jeho adresy IP. Součástí prvního řádku výpisu je pak velikost zkušebního paketu (56 slabik). Vzhledem k tomu, že hlavička paketu IP má velikost 8 slabik, 64 je celková

délka dat posílaných sítí. Každý přijatý paket potvrzený osloveným uzlem je komentován vždy samostatně na dalších řádcích výpisu. Součástí výpisu je také pořadí potvrzovaného paketu (`icmp_seq`) a čas spotřebovaný na průchod dat sítí (`time`). Po přerušení (zde stiskem kláves `^c`) **ping** zastavíme, ale program ještě provede závěrečné vyhodnocení testovaného spojení na uzel, uvede souhrnný počet odeslaných a potvrzených paketů a vyhodnotí počet ztracených paketů v procentech. Nejdelší, nejkratší a průměrnou odezvu osloveného uzlu pak uvádí na posledním řádku s označením `round-trip`. Pokud probíhá komunikace protokolů ICMP v obou uzlech, je jisté, že všechny nižší vrstvy sítě včetně vrstvy IP pracují bezchybně.

Protokoly ARP (Address Resolution Protocol) a RARP (Reverse ARP) nejsou nutnou součástí síťové vrstvy IP. Jsou používány při nutnosti identifikace hardwarové adresy síťového adaptéru protokolem IP (každý Ethernet má např. jedinečnou adresu danou kódem výrobce a svým sériovým číslem, adaptéry své adresy rozeznávají při vzájemné identifikaci). Musí pak být zajištěna jednoznačnost mezi adresou hardwaru a IP. Modul ARP pracuje jako převaděč adres IP na adresy adaptéru. Dynamicky vytváří a udržuje tabulku takových převodů. Pracuje tak, že pokud je dotázán na adresu adaptéru zadáním adresy IP, prozkoumá tabulku a v případě, že taková položka existuje, vrátí její hodnotu. V případě, že ji v tabulce nemá, dotazuje se pakem zprávy (broadcast) každého adaptéru sítě. Pokud některý z dotázaných uzlů má adresu IP hledaného adaptéru přiřazenu, odpovídá hodnotou adresy adaptéru a dotazující



Obr. 7.12 Příklad směrování



uzel (přesněji modul ARP) si svoji tabulku o odpovídající dvojici obohatí. Z příkazového řádku můžeme získat obsah tabulky ARP příkazem **arp**, např.

```
$ arp -a
```

```
valerian.vic.cz (194.1.1.1) at 0:a0:24:41:eb:a
```

```
gagarin.vic.cz (194.1.1.15) at 8:0:69:2:95:75
```

je výpis všech adaptérů evidovaných v tabulce. Adrese IP, která je ve výpisu v kulaté závorce, předchází jméno uzlu (zde v konvencích DNS, viz dál). Pokud uvedeme v parametru odpovídající adresu IP (nebo jméno hledaného uzlu), **arp** vrací daný vztah

```
$ arp 194.1.1.15
```

```
194.1.1.15 (194.1.1.15) at 8:0:69:2:95:75
```

Protokol RARP (Reverse Address Resolution Protocol) rozpoznává zpětně adresu IP podle adresy hardwaru. RARP nabývá smyslu např. u bezdiskových uzlů (např. grafické terminály X), kdy je nutno přidělit adresu IP po zapnutí počítače. Aby bylo možné dynamicky adresu IP takových uzlů měnit, byl zaveden právě RARP. Paket síťové vrstvy s žádostí o hodnotu adresy IP je zaslán serveru, který (jeho protokol RARP) odpovídá podle obsahu příslušné tabulky (např. `/etc/ethers` v případě adaptéru typu Ethernet), která má textovou podobu a je měnitelná běžným textovým editorem (jako je `vi`).

Záhlaví paketu IP, které má velikost 20 bytů, tedy musí kromě dalších informací (kontrolní součty atp.) obsahovat zejména adresu IP výchozího uzlu (místní adresa) a adresu IP koncového uzlu (vzdálená adresa). Data, která IP obohatí o záhlaví, posílá do sítě jako paket. Cesta paketu sítí je pak odvozena nejprve z cílové adresy sítě. Jde-li o síť shodnou s výchozí sítí, paket dostává přímo odpovídající adresovaný uzel. Pokud jde o síť jinou, IP používá směrovací tabulky (routing tables) pro určení cesty paketu do jiné sítě. Směrovací tabulky jsou součástí struktury jádra a správce systému je naplňuje nebo modifikuje např. příkazem **route**. Dříve používaný systém tzv. jádra Internetu (Core Internet) pro směrování paketů byl založen na skupině vzájemně propojených směrovačů, se kterými musela být každá nově vytvořená síť spojena. Obecně se však neuchytil, přestože připojované sítě a uzly nemusely obsahovat vlastní směrování paketů sítí Internet. Naproti tomu se běžně používá tzv. kooperace autonomních systémů (AS, Autonomous Systems), kdy jde o systém směrovacích domén, tj. jednotlivé sítě jsou podporovány vždy určitým směrovačem (nebo více směrovači), který je nastaven pro každý uzel sítě jako směr cesty paketů. Domluva jednotlivých směrovačů je pak výchozí pro předávání a směrování paketů. Jak již bylo řečeno, v případě komplikovanějšího spojování více sítí (např. při údržbě některé MAN pro lokalitu většího města) může cesta paketu sítí probíhat přes různé směrovače, protože pomocí procesů démonů **routerd** nebo lépe **gated**, které si vzájemně sdělují vytíženost oblastí sítí v jejich kompetenci, mohou být pakety směrovány dynamicky pokaždé jinou cestou k témuž uzlu. Přesto se dnes nejvíce setkáváme s tzv. statickým směrováním, tj. pevně zadanými směry cestování paketů, které správce systému nastavuje příkazem **route**. Příkazem oznamujeme protokolu IP spojení uzlu se směrovačem. Jednoduchý příklad ukazuje obr. 7.12.

Síť 194.1.1 je tvořena několika uzly. Kromě použití příkazu **ifconfig** pro přidělení adresy IP každému uzlu sítě musí být také pomocí příkazu **route** nastavena cesta pro pakety odkazující mimo tuto síť. Směrovač je zde počítač s přidělenou adresou 194.1.1.1, ale také s adresou 194.1.10.2 pro další síťový adaptér. To je adresa uzlu jiné sítě, která sousedí se sítí 194.1.1. V naší síti pak v každém uzlu používáme příkaz takto:

```
$ route add default 194.1.1.1 1
```

Tak určujeme cestu každého paketu zpracovávaného modulem IP tohoto uzlu, pokud je vzdálená adresa paketu mimo síť 194.1.1.1. Cesta takových paketů vede přes směrovač a pokračuje podle směrovacích tabulek tohoto směrovače, jejichž obsah je dán dalším pokračováním sítě. V uvedeném příkazu je posledním parametrem 1. Je to tzv. vzdálenost (metric), tj. počet sousedních sítí, přes které má IP hledat zadaný směrovač.

Pro testování správného nastavení sítě v uzlu je používán příkaz **netstat**. Má řadu voleb, jejichž zadáním je možné získat množství informací o různých vrstvách síťového softwaru. Jeho volbou `-r` obdržíme výpis směrovacích tabulek. V následujícím použití je volba použita s kombinací `-n`. Jde o žádost výpisu označení uzlů v číselné podobě, tj. v tečkovém formátu adres IP. Uzly totiž bývají běžně pojmenovány např. systémem DNS, jak si uvedeme v dalších článcích této kapitoly.

```
$ netstat -nr
```

Routing tables

Destination	Gateway	Flags	Refcnt	Use	Interface
127.0.0.1	127.0.0.1	UH	1	298	lo0
194.1.1.10	127.0.0.1	UH	2	4620	lo0
default	194.1.1.1	UG	126	126813	ec0

...

Výpis tabulky směrování uzlu sítě začíná uvedením odkazů na loopback (127.0.0.1) a adresy uzlu IP opět na loopback (síťové požadavky na sama sebe se vrací na úrovni IP zpět). Položka tabulky začínající **default** označuje směrování všech dalších odkazů na adresy IP, které jsou součástí jiných sítí, než je 194.1.1.1. Uzel s adaptérem 194.1.1.1 tedy směřuje pakety mimo lokální síť. Příznaky (Flags) znamenají: U (up) je funkční směrování, H (host) je indikace směrování na konkrétní uzel a G (gateway) je označení místa, kde se paketů ujímá směrovač. Sloupec **Refcnt** je počet uskutečněných odkazů na způsob směrování od navázání spojení. **Use** je počet paketů, které směrováním prošly, a **Interface** je označení síťového adaptéru.

Na obr. 7.13 vidíme složitější příklad směrování uzlů lokální sítě. Síť z příkladu 7.12 jsme zde rozdělili pomocí síťové masky 255.255.255.128 na dvě podsítě s uzly od 194.1.1.1 do 194.1.1.126 a 194.1.1.129 do 194.1.1.254. Pro doplnění, první síť bude mít adresu broadcast 194.1.1.127 a druhá 194.1.1.254.

Směrovač do ostatních sítí Internetu zůstává na adrese 194.1.1.1 a pro uzly s adresou do 126 včetně je směrování takto zachováno. Pro uzly s označením nad 129 včetně je směrování změněno na adresu 194.1.1.129, kterou jsme použili pro další síťový adaptér uzlu s adresou 194.1.1.10. Použití příkazu **route** pro **add default** v této podsíti bude pro každý uzel na adresu 194.1.1.129. Další směrování zajišťuje nový směrovač se dvěma adaptéry. Jeho směrovací tabulky musí tak být obohaceny o spojení

```
$ route add 194.1.1.10 194.1.1.129 1
```

kdy pakety IP z podsítě 128 budou směřovány do podsítě 0 a případně do dalších sítí Internetu.

Příkazem **netstat** ve směrovači, který spojuje uvedené dvě podsítě, situaci provedeme

```
$ netstat -nr
```

## Routing tables

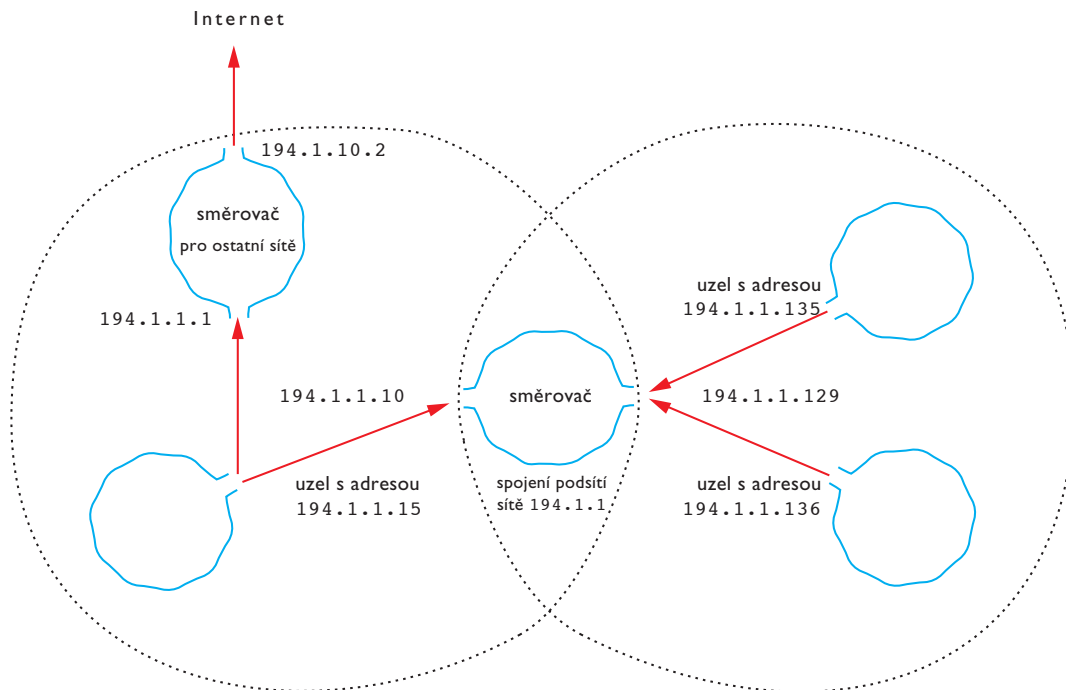
Destination	Gateway	Flags	Refcnt	Use	Interface
127.0.0.1	127.0.0.1	UH	1		lo0
194.1.1.10	127.0.0.1	UH	2	4620	lo0
default	194.1.1.1	UG	126	126813	ec0
194.1.1.10	194.1.1.129	UG	3	4555	ec2

...

Tento výpis směrování uzlu, který spojuje naše dvě podsítě, ale není ještě úplný. Pokud požadujeme také oslovování uzlů sítě 128 uzly ze sítě 0 (což je přirozené), musíme ve spojovacím uzlu zajistit i opačný průchod paketů, tedy příkazem

```
$ route add 194.1.1.129 194.1.1.10 1
```

Situace se pochopitelně komplikuje u více směrovačů nebo u spojování mnoha sítí v jednu metropolitní. Nová technologie směrování jde pak cestou tzv. *dynamického směrování* (dynamic routing), jak jsme již uvedli. Má výhodu oproti statickému směrování v tom, že se přizpůsobuje změnám prostředí sítě. Dynamické směrování je realizováno tzv. směrovacími protokoly, kterými je vždy definován způsob dynamického směrování.



Obr. 7.13 Směrování uzlů dvou podsítí

Dynamické směrování je pružná změna směrovacích tabulek. Programové moduly, které tyto tabulky aktualizují, nazýváme *směrovací protokoly* (routing protocols). V prvním přiblížení dynamické směrování rozlišuje *interní* (interior) a *externí* (exterior) směrovací protokoly. Interní jsou používány pro směrování uvnitř (lokálních) sítí a externí při správě spojování sítí v rámci např. MAN. Správce sítě ve větší míře zajímají interní protokoly, a proto jsou tyto typy směrování většinou součástí každého komerčního UNIXu.

Nejpoužívanějším interním protokolem je RIP (Routing Information Protocol), který vždy na požádání stanovuje nejlepší cestu průchodu paketu podle dotazů na nejbližší směrovače. Jinými slovy, např. démon **routed** komunikuje s okolními sítěmi, tj. démony (např. opět **routed**), kteří s ním komunikují jemu srozumitelným způsobem a sdělují mu informace o průchodu paketu sítí. Démon RIP dokáže oslovit okolní směrovače až do vzdálenosti 15 sítí. Tato *vzdálenost* (metric) bývá označována také termínem *počet hopů* (hop count). Při výpočtu se pak přidržuje pravidla, že nejvhodnější je nejkratší cesta doručení paketu. Způsob vybírání nejlepšího směru je zde nazýván algoritmem vektorového rozdílu (distance-vector algorithm). Pokud je vzdálenost směrovače, který je vyhledáván, větší než 15, pak RIP selže, není tudíž vhodný pro rozsáhlé sítě. Méně používaný protokol Hello se rozhoduje na základě časového *zpoždění* (delay) vyslaného paketu ke směrovači. Protokol Hello se vyvinul ze směrovacího protokolu pro počítače LSI-11 síť NSFNet (NSFNet backbone), který je dnes známý především jako T1 NSFNet (jde o rozšíření), což je varianta odpovídající tzv. protokolu IS-IS (Intermediate System to Intermediate System) směrovacího standardu OSI. Na rozdíl od RIP tyto protokoly nejsou omezovaly vzdálenostmi (do vzdálenosti 30000), ale přesto je jejich určení především pro menší rozměry sítí. Pro opravdu rozsáhlé sítě lze používat interní protokol OSPF (Open Shortest Path First), který ovšem bývá v UNIXu implementován zřídka. Jeho vlastností je totiž možnost udržovat několik rovnocenných cest odpovídajícího směru, což je u v UNIXu používaného protokolu IP bez užitku. Ten totiž akceptuje metodu „první vhodný je dobrý“, a zbytek směrovací tabulky ignoruje. Při implementaci OSPF proto musí být IP preprogramován.

Externí směrovací protokoly jsou určeny pro spojování a údržbu lokalit mezi sebou. Jde tedy o spojování autonomních systémů (AS) jednotlivých směrových domén, jak jsme již uvedli. Správce místní sítě většinou tyto protokoly nezajímají, protože jejich směrovací doménu externím protokolem zpřístupňuje většinou poskytovatel Internetu. Vnitřní síť obsluhuje démon **gated**, který je v současné době vyměňován za **routed**, protože je rychlejší. V opačném případě je nutné používat samostatný démon pro externí směrování, kterým je např. **eggpup**. Externí směrovací protokoly jsou používány zejména EGP (Exterior Gateway Protocol) a BGP (Border Gateway Protocol), který EGP v současné době stírá. V obou případech jde o sbírání požadavků od interních směrovačů a jejich odesílání do uzlů centrálního zpracování (core gateways), odkud po jejich zkombinování a výpočtu jsou autonomním systémům vráceny informace o optimálním směru.

Dynamické směrování na úrovni interního protokolu zajišťuje v UNIXu správce systému startem démonu **routed** při zavádění systému. Přestože démon udržuje tabulky RIP dynamicky podle informací z okolních směrovačů, je možné (a bývá to dobrým zvykem) definovat výchozí obsah tabulky RIP v souboru `/etc/gateways`. Při zajištění externího směrování démonem **eggpup** je výchozí tabulka `/etc/egp.init`. Moderní démon **gated**, který je výkonnější než současné použití **routed** a **eggpup**, je interní protokol, který rozumí i protokolům externího směrování (kombinuje RIP, Hello, BGP i EGP). Používá pro nastavení své konfigurace obsah souboru `/etc/gated.conf`.

Základy směrování, které jsme zde na několika stranách uvedli, vycházely z předpokladu použití IP jako síťové vrstvy implementace síťového spojení. Při plánování směru proto musí všechny metody vycházet z informací, které IP dokáže poskytnout, tedy z těch, které jsou obsaženy v záhlaví jeho paketů. Záhlaví paketu IP má délku pěti nebo šesti 32 bitových slov a obsahuje zejména

- verzi IP,
- délku záhlaví (IHL, Internet Hheader Length), záhlaví může být proměnné délky,
- celkovou délku paketu,
- maximální délku života paketu,
- číselné označení protokolu vyšší vrstvy,
- kontrolní součty,
- adresu výchozího uzlu,
- adresu cílového uzlu.

Není bez zajímavosti, že IP je nositelem informace tzv. čísla protokolu vyšší vrstvy. Protokoly, které zde mohou být uvedeny, jsou operačnímu systému srozumitelné podle tabulky v souboru `/etc/protocols`. Podle ní jádro stanoví, zda je požadavek určen pro síťovou vrstvu (např. je-li v záhlaví uveden protokol ICMP) nebo pro vyšší transportní vrstvu síťového spojení, která je v UNIXu nejvíce používána TCP.

Dnešní, veřejností stále více žádaná síť Internet zajistila síťovému protokolu IP nebývalý věhlas, ale současně vyčerpala jeho možnosti. Nejpalčivější je současný nedostatek adres IP. Vzhledem k tomu, že připojen dnes chce být každý, síťové adresy poskytovatelé přidělují maskované na co nejmenší počet požadovaných uzlů. S jednoznačnou identifikací síťového adaptéru dnes ale počítá každá třeba i bezdrátová grafická stanice (tzv. X-display v prostředí X-Window System, viz následující kap. 8), proto je nutno řešit rozšiřování adresace uzlů v IP. Současný návrh (leden 1998) v dokumentech RFC (např. RFC 2080 a RFC 2081) hovoří o rozšíření protokolu na tzv. IPng (Internet Protocol new generation), označovaný také dnes IPv6 (dříve IPv4). Vychází se z předpokladů minimálních (nebo lépe, žádných) změn z pohledu již přidělených adres IP nebo přidávání nových směrovacích protokolů. Pro potřeby IPng bude vyhrazeno několik adres typu A a B (uvažuje se o 10 z každé třídy), které budou přiděleny uzlům, jejichž funkce v rámci Internetu bude rozšířena o správu tzv. *autonomních domén* (Autonomous Domains, AD). Autonomní doména je část Internetu, v rámci které bude možné přidělovat adresy IP v celém rozsahu. Paket, který opouští autonomní doménu, přitom prochází serverem autonomní domény a je obohacen (zabalen, zapouzdřen, encapsulated) o unikátní adresu autonomní domény. Jednoznačnost uzlu v síti pak určuje jednak adresa IP a jednak adresa AD. Naopak při vstupu do autonomní domény bude paket IP rozbalen a vpuštěn do části Internetu jako za starých časů. Návrh původního IP od ARPA je natolik dobrý, že snese rozšíření (jeho hlavička je např. proměnné délky). Změny jsou přitom vyžadovány pouze na úrovni správy pojmenování uzlů podle DNS (Domain Name System, viz 7.4) kde bude nutné odpovídající tabulky rozšířit o adresy AD a směrování paketů v hraničních směrovačích (serverech AD); zbytek zůstane zachován.

## 7.2 Vrstva transportní, TCP,UDP

Vrstva transportní (přenosová) zastává funkci převodu dat komunikujících procesů na pakety síťové vrstvy a naopak. Jde o zprostředkování přenosu dat, která proces požaduje přenést sítí tak, aby je proces

na odpovídající straně naopak přečetl, jak on je zapsal. V UNIXu nejpoužívanější je transportní vrstva nazývaná zkratkou TCP, přestože její součástí jsou dva protokoly, jednak TCP (Transmission Control Protocol, protokol řízení přenosu) zaměřený na souvislý tok dat (pro aplikace např. **telnet** nebo **ftp**), a dále UDP (User Datagram Protocol, uživatelský protokol pro datagramy), který umožňuje přenos jednotlivých zpráv bez kontextu v toku dat (používaný např. u NFS, síťového sdílení disků). Oba protokoly jsou nejčastěji používány v kombinaci s IP, takže je způsob přenosu označován zkratkou TCP/IP jako přenosového softwaru a pod touto značkou se také rozumí způsob síťového spojení.

Na obr. 7.8 jsme uvedli způsob zpracování dat při jejich přenosu sítí jednotlivými vrstvami. Přestože jsme zde uvedli pojmy segment, paket a rámec, terminologie byla vágní. Paket je výraz pro část dat síťové vrstvy, ale je také často používán výraz datagram. Výraz datagram je přitom převzat z vyšší vrstvy. I zde může být formulován úsek dat, který nemá význam části toku dat (data stream), ale pouze ověřovací nebo zjišťovací. Jeho obsah má tedy podobu určité zprávy (message) a aktivita na síti je jím reprezentována atomicky. To je případ použití protokolu UDP. S takovým zabezpečením přenosu dat sítí však těžko vystačíme u všech aplikací. Procesy často vyžadují obousměrný souvislý tok dat, který, pro ně znamená opakovaný zápis nebo čtení dat takového toku prostřednictvím volání jádra. Při použití UDP by každý proces musel formulovat tok dat do přenosového protokolu vždy sám, což je neefektivní a ve svém výsledku nekonzistentní, protože aplikace by se v různých uzlech často nedomluvily. Funkci zajištění přenosu toku dat pak plní protokol TCP. Hovoříme-li o něm, používáme výraz segment toku dat, který je předáván síťové vrstvě. Transportní vrstva sítě TCP/IP tak umožňuje dvojí přístup k přenosu dat. Jednak je to možnost navázání spojení, které trvá po dobu celého přenosu dat (TCP), nebo pouze přenos zprávy jako dotazu nebo upozornění bez navázání stálého spojení, tj. obsah zprávy je unikátní bez kontextu s předchozí nebo následující zprávou. Kontext dat pak přisuzují zprávám komunikující procesy (UDP). Obr. 7.14 ukazuje situaci transportní vrstvy TCP a jejího zařazení.

UDP je vhodný kandidát při používání síťové komunikace typu „dotaz-odpověď“ (query-response). UDP patří do skupiny tzv. nespolehlivých (unreliable) protokolů. Nespolehlivost zde znamená, že protokol pracuje bez potvrzení, čili o to, zda data byla přijata druhou stranou, se UDP nezajímá. Proces musí sám algoritmicky ošetřit, zda na jeho „dotaz“ přišla správná „odpověď“, a pokud se tak do určité doby nestane (tzv. timeout), „dotaz“ opakovat. Výhodou je možnost minimálního zatížení sítě. Rovněž je UDP vhodný na zasílání krátkých zpráv. Záhlaví protokolu, kterým UDP obohatí data procesu a předá je IP, je jednoduché:

- číslo výchozího (efemérního) portu,
- číslo cílového (rezervovaného) portu,
- délku dat,
- kontrolní součet.

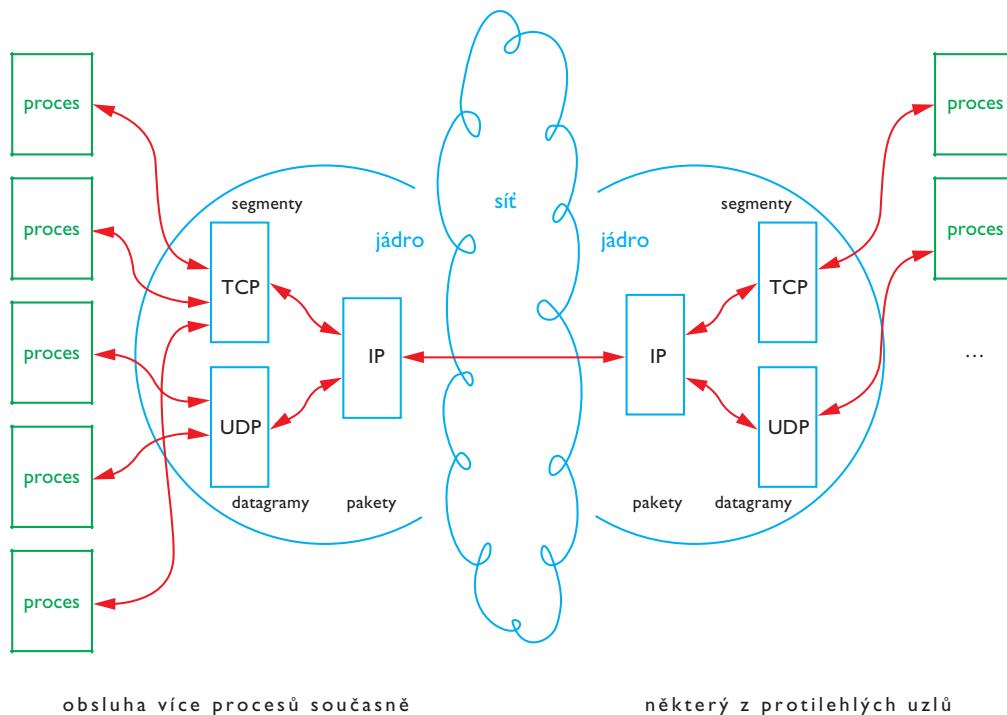
Oproti spolehlivým (reliable) protokolům je záhlaví krátké. Podstatné informace transportní vrstvy jsou čísla portu, jak bylo vysvětleno v úvodu kapitoly.

Spolehlivý protokol TCP přijímá od procesu souvislý tok dat, který přenáší sítí po částech (segmentech) balených na nižší síťové vrstvě do paketů IP. TCP je zodpovědný za přijímání potvrzení každého jím vyslaného segmentu do cílového bodu transportu. V cíli transportu segmenty potvrzuje jeho protějšek, který segmenty sestavuje opět do souvislého toku dat, který nabízí procesu, čte-li tento data ze sítě. Nutně zde před vlastním přenosem musí dojít k navázání spojení a součástí ukončení přenosu je

i zrušení spojení. TCP je označován jako spolehlivý protokol orientovaný na stálé spojení s určením pro přenos toku bytů (reliable, connection oriented, byte stream protocol). Záhloví segmentu musí tak obsahovat zejména následující hodnoty:

- číslo výchozího (efemérního) portu,
- číslo cílového (rezervovaného) portu,
- číselné označení pořadí segmentu,
- číslo potvrzovaného segmentu,
- přenosové okno,
- kontrolní součet,
- příznak zvláštní pozornosti
- provozní příznaky, aj.

Data, která mají být přenesena sítí ve výchozím bodu transportu, rozdělí modul TCP na segmenty (např. po 2000 bytech). Stanoví velikost přenosového okna (např. na 6000 bytů), což znamená, že TCP vyšle do sítě najednou tři segmenty a čeká na jejich potvrzení (nejméně 1). Teprve je-li potvrzen první segment okna, posune okno o další segmenty toku dat. Velikost okna segmentů může být v průběhu



Obr. 7.14 TCP



přenosu korigována, pokud opačná strana navrhne okno menší délky. TCP zajišťuje spolehlivost (potvrzuje data) tzv. potvrzovacím algoritmem s opětným přenosem (Positive Acknowledgment with Retransmission, PAR), což jednoduše znamená, že TCP vyšle paket, který druhá strana potvrdí jako přijatý a nezkomolený. Pokud do určitého časového kvanta (timeout) potvrzení nepříjde, TCP segment opakuje. Druhá strana přitom potvrzuje pouze přijaté a nezkomolené segmenty. Zkomolené segmenty zapomíná a nestará se o ně.

V průběhu přenosu dat je udržováno stálé spojení. Před zahájením přenosu oba moduly TCP výchozího i cílového bodu transportu spojení vytvoří tzv. podání ruky (handshaking). Navázání spojení je u TCP třicestné (three way handshake), protože jsou vyměněny tři segmenty. Výchozí uzel pošle inicializační segment a čeká na jeho potvrzení segmentem z druhé strany. Poté navazující strana vyšle segment s inicializačními hodnotami spojení, např. počáteční číselné označení toku dat. Pak dochází k vlastnímu přenosu dat. Ukončení spojení je opět realizováno třemi segmenty, kdy je vyslán segment s označením „žádná další data“. Koncový bod přenosu tento segment potvrdí a výchozí bod přenosu dalším segmentem souhlasí.

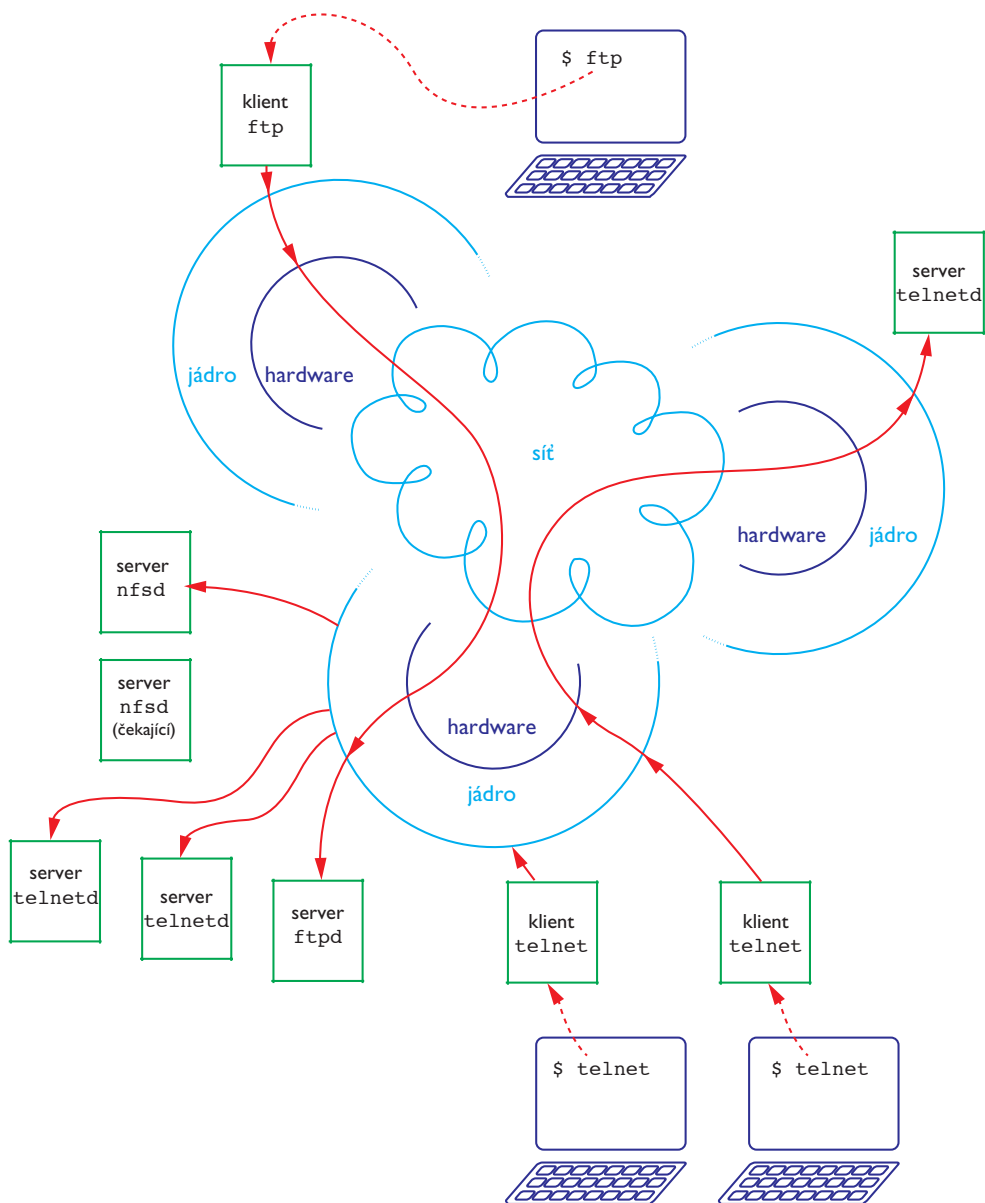
Při studiu algoritmů TCP se setkáme s řadou zdánlivých maličkostí, které je nutné ošetřovat, pokud má být spojení robustní. Např. u aplikace **telnet** jde o interaktivní přenos s proměnnou délkou segmentu (nebo o přenos segmentu délky 1 byte). Rovněž tak u této aplikace musíme zajistit přednostní přenos znaků např. přerušení z klávesnice nebo kláves XON/XOFF komunikace. U transportní vrstvy se hovoří o tzv. datech mimo rozsah (out of band data), protože musí být doručena přednostně mimo pořadí. K tomu např. slouží příznak zvláštní pozornosti v záhlaví segmentu (urgent pointer).

Problémy implementace uvedených schémat nastanou také s několikanásobným přístupem k síťovému rozhraní. Moduly jádra, které realizují TCP, UDP nebo jiný způsob transportu, musí zajistit tzv. multiplexing. TCP bude přijímat data od několika procesů současně, a to duplexně. Principiálně jsou však problémy uvedenými identifikacemi řešeny, protože operace v jádru UNIXu bude vždy atomická a omezení současného počtu přístupu k síti dáno hodnotou parametru jádra (měnitelnou regenerací jádra), navíc v daném okamžiku zajišťující horní hranici přenosu dat sítí, a tedy i průchodnost celého operačního systému.

### 7.3 Vrstva procesová

Volání jádra UNIXu, které používají procesy pro přístup k síti, představují z hlediska čtyřvrstvého modelu vrstvu procesovou. Z pohledu modelu OSI dochází k nesrovnalostem, protože pro UNIX je předěl mezi procesy a jádrem principiální, ale OSI předepisuje jemnější vrstvení, které se snaží v UNIXu sledovat SVID. Nicméně první vážná implementace sítí v UNIXu s definicí nových síťových volání jádra vznikla v systémech BSD (4.1BSD) začátkem 80. let při realizaci smlouvy mezi ARPA a skupiny BBN z Berkeley. První verze tehdy nazvaných Berkeley sockets (schránky typu Berkeley, nebo často pouze sockets, schránky) volání jádra pro potřeby procesů znamenala rozšíření platnosti volání jádra `open`, `creat`, `read`, `write`, `close`, a `lseek` pro práci se soubory i na přístup k síti. Vzhledem k tomu, že práce procesů se značně zkomplikovala a síťová nebo i místní meziprocesová komunikace vyžaduje mnohé možnosti navíc, ve verzi 4.3BSD (1986) byla definována skupina nových volání jádra, která se používá dodnes. V r. 1986 jako součást nově vydané verze UNIX SYSTEM V.3 firmy AT&T byla předána k užívání knihovna síťových služeb TLI (Transport Layer Interface), která





Obr. 7.15 Uzly, klienty a démony serverů

využívala implementace PROUDŮ (viz. kap. 6) a která monitoruje model OSI. Jde o knihovnu funkcí, která využívá volání jádra PROUDŮ. Přestože implementací síťové komunikace je v UNIXu více, my se zde zaměříme na uvedené dva nejpoužívanější způsoby práce procesů v síti typu TCP/IP.

Jak již bylo několikrát v této kapitole řečeno, síťové spojení v UNIXu je komunikace dvou procesů v různých uzlech sítě. V kontextu kap. 4 (obr. 4.2, čl. 4.7) je přitom nutné využívat architektury klient - server. Jeden z typických způsobů je přitom metoda, kdy každému procesu klient se věnuje právě jeden proces server (pro klient **telnet** vzniká v cílovém systému server **telnetd**, pro **ftp** server **ftpd** atd.), který byl vytvořen démonem **inetd** kontrolujícím síť při aktivaci síťového spojení na odpovídajícím portu. To je případ využití stálého spojení (protokol TCP). V případě bez stálého spojení (tzv. přenášení zpráv), např. u síťového sdílení disků NFS (protokol UDP, tj. bez stálého spojení), běží v cílovém operačním systému několik procesů (**nfsd** a **biod**), které jako servery obsluhují vždy požadavek jednoho klientu v okamžiku příchodu datagramu (a které jsou vytvořeny jako démoni procesem **init** při startu systému). Síťové spojení je ale vždy závislé na přítomnosti obsluhujícího serveru v cílovém systému. Nevznikne-li z popudu **inetd** nebo **init**, nemůže být požadavek klienta uspokojen. Obsluha síťových klientů je ve většině případů běžně užívaných síťových aplikací v UNIXu zcela symetrická, tj. uživatelé systému jednoho uzlu využívají síť např. ke vzdálenému přihlašování v jiných uzlech a současně jejich uzel poskytuje proces serveru nebo více serverů pro uživatele všech ostatních uzlů v síti, viz obr. 7.15. Běžně používané síťové aplikace jsou aplikace Internetu, kdy dorozumění jejich klientů a serverů je dáno rezervovanými čísly portů, jak jsme již v této kapitole uvedli. Jejich spolupráci a systémové nastavení si uvedeme v následujícím čl. 7.4.

Programovat novou síťovou aplikaci tedy znamená programovat dva procesy, klient a server. Dále je nutné zvažovat výběr typu spojení, zda jde o spojení stálé nebo pouze o přenos zpráv, definovat tedy komunikační protokol. Pro stálé spojení přitom vzniká server, který klientu slouží od navázání spojení až po jeho ukončení, zatímco v případě bez stálého spojení může být klient v průběhu jeho práce vždy při síťové komunikaci obsluhován jiným serverem. Při popisu Berkeley sockets nebo TLI budeme proto uvádět chování a používání odpovídajících volání jádra jak pro klienta, tak pro server, a dále jak pro stálé spojení, tak případ bez stálého spojení. V použití určité implementace pak obsluhujeme vždy čtyři případy různého chování procesů.

Přístup procesu k síti je často popisován jako podobenství přístupu k souborům. První implementace Berkeley sockets se pokusila toto podobenství dokonce realizovat rozšířením odpovídajících volání jádra pro práci se soubory. Přestože můžeme výsledek spolupráce dvou procesů opravdu chápat jako zápis, resp. čtení obecně nad určitým výpočetním zdrojem, práce síťových procesů je složitější. Jednak musí být navázáno spojení a jednak jde o řadu časových prodlev, které vznikají mimo místní jádro a které toto jádro neovlivní ani neodhadne. Analogii lze vidět (možná lepší) také v komunikaci dvou místních procesů např. pojmenovanou rourou, ale i zde je situace pouze přibližná, a přestože je často v literatuře podobenství uváděno formou různých tabulek s odpovídajícími volání jádra, my budeme v dalším textu možnosti takového podobenství akceptovat, explicitně uvádět, ale implicitně nevnučovat. Jednotu v přenosu dat v budoucnu totiž jistě přinese standard, a to v kontextu současného vývoje sítí ve smyslu práce na síťových operačních systémech (např. Plan 9, viz. kap. 11), kdy bude přístup k výpočetním zdrojům, ať už místního nebo vzdálených uzlů, transparentní a odpovídající volání jádra jednotná.

### 7.3.1 BSD sockets

Výchozím termínem pro orientaci je asociace, jak byla uvedena v úvodu kapitoly. Z pěti {protokol, místní adresa, port, vzdálená adresa, port} je socket právě část pro jednu stranu spojení, tzv. *poloviční asociace*, tj. {protokol, místní adresa, port} a {protokol, vzdálená adresa, port}. Berkeley sockets byly koncipovány sice především pro síťovou komunikaci procesů, ale socket je možné také vytvořit pro spojení a komunikaci procesů místního UNIXu. Jsou definovány tři základní způsoby používání schráněk, tzv. domény, které jsou typu:

UNIX, kdy se jedná právě o místní komunikaci procesů. Na místě portů jsou při používání uvedeny 0, namísto adres IP jsou uvedena jména souborů (např. v oblasti /tmp). Při použití tak vznikají soubory typu `S_IFSOCK` (s ve výpisu např. příkazu `ls`). Na místě protokolu je uváděno `unixstr`, což je proud (stream), tj. stálé spojení, nebo `unixdg` bez stálého spojení. Nedochozí k balení dat ani využívání protokolů sítě, což je logické. Spojení je ošetřeno jedním jádrem a je proto spolehlivé. BSD systémy implementují pomocí této domény roury.

XNS, kdy se jedná o podporu síťových protokolů XNS (Xerox Network Systems) firmy XEROX. Jde o protokoly, které jsou podobné TCP/IP. Dříve často používané byly vyvinuty firmou v 70. letech. Systémy BSD podporují hardware i software XNS, ale my se jimi budeme zabývat pouze okrajově.

INTERNET je doména pro komunikaci procesů za využití protokolů Internetu, tedy způsobu TCP/IP. Bude především předmětem dalšího popisu.

Procesy, účastníci síťové komunikace od svého vzniku až po zánik, prochází několika stavy, jak ukazuje obr. 7.16 a 7.17.

Komunikace procesů v různých uzlech při zajištění stálého spojení (obr. 7.16) probíhá na straně uzlu se serverem nejprve v přípravě síťového spojení, tj. v naplnění hodnot pro schránku. Struktura schránky (hodnoty poloviční asociace) je definována v `<sys/socket.h>` a server ji naplní ve volání jádra `socket` a `bind`. Voláním jádra

```
#include <sys/types.h>
#include <sys/socket.h>
```

```
int socket(int family, int type, int protocol);
```

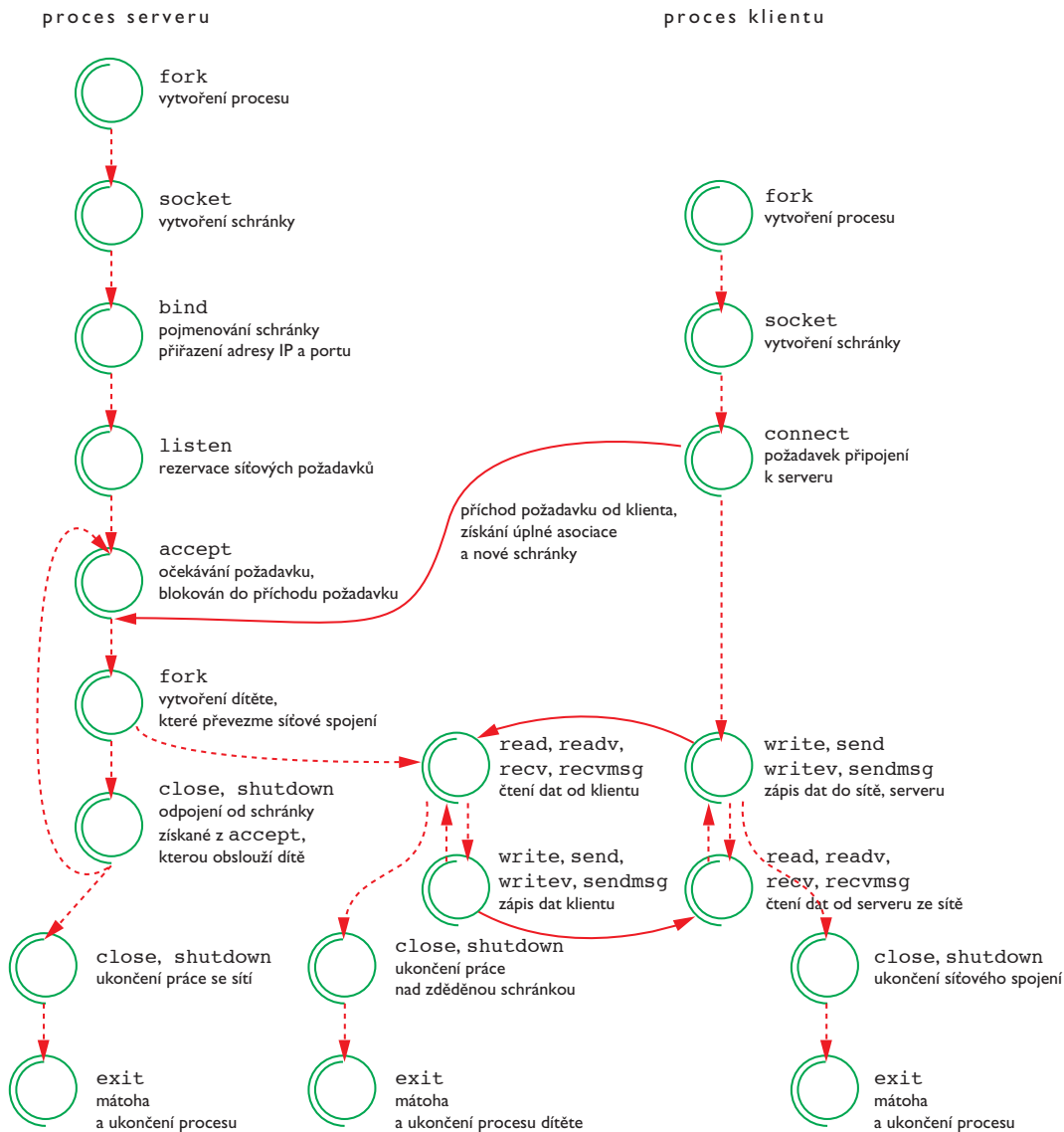
definujeme jednak doménu schránky ve `family` (`AF_INET` je INTERNET) a v `type` způsob spojení. `SOCK_STREAM` je stálé spojení, tedy použití protokolu TCP, a `SOCK_DGRAM` je bez stálého spojení, tj. použití UDP. Můžeme také použít `SOCK_RAW`, které implikuje tzv. přímé spojení. Proces pak pracuje přímo s protokolem IP. Konečně v `protocol` definujeme protokol. U domény INTERNET `IPPROTO_UDP`, `IPPROTO_TCP`, `IPPROTO_RAW` a `IPPROTO_ICMP`, vždy v odpovídajícím kontextu s `type`. Poslední dvě konstanty pak rozlišují použití IP nebo ICMP. Např.

```
sd=socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

otevívá schránku stálého spojení s protokolem TCP. Program **ping** (viz čl. 7.1) bude pravděpodobně používat schránku

```
sd=socket(AF_INET, SOCK_DGRAM, IPPROTO_ICMP);
```

Návratová hodnota volání jádra `socket` je deskriptor schránky, který je dále procesem používán pro práci ostatních volání jádra se schránkou. Teprve volání jádra



Obr. 7.16 Stavy procesů síťové komunikace v BSD, stálé spojení

```
int bind(int sockfd, struct sockaddr *myaddr, int addrlen);
```

přihadí procesu podle `myaddr` odpovídající číslo portu a síťovou adresu poloviční asociace. Argument `sockfd` je použit z návratové hodnoty `socket`. `myaddr` je ukazatel na strukturu s požadovanou adresou (tj. 16 bitovým číslem portu a 32 bitovou adresou IP) a `addrlen` je velikost této struktury. Tak např. fragment programu pro server může obsahovat

```
struct sockaddr adrserve;
adrserve.sin_family=AF_INET;
adrserve.sin_addr.s_addr=htonl(INADDR_ANY);
adrserve.sin_port=htons(SERV_TCP_PORT);
bind(sd, &adrserve, sizeof(adrserve));
```

Na obr. 7.16 uvádíme příklad serveru poskytující stálé spojení, který očekává požadavky klientů na dané schránce. Dalším voláním jádra

```
int listen(int sockfd, int backlog);
```

si server rezervuje výhradní přístup k vytvořené schránce. V parametru `backlog` stanovuje velikost fronty v požadavcích ze sítě, kterou jádro bude procesu udržovat (běžně se používá hodnota 5).

Doposud stále nebylo provedeno síťové spojení, dokonce ani nebyla síť aktivována. Server uvádí síť do aktivního stavu akceptováním požadavků voláním jádra

```
int accept(int sockfd, struct sockaddr *peer, int addrlen);
```

Proces serveru je nyní zablokovaný až do příchodu požadavku ze sítě. V případě, že se tak stane, jádro vytváří novou schránku stejných vlastností, jako je `sockfd`, a do `peer` odevzdá volajícímu procesu obsah schránky, kterou definoval klient. `addrlen` je doplňující hodnota velikosti obsahu schránky podobně jako u `bind`. V případě úspěšného `accept` bylo spojení navázáno a server obvykle vytváří pro obsluhu požadavku proces dítěte. Dítě dědí schránku a používá volání jádra pro čtení nebo zápis pro přenos dat mezi jím a klientem. Server schránku získanou z `accept` uzavírá pomocí `close` (je to totéž `close`, kterým se uzavírá deskriptor souboru, zde nazývaný deskriptor schránky `sockfd`) a vybírá další požadavek ze sítě nad odpovídající schránkou. Schránku může také uzavřít pomocí volání jádra `shutdown` (které umožňuje větší komfort práce při uzavírání schránky).

Inicializace síťového spojení na úrovni protokolu TCP (tj. v předchozím článku komentované třicestné navázání spojení) proběhne na základě volání jádra klientu

```
int connect(int sockfd, struct sockaddr *servaddr, int addrlen);
```

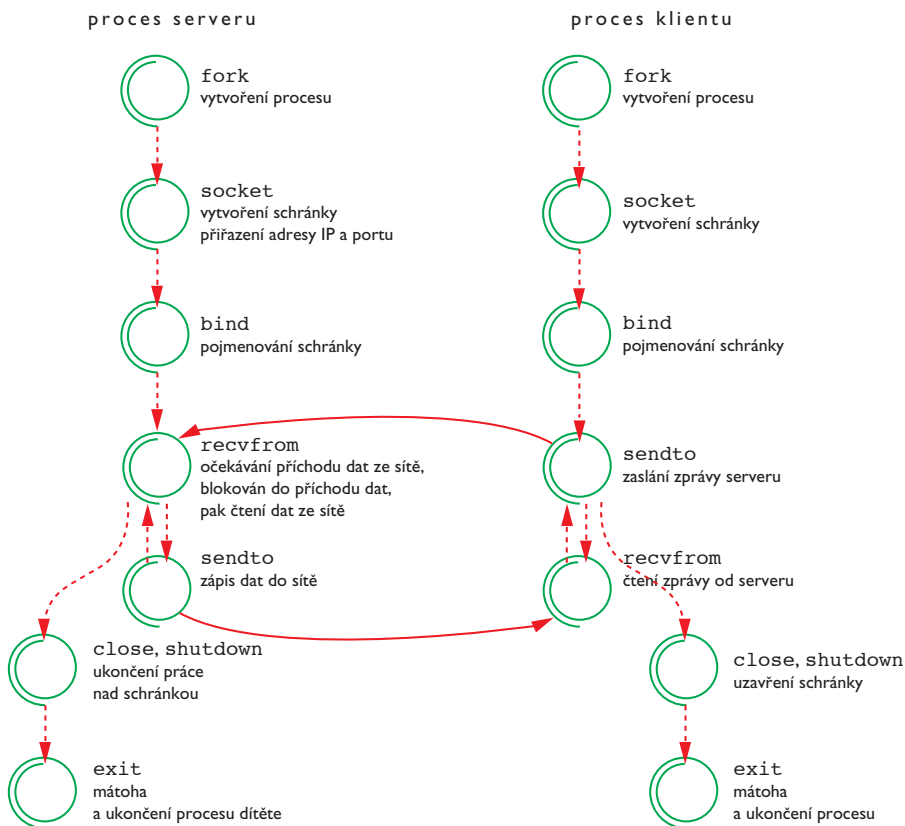
Schránku `sockfd` klient dříve vytvořil voláním jádra `socket` a v argumentech `servaddr` a `addrlen` zadává identifikaci serveru v protilehlém uzlu. Volání jádra `connect` vytvoří spojení, které je v dalším přenosu dat sítí pouze odkazováno při čtení nebo zápisu dat sítí. V případě programování schránek bez stálého spojení je používáno jiné volání jádra, kdy je při každém síťovém přenosu nutné zadávat protilehlou adresu a navazovat spojení (viz obr. 7.17 a následný popis).

U stálého spojení mohou procesy pro přenos dat použít volání jádra `read` a `write` pro čtení nebo zápis do sítě jako do souboru. Při jejich používání nad schránkou musí však programátor uvažovat také situaci, kdy např. proces nepřechte zadaný počet znaků najednou, a to ne z důvodu ukončení síťového přenosu, ale prostě proto, že data ještě neprošla sítí. Schránky Berkeley nabízejí dále rozšířená volání jádra `readv` a `writv`, pomocí kterých může proces opatřit data pro přenos sítí záhlavím aplikace (viz

obr. 7.8), usnadňují tak programátoru zapouzdření na úrovni vrstvy procesové. Jak je uvedeno na obr. 7.16, lze použít i volání jádra `send` a `recv`, což jsou varianty zápisu a čtení pro použití pouze nad schránkou. Nejobecnější volání jádra pro přenos dat sítí s možností zapouzdření jsou `sendmsg` a `recvmsg`.

Použití datagramů v síťovém přenosu ukazuje obr. 7.17.

Na rozdíl od stálého spojení se situace zjednoduší, protože proces serveru čte všechny požadavky ze sítě. Každý přitom zpracuje bez vytváření dětského procesu. Po vytvoření schránky (pomocí `socket`), navázání schránky na adresu (`bind`) a rezervování fronty (`listen`) se server zablokuje do čekání na požadavek pomocí volání jádra `recvfrom`, kterým při každém požadavku, tj. `sendto` klientu, zadává



Obr. 7.17 Stavy procesů síťové komunikace v BSD, bez stálého spojení

v parametru adresu (č. portu a adresu IP). Server i klient bez stálého spojení mohou také podobným způsobem použít `sendmsg` a `recvmsg`, u kterých je možné i zadávat u každého přenosu dat síť opětovně spojovací adresu.

Programování schránek vyžaduje podobně jako TLI dobrou průpravu programátora a v této knize se věnujeme pouze principům. Čtenáře jistě napadnou situace, které je nutno při programování řešit a na které uvedená volání jádra zatím nestačí. Důležitá volání jádra, která lze při programování schránek používat jsou např. `fcntl` a `ioctl` a speciálně `setsockopt` a `getsockopt`, která nastavují různé vlastnosti schránek. Programátor sítě nalezne mnohé poučení a podrobný rozbor programování sítí v UNIXu v [Stev90]. Schránky nejsou součástí SVID.

### 7.3.2 TLI

Transport Layer Interface (TLI) je prostředí programátora sítí v UNIX SYSTEM V. AT&T tak realizovala transportní vrstvu modelu OSI v operačním systému UNIX. TLI je knihovnou funkcí, nikoliv seznamem volání jádra. Volání jádra, která funkce TLI používají při realizaci síťových služeb, jsme poznali v kap. 6 u popisu technologie STREAMS. Definici TLI a její úplný popis lze studovat v SVID.

Základní termíny, které popis TLI uvádí, jsou *cílový bod transportu* (transport endpoint) a *správce transportu* (transport provider). Cílový bod transportu (nebo přenosu) je poloviční asociace, jde o analogii schránky z Berkeley. Správce transportu je prostředí, které procesům umožňuje provádět přenos dat mezi jednotlivými uzly sítě. Vzhledem k obecné možnosti (pomocí STREAMS) implementace různých protokolů mohou být v uzlu implementovány různé způsoby přenosu, tj. může být k dispozici několik správců transportu (např. pro protokoly XNS jeden, pro TCP/IP jiný atd.). TLI přitom není správce transportu, TLI různé správce transportu, které jsou v jádru implementovány, zpřístupňuje procesům. TLI je tedy souhrn funkcí, pomocí kterých proces využívá různé správce transportu. Vzhledem k tomu jsou formální definice např. pro cílový bod transportu obecnější než jsme poznali u schránek. Toto větší zobecnění přineslo zpočátku řadu obtíží, se kterými se tvůrci TLI museli potýkat, takže přestože byla transportní vrstva TLI uvedena jako součást UNIXu již v r. 1986 (ale několik let po vzniku schránek!), provozní stability bylo dosaženo teprve koncem 80. let. Původní velkorysá myšlenka obecného pojetí sítí v UNIXu tak byla realizována.

TLI je navrženo a podporuje služby pro dva základní režimy komunikace síťových procesů v technologii klient - server. Jednak je to služba režimu stálého spojení (connection-mode service) a dále je to služba režimu bez stálého spojení (connectionless mode service).

Režim stálého spojení je provozován ve čtyřech základních fázích: inicializační (deinicializační), navázání spojení, přenosu dat a uvolnění spojení. Režim bez stálého spojení je provozován pouze ve dvou fázích: inicializační (deinicializační) a ve fázi přenosu dat.<sup>4</sup>

Při používání funkcí TLI může proces pracovat synchronně (čeká zablokován na výskyt události, např. server očekává oslovení klientem) nebo asynchronně (proces je na událost upozorněn, přitom není zablokován a věnuje se jiné činnosti). Implicitní je synchronní režim. Proces v okamžiku zpřístupnění správce transportu (při použití funkce `t_open`) může použít příznak `O_NDELAY` nebo `O_NONBLOCK` a pracovat asynchronně (v průběhu práce se správcem transportu může měnit režim synchronní na asynchronní a zpět použitím uvedených příznaků nebo jejich negace ve volání jádra `fcntl`). V případě asynchronního přenosu může nastat až sedm různých událostí, na které je proces upozorněn (viz také

signál SIGPOLL). Proces událost analyzuje pomocí funkce `t_look`, jejíž návratovou hodnotou v případě události je jedna z možností:

<code>T_LISTEN</code>	klient má požadavek na spojení (pouze pro stálé spojení),
<code>T_CONNECT</code>	spojení bylo navázáno (pouze pro stálé spojení),
<code>T_DATA</code>	přišla data,
<code>T_EXDATA</code>	data byla odeslána (pouze pro stálé spojení),
<code>T_DISCONNECT</code>	spojení bylo ukončeno (pouze pro stálé spojení),
<code>T_ORDREL</code>	spojení bylo řádně ukončeno (pouze pro stálé spojení, při jeho uvolnění),
<code>T_UDERR</code>	nastala chyba v posledním odeslání dat (pouze bez stálého spojení).

Chyby, které nastanou při práci síťových procesů, lze sledovat pomocí mechanismu návratové hodnoty – 1, kterou vrací každá funkce TLI, pokud došlo k chybě. Současně TLI naplní obsah proměnné `t_errno` (proces ji definuje `extern int t_errno`) typem chyby, která nastala (pozor, obsah proměnné není nulován v případě úspěchu). Celočíselné vyjádření typu chyby lze převádět na textový řetězec funkcí `t_error`. Vzhledem k tomu, že TLI je knihovna funkcí, je možné také analyzovat chybu síťového spojení na úrovni volání jádra. Pokud se objeví v `t_error` chyba `TSYSERR`, volání jádra, které selhalo, uložilo typ chyby do obsahu proměnné `errno` (viz kap. 1).

Obr. 7.18 ukazuje stavy procesů klientu a serveru v síťovém spojení pro režim stálého spojení, zatímco obr. 7.19 pro režim bez stálého spojení.

Proces server vytváří koncový bod transportu funkcí

```
#include <tiuser.h>
#include <fcntl.h>
```

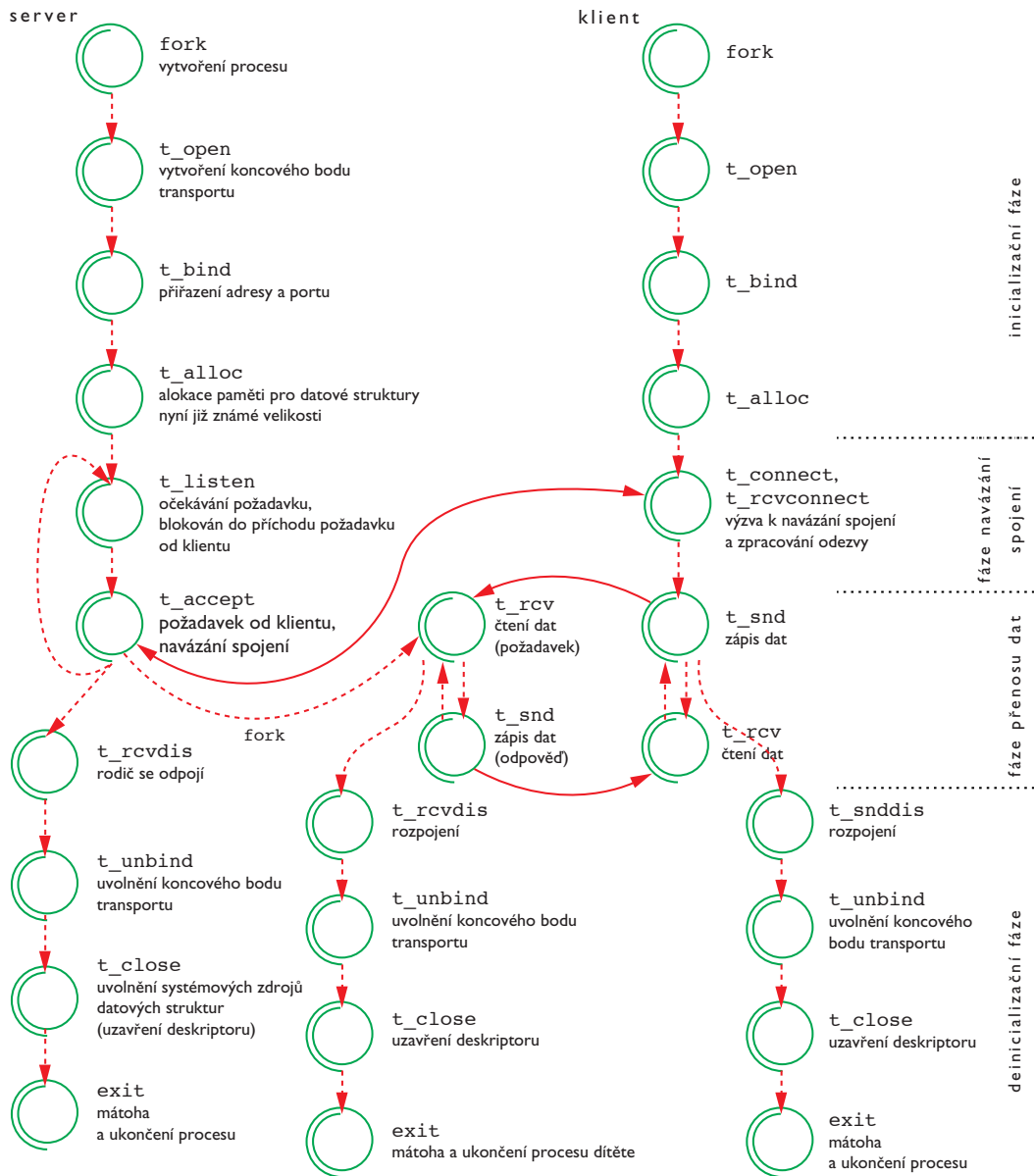
```
int t_open(const char *path, int oflag, struct t_info *info);
```

což je otevření souboru `path` požadovaného správce transportu (např. `/dev/tcp`, `/dev/udp`, `/dev/ip`, ale i `/dev/iso_cots`). Dojde tak k vytvoření koncového bodu transportu. Funkce vrací deskriptor souboru (v dalších funkcích našeho textu označovaný jako `fd`), tedy odkaz do tabulky otevřených souborů procesu. V argumentu `oflag` zadáváme způsob otevírání (synchronní `O_NDELAY` nebo asynchronní `O_NONBLOCK`, v logickém součtu s `O_RDWR`). Funkce po úspěšném provedení naplní obsah struktury `info` informacemi o správci transportu, jako je např. velikost adresy, velikost jednotky přenosu dat, typy nabízených služeb atd. Na základě vybraného správce transportu funkcí `t_open` požádáme o konkrétní přiřazení koncového bodu transportu. Pro TCP/IP je to adresa IP a číslo portu. Stane se tak funkcí

```
int t_bind(int fd, struct t_bind *req, struct t_bind *ret);
```

kde struktura `t_bind` obsahuje prvky síťové adresy. V parametru `req` můžeme požadovat přidělení určité adresy, nebo její hodnotu ponechat na správci transportu. Obsah parametru `ret` je pak naplněn správcem transportu skutečnou hodnotou adresy. `t_bind` je velmi obecná, protože TLI je vrstva transportní podle ISO a podporuje tak zcela obecný princip práce s nižšími vrstvami. Syntaxe i použití funkce tak musí být zcela nezávislé na použitém přenosovém protokolu. Funkce `t_alloc`, k níž je doplňková `t_free`, je rozšiřující funkce, kterou alokujeme potřebnou paměť pro datové struktury použitého správce transportu (velikost paměti, kterou alokujeme, se dovídáme ve struktuře `info` funkce





Obr. 7.18 Stavy procesů síťové komunikace TLI, stálé spojení

`t_open`). TLI neprovádí tuto alokaci dynamicky sama a nechává tak programátorovi větší volnost práce s pamětí.

Fáze navázání spojení je zajištěna u stálého spojení především funkcí `t_listen`, kterou používá server pro očekávání klientu. Ten se ozývá funkcí `t_connect`. Funkcí

```
int t_listen(int fd, struct t_call *call);
```

získá server ve struktuře `call` informace o protilehlém koncovém bodu transportu a

```
int t_connect(int fd, struct t_call *sndcall,
              struct t_call *rcvcall);
```

zadáva tento koncový bod transportu v `sndcall`, zatímco v `rcvcall` získá klient odpovídající bod transportu serveru. Po úspěchu `t_listen` server funkcí

```
int t_accept(int fd, int resfd, struct t_call *call);
```

požádá o deskriptor souboru `resfd` pro nový koncový bod transportu a správce transportu jej na základě obsahu `call` (známého z funkce `t_listen`) přiřadí. Vznik nového koncového bodu transportu pro totéž spojení je důležitý, vytváříme-li pro obsluhu síťového požadavku nový proces (náš případ), protože server tak může očekávat požadavky na starém koncovém bodu transportu (deskriptor `fd`), zatímco klient nadále komunikuje s novým bodem transportu (deskriptor souboru `resfd`). Pokud nebudeme programovat server s ošetřením požadavků procesem dítěte (tzv. konkurentní server), nepoužijeme `t_accept` (náš případ na obr. 7.19, jde o tzv. iterativní server).

Data jsou přenášena funkcí

```
int t_snd(int fd, const void *buf, unsigned nbytes, int flags);
```

pro zápis (chcete-li odeslání) dat a

```
int t_rcv(int fd, const void *buf, unsigned nbytes, int flags);
```

pro čtení (nebo chcete-li příjem) dat. Význam a použití argumentů `fd`, `buf` a `nbytes` je analogické s voláními jádra `read` a `write` pro práci se soubory. Příznak `flags` navíc může obsahovat informaci `T_EXPEDITED` pro práci s daty mimo rozsah nebo `T_MORE` jako informace o konci dat.

Funkcí

```
int t_snddis(int fd, struct t_call *call);
```

proces buďto uzavře již navázané spojení nebo spojení odmítne. Proč k tomu dochází, uvádí ve struktuře `call`, jejíž obsah může protilehlý proces získat funkcí

```
int t_rcvdis(int fd, struct t_discon *discon);
```

v obsahu struktury `discon`. Takový způsob ukončení spojení je tzv. *tvrdý* (abortive). Používá se také *regulérní* (orderly), který je na rozdíl od tvrdého ukončení pro správce transportu nepovinný (nemusí jej implementovat). Pro regulérní zrušení spojení jsou definovány funkce

```
int t_sndrel(int fd);
```

```
int t_rcvrel(int fd);
```

Regulérní zrušení zaručí na rozdíl od tvrdého zrušení obsluhu prozatím nedoručených dat. Jak ale vidíte na obou obrázcích, programátoři jej obvykle nepoužívají. Konečně

```
int t_unbind(int fd);
```

zruší koncový bod transportu a

```
int t_close(int fd);
```

uvolní deskriptor souboru.

Pro případ bez stálého spojení je na obr. 7.19 patrná odlišnost v použití funkcí ve fázi přenosu dat. Funkce

```
int t_rcvdata(int fd, struct t_unitdata *unitdata, int *flags);
```

přijímá data ze sítě podle deskriptoru `fd`. Data jsou přijata do položek struktury `unitdata`, kde si kromě vlastních dat můžeme také přečíst, odkud data přišla (adresu a port). V příznaku `flags`, je-li nastaven na pravdivou hodnotu, je sděleno, že data ještě nebyla ukončena. Naopak funkce

```
int t_sndudata(int fd, struct t_unitdata *unitdata);
```

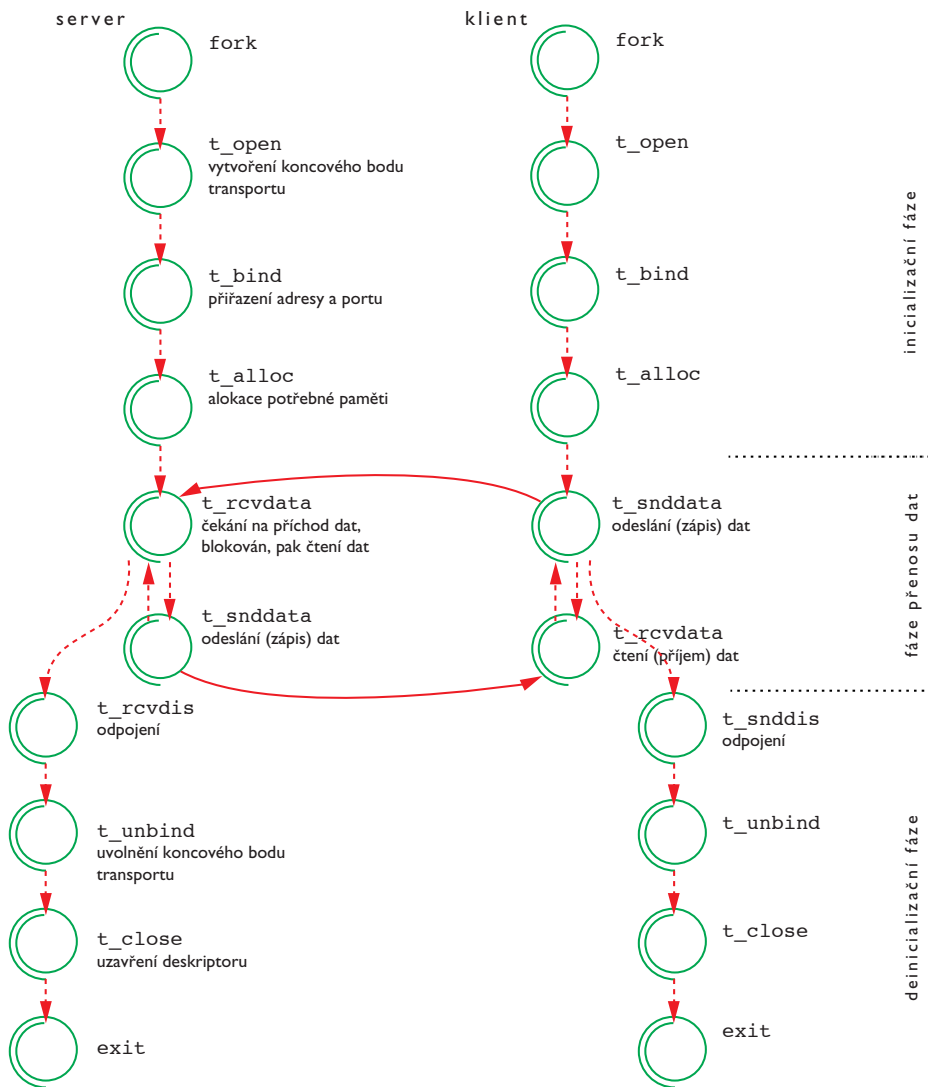
data do sítě vysílá. Programátor je zde odpovědný za obsah struktury `unitdata`, kde odkazuje na data požadovaná pro zápis do sítě.

Uvádět příklady programů procesů síťových spojení znamená popsat dalších několik stran textu, který určitě nebude zcela korektní a navíc přesahuje rozsah, který jsem této tematické kapitole vyčlenil. Čtenář, který by rád v problematice programování sítí v UNIXu pokračoval, necht' studuje publikaci [Stev90], kde lze krásu síťové komunikace procesů pochopit hlouběji. Při sestavování programu síťové aplikace (příkazem `ld`), který využívá knihovnu TLI, programátor používá volbu `-lnsl` (`ns1` je zkratka pro Network Services Library, knihovna síťových služeb), schránky jsou součástí jádra, ale mnohé implementace SYSTEM V je emulují pomocí TLI. V takovém případě je nutné poradit se s provozní dokumentací, kde je jméno knihovny schránek uvedeno.

Důležitý je však ještě aspekt transparentního používání. Znamená to, že síťové aplikace v UNIXu lze používat navzájem proti sobě, ať už mají uzly své síťové aplikace programovány v prostředí schránek nebo TLI. Na uvedených příkladech obr. 7.16 - 7.19 lze tedy vzájemně vyměňovat procesy klientů a serverů. Implementace v jádru a nižší vrstvy síťového spojení totiž rozdíl mezi schránkou a koncovým bodem transportu ošetřil.

### 7.3.3 Síťový programovací jazyk RPC

Na obr. 7.8 jsme uvedli označování (balení) dat nejenom u přenosových protokolů, ale také u vrstvy procesové. Jak je naznačeno, data zapisovaná procesem do jádra obsahují také určité záhlaví, které je doplněno aplikací. Pro zobecnění práce síťových procesů vznikl způsob reprezentace přenášených dat, který je označován XDR (eXternal Data Representation, externí reprezentace dat). Původně byl XDR navržen jako součást prostředí síťového programování v RPC (viz dál) firmy Sun Microsystems (při programování NFS, vzdáleného sdílení disků). Jeho původní definice je v RFC 1014 z r. 1987, poslední dokument v RFC 1832 z r. 1995. XDR je dnes definován v SVID a AT&T jej specifikuje jako realizaci prezentační vrstvy síťového modelu OSI podle ISO. UNIX SYSTEM V používá XDR při programování v RPC a pro realizaci RFS (tj. vzdálené sdílení souborů, analogie NFS). RPC i RFS je rovněž definováno v SVID.



Obr. 7.19 Stavy procesů síťové komunikace TLI, bez stálého spojení

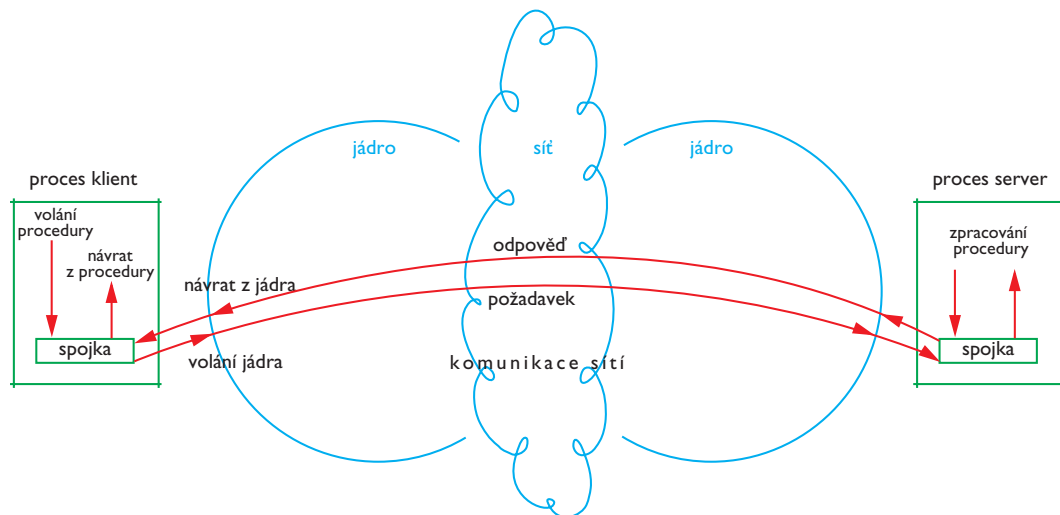
U XDR najdeme přibližnou analogii s X.409 (ISO Abstract Syntax Notation). Hlavní rozdíl je v pojetí datových typů. U XDR je implicitní, X.409 používá striktně explicitní typy. Znamená to, že v X.409 je u každého přenášeného datového pole vždy uveden jeho typ, kdežto XDR má implicitní datové typy, ve kterých neoznačená data uvažuje. XDR není programovacím jazykem (tím se stává teprve při použití např. s RCP). Jedná se pouze o definici dat. Jejich správnou reprezentaci v různých implementacích UNIXu nebo v různých operačních systémech zajistí právě vždy konkrétní implementace XDR.

Problémy, na které síťové aplikace narážejí a o kterých jsme se zatím nezmiňovali, jsou totiž např. v pořadí slabik přenášených dat. Při reprezentaci dat na více než jednom byte používají operační systémy různé pořadí těchto bytů. Typické jsou u 16 bitových hodnot rozdíly mezi pojetím horní a dolní byte. Způsob pořadí je označován jako *little endian* (malý endian), tj. první je dolní bytu, a *big endian* (velký endian), tj. první je horní byte. Situace se pochopitelně dále komplikuje u 32 bitových a delších hodnot. Způsob ukládání dat je dán architekturou hardwaru, tj. výrobcem stroje, a přenositelný software se s tím musí pouze vyrovnat. Z tohoto pohledu implementace XDR uvažuje např. všechna data jako 32 bitové hodnoty a u systémů, které nevyužijí celý rozsah, je použita pouze část. Po přenosu sítí je obsah 32 bitové hodnoty reprezentován podle uvažovaného pořadí slabik. Problém vzájemného porozumění síťových aplikací je také s tím související kódování nejen binárních, ale i znakových hodnot dat, tj. používání ASCII nebo EBCDIC. XDR je nezávislé na použití přenosového protokolu. Data jsou definována stejným způsobem jak pro stálé spojení, tak pro přenos dat bez stálého spojení.

Syntaxe XDR je podobná jazyku C. Čtenář by se měl orientovat v provozní dokumentaci, programátor znalý jazyka C nebude mít problémy. Jak vyplývá z předchozího textu, jde pouze o popis dat, tj. XDR obsahuje pouze datové struktury. Klíčová (tj. vyhrazená) slova jsou `bool`, `char`, `case`, `const`, `default`, `double`, `enum`, `float`, `hyper`, `opaque`, `string`, `struct`, `switch`, `typedef`, `union`, `unsigned` a `void`. Gramatika je jednoduchá, umožňuje vyjádření různě strukturovaných typů dat. Pro manipulaci s daty je pak používána knihovna funkcí XDR. Jednotlivé funkce jsou pojmenovány s předponou `xdr` (např. `xdr_create`, `xdr_array`, `xdr_bytes`, `xdr_double` atd.). Při sestavování programem `ld` pak programátor používá volbu `-lrpcsvc`, protože XDR je považováno za součást RPC.

XDR je označován jako standard pro obecnou definici dat. RPC (Remote Procedure Call, vzdálené volání procedury) je algoritmickou částí prvního síťového jazyka UNIXu, ale není součástí tohoto standardu. Doporučení SVID definuje RPC jako síťový programovací jazyk pro UNIX SYSTEM V. Výchozí popis RPC tak, jak byl vyvinut firmou Sun Microsystems, je také uveden v dokumentu RFC 1057 z r. 1988. Jako standard je definován v RFC 2203 z r. 1997. I RPC je z pohledu modelu OSI zařazováno do šesté, tj. prezentační vrstvy.

Princip RPC vychází z pojmu *spojka* (stub). Při programování je běžné využívání volání procedur. Proces, který běží v operačním systému, volá procedury pro zajištění určité opakující se části kódu. Termín, který se dnes používá při realizaci mechanismu volání procedury v rámci uzlu, je místní volání procedury (Local Procedure Call, LPC). Je zajištěn místními prostředky operačního systému, které jsou v principu synchronní povahy (pracují v rámci téhož jádra nebo dokonce jednoho procesu). Vzdálené volání procedury je mechanismus zajištění provedení požadované procedury operačním systémem, která je ovšem provedena v jiném uzlu sítě. Proces klient tak požaduje obsluhu serverem v uzlu sítě, jak je u práce v síti v UNIXu běžné. Situaci ukazuje obr. 7.20.



Obr. 7.20 Vzdálené volání procedury

Volání procedury klientu předává řízení spojce. Ta převezme jméno procedury a její parametry, tyto hodnoty zabalí do formy síťového požadavku a pomocí volání jádra aktivuje zápis těchto dat do sítě. Data jsou přenesena sítí (vybraným protokolem pro stálé nebo bez stálého spojení). V odpovídajícím uzlu sítě na ně čeká proces server, který pomocí své spojky data z jádra přečte. Získaná data zpracuje a výsledek předá své spojce, která pomocí síťových volání jádra zapíše výsledky do sítě, odkud je na druhé straně čte spojka klientu a předává je jako návratové hodnoty volané procedury. Práce procesu klient je tak transparentní z pohledu volání procedury, protože jak místní, tak vzdálené volání má pro něj shodný způsob ovládání. RPC přitom musí ošetřit průchod dat sítí tak, aby proces nemusel opakovaně žádat o přenos nebo zjišťovat, zda byla data doručena sítí bezchybně. Ke správnému chování procesů klientu a serveru je nutná podpora správce systému, který musí v uzlu s procesem serveru zajistit probuzení (nebo start) odpovídajícího procesu na daném portu. Rovněž tak na straně klientu musí být řečeno, který uzel obsahuje protějšek síťové komunikace. Situace je ošetřena zcela standardním způsobem pomocí superserveru **inetd** a rozšířením tabulky `/etc/inetd.conf` nebo trvalou přítomností odpovídajících procesů serverů v systému, které se startují v době zavádění v průběhu práce procesu **init**, což uvedeme v následujícím čl. 7.4. SVID definuje formát tabulky potřebný pro definici serverů při práci klientů na odpovídajících portech. Jméno souboru s takovou tabulkou však neuvádí. Rovněž tak pro čísla portů používá termín *číslo programu RPC* (RPC program number).

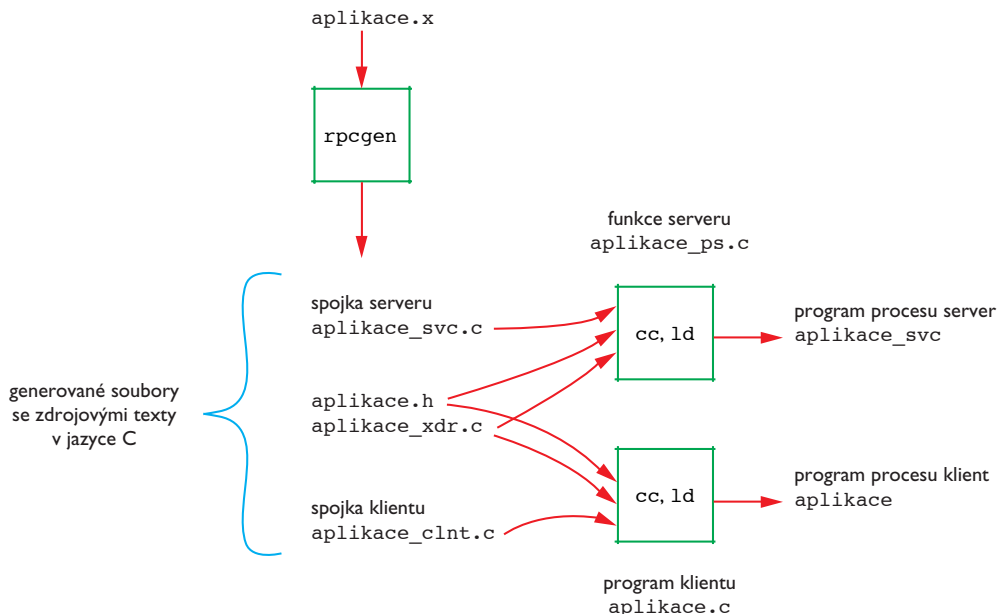
Prostředky programátora při vytváření aplikace RPC jsou uvedeny na obr. 7.21.

Klíčový je program **rpcgen**, který na základě zdrojového textu v souboru s příponou `.x` (aplikace `.x`, zdrojový text v XDR a RPC) generuje soubory s příponami `_svc.c` (spojka serveru), `_clnt.c` (spojka klientu), `_xdr.c` (definice dat) a `.h` (hlavičkový soubor s definicemi pro klient i server). Generované soubory jsou zdrojové texty pro jazyk C. V souboru s příponou

.x popisujeme data síťové spolupráce a jména procedur s jejich parametry, které představují typ a rozsah dat zasílaných sítí. Algoritmická část serveru (tj. funkce, které spojíka serveru ve vzdáleném uzlu využívá) je obsahem souboru, který programátor pojmenuje dle své vůle (na obr. 7.21 jsme jej nazvali `aplikace_ps.c`). Důležité však je, aby pokrýval všechny externí symboly použité ve spojíce serveru (tj. definované v .x). Funkci `main` programu pro server obsahuje generovaný `_svc.c`. Program procesu klient (který využívá procedury definované v .x) je rovněž dodaný programátorem (na obr. 7.21 pod jménem `aplikace.c`) a musí obsahovat funkci `main`. `cc`, kompilátor jazyka C, a sestavující program `ld` zpracují výsledky práce programátora a programu `rpcgen` a vytvoří program síťové aplikace pro klient (u nás `aplikace`) a server (`aplikace_svc`). Po instalaci, spuštění serveru ve vzdáleném uzlu a po oslovení uzlu klientem může být aplikace testována.

Syntaxe zápisu procedur v jazyce RPC je shodná s XDR. Navíc obsahuje část definic s klíčovým slovem `program-def`, kde programátor definuje jméno programu a jména jeho verzí. Definice verze je uvedena klíčovým slovem `version-def`. V rámci definice vyjmenujeme jména používaných procedur. Procedury definujeme klíčovým slovem `procedure-def` a zde zadáváme jména procedur a definice jejich parametrů. Programátor znalý jazyka C se dokáže rychle orientovat.

V jazyce RPC (a XDR) je např. programována aplikace vzdáleného přístupu k souborům (NFS nebo RFS, viz následující čl. 7.4). Uzel, který zveřejňuje svazky do sítě, startuje procesy serverů, které zajišťují odezvu klientům z jiných uzlů sítě. Po provedení příkazu `mount` v uzlu s klienty jádro uzlu klientů zpřístupní zveřejněný svazek pro použití libovolným procesem běžně používajícím volání jádra čtení



Obr. 7.21 Programování v RPC

a zápisu do souborů (`read`, `write`), takže aplikace klientu NFS tak není programována typicky, jak bylo uvedeno, protože obsluhu provede skrytě jádro. Klientem, který je programován v RPC, je zde systémový program `mount`.

## 7.4 Síťové aplikace a jejich provoz

Síťová komunikace podle modelu OSI je pro uživatele představována 7. aplikační vrstvou. V době aktivace a využívání síťové aplikace je podporována nižšími vrstvami, tak jak jsme uvedli v předchozí části kapitoly. Síťová aplikace musí být také správně nastavena a provozována, tj. musí pracovat na správném čísle portu, využívat odpovídající přenosové a komunikační protokoly a správně se dorozumět se svým protějškem v rámci komunikace typu klient - server. Správné nastavení provozu síťové aplikace je otázkou správného obsahu odpovídajících tabulek, které využívá především server, tj. procesy typu démon, které pracují pro zajištění síťových požadavků klientů. Obsahy tabulek síťových aplikací, start, stop a další komunikace s démony je uváděna jako součást 6. prezentační vrstvy modelu OSI, přestože vrstvu prezentační lze striktně v UNIXu přiřadit pouze prostředku definice dat XDR.

Jak vyplývá z předchozího, proces může oslovit různé vrstvy síťové komunikace, které zajišťuje jádro. **ifconfig**, který přiřazuje síťovému adaptéru adresu IP, musí mít možnost komunikovat se síťovou vrstvou. Testovací program správného směrování **ping** nebo další směrovací procesy (**routed**, **gated** atd.) oslovují také síťovou vrstvu a na zásobníku protokolů pracují pouze s IP (případně ICMP atp.). Přestože se jedná o výjimky, které navíc podléhají privilegovanému přístupu, jsou tyto procesy síťového provozu součástí vrstvy procesové (v OSI 6. nebo i 7. vrstvy). Naopak aplikace **telnet**, **ftp** nebo NFS jsou aplikace 7. vrstvy, striktně využívající zásobník protokolů a systémové nastavení 6. vrstvy. V této trochu vágní terminologii síťové aplikace budeme sledovat především provozní charakter sítí v UNIXu. Síťová aplikace pro nás bude skupina procesů. Jaký proces a s jakou vrstvou pracuje sice budeme uvádět, ale vyhneme se jednoznačnému zařazování.

Síťové aplikace, podobně jako procesy obecně, tedy můžeme rozdělit na systémové a uživatelské. Základní systémovou síťovou aplikací je DNS (Domain Name System, způsob pojmenování domén), která provádí náhradu číselného označování uzlů adresami IP na jejich označování jmény. Systémová síťová aplikace je také tzv. síťový superserver **inetd**, což je démon, který přebírá síťové požadavky a konkurentním způsobem (startuje děti a předává jim síťový požadavek) zajišťuje start správných procesů serverů. Systémová síťová aplikace je také démon **sendmail**, který zajišťuje provoz elektronické pošty. Systémové síťové aplikace jsou démony **nfsd** pro provoz sdílení svazků prostřednictvím NFS nebo démony **httpd** pro zajištění služeb serveru WWW. Postupně se těmito a dalšími systémovými síťovými aplikacemi budeme z pohledu jejich principů zabývat v dalším textu tohoto článku. Síťové aplikace strany klientu budeme popisovat, pouze bude-li to pro princip provozu sítě nezbytné. Jinak formáty uživatelských příkazů **telnet**, **ftp**, **rlogin**, **rnp**, **elm**, **netscape** a dalších nalezne uživatel v provozní dokumentaci.

Více než jakýmkoliv jiným způsobem je operační systém a data, která obsahuje, vystaven rizikům zneužití a (ať už vědomému nebo nevědomému) paralyzování provozu výpočetního systému. Při práci se síťovými aplikacemi je proto nutné dobře sledovat bezpečnostní hlediska provozu. Citlivá místa síťové komunikace z hlediska bezpečnosti a jejich ošetření různými způsoby (organizačními, nebo softwarovými) probereme v samostatné kap. 9.



### 7.4.1 Pojmenování uzlů sítě (NIC, DNS)

Jméno operačního systému lze v UNIXu získat voláním jádra `uname`. Jak SVID, tak POSIX jej definují, stejně tak jemu odpovídající příkaz `uname`.

```
#include <sys/utsname.h>
```

```
int uname(struct utsname *name);
```

Struktura `name` má přitom definovány všechny položky jako pole typu `char`. Položka `sysname` po návratu z jádra obsahuje jméno operačního systému, `nodename` jméno jako označení systému pro síťovou komunikaci, `release` a `version` jako podrobnější informace o systému (vydání a verze) a konečně v `machine` je textové označení hardwaru, na kterém operační systém běží. Příkaz `uname` vypisuje uvedené informace na standardní výstup, a to podle toho, jaká volba byla použita (např. `-a` znamená výpis všech uvedených textů). Přestože je `uname` i `uname` součástí každého systému, používá se pouze pro informační účely. Není totiž definován způsob, jak lze hodnoty získané pomocí `uname` jakkoliv měnit. Standard zdůvodňuje přítomnost `uname` historicky a komentuje, že textové hodnoty identifikace operačního systému bylo možné (a lze) měnit pouze při kompilaci a stavbě jádra. Jak uvidíme v kap. 10, tato operace není jednoduchá a hlavně je časově náročná. Systémy BSD proto zavedly odpovídající volání jádra `gethostname` a `gethostid` pro získání jmenové a číselné identifikace operačního systému a `sethostname` a `sethostid` pro jejich změnu. Ani SVID ani POSIX je neuvádí. Jejich formát v systémech nejen BSD je

```
#include <unistd.h>
```

```
int gethostname(char *name, int namelen);
```

```
int sethostname(char *name, int namelen);
```

```
int gethostid(void);
```

```
int sethostid(int hostid);
```

kde `name` je jméno operačního systému a `namelen` je délka tohoto jména. `sethostname` je pouze pro privilegovaný proces. Délka jména operačního systému bývá omezena parametrem jádra (viz `<sys/param.h>`) obvykle hodnotou 64. Číselná identifikace `inhostid` je nastavována na hodnotu některé adresy IP. `sethostid` používá opět pouze privilegovaný proces a běžně je číselná identifikace nastavována při startu operačního systému. Tato číselná hodnota je však pouze označením jádra a nemá nijak definovaný vztah k určitému síťovému adaptéru uzlu. Adaptéru se adresy IP přidělují pomocí `ifconfig`, jak jsme uvedli u popisu IP. Dochází přitom k zápisu do vnitřních tabulek běžícího jádra.

Jméno uzlu, tj. jméno operačního systému pro komunikaci v síti, nastavuje správce systému pomocí příkazu `hostname`. Jméno zadává v parametru. Bez parametru je příkaz použitelný i pro neprivilegovaného uživatele, pak vypisuje jméno uzlu na standardní výstup. Příkaz `hostname` vznikl v systémech BSD a pro získání nebo nastavení jména uzlu používá volání jádra `sethostname` a `gethostname`, např.:

```
# hostname freak
```

```
# hostname
```

```
freak
```

#

Nastavení jméno uzlu (tj. operačního systému) lze uplatnit v síťové komunikaci, ale jeho nastavení pomocí **hostname** samo o sobě zveřejnění neznamená. Jméno musí být ještě uvedeno v tabulkách pojmenování uzlů sítě. Jméno operačního systému musí být totiž spojeno s adresou IP síťového adaptéru a veřejně oznámeno. Dnes jsou používány dva způsoby záměny adres IP za jména uzlů. Jednak je to starší způsob používání tabulek NIC a jednak použití distribuovaného systému jmen uzlů DNS.

Implementace IP na místě síťové vrstvy nám umožňuje označovat uzly číselnými adresami IP tak, že jsou uzly jednoznačně rozpoznatelné v rámci sítě nebo v sítích, ke kterým je místní síť připojena (např. v Internetu). Záměna adres IP za jména je tak pouhé přiřazování jmen adresám IP. Problém je takový, že toto přiřazování musí být jednotné v rámci celé sítě a všech sítí, ke kterým je místní síť připojena (např. v Internetu). Záměna adres IP za jména tak musí být viditelná ve všech uzlech sítě a sítě sítí. Nejhorší případ je ten, kdy evidujeme v každém uzlu sítě tabulku odpovídajících jmen uzlů místní sítě a některých dalších uzlů ostatních sítí. Připojení dalšího uzlu k síti pak znamená rozšířit tabulky ve všech uzlech o tutéž položku. Takové chování vykazuje systém NIC, který správnost obsahu tabulek zajišťuje také ve spojení s NIS (dříve Yellow Pages). Důmyslnější systém pro označování uzlů jmény vznikl v systémech BSD a byl pojmenován DNS. Byl zaveden systém tzv. primárních serverů, které obsahují informace o všech uzlech určité oblasti (tzv. domény). Oblasti jsou hierarchicky uspořádány, např. oblast pro Českou republiku má označení **cz**. Doménu **cz** eviduje server, který obsahuje odkazy na další servery částí České republiky, např. **cuni.cz** je oblast (tzv. doména 2. řádu), která určuje lokalitu Karlovy univerzity v Praze. Doménu **cuni.cz** udržuje server, který eviduje další domény (3. řádu), které mohou znamenat jednotlivé fakulty, např. **mff.cuni.cz** atd. Server určité domény, je-li osloven dotazem na doménu, za kterou nezodpovídá, předává požadavek serveru vyšší domény. Vyřízené požadavky si dále servery domén udržují ve vyrovnávacích pamětech (cache), takže dotazy sítí se tak minimalizují. Změna v tabulkách serverů domén se objeví až po vynuceném zapomenutí vyrovnávací paměti serverů, k čemuž dochází automaticky u každého serveru přibližně dvakrát denně. DNS je důležitý především pro komunikaci rozsáhlých sítí (jako je Internet), kdy správce systému nemůže předpokládat, jaký uzel uživatel v následujícím okamžiku osloví. Přitom používání adresy IP přímo na místě jména uzlu sice bude fungovat, ale nemá uživatelskou budoucnost.

Pro získání adresy IP uzlu na základě znalosti jeho jména (a opačně) používají síťové aplikace funkce **gethostbyname** a **gethostbyaddr**. Funkce pracují pro oba způsoby rozlišování uzlů, tj. jak NIC (za podpory NIS), tak DNS – záleží na systémovém nastavení.

NIC (DDN Network Information Center) vychází z definic jmen uzlů tabulky **/etc/hosts**, případně **/etc/networks** pro přiřazení jmen uzlů a sítí adresám IP. Tabulky jsou dodnes používány v každém uzlu. Jejich obsah je definice adres IP s ekvivalenty jmen, pod kterými na adresy IP můžeme odkazovat z jakékoliv síťové aplikace. Např. **/etc/hosts** může mít obsah

```
# Tabulka jmen uzlů odpovídajících adres IP
```

#

```
# Neodstraňujte v žádném případě následující řádek
```

```
127.0.0.1      localhost
193.86.115.101 freak      loghost
147.229.112.10 animal.fff.csuni.cz  animal
```

128.66.15.2      ora.com      ora

...

kde na každém řádku souboru jsou slovní ekvivalenty vždy jedné adresy IP. Uzel `freak` je definice našeho uzlu, nemá jiné přezdívky, zatímco uzel `animal` je přezdívka uzlu `animal.fff.csuni.cz`. Přezdívka totiž není `loghost`, ale označení, které využívají systémové procesy (démony) pro zaměření své pozornosti pro vedení statistiky přihlašování uživatelů na tomto uzlu. Uzel `localhost` má předznačenu poznámku, že nesmí být z tabulky vypuštěn. Adresa `127.0.0.1` (jak vyplývá z čl. 7.1) je rezervována pro odkaz uzlu na sebe sama. Jde o tzv. `loopback` (zpětná smyčka), kdy síťová vrstva IP síťový požadavek vrací ihned vyšším vrstvám jako právě přichází ze sítě. Řada systémových aplikací (např. grafické prostředí X) využívá jméno tohoto uzlu k síťovým odkazům sama na sebe. Správce používá `loopback` také pro test správného chování vrstev sítě nad síťovou vrstvou IP. NIC znamená nabídku rozsáhlé tabulky (se jménem `hosts.txt`), ze které lze zkonstruovat tabulky `/etc/hosts` a `/etc/networks`. Tabulku lze získat v lokalitě `nic.ddn.mil` Internetu, systémy BSD pro tento účel nabízejí příkaz `gettable`. Tabulka má formát popisu známých uzlů a sítí v Internetu a teprve použitím příkazu `htable` v pracovním adresáři vzniknou tabulky `hosts` a `networks`. Jejich přesunutím do adresáře `/etc` začínají být akceptovány síťovými aplikacemi.

Používat NIC se obecně nedoporučuje, protože je těžkopádné udržovat tabulky u větších sítí a zvláště v případě připojení k Internetu. Dnes je nejrozšířenější a nejpoužívanější způsob rozlišování jmen uzlů v Internetu služba DNS, která je automaticky modifikována při registraci nových lokalit nebo uzlů v celém světě. NIC nijak negarantuje rychlé změny v nabízené `hosts.txt` a ani tabulku nijak nedistribuuje. Přesto používání tabulek `/etc/hosts` a `/etc/networks` stále přežívá, protože je jednoduché a pro místní síť se stálým počtem uzlů nebo pro spojení několika místních sítí bez připojení k Internetu dostačující.

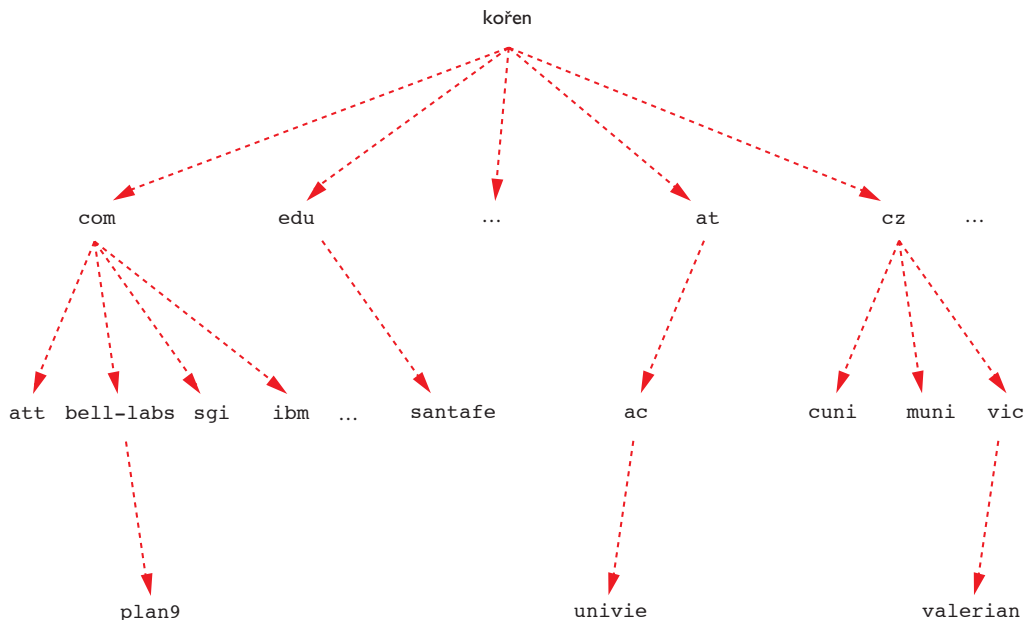
DNS (Domain Name System) je síťová služba pojmenování uzlů sítě TCP/IP, která vznikla v systémech BSD a implementuje ji především programový systém BIND (Berkeley Internet Name Domain). Rozděluje síť do tzv. domén, které mají vzájemnou hierarchii, jak jsme již naznačili. Každá doména je definována ve vybraném uzlu sítě. Informace o doménách vyšší a nižší úrovně se přitom zobrazuje pomocí démonu `named`. V době vzniku DNS (1983) byly Spojené státy rozděleny do domén prvního řádu podle předmětu zájmů práce v síti Internet. Při rozšíření Internetu za hranice USA došlo k přidělování domén prvního řádu podle geografie. Znamená to, že v tzv. výchozím (nebo kořenovém) serveru jsou registrovány např. domény `com` (komerční organizace), `edu` (vzdělávací instituce), `mil` (vojensství), `gov` (státní instituce), které se vztahují na síť především Spojených států, ale také domény `uk` (Spojené království), `fr` (Francie), `de` (Německo), `cz` (Česká republika) atd. Je registrována také doména `us` (Spojené státy), ale je málo používaná. Seznam všech domén prvního řádu lze získat z anonymního serveru FTP `ftp.wisc.edu` adresáře `connectivity_table`. Kořenový server registruje seznam takových domén a adresy IP serverů, které poskytují informace o dalším pokračování stromu domén. Hierarchickou strukturu DNS ukazuje obr. 7.22.

Např. doména první úrovně `cz` je registrována v kořenovém serveru. Definice domén její druhé úrovně, tj. např. `cuni` nebo `vic`, poskytuje server, na nějž je odkaz z kořenového serveru. Odkaz na doménu třetí úrovně, např. `vic`, je odkazem na adresu IP, která zajišťuje tuto oblast, tedy kde pracuje server `named`, který registruje další domény nižší úrovně. Nemusí to tak ale být vždy. Např. doména

`science` nemusí mít vlastní server a popis jejího dalšího pokračování je součástí serveru domény `cz`. Snaha organizací, které poskytují Internet, je přepouštět odpovědnost za doménu nižší úrovně vždy na majitele domény. Také pro majitele je to výhodnější, protože organizace pokračování jeho domény je plně v jeho kompetenci. Doména `valerian` označuje již konkrétní uzel. Odkaz na něj pak znamená použít zápis `valerian.vic.cz`, např.

`kitchen.santafe.edu` \$ **telnet** `valerian.vic.cz`

Síťová aplikace **telnet** požádá o identifikaci adresy IP podle nejbližšího serveru DNS. Dotaz může být vznesen z libovolného uzlu připojeného k Internetu (jak je naznačeno z `kitchen.santafe.edu`). Předpokládáme tedy, že náš uzel není součástí domény `cz` a dotaz na tuto doménu je uskutečněn v našem uzlu poprvé. Funkce `gethostbyname` oslovuje nejbližší server DNS, který spravuje doménu `santafe.edu`. Server nemá registrovanou doménu `cz` našeho dotazu, a proto nám odpoví odkazem na kořenový server. Ze znalosti adresy IP se serverem domény `cz` získáme adresu IP serveru domény `vic.cz`, který nám po oslovení odešle adresu IP uzlu `valerian`. Tato situace nastane také za předpokladu, že námi postupně oslovené servery jsou takovým požadavkem osloveny poprvé. Pokud totiž např. server domény `edu.santafe` rozpoznání domény `vic.cz` již pro nějakou aplikaci požadoval, server pro `edu.santafe` má výslednou adresu IP serveru domény `vic.cz` uloženu ve vyrovnávací paměti (cache) a odpověď bude proto rychlejší. Vyrovnávací paměti serverů DNS mají svou expirační dobu.



Obr. 7.22 Příklad stromu DNS

Méně používané dotazy se proto po čase ztrácejí a musí být obnovovány popsanou cestou. Evidence nové domény nebo její změna kdekoliv v síti je ale také podmíněna takovou expirací.

BIND je síťová aplikace, která implementuje DNS. Využívá stromu domén, který je distribuovaně poskytován jednotlivými servery, démony **named**. Říkáme také, že BIND má dvě části: *dotazovací* (tzv. resolver, tj. síťový klient) a *rozlišovací* (tzv. name server, server jmen, síťový server).

Uzel v síti může převádět jména na adresy IP (tj. rozlišovat) dotazem na nejbližší server DNS. Dotaz jakékoliv síťové aplikace uzlu je směrován definovanému serveru. Knihovni funkce (gethostbyname) klientu síťové aplikace, která vznesla dotaz, se v UNIXu orientuje podle toho, zda je součástí operačního systému správa domény. Pokud ano, pracuje v uzlu démon **named** a stačí nastavit jméno uzlu na jméno kvalifikované tímto serverem, např.

```
$ hostname valerian.vic.cz
```

v uzlu **valerian** postačuje funkce **gethostbyname** pro odkaz na správný server domény **vic.cz**. Je však nutné, aby uzel **valerian** byl také tímto způsobem definován v tabulkách serveru domény **vic.cz** (tzv. FQDN, fully qualified domain name). Dotaz DNS je tedy lokální. Říkáme také, že tato definice uzlu, který používá vlastní server DNS, je implicitní (default). Znamená to, že klient i server síťové aplikace DNS běží v tomtéž uzlu.

Pokud je v uzlu v UNIXu přítomen soubor **/etc/resolv.conf** (a v uzlu také neběží démon **named**), orientuje se funkce **gethostbyname** podle tabulky, kterou soubor obsahuje. Tabulka má jednoduchý tvar. Kromě komentářů, jejichž řádky začínají znakem **#**, obsahují její řádky buďto klíčové slovo **nameserver** a adresu IP uzlu se serverem DNS, nebo klíčové slovo **domain**, za kterým následuje jméno domény, např.

```
# Domain name resolver, file /etc/resolv.conf
#
domain mff.cuni.cz
nameserver 147.229.15.3
nameserver 147.220.1.2
```

Klíčovým slovem **domain** určuje plné jméno DNS dotazů na jména bez domény (tj. uzlů místní domény) a dvě adresy IP serverů DNS. Je obvyklé zadávat adresy dvě pro případ výpadku některého ze serverů.

Démon **named** je serverem DNS. Definice domény, kterou spravuje a poskytuje, je uložena v několika souborech. Klíčový je obsah souboru **/etc/named.boot**, ve kterém je definován typ serveru a podle něj jména dalších souborů, jejichž obsah definuje strukturu domény a její vazbu na okolní servery DNS. Typ serveru DNS může být *primární* (primary), *sekundární* (secondary) nebo *pouze zpřístupňující* (caching-only). Primární je plnohodnotný server, který v dalších tabulkách plně definuje doménu, označení uzlů v této doméně, případně odkazy na uzly, ve kterých doména pokračuje. Sekundární má také charakter autority domény, ale je bez vlastních definic, které obsahuje primární server, na nějž je napojen. Konečně pouze zpřístupňující je jen vyrovnávací paměť, která udržuje použité odkazy na ostatní servery. Tento typ serveru nemá charakter vlastní autority.

Soubory odkazované z **/etc/named.boot** jsou použity podle typu serveru. Soubor **named.ca** obsahuje informace pro inicializaci vyrovnávací paměti serveru a je uveden u každého typu serveru.

`named.hosts` definuje strukturu domény a převod jmen na adresy IP. Opačný postup, převod adres IP na jména domény, jsou definovány v souboru `named.rev`. Oba soubory jsou uvedeny pouze u primárního typu serveru. Obsahem souboru `named.local` je rozpoznání samotného uzlu při odkazu aplikací sama na sebe (tzv. loopback) a obvykle je použit u každého typu serveru.

Tabulka v `/etc/named.boot` může obsahovat klíčová slova, kterými začíná každý její řádek (komentáře jsou řádky začínající znakem `;`). Klíčové slovo **directory** uvádí jméno adresáře, který obsahuje odkazované soubory. **primary** a **secondary** určuje typ serveru, **cache** soubor s vyrovnávací pamětí. **forwarders** uvádí seznam serverů, na které jsou dotazy případně převáděny, a **slave** označuje schopnosti serveru pouze přesměrování na jiné servery DNS.

Pokud budeme vytvářet primární server, použijeme klíčové slovo **primary** několikrát k oznámení jmen souborů, ve kterých je konfigurace domény uvedena. Např.

```
; primary name server /etc/named.boot example
; (příklad souboru /etc/named.boot primárního serveru jmen)
;
directory                                /var/named
primary      vic.cz                      named.hosts
primary      184.229.IN-ADDR.ARPA       named.rev
primary      0.0.127.IN-ADDR.ARPA       named.local
cache        .                          named.ca
```

definujeme doménu `vic.cz`. Její další popis je přitom uložen v souborech adresáře `/var/named`. `named.hosts` definuje strukturu domény `vic.cz` na síťové adrese `184.229.0.0`, její jména a odpovídající adresy, případně definice domén navazujících řádů. Doména `IN-ADDR.ARPA` je ta, pro kterou je náš server primární. Druhý záznam `primary` pak označuje tento server jako opačný k doméně `184.229.IN-ADDR.ARPA` a informace pro něj jsou obsahem souboru `named.rev`. Třetí záznam `primary` je definice zpětné (loopback) domény na sebe samu, a to jako domény primární. Doména `IN-ADDR.ARPA` je zde zpětnou doménou, převádí jméno `localhost` na adresu `127.0.0.1` v souboru `named.local`. Poslední řádek (`cache`) je definice vyrovnávací paměti serveru do souboru `named.ca`, který bude za tímto účelem vytvořen a používán. Dva poslední záznamy uvedeného příkladu jsou při použití samostatně definicí serveru DNS typu pouze zpřístupňující (viz další text). Z příkladu vyplývá, že pro získání registrace domény primárního serveru je nutná registrace v doméně `IN-ADDR.ARPA`. Pokud vaši doménu zajišťuje poskytovatel Internetu, tuto registraci zajistí.

Obsah souborů vyjmenovaných v `/etc/named.boot` si nyní probereme v ukázkách. Všechny mají podobný formát. Validní záznam DNS v těchto tabulkách (tzv. DNS resource record nebo jen NS record) má obecný tvar:

```
[name] [ttl] IN type data
```

kde `name` je jméno uzlu nebo domény. Je-li ukončeno znakem `.`, je to absolutní jméno v rámci DNS, jinak je vztaženo k definované doméně. `ttl` (time-to-live) je počet vteřin, po který má být tento záznam platný ve vyrovnávacích pamětech ostatních serverů DNS. Obvykle je tato položka vynechána, protože se používá označení životnosti celého souboru (viz následující příklady). **IN** je klíčové slovo a určuje

typ záznamu (Internet DNS). Znamená to, že mohou být i jiné typy záznamu, ale v DNS jiný nepoužijeme. **type** je typ záznamu a na jeho místě lze použít

SOA	(start of Authority) je záznam pro záhlaví záznamů DNS,
NS	(name server) označuje uzel, který obsahuje server DNS,
A	(address) je konverze jména uzlu na adresu IP,
PTR	(pointer) opačně, konverze adresy IP na jméno uzlu,
MX	(mail exchange) je označení uzlu, ve kterém pracuje doručovací program pošty domény. V souvislosti s poštou lze použít také záznam typu MB (mailbox, definice poštovní schránky uživatele), MR (záměnné jméno poštovní schránky uživatele), MINFO (řídící informace seznamu pošty) a MG (definice členů poštovní skupiny uživatelů), jsou ale považovány za experimentální.
CNAME	(canonical name) je další jméno uzlu (alias),
HINFO	(host information) slovní popis hardwaru uzlu,
WKS	(advertise network services) zveřejňuje síťovou službu uzlu.

Konečně poslední položka záznamu je **data**, která obsahuje data podle typu záznamu (např. adresa IP). Ukažme si příklady. Klíčový soubor primárního serveru **named.hosts** pro definici domény, jejích uzlů a následných domén může obsahovat

```
; hosts to IP addresses records example named.hosts
; (příklad souboru named.hosts)
;
@      IN      SOA      valerian.vic.cz.  skoc.valerian.vic.cz.  (
                                97071001      ; verze obsahu souboru
                                43200         ; akt. sek. serveru dvakrát denně
                                3600          ; opakovaný přístup každou hodinu
                                3600000       ; expirace za 1000 hodin
                                2419200      ; délka života záznamů je 1 měsíc
                                )
; definice serverů domén a serverů elektronické pošty
      IN      NS      valerian.vic.cz.
      IN      NS      ns.bm.cesnet.cz.
      IN      MX      10 valerian.vic.cz.
      IN      MX      20 gagarin.vic.cz.
; definice místních uzlů
localhost IN      A      127.0.0.1
valerian   IN      A      184.229.10.10
           IN      MX      5 valerian.vic.cz.
loghost    IN      CNAME  valerian.vic.cz.
timothy    IN      A      184.229.10.11
gagarin     IN      A      184.229.10.12
; doména dalšího řádu
dudak.swandini IN  A      184.229.20.1
swandini    IN      NS      dudak.swandini.vic.cz.
```



IN NS timothy.vic.cz.

Doména `vic.cz` z `/etc/named.boot` je definována v prvním záznamu SOA v uzlu `valerian.vic.cz` (jeho adresa je uvedena v jednom z následujících záznamů). Hodnota `skoc.valerian.vic.cz` definuje poštovní schránku `skoc` v uzlu `valerian.vic.cz` jako kontaktní pro tuto zónu. Následující hodnoty v kulatých závorkách jsou důležité pro informace ostatních serverů z pohledu doby platnosti dále uvedených záznamů. První je verze obsahu souboru. Podle tohoto označení rozpozná dotazující se server, zda byla v souboru provedena změna. Správce domény by tedy měl pečlivě dbát na změnu osmiciferného označení souboru, protože jinak mohou provedené změny v záznamech zůstat nepovšimnuty (číslo verze je obvykle odvozováno z data provedené změny). Hodnota 43200, tzv. refresh, je čas ve vteřinách, po kterém sekundární server (vlastně každý server, který si již záznam vyžádal) požádá primární o aktualizaci záznamů. Řídící je přitom číslo verze SOA. 3600 je počet vteřin, za který sekundární server opakuje svůj požadavek aktualizace, pokud poslední požadavek aktualizace selhal. 1000 hodin je přibližně 42 dnů, což vyjadřuje další hodnota, a to počet vteřin, po jejichž uplynutí považují ostatní servery DNS server za nedostupný, pokud v průběhu celého časového intervalu nebyl ani jednou uspokojen jejich dotaz. Poslední hodnota je implicitní hodnota `ttl` pro všechny následující záznamy. Následující záznamy MX definují poštovní servery celé domény (obvykle jsou uváděny dva pro případ výpadku jednoho z nich). Záznamy NS definují uzly se servery DNS. Další záznamy definují jména uzlů přiřazená adresám IP (typ A). Za každým takovým záznamem bývá zvykem použít HINFO pro další informace o uzlu. Povšimněte si záznamu CNAME pro `loghost`. Výsledek má stejný efekt jako jméno uzlu `loghost` v tabulce `/etc/hosts` u NIC. V příkladu jsme také naznačili, jak mohou záznamy pokračovat pro definici další domény (`swandini` a uzel `dudak`).

Soubor `named.rev`, který má jednodušší tvar, pak pro náš příklad bude obsahovat

```
; IP addresses to hosts records example named.rev
; (příklad souboru named.rev)
;
@      IN      SOA      valerian.vic.cz.  skoc.valerian.vic.cz.  (
                                97071103      ; verze obsahu souboru
                                43200         ; akt. pro sek.server 2krát denně
                                3600          ; opakovaný přístup každou hodinu
                                3600000       ; expirace za 1000 hodin
                                2419200      ; délka života záznamů je 1 měsíc
                                )
                                IN      NS      valerian.vic.cz.
                                IN      NS      ns.bm.cesnet.cz.
10.10      IN      PTR      valerian.vic.cz.
10.11      IN      PTR      timothy
10.12      IN      PTR      gagarin
20         IN      NS      dudak.swandini.vic.cz
```

Důležité je dát do kontextu také odkaz na `named.rev` z výchozího souboru `/etc/named.boot`, kde je naše síťová adresa navázána na doménu `IN-ADDR.ARPA`, což je označení kořenové domény. Následující řádek souboru `named.boot` pak odkazuje na `named.local`. Je rovněž označen jako primární. Obsah souboru je přitom jednoduchý:



```
; convert address 127.0.0.1 to localhost example
; (příklad souboru named.local, rozpoznání sama sebe)
;
@      IN      SOA      valerian.vic.cz.  skoc.valerian.vic.cz.  (
                                1          ; verze obsahu souboru
                                36000       ; aktualizace po 10 hodinách
                                3600        ; opakovaný přístup každou hodinu
                                3600000     ; expirace za 1000 hodin
                                36000       ; délka života záznamů
                                )
      IN      NS      valerian.vic.cz.
1      IN      PTR     localhost.
```

Naposledy uvedený soubor pro lepší výkon (a menší zatížení sítě) může být uveden také při definici sekundárního serveru. Definice sekundárního serveru v `named.boot` pak může být např.

```
; secondary name server /etc/named.boot example
; (příklad souboru /etc/named.boot sekundárního serveru jmen)
;
directory                                /var/named
secondary  vic.cz                        184.229.10.10    vic.cz.hosts
secondary  184.229.IN-ADDR.ARPA          184.229.10.10    184.66.rev
primary    0.0.127.IN-ADDR.ARPA          named.local
cache      .                             named.ca
```

kteřý může obsahovat uzel s jinou adresou IP než doposud použitou v konfiguračních souborech. Odkaz na sekundární server od různých klientů sítě má stejný efekt jako přímý odkaz na server primární, ale pokud je uzel v menší vzdálenosti, využíváme jeho přiblížení informací prostřednictvím jeho vyrovnávací paměti.

Konečně při konstrukci serveru jen jako vyrovnávací paměti (pouze zpřístupňující) v `named.boot` píšeme

```
; caching-only name server /etc/named.boot example
; (příklad /etc/named.boot pouze zpřístupňujícího serveru jmen)
;
primary    0.0.127.IN-ADDR.ARPA          /var/named/named.local
cache      .                             /var/named/named.ca
```

Jak vidíme, takový typ serveru nemůže mít definovanou autoritu, tj. není zařazen do hierarchie domén. Jako přiblížená vyrovnávací paměť pro další servery, na které odkazuje prostřednictvím své vyrovnávací paměti, je výhodné jej používat pro místní sítě, které mají svou (jednoduchou a stálou) doménu definovanou na některém primárním serveru. Běh démonu **named** a přítomnost souborů `/etc/named.boot`, `named.local`, `resolv.conf` a `named.ca` pak vcelku jednoduše zajistí vazbu na Internet.

Poslední soubor, který nám zbývá k diskuzi, je soubor s inicializací vyrovnávací paměti serveru DNS. Tyto informace se doporučuje převzít z veřejného anonymního serveru FTP `ftp.uu.net`, a to

z adresáře `networking/ip/dns/bind/bind.4.8.3.tar.Z`. Získání tohoto souboru a jeho prostudování ponecháváme čtenáři za domácí úkol.

Správné nastavení serveru DNS dokáže testovat příkaz **nslookup**, jehož interaktivní komunikace se správcem odhalí základní nedostatky, případně potvrdí provozuschopnost serveru (viz provozní dokumentace). Příklad je uveden v kap. 10. S označováním uzlů v rámci DNS budeme pracovat v dalších částech kapitoly a uvedené informace autor považuje za minimum, které musí správce domény o DNS znát; pokud text čtenáři není dostatečně jasný, doporučujeme jej pročíst ještě jednou.

SVID v části Remote Services akceptuje dokument RFC-819 (The Domain Naming Convention for Internet User Applications), který je základem pro definici DNS podle ARPA. SVID akceptuje přidělování jmen sice v hierarchii DNS, ale pouze ve dvou úrovních, navíc bohužel v přesně opačném významu než ARPA. Doménou označení začíná, za tečkou je uveden uzel (nebo zdroj, resource, `para.freak` je označení pro uzel `freak` domény `para`). Pro podporu této struktury definuje příkazy **dname** pro nastavení nebo získání jména uzlu (analogický **hostname**) a **nsquery**, pomocí kterého lze získat informace o libovolném uzlu v rámci DNS. Správce systému podle SVID startuje autoritu DNS příkazem **rfstart** a zastavuje **rfstop**. Aktualizaci tabulek DNS primárního serveru pak provádí příkazem **rfadmin**. Informace serverů DNS jsou uloženy v adresáři `/etc/nserve`, pro autoritu pak v podadresáři `auth.info`.

## 7.4.2 Síťový superserver `inetd`

Síťový server v UNIXu je proces, který zajistí obsluhu požadavků klientu. Takový proces je démon, který je zablokován v očekávání příchodu síťového požadavku. Start síťového serveru zajišťuje přítomnost příkazového řádku jeho startu v některém ze systémových scénářů, jako je např. `/etc/rc` nebo v tabulce `/etc/inittab`, jak je podrobně komentováno v kap. 10. Démon **named** popsaného DNS je např. takto startován. Ale na rozdíl od **named** je řada síťových aplikací, které mohou být aktivovány méně často. Obsluhu vzdáleného přihlášení např. realizuje démon **telnetd**. Nelze dopředu říct, jak často bude odblokován síťovým požadavkem. **telnetd** je na rozdíl od **named** navíc vždy jedinečný pro každý klient. Počet démonů **telnetd** by tedy vždy limitoval počet současných klientů a démony by po většinu času zbytečně využívaly tabulky procesů jádra. Základní metoda provozu takových síťových aplikací je v přítomnosti jediného démonu **inetd**, který je zablokován v čekání na síťový požadavek. V okamžiku příchodu síťového požadavku klientu vytvoří tento démon dítě (program určí podle čísla portu), kterému síťový požadavek předá. Technologie přítomnosti jednoho síťového démonu **inetd** vznikla v systémech BSD. V literatuře je proto často spojován se síťovými aplikacemi z Berkeley, ale jeho činnost je obecná a je přítomen v každém UNIXu (přesto jej SVID neuvádí).

Proces **inetd** (internet daemon), tzv. *síťový superserver*, sleduje a zpracovává síťové požadavky podle tabulky v souboru `/etc/inetd.conf`. Každý řádek souboru popisuje proces dítěte, který **inetd** vytváří:

```
# příklad tabulky síťového superserveru /etc/inetd.conf
# služba schránka prot. čekání uživatel  program      argumenty
#
ftp      stream      tcp      nowait  root /etc/ftpd      ftpd -l
telnet   stream      tcp      nowait  root /etc/telnetd      telnetd
```

```
time    dgram      udp    wait    root internal
...
```

Např. služba **ftp** je zajištěna startem programu ze souboru `/etc/ftpd` (s argumentem **-l**). Démon **ftpd** má přiřazeného po dobu své existence (tj. obsluhy klientu **ftp**) efektivního uživatele **root** (myslíte, že je to bezpečné?). **ftpd** bude využívat na úrovni transportní proud bytů, zde protokol TCP (možnost jsou **stream** pro TCP, **dgram** pro UDP a **raw** pro práci přímo s IP). Po vytvoření dítěte **ftpd** nebude **inetd** čekat na jeho dokončení. Na příkladu služby **time** demonstrujeme použití **inetd** s čekáním. **internal** na místě programu je přitom označení služby, kterou superserver realizuje vlastními prostředky a nevytváří pro ni nové dítě. Zde jde o službu časové synchronizace uzlů.

Tabulka v `/etc/inetd.conf` je výchozí pro práci superserveru, ale není jediná. Co bystrému čtenáři okamžitě schází, je právě číslo portu, podle kterého se **inetd** rozhoduje, kterou službu použije. Registrace čísel portů je obsahem souboru `/etc/services`, kde je každá služba registrována na samostatném řádku. Např.

```
# příklad tabulky registrace čísel portů /etc/services
# služba    port/prot.  přezdívky
ftp         21/tcp
telnet     23/tcp
time       37/udp      timeserver
...
X0          5800/tcp
X1          5801/tcp
```

je fragment takové tabulky. Položka **služba** odpovídá položce téhož textu v `inetd.conf`. Dvojice čísla portu a označení protokolu je uvedena jako následující položka. V tabulce jsme uvedli také příklad definice portů pro práci grafických terminálů (X0 a X1), které nemají ekvivalent v `inetd.conf`. Proč tomu tak je, uvidíme po přečtení kap. 8.

Položka **prot.** (protokol) v definici `inetd.conf` je odkaz do tabulky souboru `/etc/protocols`. Jsou v ní registrována jména protokolů s jejich číselným označením. Číselná označení slouží pro identifikaci protokolu jádrem. Čtenář si jistě vzpomene na položku číselného označení protokolu vyšší vrstvy v záhlaví paketu IP. Klient tak sděluje systému se serverem protokol vyšší vrstvy, se kterým pracuje. V protokolu se musí klient a server shodovat. Pokud se tak nestane, síťová aplikace selže. Tabulka v `/etc/services` proto musí mít shodný obsah, např.

```
# protokly Internetu
ip      0      IP      # pouze ip, tzv. pseudo protocol number
icmp    1      ICMP   # opět pouze vrstva síťová, aplikace ping nebo RIP
tcp     6      TCP
egp     8      EGP    # exterior gateway protocol, aplikace směrování
udp     17     UDP
```

je typické číselné přiřazení. Každý řádek je věnován jednomu protokolu. Za položkou číselného označení protokolu následují přezdívky. Zde jsou použity pro zamezení kolize malých a velkých písmen. Následující komentáře začínají znakem **#** a pokračují až do konce řádku.

Podobně jako u většiny síťových démonů i u **inetd** může privilegovaný uživatel použít signál **SIGHUP**, po jehož přijetí démon reaguje opětovným načtením všech konfiguračních tabulek. Při změně uvedených tabulek proto správce zadává příkaz

```
# kill -1 PID
```

kde PID je číselná identifikace démonu **inetd** a signál 1 je **SIGHUP**.

Programátor síťové aplikace v prostředí RPC také nesmí zapomínat na dohodu se správcem systému při instalaci výsledné síťové aplikace. V uvedených tabulkách musí být registrovány servery, které jsou probouzeny za účelem realizace spojky RPC. Která čísla portu správce systému aplikaci přidělí, je přitom v jeho kompetenci. SVID za tímto účelem používá termín RPC program number data base (databáze čísel programů RPC) a definuje obsah tabulky **rpc** (není ale explicitně uvedeno jméno souboru), ve které na každém řádku definuje jméno programu serveru RPC, jeho číselnou hodnotu a přezdívky, např.

```
# rpc
rpcbind      100000      portmap sunrpc
nfs          100003      nfsprog
...
```

Další vývoj správy aplikační vrstvy sítí může být proto zajímavý. Jasný signál pro sjednocení správy síťových aplikací přinese rozšiřující vydání standardu POSIX, který má být určen pro správu systému. Jeho vydání se očekává do roku 2000.

### 7.4.3 Síťové periferie (tiskárny, terminály, pásky, RFS a NFS, NIS)

Zpřístupněním periferie procesům z jiných uzlů dochází k jejímu síťovému využívání. Typickým požadavkem jsou tiskárny, kdy vznikají tzv. síťové tiskárny. Softwaru, který tiskárnu nabízí (tj. zveřejňuje) do sítě, říkáme tiskový server. Vzdálené využívání periferií (remote devices) je také potřebné např. u magnetických pásek a disků. Zajímavá je práce se síťovým terminálem. Při vzdáleném přihlašování, tj. pro požadavky klientů, jako jsou **rlogin** nebo **telnet**, musí být přiřazena ve vzdáleném systému periferie terminálu. Vývoz takové periferie je přitom emulován. Jinými slovy, vzdálený systém musí pro proces klientu vytvořit emulaci terminálového zařízení, kde pod linkovou disciplínou terminálu není ovladač fyzického terminálu, ale síť. Terminál, který je takto emulován ve vzdáleném systému pro obsluhu klientu a je alokovan pro potřeby serveru (**rlogind** nebo **telnetd**), je označován termínem pseudoterminál (pseudo-terminal device, pseudotty). Pseudoperiferie jsou implementovány použitím PROUDŮ, jak uvidíme v dalším textu. Pro využívání vzdálených systémů souborů byla vytvořeny aplikace NFS (Network File System) a RFS (Remote File Sharing, podle SVID), principiálně shodné implementace i využití, ale bohužel různých jmen systémových příkazů. Z pohledu využívání systému souborů vzdáleného uzlu se logicky nabízí také možnost práce se speciálními soubory, tj. s periferiemi vzdáleného uzlu. Jak RFS, tak NFS tuto možnost nabízejí, ale používá se výjimečně nebo pro speciální síťové aplikace, protože jak jsme uvedli v kap. 6, se speciálním souborem pracuje vždy určitý démon, který periferii řídí a frontuje nebo jinak usměrňuje požadavky na přístup k periférii (typické je to u tiskáren, jak uvidíme dále). Je proto lépe používat místní řídicí démon periferie také jako síťový server požadavků na periférii, které přicházejí ze sítě, než řídit periférii systému, (tj. alokovat vzdálený speciální soubor). Místním i vzdáleným uživatelům je tak periferie dostupná stejným způsobem.

Sdílení tiskárny v místním uzlu jsme komentovali v kap. 6, kde jsme uvedli dva nejrozšířenější způsoby přístupu uživatele k frontě tiskárny pomocí programů **lp** a **lpr** a systémové zabezpečení tisku obsahu fronty démony **lp sched** nebo **lpd** (viz obr. 6.10 a 6.11). Pro vzdálený tisk můžeme koncepci ponechat, ale rozšířit ji o síťovou komunikaci démonů v okamžiku, kdy démon rozpozná (např. **lpd** položkou v tabulce `/etc/printcap`), že tiskárna, na kterou má obsah fronty tisknout, je v jiném uzlu. V tom případě otevírá síťové spojení odpovídajícího uzlu a požadavky pro tisk (tj. soubory s obsahem pro tisk i řídicí soubor tiskové úlohy) přenáší za podpory aktivovaného vzdáleného démonu vzdálené tiskárny do vzdálené fronty pro tisk. Podívejme se na obr. 7.23, kde uvádíme příklad používanějšího vzdáleného tisku **lpr** systémů BSD.

Démon tisku **lpd** v případě neprázdné tiskové fronty (kterou plní klienty tisku **lpr**) vytváří vždy na každou tiskovou úlohu dítě **lpd**. V případě, že podle obsahu `/etc/printcap` rozpozná uživatelův požadavek na vzdálené tiskárně, aktivuje síťovou službu **printer** (který teď jako klient síťové komunikace získá port pomocí funkce `getservbyname`). Ve vzdáleném systému démon **lpd** očekává příchod tiskového požadavku, blokován v operaci čtení ze schránky (jsme v BSD) jak typu UNIX (místní tisky), tak typu INTERNET (požadavek tisku síťového klientu). Jak pro místní, tak pro vzdálený tisk vytváří **lpd** dítě, které v případě převzetí síťového spojení převezme všechny datové i řídicí soubory tiskových úloh klientu tisku a uloží je do adresáře tisku odpovídající tiskárny. Po ukončení síťové komunikace dítě projde celou frontu tiskových úloh a tiskne je na tiskárně. Jeho otec **lpd** mezitím čeká na jeho dokončení a teprve potom přijímá další požadavek tisku, ať už od místních **lpr** nebo síťových klientů **lpd**.

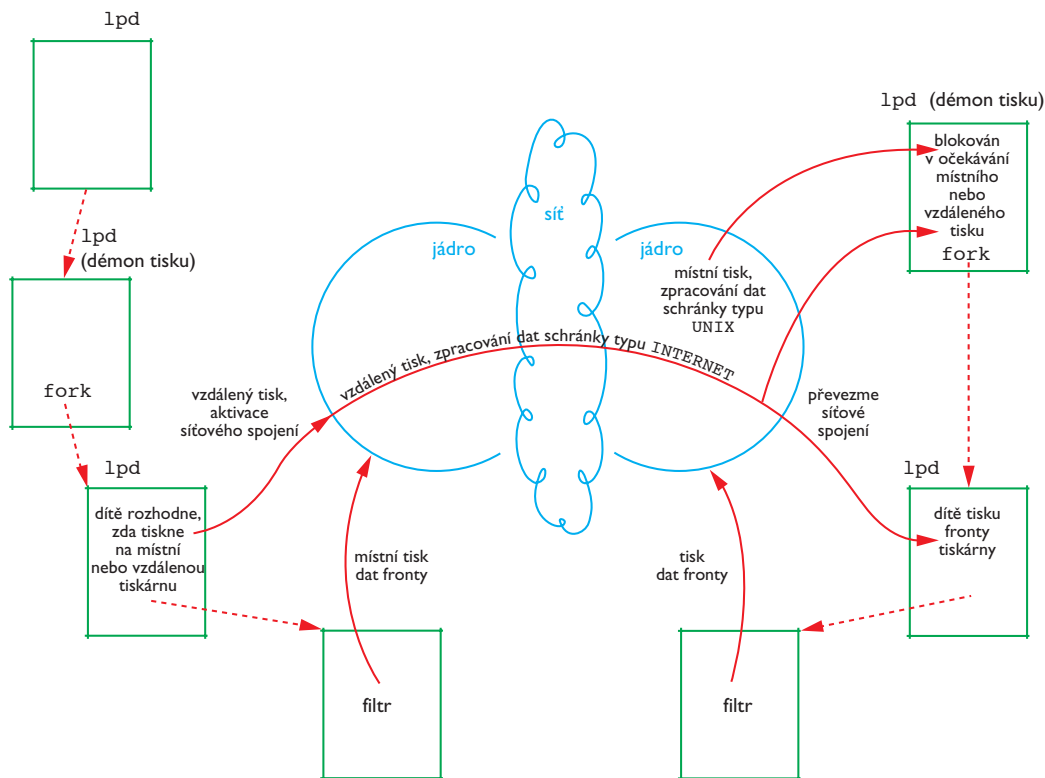
Čtenář by si měl pozorně prohlédnout obr. 7.23, ale i obr. 6.10 a 6.11, a prostudovat tabulky `/etc/services`, resp. `/etc/inetd.conf` pro případ tiskových serverů. Z těchto informací dokáže získat přehled o vzdáleném tisku, který vychází z obsluhy tiskových front v SYSTEM V (**lp** a **lp sched**).

*Pseudoterminál* je programová emulace funkcí periferie terminálu. Dobře se implementuje pomocí PROUDŮ (viz obr. 4.7 a 4.8). Ve svém výsledném efektu jde vlastně o pojmenovanou obousměrnou rouru, na jejíž jedné straně je proces běžící na terminálu (nejčastěji některý shell) a na straně druhé proces, který uživateli zpřístupňuje takto vzniklé terminálové prostředí. Využití pseudoterminálu v síťové aplikaci **rlogin** ukazuje obr. 7.24.

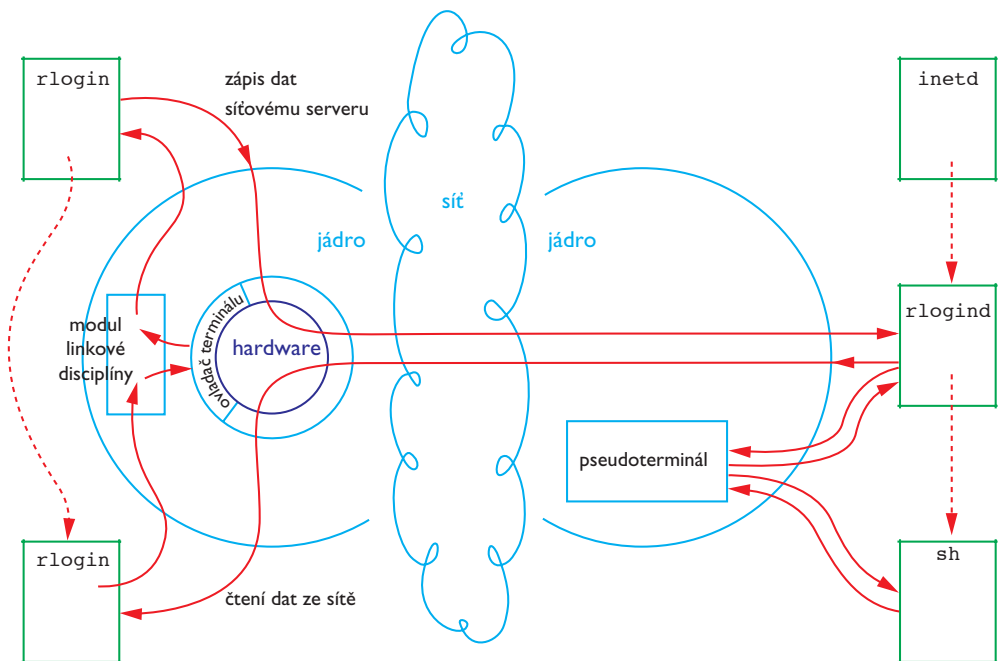
Všimněme si nyní prozatím pravé strany obrázku, a sice procesu síťového serveru **rlogind**. V případě příchodu síťového požadavku na portu patřícího službě **rlogin** startuje síťový démon **inetd** server **rlogind**. Server pomocí volání jádra **open** otevírá PROUDOVÉ zařízení `/dev/ptm/N`, což provede parametrem **PTY\_MASTER** na místě jména souboru. Jméno konkrétního speciálního souboru (N) mu je přiděleno jádrem podle volných pozic registrovaných jádrem. Otevírá tak stranu pseudoterminálu, kterou označujeme jako *vládce* (master). Jak ukazuje detail na obrázku, je to část pseudoterminálu, která by u fyzického terminálu příslušela ovladači terminálu. **rlogind** poté vytvoří dítě, které před proměnou na proces shell otevírá druhou stranu pseudoterminálu označovanou jako *otrok* (slave, tj. `/dev/pts/N` parametrem **PTY\_SLAVE** na místě jména souboru v **open**). Využití speciálních souborů přidělovaných jádrem pak zaručí po otevření na straně slave již přítomnost modulu linkové disciplíny v jádru, takže shell, který vznikne poté z dítěte prostřednictvím **exec**, získává otevřené deskriptory na místě 0, 1 a 2 pro práci se standardním vstupem a výstupem jako při práci s fyzickým terminálem. Rovněž tak získá pseudoterminál jako řídicí terminál.

Na straně klientu proces **rlogin** získává řídicí terminál od rodiče (tj. některého shellu pracujícího uživatele) a tím i místní linkovou disciplínu. Je-li navázání síťového spojení do požadovaného uzlu úspěšné, vypíná zobrazování uživatelem zapisovaných znaků na klávesnici místní linkové disciplíny (echo), protože očekává jejich opakování vzdáleným modulem linkové disciplíny. Pro čtení dat ze sítě vytváří dětský proces **rlogin**, který je zodpovědný za čtení dat ze sítě, zápis přitom provádí sám. Toto rozdělení je výhodné, protože pozastavení práce rodiče v místním systému neznamena, že data nejsou ze sítě čtena. Data vzdáleného systému (procesu shellu) se proto objevují, kdykoliv přijdou. Je to důležité pro práci s řídicími znaky terminálu, např. zajištění režimu X-ON/X-OFF atp.

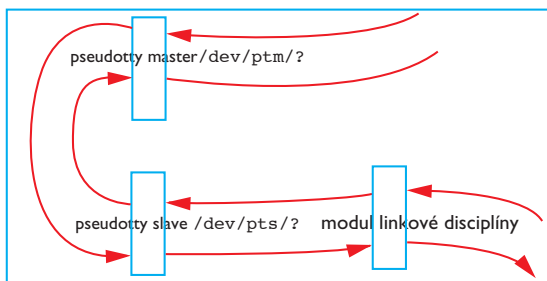
Pseudoterminál je tedy oboustranné párové zařízení s označením master a slave. Oba jeho konce jsou přitom spojeny tak, že cokoliv je zapsáno na straně slave, je čteno na straně master a naopak. Pseudoterminál přitom podléhá na straně slave linkové disciplíně, která je ovlivnitelná s ní pracující aplikací způsobem běžným při práci s terminálem. Jádro má obvykle k dispozici omezené množství pseudoterminálů (běžně 32), které postupně přiděluje podle požadavků síťových (ale i jiných) aplikací pro simulaci



Obr. 7.23 Vzdálený tisk lpr



detail pseudoterminálu:  
spojení dvou proudů využitím obousměrné pojmenované roury



Obr. 7.24 Použití pseudoterminálu v aplikaci `rlogin`

terminálového zařízení. Přidělený speciální soubor může proto být po každém vzdáleném přihlášení z téhož místního do téhož vzdáleného systému různý. Jména speciálních souborů, které jsme uvedli, pak vychází z SVID. Výrobci ve svých verzích však používají mnohdy jiná jména. Pro síťovou aplikaci je to však věc transparentní a týká se pouze implementace jádra.

Pseudoterminály jsou využívány pro otevírání oken terminálového přístupu k místnímu nebo vzdálenému systému v grafickém prostředí oken X, což budeme komentovat v kap. 8. Lze je ale také použít lokálně pro simulaci více terminálů na tomtéž fyzickém terminálu. Namísto síťové aplikace pak pracuje pro distribuci vstupů a výstupů z fyzického terminálu zvláštní proces (různých jmen, na obr. 7.25 `multigtty`), pod jehož řízením jednotlivá sezení probíhají (přepínání mezi sezeními se provádí obvykle kombinací kláves Alt-F1, Alt-F2 ...). Situaci stručně ukazuje obr. 7.25.

Také magnetopáskové jednotky lze ovládat vzdáleně. Znamená to, že magnetickou pásku, která je připojena jako periferie k místnímu systému, nabízí tento systém do sítě prostřednictvím aktivovaného serveru (opět démonem **inetd**). Příkladem síťového serveru magnetické pásky je **rmt** (bývá v adresáři `/etc`), který poskytuje data vzdálenému klientu. Klientem může být např. **tar** nebo **cpio**. Jejich práce je orientována nikoliv na zařízení místní, ale síťové, čili je definované uzlem (v konvencích NIC, DNS nebo adresou IP) a službou magnetické pásky. Např.:

```
$ tar -tvf rmt@valerian:/dev/rmt0
```

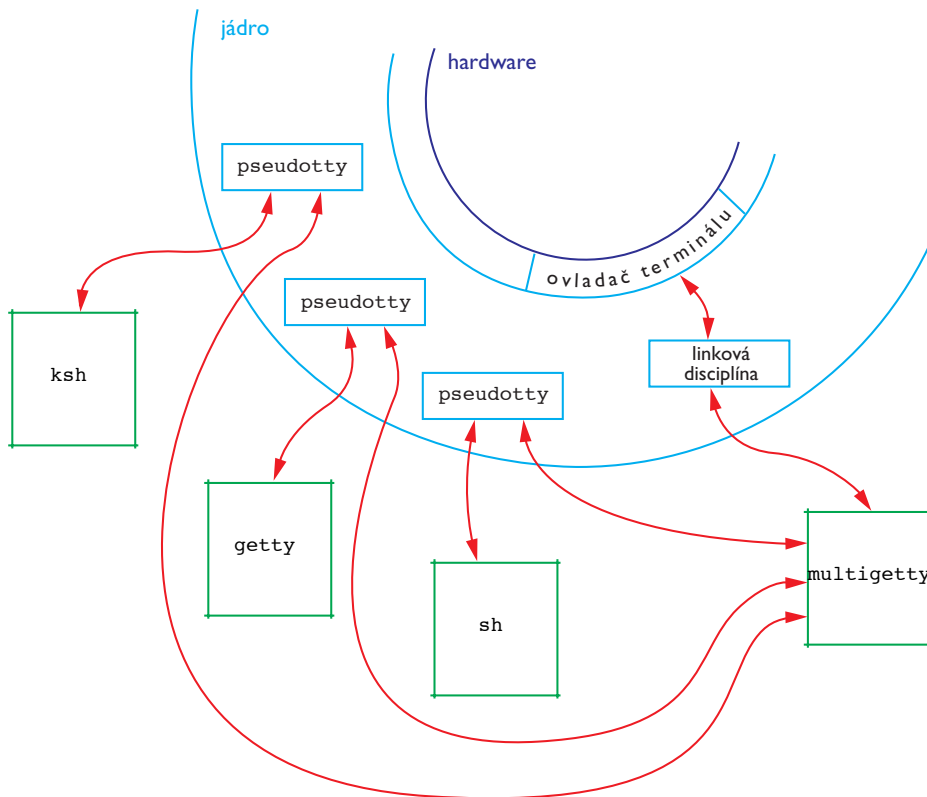
...

kde **rmt** je jméno uživatele, pod jehož sezením přístup probíhá (může být vynechán včetně znaku `@`), **valerian** je jméno uzlu (lze jej nahradit adresou IP) a `/dev/rmt0` je jméno speciálního souboru pásky (i ten lze vynechat včetně znaku `:`). V zásadě klient (v našem příkladu rozšířený **tar**) ani server neobsahují žádný nový aspekt práce v sítích. Podobně je tomu i u méně používaných aplikací BSD vzdáleného provádění příkazů **rexec**, vzdáleného shellu **rsh** nebo **remsh** (přestože se někdy hodí více, používá se **rlogin** nebo **telnet**) a kopie souborů sítí **rcp** (používá se **ftp**). Všechny případy jsou implementovány a instalovány na již diskutovaných principech a nové aspekty jsou spíše věci programátora těchto aplikací (viz [Stev90]).

Sdílení disků se věnují služby NIS a NFS. Síťová aplikace NIS (Network Information Services) byla původně vyvinuta opět jako součást systémů BSD a měla název Yellow Pages (YP)<sup>5</sup>. Je programována v RPC a její definice není součástí SVID. NFS (Network File System) je původně produktem firmy Sun Microsystems, ale dnes stěží najdeme komerční UNIX, který by aplikaci nepodporoval. SVID definuje velmi podobnou službu RFS (Remote File Sharing), která je také programována v RPC a XDR, ale prakticky ji používá jen SYSTEM V. V současné době je silným konkurentem produktů průmyslových implementací UNIXu, a sice AFS (Andrew Filesystem), jehož budoucnost se zatím očekává. Převážná většina implementací UNIXu však podporuje NFS a v současné době také v SVID zmiňovaný produkt DFS (Distributed File System).

NIS poskytuje správci systému distribuovanou údržbu systémových tabulek uzlů místní sítě. Systémové tabulky, které NIS spravuje, jsou především databáze uživatelů v `/etc/passwd`, `/etc/group`, definice uzlů sítě v `/etc/hosts`, `/etc/networks`, síťové služby a protokoly v `/etc/services` a `/etc/protocols`. Prakticky to znamená, že v rámci místní sítě pracuje jeden síťový server NIS (proces **ypserv**), který má k dispozici databázi systémových tabulek platných pro celou místní síť. Uzly, které chtějí služeb serveru využívat, k tomu delegují klientu **ypbind**, na který se obracejí systé-





Obr. 7.25 Použití pseudoterminálů pro možnost více sezení na fyzickém terminálu

mové aplikace. NIS nedokáže pracovat přes směrovače, ale dokáže procházet mosty. Je to tedy služba místního charakteru. Server NIS pracuje pro určitou oblast, které říkáme doména NIS (v rámci místní sítě může pracovat několik serverů NIS). Přestože tato doména je obsahem i funkcí zcela odlišná od domény DNS, doporučuje se používat shodného jména. Jméno domény je definováno správcem systému příkazem **domainname**. Např. pomocí

```
# domainname vic.cz
```

nastavíme v systému doménu, a to v uzlu serveru nebo klientu služby NIS. Bez parametru zobrazí i neprivilegovanému uživateli současné jméno domény. Doména musí být nastavena uvedeným způsobem vždy při startu systému. Proto je příkaz pro její nastavení umístěn v některém ze startovacích scénářů, např. v `/etc/rc.local` (viz kap. 10). Rovněž tak musí být ve startovacím scénáři uveden start démonu NIS **ypserv** nebo **yplibind**. Oba procesy jsou podobně jako **named** v DNS trvale běžící, přestože jsou zablokovány v čekání na výskyt události na odpovídajícím portu.

Součástí poskytování služby NIS jsou v uzlu s **ypserv** vytvořeny tzv. mapy NIS (NIS maps). Mapy jsou vytvořeny ze současných verzí systémových tabulek uzlu pomocí příkazu **ypinit** (s volbou **-m**) do adresáře `/var/yp/NISdomain`, kde `NISdomain` je jméno domény NIS (v našem příkladu `/var/yp/vic.cz`). Pro aktualizaci map pak používáme příkaz **ypmake**. Příkazů, případně knihovních funkcí, je více. Čtenář je může najít v dokumentaci NIS.

Důležitým souborem při využívání NIS je však ještě tzv. databáze uzlů, uživatelů a domén NIS, která je v uzlu serveru soustředěna v souboru `/etc/netgroup`. Jde o vytvoření odkazových jmen na uzly v kontextu domén a uživatelů v těchto uzlech. Databáze je využívána službami vzdáleného přihlašování (např. **rlogind** při správném nastavení `/etc/hosts.equiv` nebo `.rhosts` v domovském adresáři nevyžaduje heslo pro přístup vzdáleného uživatele) nebo službou NFS pro síťové identifikace. Formát `/etc/netgroup` vychází z definice jména a seznamu uzlů nebo dalších jmen. Každý řádek souboru je definice nového jména (říkáme jména skupiny NIS):

```
jméno_skupiny člen [člen] ...
```

kde člen je buďto opět jméno\_skupiny nebo seznam ve formátu

```
(uzel,uživatel,doména)
```

např.

```
sgi          (indy,,) (indigo2,,) (origin,,)
dec          (hp832,,) (hp735,,)
vsechny      sgi dec
```

kde jsme na místě uživatelů a domén ponechali prázdné položky: implicitně znamenají všichni uživatelé, vynechaná doména je platnost pro všechny domény. Soubor `/etc/netgroup` je také transformován do podoby mapy NIS při stavbě domény.

Poslední poznámka se vztahuje k registraci uživatelů. V uzlu klientu NIS podléhají síťovému prozkoumání službou NIS uživatelé, kteří jsou označeni v prvním sloupci jejich řádku registrace v `/etc/passwd` znakem + (tzv. NIS cookie). Uživatelé zdánlivě nechránění heslem jsou tak ve skutečnosti prověřováni serverem NIS.

Základní myšlenkou sdílení disků je zpřístupnit uživatelům disky prostřednictvím procesu serveru a také v jiných uzlech pracovat se strukturou svazků uzlu serveru, a to připojením některého adresáře uzlu klientu k adresáři uzlu serveru. Uživatelé pracující v uzlech klientů pak manipulují s daty takového vzdáleného svazku stejným způsobem jako s daty místních svazků, tj. mohou soubory otevírat, číst nebo je přepisovat.

Každý proces v uzlu klientu při otevírání vzdáleného souboru prochází jádrem (volání jádra `open`), které udržuje evidenci o připojených svazcích. Jádro v tomto případě rozpozná, že jde o vzdálený svazek, a předá požadavek pro přenos dat z odpovídajícího uzlu procesu klientu. Proces server ve vzdáleném uzlu požadavku vyhoví, provede odpovídající operaci a její výsledek opět předá procesu klientu síťovým spojením. Jádro převezme data od procesu klientu a předá je místnímu procesu. Podívejme se na obr. 7.26.

V NFS (Network File System) je vzdálený svazek připojen voláním jádra `mount`. Správce systému tedy použije příkaz **mount** (pro odpojení **umount** nebo analogicky volání jádra `umount`) způsobem:

```
# mount valerian:/usr2/people /usr2/people
```

Na místě speciálního souboru je tedy použita síťová specifikace jména vzdáleného uzlu oddělená dvojtečkou od úplné cesty k adresáři, který má být připojen na místní adresář použitý v dalším parametru příkazu. Procesem klient, který od této chvíle zajišťuje přenos dat vzdáleného svazku, je **biod** a běží v několika instancích (4 až 8) jako démon. Strana serveru musí být zajištěna démonem **nfsd**, který běží také několikrát a který vyhovuje spojkám RPC klientů **biod**. Na straně serveru musí být ještě k dispozici proces **rpc.mountd** (nebo jen **mountd**), který zpřístupňuje svazky pro vzdálený **mount**. Vzhledem k jeho nepřilíh častému oslovování bývá startován démonem **inetd** vždy na požádání. Typická je ještě implementace NFS, kde se používají demony **rpc.lockd** (na obou stranách spojení) k zamykání souborů, a jimi využívaný démon **rpc.statd**, který sleduje síťové služby a využívá se např. při havárii NFS. Při výpadku serveru NFS totiž klienty čekají na jeho oživení – obnovu zámků souborů pak provádí **rpc.statd**.

Uzel serveru musí oznamovat do sítě svazky, které je možné vzdáleně připojovat. Vedle již uvedeného démonu **nfsd** používá správce systému příkaz **exportfs**, kde v argumentu uvádí jméno adresáře, jehož podstrom bude zpřístupněn. Vývoz adresáře může přitom být libovolný adresář některého ze svazků. Vývoz navíc dále pokračuje případnými dalšími připojenými svazky na některý z adresářů vyváženého stromu. Export je tedy uskutečňován na logické úrovni a klient se nezajímá o další rozdělení importovaného stromu adresářů na svazky disků. Podmínkou pro vývoz adresáře je jeho evidence v tabulce souboru **/etc/exports**. Má jednoduchý tvar: každý řádek obsahuje cestu k adresáři, který je exportován. Mezerou oddělený může být parametr pro **exportfs**. Např.

```
/usr2
/usr/local/dosapp -ro
```

je tabulka dvou vyvážených adresářů, druhý adresář je přitom přístupný pouze pro čtení. Pro import např. adresáře **/usr2/people** našeho předchozího příkladu je uvedení pouze adresáře **/usr2** dostačující, protože připojovat se může klient do libovolného místa exportovaného stromu adresářů. Tabulka **/etc/exports** je důležitá a většina instalací ji používá při startu NFS pro export všech jejích adresářů (**exportfs** má volbou **-a**), ale aktuální export adresářů jádro udržuje také ještě v binárním souboru **/etc/xtab**, který je také zdrojem informací příkazu **showmount**. Příkaz je dostupný v uzlu exportovaných adresářů pro jejich výpis, ale používají jej také uzly klientů, např.

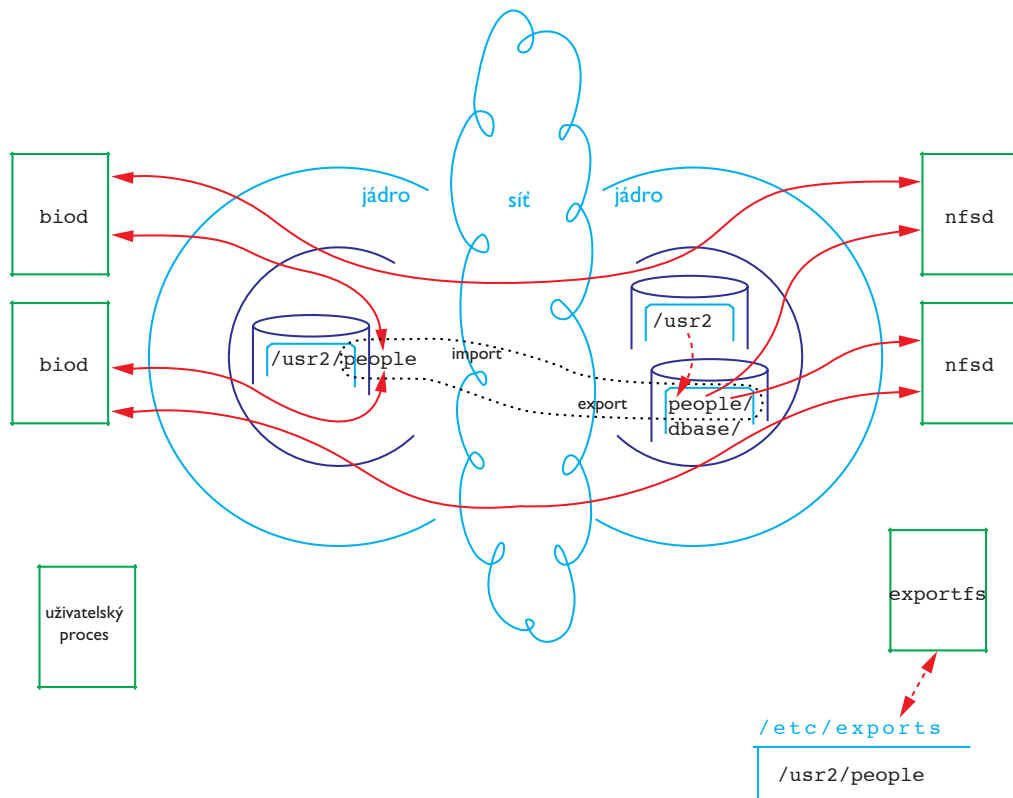
```
$ showmount -e valerian
export list for valerian
/usr2          (everyone)
/usr/local/dosapp (everyone)
```

Informace **(everyone)** znamená, že adresář je dostupný každému klientu z libovolného uzlu. Seznam uzlů klientů totiž lze v uzlu serveru zúžit jejich vyjmenováním pomocí parametru **-access** v tabulce **/etc/exports**, např.

```
/usr/man -access=timothy:gagarin
```

Seznam uzlů **timothy,gagarin** se pak objeví na místě **(everyone)**.

Start aplikace NFS je odvozena od strany klientu nebo serveru, přestože najdeme běžně instalace sítí, kde uzly svazky jak dovážejí, tak vyvážejí. Správce sítě by měl však takovou anarchii mít dobře zdůvodněnou. Start démonů, které musí být při provozu NFS trvale přítomny v systému, je uveden v tabulkách startu systému (speciálně jeho nastavení sítě). Pro uzel klientu je to démon **biod** (s para-



Obr. 7.26 Vzdálené svazky NFS

metrem počtu instancí běhu), **rpc.statd** a **rpc.lockd**. Na straně serveru pak démon **nfsd**, případně **rpc.mountd**, pokud není uveden jako služba v `/etc/inetd.conf`, a export adresářů příkazem **exportfs -a**.

V současné době se kromě nejrozšířenější aplikace NFS používají také další implementace síťového distribuovaného systému souborů. Je to jednak průmyslová implementace AFS (Andrew Filesystem) a jednak v SVID definovaný RFS (Remote File Sharing). SVID uvádí (ale nedefinuje) také NFS jako alternativu RFS stejného významu a zajišťuje kompatibilitu RFS s NFS.

Na rozdíl od NFS, který pracuje především s protokolem bez stálého spojení (UDP nebo v OSI definovaný TP4), RFS pracuje s protokoly stálého spojení. Termíny, které definuje SVID, jsou sdílení (sharing) stromů adresářů, jejich vzdálené připojování (remote mounting). Uzel sítě, který nabízí disky pro sdílení, je systém serveru (server system), a naopak uzel, jehož aplikace disky sdílí pomocí jejich připojení, je systém klientu (client system). RFS exportuje adresáře do sítě příkazem **share.unshare** (jsou stále definovány **adv** a **unadv**, které mají podobný význam – SVID uvádí, že budou do budoucna

vypuštěny) je naopak z nabídky pro síť vyjímá (u NFS používáme příkaz **exportfs** s volbou **-u**). **unshare** nemá žádný vliv na již připojené svazky klientů, ale odmítá nové klienty při pokusu o připojení. Připojení je i zde prováděno příkazem **mount**. Pokud vyvážíme svazek typicky jako RFS, označujeme jej jménem, které se pak používá na místě jména uzlu a dvojtečkou odděleného vzdáleného adresáře u NFS, např.

```
# share -F rfs /usr2/people PEOPLE
```

a v uzlu klientu pak píšeme

```
# mount -F rfs PEOPLE /usr2/people
```

Ale také

```
# share -F nfs /usr2
```

a u klientu

```
# mount -F nfs valerian:/usr2/people
```

Zveřejněné adresáře v RFS jsou viditelné příkazem **dfshares**. Použitím voleb si můžete vybrat pouze určitý typ (NFS nebo RFS volbou **-F**, typ je také případně uveden v `/etc/dfs/fstypes`) nebo uzel (**-h**). Informace o exportovaných a připojených adresářích, které jsou kořeny vzdálených stromů, si správce vyžádá příkazem **dfmounts** (**rmntstat** bude z SVID vypuštěn). Jeho volby i samotný výpis je podobný **dfshares**. Dále má ještě správce systému možnost používat příkaz **fusage**, který jej informuje o statistice přenášených diskových dat, a **fumount** (forced unmount), pomocí něhož dává z uzlu serveru pokyn ke spuštění skriptů v uzlech klientů pro odpojení svazků již do sítě nenabízených. V SVID je pro práci s RFS rezervován adresář `/etc/dfs` (distributed filesystems), speciálně `/etc/dfs/rfs`. Pro práci RFS musí být startován také systém autorit DNS, který je v SVID startována příkazem **rfstart**. Komentář uvádí, že příští verze SVID bude obsahovat část DFS (Distributed File Systems), která nahradí současné RFS.

Při práci místní sítě se sdílenými disky se více než kdekoli jindy objevuje nutnost udržovat tabulky s registrací uživatelů ve všech uzlech konzistentní. Obsahem i-uzlu každého souboru je, jak víme, číselná identifikace uživatele, jejíž registrace je jedinečná v tabulce `/etc/passwd` (analogicky pro skupinu v `/etc/group`). Uživatel, který potřebuje participovat na připojeném svazku z různých uzlů sítě, musí mít svoji číselnou identifikaci shodnou ve všech takových uzlech. Tento problém řeší NIS, jak jsme si ukázali, ale SVID také definuje distribuovanou databázi uživatelů sítě (tzv. user and group ID mapping). Pro stavbu distribuované databáze uživatelů se v uzlu serveru používá příkaz **idload**, který vychází z informací podle obsahu souborů `uid.rules` a `gid.rules` v adresáři `/etc/dfs/rfs/auth.info`. Podle vyjádření SVID je takto vzniklá databáze akceptovaná demony RFS.

#### 7.4.4 Pošta (sendmail)

*Elektronická pošta* (electronic mail, e-mail, slangově email) je způsob předávání dat mezi jednotlivými uživateli operačního systému pomocí jim evidovaných poštovních schránek (mailboxes), jak jsme již uvedli v kap. 5 (čl. 5.2)<sup>6</sup>. Využívání sítí a zejména sítě sítí (Internetu) otevřelo elektronické poště nový rozměr. Posílání pošty je možné každému uživateli registrovanému v libovolném operačním systému uzlu, který je připojen k síti. Principiálně je elektronická pošta odvozena od klasické, ovšem namísto

v papíru zabalené další papíry, které jsou médiem pro zápis zasílaných informací inkoustem nebo tiskovou barvou a přenosu pomocí nákladných institucí pošt (lidé, auta, vlaky), jsou elektronická data přesunuta do jiného uzlu počítačové sítě a připojena k souboru poštovní schránky požadovaného uživatele. Doručení zajistí trvale pracující poštovní software ve všech uzlech sítě, které příjem a odesílání pošty podporují. Důležitou roli hraje také doba doručení, která se u elektronické pošty pohybuje v rozmezí několika minut.

Rozšíření elektronické pošty pro síť musela provázat úprava poštovního softwaru, jejího rozšíření z místní na síťovou aplikaci. V odst. 5.2 jsme již uvedli, že je tento software rozdělen na tři části. Jednak je to část uživatelská (user agent), se kterou pracuje uživatel, tedy jde o program, který umožňuje uživateli čtení nebo odesílání pošty, a část systémová, která poštu přijme (delivery agent, agent doručení) nebo přeneše (transport agent, agent přenosu). Na rozdíl od dosavadního způsobu práce síťových aplikací není user agent klientem síťové aplikací; klient i server je zajišťován systémovou částí, která přebírá od uživatele požadavek pro přenos pošty a pro spojení s odpovídajícím uzlem přebírá roli klientu, který poštu vloží do poštovní schránky uživatele. Např. démon **sendmail** je agentem transportu, server **smtpd** (síťová služba SMTP, Simple Mail Transfer Protocol) je agentem doručení.

Přestože **sendmail** není jediným agentem transportu, je nejpoužívanější a na jeho příkladu si princip chování elektronické pošty demonstrujeme.<sup>7</sup>

Adresace uživatele v rámci místního uzlu je jednoduchá a zůstává stejná i v případě připojení uzlu do sítě. Registrovaný uživatel je jiným uživatelem označen jménem podle `/etc/passwd`. Např.

```
$ mail petr
```

```
Milý Petře,
```

```
Nepřepeři nám vepře
```

```
^d
```

je odeslání pošty uživateli **petr** v rámci uzlu, kde jsme přihlášení. Adresace v případě uživatele v jiném uzlu sítě je rozdělena na dvě části: opět jméno uživatele registrovaného v `/etc/passwd`, ovšem vzdáleného uzlu, a za znakem `@` označení vzdálené domény příjmu pošty. Např.

```
$ mail petr@vic.cz
```

**vic.cz** přitom není jméno vzdáleného uzlu, ale jméno domény. Pošta je odeslána teprve podle záznamu typu **MX** serveru DNS odpovídající domény (viz odst. 7.4.1), který určuje uzel s poštovním serverem (z příkladu v 7.4.1 je to např. uzel **valerian.vic.cz**). Bývá zvykem vytvářet více než jeden záznam **MX** pro určitou doménu. Jde o definice dalších uzlů, kterým je pošta doručena v případě, že hlavní uzel domény pro poštu je právě nedostupný. Náhradní uzel poštu převezme a ponechá si ji obvykle 2 dny, v průběhu kterých se opakovaně pokouší poštu doručit na správné místo. Pokud k tomu nedojde, je pošta odeslána zpět s oznámením dvoudenního výpadku cílového uzlu.

Poštovní schránky uživatelů bývají uloženy v adresáři `/var/mail` (nebo `/usr/mail`). Program, se kterým pracuje uživatel, používá pro identifikaci poštovní schránky obsah proměnné **MAIL** v shellu. Proměnná **MAILCHECK** obsahuje vteřinový interval (implicitně 600, tj. 10 minut), po jehož uplynutí shell zjišťuje, zda se v poštovní schránce objevila nová pošta. Pokud ano, shell vypisuje na standardní výstup uživateli zprávu **You have mail**. Pro kontrolu může uživatel v systémech odpovídajících **SVID** používat také příkaz **mailcheck**, který provádí totéž. Uživatelé si jej obvykle vsouvají do souboru `.profile` domovského adresáře takto:

```
mailcheck 2>/dev/null
```

Uživatelských programů pro čtení a odesílání pošty je celá řada. SVID uvádí pouze **mail** a schopnější **mailx**, POSIX definuje pouze **mailx**. Výchozí nastavení po spuštění programu je pro **mailx** definováno v proměnné **MAILRC** v shellu (bývá to soubor **.mailrc** v domovském adresáři uživatele). **mailx** totiž pracuje s řadou vnitřních proměnných. V UNIXu známé a používané jsou také programy **elm** nebo **mh** nebo **mailtool** firmy Sun. Dnes je však prakticky každý UNIX vybaven grafickým prostředím a většina uživatelů používá poštovní programy v rámci práce na pracovní ploše (např. **zmail**), které jsou ovladatelné myší a nemají alfanumerický charakter příkazového řádku shellu. Konvence uvedených importovaných proměnných shellu však dodržují.

Agent transportu **sendmail** (pochází z BSD) běží jako démon síťové aplikace na portu 25. Jemu odpovídající server **smtpd** je agentem doručení. Znamená to, že **sendmail** přebírá poštu od uživatelských programů a stává se klientem vždy vzdáleného poštovního uzlu, se kterým navazuje spojení jeho vyhodnocením podle DNS. Jako klient mu v cílovém uzlu transportu pošty odpovídá **smtpd**, který je startován démonem **inetd**. Vzhledem k potřebě synchronizace je dnes **smtpd** také často startován jako dítě procesu **sendmail**. SMTP je totiž protokol přenosu pošty (viz RFC 821) a implementace jeho serveru **smtpd** jako agentu doručení pošty je dnes považována za standardní i bez přítomnosti démonu **sendmail**.

**sendmail** je síťová aplikace, která odesílá a přijímá poštu podle SMTP. Pro odesílání pošty je možné nastavit interval, po který se uživateli odesílaná pošta hromadí v místním uzlu a je pak odesílána najednou, přestože se tato možnost dnes používá výjimečně, např. při ladění. Základním konfiguračním souborem je **/etc/sendmail.cf**, který je složitý a úzce souvisí s informacemi DNS.

**sendmail.cf** obsahuje především informace o prostředí, ve kterém je **sendmail** provozován (definice domén v proměnných, pomocných maker atd.), prepisovací pravidla adres pošty a definici agentů doručení pošty. Pošta totiž může být doručena nebo odesílána i jiným způsobem než pomocí TCP/IP (např. pomocí UUCP, viz 7.5). Vzhledem ke složitosti práce s obsahem souboru **sendmail.cf**, je mnohdy vítána možnost jej měnit interaktivním pomocným programem (např. **configmail** v IRIXu).

**sendmail** pracuje také se substitucemi pro adresáta pošty. Příkaz **newaliases** (který je totéž co **sendmail -bi**) vytváří binární data přezdívek v souboru **/etc/aliases.db** podle editovaného textového souboru **/etc/alises**. Obsah souboru **aliases** je jednoduchý. Každý řádek obsahuje texty ve tvaru

```
přezdívka: jméno_1, jméno_2, ...
```

kde **jméno\_?** jsou jména uživatelů, kterým bude pošta doručena, pokud přijde pod jménem **přezdívka**. Na místě jména uživatele může být uvedena také přezdívka. Uživatel může být také určen adresou pošty ve tvaru **uživatel@doména**. Ve chvíli, kdy **sendmail** rozpoznal poštovní schránku uživatele, orientuje se ještě podle obsahu souboru **.forward**, který může mít uživatel poštovní schránky ve svém domovském adresáři. **.forward** obsahuje poštovní adresu uživatele, kam má být příchozí pošta ihned přeměrována, aniž by byla připojena do poštovní schránky místního uzlu.

Poštovní uzel konfigurovaný v DNS a **/etc/sendmail.cf** je obvykle využíván uživateli místní sítě. Pro pohodlí práce uživatele je často adresář poštovních schránek **/var/mail** exportován do sítě pomocí NFS a pošta je tak čitelná v libovolném uzlu sítě. Pro potřeby vstupu do poštovního uzlu



a čerpání obsahu poštovní schránky síťovými klienty byl zkonstruován protokol POP (Post Office Protocol, dnes známá verze POP3). Jako server je startován démonem **inetd** na portu 110 a při prokázání uživatelským heslem sezení předává vzdálenému klientu obsah poštovní schránky. POP je často používán síťovými klienty jiných operačních systémů (např. MS Windows). Uživatel musí být opatrný, protože při přenosu schránky server implicitně přenesenou poštu na poštovním uzlu ruší. Klient by tak měl být provozován na skutečně osobním, nikoli veřejném počítači, protože pošta se tak přemístí do uzlu klientu. Je proto výhodnější se přihlásit pomocí **telnet** a číst poštu uživatelským agentem, nebo použít grafické prostředí X a z místního grafického adaptéru si vytvořit grafický terminál práce v poštovním uzlu.

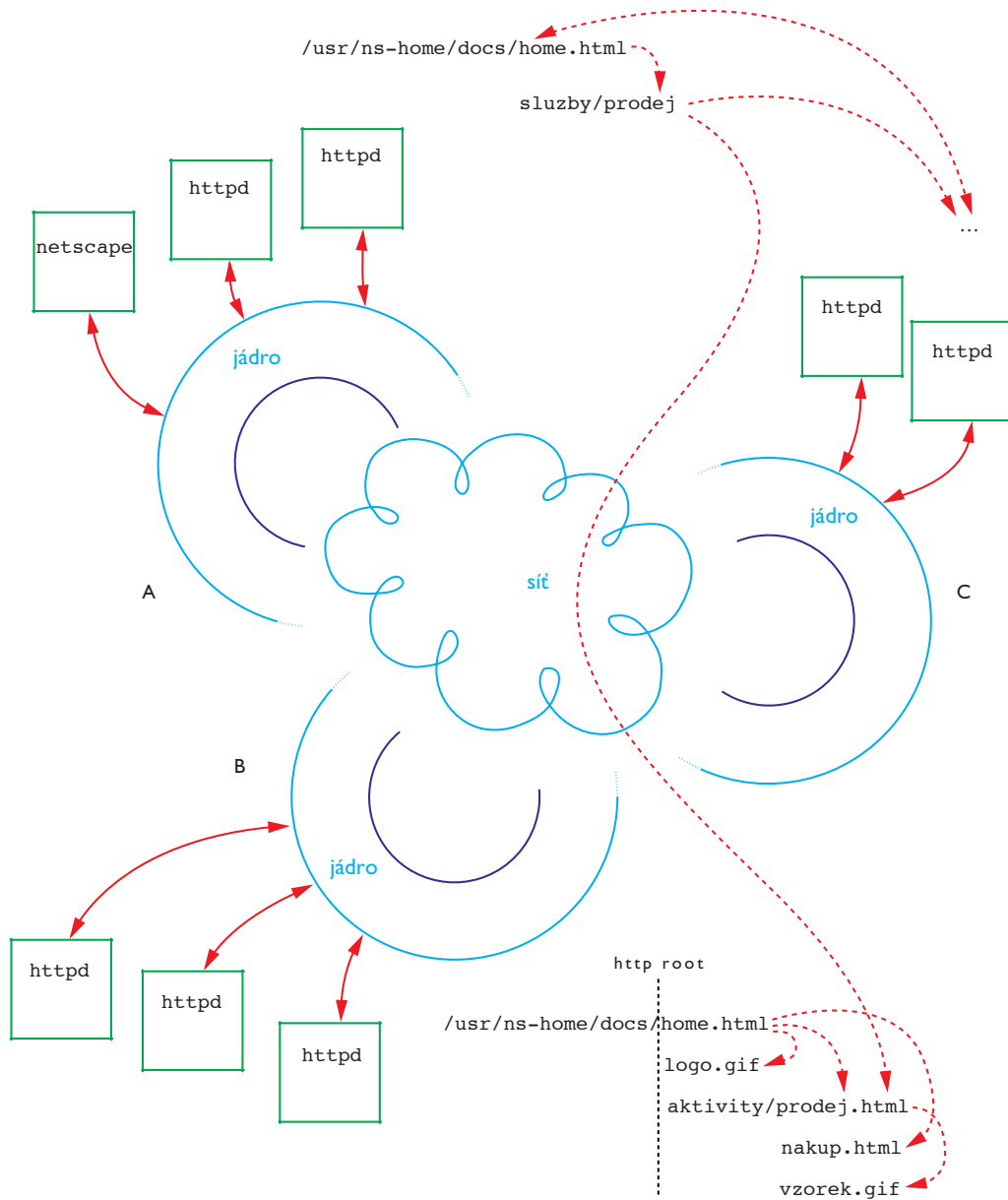
## 7.4.5 Internet a Intranet

Kořeny dnešní největší celosvětové sítě Internet sahají do hluboké minulosti výpočetní techniky tedy do konce 60. let, a sice k projektu vlády Spojených států ARPA a vzniku a provozování jejich vlastní sítě ARPAnet. Výzkum v oblasti IP měl zajistit provoz sítě, přestože její část mohla být náhle zničena. To přineslo charakter síťové vrstvy IP, jak byla popsána v této kapitole, kdy se zbývající část sítě snaží jiným směřováním paketů zajistit požadované spojení. Každopádně jsou pak fyzicky od sebe oddělené lokality provozuschopné samy o sobě. Veřejný zájem o spojování sítí mezi sebou přinesla teprve vlna technologického pokroku na začátku 80. let v podobě stále levnějšího hardwaru pracovních stanic nebo PC a jejich síťových rozhraní. Spojení v místních sítích umožnilo distribuovaný způsob práce, začalo se ustupovat od střediskových silných výpočetních systémů. V té době se o rozšíření způsobu spojování sítí způsobem daným ARPA nejvíce zasloužila NSF (National Science Foundation, Národní vědecká nadace), která vytvořila tzv. síť NSFNet a poskytla ji především univerzitám (hovoříme stále o Spojených státech). Vývoj NSFNet pak poznamenala implementace protokolů ARPA TCP/IP v univerzitních systémech UNIX (tehdy 4.2BSD), kterou provedla skupina BBN. Obliba UNIXu a síťových služeb, jako je pošta, **telnet** nebo **ftp**, se spojila a akademický svět začal podporovat stále rostoucí veřejnou síť Spojených států na bázi operačního systému UNIX. Jméno Internet bylo odvozeno od významu internet-working (nebo internet), spojování sítí, což byl běžný termín ARPA při vývoji IP. V té době se totiž do Internetu začleňovaly další současně se rozrůstající národní sítě sítí.

V průběhu 80. let dosáhl Internet velkého rozšíření a po školách se o Internet začínal zajímat i průmysl a obchod. Internet náhle přestával být veřejnou sítí podporovanou státem. Začínal se stávat prostředkem, ale i předmětem obchodu. Komerční využití však shledalo síť málo bezpečnou z pohledu nejen bankovních transakcí, ale i z hlediska nechtěného zveřejnění interních dat firem. Současný požadavek zachování soukromí z pohledu postulovaných lidských práv přinesl pro 90. léta vývoj Internetu především z pohledu bezpečnosti (viz. kap. 9) a tato situace trvá dodnes.

UNIX byl a zůstává každopádně dominantním operačním systémem Internetu. Mnohé zmiňované síťové aplikace nesou označení aplikace Internetu, protože byly v UNIXu implementovány podle definic ARPA. Konečně hloubka zapuštění kořenů Internetu v UNIXu je zřejmá už od síťové vrstvy IP. Veřejné dokumenty RFC (Requests For Comments, viz úvod kapitoly) jsou pro správce toužícího po erudici v Internetu základním zdrojem informací o současném vývoji. Jejich poskytování je součástí služeb centra správy Internetu. Hlavní směry vývoje Internetu určuje Internet Society neboli ISOC, ze které vychází tzv. rada Internetu IAB (Internet Architecture Board), skupina odborníků, která přijímá stan-





Obr. 7.27 Stránky WWW

dardy a rozděluje zdroje Internetu. Přidělování adres IP a domén DNS pak přesouvá rada na hlavní poskytovatele Internetu v jednotlivých zemích světa. O koho se jedná, ví každý dealer Internetu. Internet se dodnes nepodařilo zcela komercializovat a akademický Internet, který je financován z vládních zdrojů každé země, má stále svou důležitost. Technicky oddělit komerční a akademický Internet je sice obtížné, ale nikoliv nemožné, jak vyplývá z popisu síťové vrstvy IP z pohledu směřování paketů. Takže je možný přístup pracovníka školy a komerční firmy současně, kdy je jednomu počítán počet přenesených bitů a druhému nikoliv, přičemž ten platící je zvýhodněn rychlejším přístupem.

Největší zájem o služby Internetu se týká aplikací elektronické pošty a zobrazování zdrojů informací. Programátoři pak dále rádi vyhledávají volně šiřitelné programové vybavení pro svá PC, které zveřejňují jejich kolegové. Profesionálové dnes běžně využívají Internet k získání oprav chyb v profesionálních produktech renomovaných firem, na jejichž uzlech jsou obvykle aktuální opravy k dispozici pro přenos do místního systému. Jako informační servery vznikly v průběhu vývoje Internetu síťové aplikace GOPHER, WAIS, ARCHIE, anonymní FTP nebo VERONICA, které byly implementovány ještě v době textového přístupu. Přestože dnes existují jejich grafické podoby, je nejvíce používaná grafická podpora informačních databází WWW (World Wide Web), která integruje služby svých předchůdců. Uživatelé dává možnost cestovat v Internetu v grafickém okně (např. klientu **netscape**) obvykle plném statických i pohyblivých obrázků, zvuků a textů. Výchozím kódovacím jazykem WWW je HTML (Hypertext Markup Language), jehož způsob kódování poskytovaných informací vychází ze způsobu formátování dokumentů programů **roff** nebo **nroff** UNIXu. Dalším mezníkem, ke kterému došlo, je zjevně vznik síťového programovacího jazyka JAVA, ve kterém je možné programovat i složité síťové aplikace grafického Internetu a jemuž patří budoucnost.

Každá zmíněná aplikace Internetu má podobný princip implementace, jako jsme uvedli v této kapitole. Principiálně jde o komunikaci dvou procesů technologií klient - server. Např. poskytovaná textová databáze GOPHER pracuje na portu 70. Serverem pro proces uživatele klienta **gopher** je démon **gopherd**. Samozřejmě je nutné aplikaci v uzlu serveru instalovat a konfigurovat pomocí tabulek, zde **gopherd.conf**. GOPHER ovšem jako i další aplikace Internetu umožňuje uživatelům pomocí menu aktivovat servery v dalších uzlech, na které je proveden odkaz. Uživatel tak pomocí klienta cestuje Internetem z uzlu na uzel a nemusí přitom znát umístění ani adresaci nebo označení uzlů, se kterými právě pracuje.

Princip odkazů na servery informací v jiných uzlech je jedna z hlavních vlastností, ze které vychází WWW. Proces server je zde démon **httpd** (jichž bývá v uzlu trvale jádrem registrováno 16). Klientem je proces **netscape** nebo **Mosaic**, případně ve světě PC **explorer**. Aplikace WWW pracuje na portu č. 80. HTML je odkazovací formátovací jazyk, ve kterém je zakódována informační databáze poskytovaná každým uzlem serveru WWW. Server **httpd** posílá informace v HTML klientu, který je teprve interpretuje (formátuje) a zobrazuje uživateli ve svém okně. Pomocí HTML se informace rozdělují do *stránek* (pages). Klient zobrazí vždy jednu stránku, o kterou uživatel požádal. Informační databáze každého uzlu je množina takových stránek. Každá stránka je uložena vždy v jednom souboru. Klient odkazuje vždy na jednu stránku a uživatel si z ní po zobrazení vybírá případnou cestu ke stránce další. Znamená to, že stránky v uzlu serveru tvoří postupně odkazovaný strom stránek. Klient ovšem může začít na libovolné stránce tohoto stromu a nezávislých stromů stránek může být v uzlu více. Termín strom z teorie grafů jsme použili pouze pro první přiblížení, ale zdaleka nevyhovuje skutečnosti. Stránky totiž mohou být propojeny vzájemnými odkazy zcela libovolně, takže mohou vznikat (a běžně

vznikají) cykly, (tj. odkazy zpět na výchozí stránku). Stejně tak mohou být dvě množiny stránek propojeny do libovolného místa (a nikoliv pouze na výchozí stránku množiny). Odkaz na jinou stránku nazýváme *hyperlink*, výchozí stránku určité množiny označujeme termínem *domovská stránka* (home page). Množiny většinou určují téma nebo předmět zájmu informační databáze, přestože uzly mají většinou svoji výchozí domovskou stránku (uzel patří obvykle určité organizaci nebo firmě), ve které se předměty zájmů rozvětvují. Množiny stránek jsou takto umísťovány v uzlech sítě Internet a vytváří tak jednotlivé databáze zveřejňované servery **httpd**, servery WWW. Odkaz na určitou stránku pak může znamenat odkaz v rámci uzlu, ale struktura WWW umožňuje odkazovat i libovolnou stránku v rámci celého Internetu. Klient rozpozná odpovídající odkaz jako odkaz do jiného místa Internetu (podle označení DNS) a požádá si tamní server **httpd** o obsah odkazovaného souboru, který po přenesení uživateli v okně klientu zobrazí. Uživatel tak mění zdroje informací Internetu a mnohdy ani neví, že stránky přicházejí každou chvíli z jiné části světa (přestože klienty probíhající odkaz na stránku vždy komentují). Podívejme se na obr. 7.27.

Každý uzel (A, B nebo C) na obrázku poskytuje služby WWW. V uzlu A je právě spuštěn klient **netscape**, který je uživatelským procesem zobrazujícím stránky serverů. Šípkami jsou znázorněny odkazy jednotlivých souborů (stránek). Odkazy jsou jednosměrné: z výchozí stránky uzlu A vede odkaz jak na místní soubor, tak na soubory vzdálených uzlů B a C. Zdrojové texty stránek HTML mají konvenci přípony `.html` (nebo jen `.htm`), přesto je na obrázku znázorněn i odkaz z některých stránek na soubory jiného ukončení. Soubory se zakončením `.jpg` nebo `.gif` jsou konvencí pro rozpoznání souborů s obsahem ve grafickém formátu JPEG (grafický standard definovaný skupinou Joint Photo-Graphic Experts Group) a GIF (Graphics Interchange Format vyvinutý firmou CompuServe). Klient WWW totiž umí tyto grafické soubory zobrazit a autor dokumentů v HTML je vkládá na určité místo stránky odkazem na jejich jména.

Klient a server WWW využívají definovaného protokolu aplikace HTTP (Hypertext Transfer Protocol) pro vzájemné dorozumění. Znamená to, že jejich aplikační úroveň je definována ze strany serveru zápisem v HTML (server poskytuje obsahy odkazovaných souborů) a ze strany klientu požadavky, které formulují odkaz. Tato strana klientu byla definována jako tzv. dotazy URL (Uniform Resource Locator, jeho obecnější podoba je URI, Uniform Resource Identifier, ale URI se používá velmi málo, viz RFC 1737), tj. jednoznačná definice určitého zdroje v rámci Internetu. Zdrojem přitom může být nikoliv pouze jméno souboru s HTML, ale obecná definice jak datové části zdroje, tak metoda jeho získání, přesněji označení síťové aplikace, která jej využije pro potřeby klientu. Obecný formát URL má tvar:

```
service://user:password@host:port/path
```

Jednotlivé části definice uvedené kurzivou mohou přitom být nepovinné. Typické použití pro odkaz klientu WWW na určitou stránku je např.

```
http://www.ffa.vutbr.cz/skoc/home.html
```

kde je definován soubor `/skoc/home.html` (domovská stránka autora této knihy) v uzlu domény DNS `www.vic.cz`. Služba je **http**, protokol HTTP. Jak vyplývá z obecné definice URL, jde o snahu sjednotit Internet z pohledu odkazu na zdroj. Na místě `service` tak můžeme použít i jiné služby Internetu, např. **ftp** pro přenos souborů. V dalších částech URL pak určujeme uzel a cestu k souboru, který požadujeme přemístit do uzlu klientu. Zde se uplatní také `user` a `password`, pokud nevyužíváme anonymní FTP. Klient v případě FTP vytváří po navázání spojení se serverem na odpovídajícím portu

dítě, které spojení zdědí a o přenos se jako klient FTP postará. Nebo lze použít na místě **service gopher, wais, telnet, mailto, file** aj. Některé (např. **gopher**) provádí klient v rámci své práce v okně, na jiné vytváří nové dítě (např. **telnet**). Část **host**, jak už z příkladu vyplývá, je označení uzlu v rámci DNS, kde se zdroj nachází, ale na jeho místě lze také použít adresu IP. Cesta ke zdroji **path** je cesta ke jménu souboru, který znamená požadovaný zdroj. Výchozí adresář, tj. kořen (root) tohoto odkazu, tedy adresář označovaný **/**, zde obecně neznamená totéž co kořenový adresář při běžné práci uživatele v UNIXu. Každá označená služba zde podléhá výchozímu adresáři serveru, který službu v daném uzlu poskytuje. Znamená to, že např. pro anonymní službu **ftp** bude adresář **/** shodný s adresářem uživatele se jménem **ftp** osloveného uzlu (takový je úzus pro nastavení anonymního FTP). Pro **telnet** je **/** domovský adresář uživatele **user**. Server HTTP (démon **httpd**) podléhá nastavení správcem uzlu v době instalace. Dnes je výchozí adresář umísťován do různých částí systémového stromu adresářů, např. server firmy Netscape Communications Corp. používá adresář **/usr/ns-home/docs** (**/usr/ns-home** je výchozí adresář veškerého softwaru serveru), odkud jsou umísťovány soubory se stránkami WWW, viz také obr. 7.27. Toto nastavení adresáře **/** pro službu Internetu s označením dosažitelnosti pouze jistého podstromu souvisí s bezpečností ostatních dat uzlu. Některé služby však spojují **/** s výchozím adresářem uzlu. Příkladem může být služba **file**. Jde o požadavek zobrazení obsahu souboru, např.

```
file://localhost/usr/ns-home/docs/logo.gif
```

je pokyn pro otevření souboru v místním uzlu s uvedenou cestou, ale lze zjednodušeně také psát

```
file://usr/ns-home/docs/logo.gif
```

Odkazy URL používá klient v rámci HTTP, server poskytuje zdrojový kód stránky WWW v HTML. Ve zdrojovém kódu HTML se běžně vyskytují další odkazy URL, které klient vyřizuje navázáním spojení s odpovídajícím serverem. Např.

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML>
<HEAD>
<TITLE>Výchozí dokument uzlu B</TITLE>
</HEAD>
<BODY BGCOLOR="white">
<P ALIGN=CENTER>
<IMG SRC="/logo.gif" BORDER=0 ALT="[LOGO]"><BR>
Naše nabídka je prostá:<BR>
<A HREF="/activity/prodej.html">prodáváme</A>,
<A HREF="/activity/nakup.html">kupujeme</A> a
<A HREF="http://www.A.imagine/home.html">spolupracujeme</A>.
</P>
</BODY>
</HTML>
```

je zdrojový text dokumentu v souboru **home.html** uzlu B. Stránku formátuje klient v okně. Nejprve zobrazí obsah místního souboru **logo.gif**. Označení **IMG** je v HTML identifikace obrázku. Obrázek bude bez rámečku (**BORDER=0**) a pokud by klient záměrně nevyřizoval odkazy na obrázky (ale pouze

texty, přenos je pak rychlejší), bude na místě obrázku uveden text `LOGO`. Pod obrázkem bude uveden ve středu dvou řádků (`ALIGN=CENTER`) text takto

[ `LOGO` ]

Naše nabídka je prostá:

prodáváme, kupujeme a spolupracujeme.

Podtržené texty jsou citlivé na kliknutí myši. Text prodáváme je přitom vázán na odkaz URL `/aktivita/prodej.html`, kde všechny části s výjimkou cesty k souboru (podle výchozího adresáře démonu `httpd`) jsou vynechány – jedná se totiž o místní odkaz. Analogický je případ na myš citlivého textu kupujeme. Třetí citlivý text spolupracujeme využívá širší odkaz URL. V uzlu `www.A.imagine` (je naše imaginární síť DNS) požaduje obsah souboru `home.html` ve formátu protokolu HTTP (v jazyce HTML). Kliknutím na text spolupracujeme dojde k navázání spojení mezi klientem a serverem uzlu A, po přenosu obsahu souboru `home.html` bude jeho obsah formátován v okně klientu a náš dosavadní text bude novou stránkou vystřídán. Uvedený příklad textu HTML obsahuje také základní konstrukce pro definici textu jako HTML (mezi `<HTML>` a `</HTML>`), je vymezena hlavička (mezi `<HEAD>` a `</HEAD>`), její titul (mezi `<TITLE>` a `</TITLE>`) a vlastní tělo textu HTML (mezi `<BODY>` a `</BODY>`). Text je uveden v paragrafu (mezi `<P>` a `</P>`), odkazy se požadují označením mezi `<A>` a `</A>`. Vůbec první řádek textu je popisný a může být vynechán.

Aplikační protokol HTTP je neustále ve vývoji (viz dokumenty RFC), požadavky na schopnosti formátovacího jazyka HTML totiž stále vzrůstají. Obľiba WWW aplikací přináší snahu přenášet a zobrazovat zvuky nebo video, pokud možno všechno najednou, jak zní označení multimediálních aplikací. K tomu, aby se stala práce ve WWW více profesionální, byl vyvinut programovací jazyk JAVA. Jedná se o velmi silný objektově orientovaný programovací jazyk s velkou podporou výstavby grafických aplikací, jehož použití je jak místní, tak i síťové. Jeho chování je typické pro přenositelné aplikace na úrovni binárního kódu. Proveditelné programy jsou sice překládány kompilátorem, ale pouze do mezikódu, pro který musí existovat na různých platformách interpret. Tím je dosažena proveditelnost na libovolném hardwaru. Interpretem jazyka JAVA je dnes každý klient WWW. Program, na který je proveden odkaz, je tedy přenesen do uzlu klientu, který jej pak interpretuje na obrazovce uživatele (třeba již mimo hranice okna klientu). Protokol HTTP byl ihned po úspěchu jazyku JAVA rozšířen o možnosti vkládat do textu jazyka HTML tzv. JAVA applets (jablíčka JAVY, mezi `<APPLET>` a `</APPLET>`), což jsou fragmenty programovacích sekvencí, jejichž syntaxe vychází z JAVY, ale je její podmnožinou. Každopádně tak dochází k rozšíření HTML pro potřeby programování, a to pouze tam, kde se nevystačí s konstrukcemi formátování textů nebo umísťování obrázků. Jazyk JAVA je považován za následníka jazyka C.

Komfort uživatelského prostředí, které používá každý klient WWW, vedl výrobce pracovních stanic a PC k myšlence vytvořit prostředí téhož typu pro práci nikoliv pouze v Internetu, ale také pro potřeby místní sítě, místní sítě sítí anebo několika různých podsítí v rámci Internetu (firma má běžně filiálky v různých částech země, státu i světa). Pro takto využívané síťové aplikace se začal používat termín *Intranet*. Intranet je především postaven na komfortu grafických aplikací, které jsou klienty databázových distribuovaných serverů. V současné době se jedná o atraktivní artikl softwarových firem, které zajímá především komerční stránka. Vznikají různé varianty, ale žádná z nich se neukazuje jako významná pro jednoznačný směr prostředí Intranet. Z pohledu tématu naší kapitoly se jedná o uplatnění známých postupů aplikační vrstvy sítě a jedná se více o nově definované aplikační protokoly mezi klientem a serverem.

## 7.5 Protokol IP jako aplikace (UUCP, PPP, SLIP)

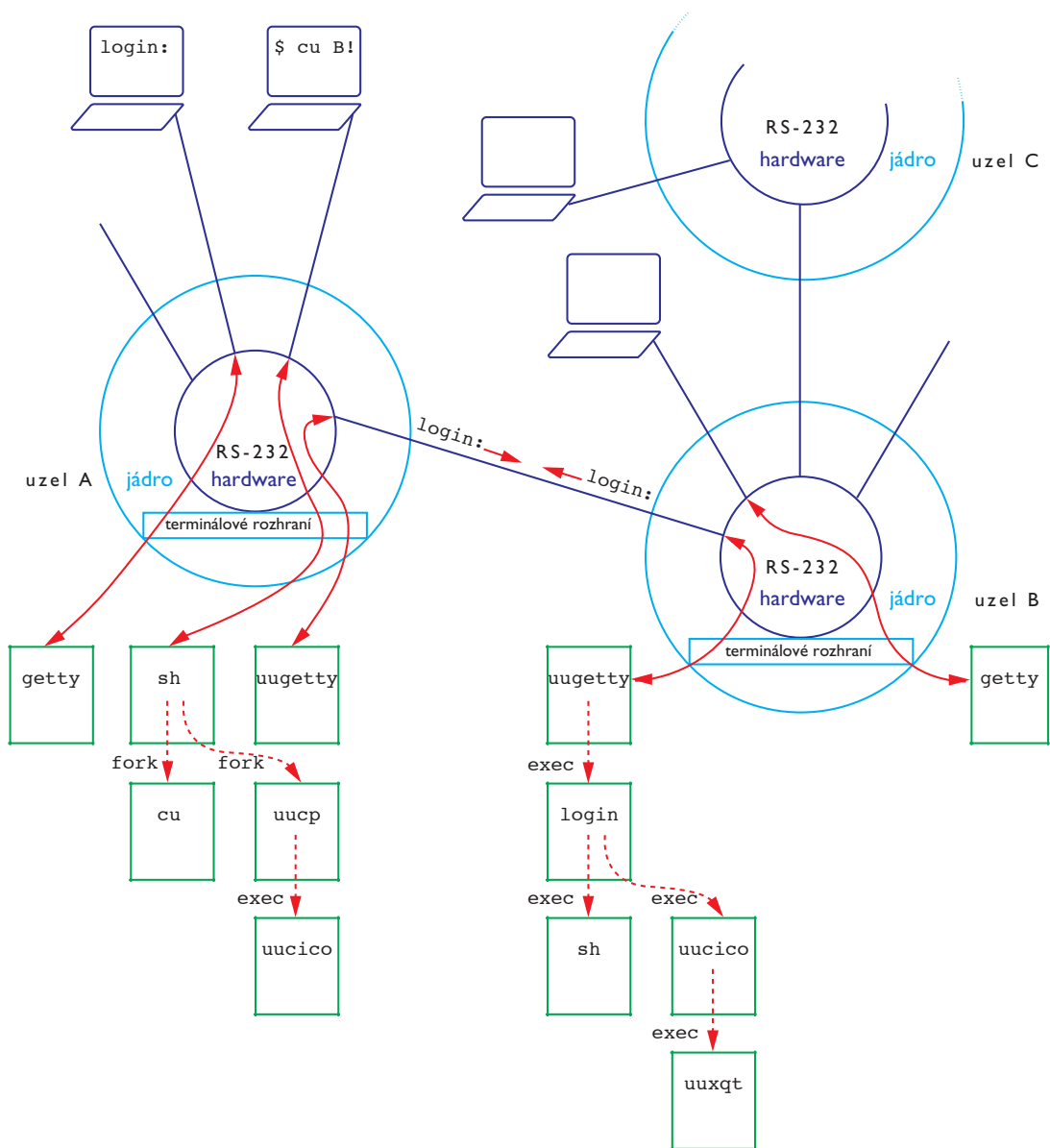
Již v UNIX version 7, první veřejné verzi operačního systému UNIX začátkem 80. let, byl její součástí komunikační software nazvaný UUCP. Jeho název byl odvozen z jeho nejpoužívanějšího příkazu pro kopii souborů mezi dvěma uzly **uucp** (UNIX to UNIX Copy), jehož syntaxe byla odvozena z příkazu **cp**. UUCP je spojení dvou operačních systémů UNIX terminálovým rozhraním. Namísto fyzického terminálu je kabelem připojen druhý počítač opět svým terminálovým rozhraním. Hardware je spojen přímo (tzv. null-modem), ale mezi systémy může být vložena i telefonní ústředna. Systém UUCP je nastaven tak, že pro terminálové rozhraní pracuje proces umožňující přihlašování do systému, jak jsme si ukázali v kap. 2. Základní situaci ukazuje obr. 7.28.

Proces **uugetty** je pouze upravená verze procesu **getty**, a to tak, aby mohlo být terminálové rozhraní využito z obou systémů symetricky, jak si ukážeme dál. Vzdálené přihlášení do systému pak realizuje proces **cu**, který na našem obrázku použil uživatel přihlášený na terminálu v uzlu A. Vzniklý proces **cu** podle parametrů příkazového řádku použije odpovídající terminálové rozhraní k emulaci terminálu pro vzdálený systém. Na obrazovce uživatele systému A se objeví text **login:**, který ze systému B zapisuje do svého terminálového rozhraní proces **uugetty**. Přihlášení a práce uživatele v systému B pak pokračuje tak, jak je uživatel zvyklý. Po zadání jména a hesla registrovaného v uzlu B je proces **uugetty** voláním jádra **exec** postupně proměněn v **login** a odpovídající shell. V průběhu sezení řízeného procesem **cu** může uživatel použít znak výluky ~ pro přepnutí komunikace z procesu vzdáleného shellu na místní proces. Za znakem ~ následuje příkaz pro **cu**. Může to být znak . pro ukončení **cu** (spojení zaniká, skončí vzdálený shell a proces **init** ve vzdáleném systému opětne vytváří **uugetty**) nebo texty **take** nebo **put** s možností přenosu souborů ze vzdáleného systému do místního či naopak; viz provozní dokumentace příkazu **cu**.<sup>8</sup>

Další příkaz je **uucp**, kterým je možné provádět kopii souborů mezi systémy. Pokud je terminálové rozhraní volné (tj. není obsazeno příkazem **cu**), aktivuje **uucp** ve vzdáleném systému proces **uugetty** stejným způsobem, jako to provede uživatel prostřednictvím **cu**. V každém systému UNIX je implicitně registrován uživatel se jménem **uucp**, pod kterým se proces **uucp** ve vzdáleném systému přihlašuje. Nahlédneme-li do souboru **/etc/passwd**, nalezneme na místě výchozího shellu uživatele **uucp** odkaz na soubor s programem s relativním jménem **uucico** (Unix to Unix Copy In Copy Out)<sup>9</sup>. Jedná se o proces, který zabezpečuje protokol UUCP a ve který se po úspěšném přihlášení promění proces **login**. Proces **uucp**, který přihlášení inicioval, se po ohlášení **uucico** ve vzdáleném uzlu promění pomocí **fork** také na proces **uucico**. Oba spolupracující procesy nyní provedou uživatelem požadované přenosy souborů. Po uskutečnění přenosu procesy skončí svoji činnost, přihlášení zaniká, terminálové rozhraní je uvolněno. Zánik procesu **uucico** ve vzdáleném systému registruje tamní **init**, který nově startuje **uugetty** pro potřeby další aktivity UUCP.

Další příkaz UUCP, který SVID definuje, je **uux** pro provedení příkazu ve vzdáleném uzlu (v parametrech jmen souborů příkazu můžeme používat jak místní, tak vzdálené soubory). Princip komunikace mezi dvěma uzly pomocí terminálového rozhraní přitom zůstává zachován. Proces serveru, který požadavek na provedení příkazu převeze a realizuje, je **uuxqt**.

Přenos souborů prostřednictvím **uucp** není možný do libovolného adresáře vzdáleného systému. Přístupová práva v UNIXu podléhají jednotlivým uživatelům, proto UUCP ukládá přenášené soubory pouze do míst, kde je povolen zápis pro uživatele **uucp**. Pokud uživatel nezná strukturu adresářů a přístupovo-



Obr. 7.28 Schéma zapojení UUCP



vých práv vzdáleného uzlu, používá notaci `~/` a data tak přenáší do veřejné oblasti, kterou získává proces **uucico** ve vzdáleném uzlu z obsahu proměnné prostředí procesu (proměnné shellu) `PUBDIR`, což bývá např. adresář `/usr/spool/uucppublic`. Do téhož adresáře umísťuje **uucico** soubory, jejichž zápis do určitého adresáře právě z důvodů přístupových práv selhal.

Adresace uzlu v rámci uzlů UUCP je notací

`jméno_uzlu!`

Za znakem `!` pak následuje cesta k souboru nebo adresáře. `jméno_uzlu` je jedno z registrovaných jmen dosažitelných z uzlu prostřednictvím nastavených terminálových linek v tabulkách UUCP. Např.

```
$ uucp "*.c B!~/src"
```

je kopie všech zdrojových souborů jazyka C do podadresáře `src` adresáře podle `PUBDIR` vzdáleného uzlu se jménem `B`.

Získat seznam všech takových uzlů může uživatel příkazem **uuname**, volbou `-l` získá jméno místního uzlu. Přestože takové jméno uzlu UUCP nijak nesouvisí se jménem uzlu podle odst. 7.4.1, je možné nastavit agent transportu a doručení elektronické pošty prostřednictvím UUCP. Uzlů sítě UUCP totiž může být více a terminálová rozhraní mohou spojit systémy UUCP různým způsobem. Podle obr. 7.28 lze např. adresovat uzel C z uzlu A způsobem

```
B!C!
```

Pokud je uzel A navíc ještě připojen k Internetu a adresovatelný prostřednictvím DNS např. jako `A.imagine.cz` s poštovní doménou `imagine.cz`, pošta uživateli se jménem `petrn` uzlu C, za předpokladu správného nastavení agentů pošty, je adresovatelná

```
B!C!petrn@imagine.cz
```

Uzel A je tak bránou (gateway, je totiž současně uzlem `A.imagine.cz`) mezi sítěmi TCP/IP a UUCP.

Pro cílený přenos souborů určitým uživatelům bylo UUCP takto rozšířeno o příkazy **uuto** a **uupick**. Příkaz **uuto** posílá požadované soubory do požadovaného uzlu s umístěním do veřejné oblasti, ovšem s označením uživatele, kterému soubory mají patřit. Např.

```
$ uuto "*.c B!C!petrn"
```

je kopie souborů do systému C s označením pro uživatele `petrn`. Uživatel `petrn` v sezení uzlu C prohlíží všechny takto proběhlé transakce příkazem **uupick** a rozhoduje, jak soubory z veřejné oblasti UUCP přemístí. Odesílající uživatel může volbou `-m` v příkazu **uuto** požadovat oznámení proběhlé transakce UUCP cílovému uživateli poštou.

Všechny transakce UUCP jsou monitorovány a ukládány do systémového souboru (obvykle `/usr/spool/uucp/LOGFILE`). Uživatel může proběhlé transakce prohlížet příkazem **uulog**, volbou `-s` se přitom může zaměřit pouze na konkrétní uzel. Privilegovaný uživatel má navíc možnost prohlížet a modifikovat frontu požadavků UUCP příkazem **uustat** (v některých systémech **uuq**), a to ve všech definovaných uzlech, pokud fronty UUCP tyto uzly podporují (požadavek může být prováděn ihned).

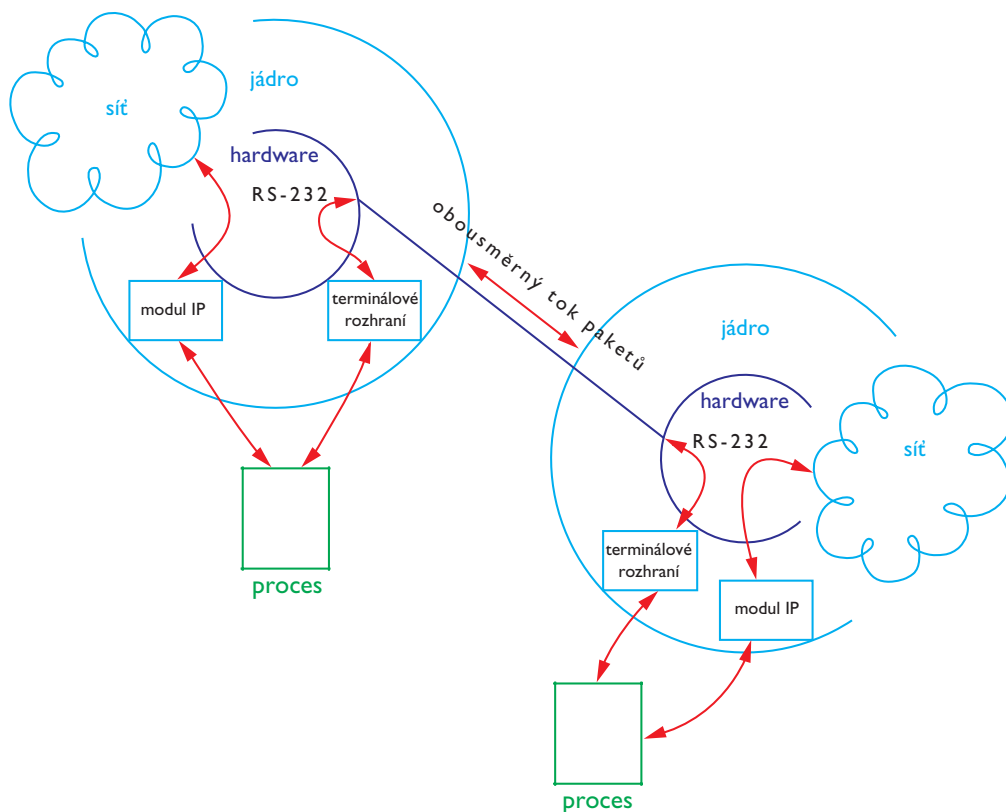
Nastavení a správa UUCP je prováděna obvykle editací systémových tabulek v adresáři `/usr/lib/uucp` (ale někdy také v `/etc/uucp`). Důležité soubory pro nastavení UUCP jsou `L.sys` (někdy `Systems`), kde je definován seznam sousedních uzlů, `L-dialcodes` (nebo `Dialers`) s tele-



fonními čísly terminálových linek vzdálených uzlů připojených přes telefonní ústřednu, **L-devices** (nebo **Devices**), ve kterých je soupis typů modemů, které lze využívat v **L-dailcodes**. Správce uzlu může také v souboru **USERFILE** definovat jména výhradních uživatelů, kteří mohou využívat UUCP, a v souboru **L.cmds** vyjmenovat příkazy, které je možné provádět příkazem **uux** ze vzdálených uzlů. Přezdívkový uzlů jsou umístěny v souboru **L.aliases**. V tomto adresáři bývají uloženy také programy **uucico** nebo server **uuxqt**.

Soubory UUCP, které vznikají při samotném provozu, bývají umístěny v adresáři **/usr/spool/uucp**. Jsou to např. již zmiňovaný **LOGFILE** se záznamem transakcí UUCP, dále ho doplňuje **SYSLOG** a **ERRLOG** se záznamem vyskytnutých chyb UUCP.

Správce systému pak nesmí zapomínat také na doplnění tabulky **/etc/inittab** pro start procesu **uugetty**, jehož parametrem je terminálové rozhraní a způsob jejího nastavení. Proces **uugetty** přitom může být startován z obou stran terminálového rozhraní, tj. v každém systému (viz obr. 7.28). Při aktivaci z jedné strany zůstane místní **uugetty** zablokován.



Obr. 7.29 Vrstva IP jako aplikace

UUCP je případ síťového spojení počítačů typu peer-to-peer. Jde o síť dvou počítačů. Přitom hardware, který je spojuje, realizuje obsluhu právě dvou uzlů. Vzhledem k tomu, že spojení dvou počítačů telefonní linkou je tento typ a UUCP od začátku dokázalo pracovat s terminálovým rozhraním za účasti dvou modemů jako s vytáčenou nebo pevnou telefonní linkou, hledala se cesta obsluhy takového spojení typem sítě TCP/IP za využití již prověřeného UUCP. Základní myšlenka je opět ve vzdáleném přihlášení, kde se na místě procesu shellu startuje proces, který bude terminálovové rozhraní spojit s vyhrazenou adresou IP. Tento proces bude také síťovou aplikací pracující s vrstvou IP. Znamená to, že bude spolupracovat s IP v jádru tak, že modul IP jej aktivuje vždy při odkazu na jemu vyhrazenou adresu IP. Předává mu pakety IP a naopak, pakety přicházející terminálovým rozhraním jsou předávány modulu IP v jádru k dalšímu směrování. Podívejme se na obr. 7.29.

Jde o trochu zvláštní (ale funkční) rozšíření síťové vrstvy mimo oblast jádra. Jak je na obrázku naznačeno, oba uzly mohou dále používat jiný druh síťového spojení, tj. oba jsou účastníky další sítě. Síť peer-to-peer v tomto případě hraje tedy i roli směrování. Oba uzly jsou směrovače s odpovídající adresou IP pro další síť TCP/IP, jejíž je každý uzel také součástí.

Realizace takového způsobu rozšíření vrstvy IP je pomocí aplikačního (protože na úrovni procesu) protokolu SLIP (Serial Line Interface Protocol) nebo mladšího PPP (Point to Point Protocol).

V rámci PPP je použit pro realizaci spojení proces **ppp**. Při jeho práci používáme opět termín server a klient. Mechanismus vytvoření spojení totiž probíhá startem procesu **ppp** v uzlu, který na odpovídající terminálové lince očekává výzvu k přihlášení vzdáleného systému (řetězec končící na `login:`). Tento proces klientu provede sekvenci odeslání textů jména a hesla smluvného uživatele (např. `pppcli`). Uživatel je ve vzdáleném systému rozpoznán podle tabulky v `/etc/passwd` a přihlašovací proces **login** se promění na proces **ppp**, který je serverem spojení PPP. Klient i server provedou dohodu o nastavení podle svých konfigurací adresy IP (kdy každá strana ohlásí svoji adresu IP vztaženou k používanému terminálovému rozhraní) a směrování a zablokují se do příchodu prvních paketů jim odpovídající adresy IP. Každá strana spojení (odpovídající proces **ppp**) je konfigurována ve svém souboru `ppp.conf` (obvykle v adresáři `/etc`). Na straně klientu je definován server a opačně. `ppp.conf` přitom využívá konfiguračních souborů UUCP, jako je např. soubor s definicí vzdálených systémů `L.sys`, použitého modemu `L.devices` nebo případného vytáčeného telefonního čísla `L.dialcodes`. Zde je také definována adresa IP a způsob směrování. Podrobný popis obsahu `ppp.conf` pak nalezne čtenář v provozní dokumentaci příkazu **ppp**. Na straně serveru je důležité nezapomínat na start procesu **getty** (v `/etc/inittab`), který klientovi zpřístupňuje vzdálený uzel. Na rozdíl od UUCP je terminálové rozhraní využíváno v okamžiku přihlašování klientu vždy pouze z jedné strany. Použití procesu **uugetty**, a to dokonce z obou stran, zde ztrácí smysl. Na straně klienta je proces **ppp** startován z příkazového řádku privilegovaného uživatele nebo z některého ze scénářů při startu systému.

Na rozdíl od PPP vyžaduje SLIP označení pevnou adresou IP na obou stranách spojení. Vzhledem k tomu, že neobsahuje žádné zabezpečení pro případ nekvalitních spojů nebo dokonce kompresi dat (opět na rozdíl od PPP), je v naší zemi pro modemové spojení dvou uzlů prakticky nepoužívaný. I SLIP využívá konfigurační soubory UUCP podobně, jak jsme naznačili u PPP. SLIP realizuje proces **slattach** nebo **sliplogin**. Pomocí **slattach** spojíme sériové rozhraní s adresou IP, ale při startu musíme (obvykle v parametrech příkazu) stanovit také adresu IP vzdáleného uzlu. Vzdálený uzel používá také nastavení pomocí **slattach** se zadáním adres IP v opačném pořadí. Před samotným

startem procesu **slattach** je přitom nutno navázat spojení obou uzlů prostřednictvím UUCP. I zde proto lze hovořit o straně klientu a serveru, ovšem pouze v UUCP, procesy **slattach** pak provozují protokol SLIP podle síťových požadavků a předávají si vzájemně pakety IP. Používáný je také proces **sliplogin**, který je uživatelem startován po přihlášení do vzdáleného uzlu namísto některého z shellů, a to odpovídajícím zápisem v tabulce souboru `/etc/passwd`. Přichází uživatel ve svém místním systému pak startuje proces stejného jména, tj. **sliplogin**, který je klientem vzdáleného **sliplogin**. Klient je konfigurovaný na odpovídající adresy IP v příkazovém řádku. Server využívá konfiguračního souboru `/etc/hosts.slip`, kde jsou na jednotlivých řádcích uvedena jména uživatelů a konfigurace adres IP pro spojení SLIP s klientem. Tak jsou sice adresy IP zadány pevně, ale pro různé uživatele se mohou měnit. V daném okamžiku však mohou být sériovým rozhraním spojeni pouze dva uzly.

UUCP a následně PPP nebo ještě SLIP jsou omezeny, ať už přenosovými rychlostmi telefonického spojení nebo pouze sériového rozhraní terminálu. Přesto je dnes jejich použití značné a předvedená metoda vzdáleného přihlašování je používána pro vstup do sítí UNIXu i z jiných typů operačních systémů (např. MS Windows-NT). Vzhledem k rychlému vývoji nových typů telefonických spojů (např. ISDN) se očekává nový způsob podpory, který pak nebude koncipován jako míchání síťové a aplikační vrstvy, ale bude vyhovovat současným standardům síťového spojení.

<sup>1</sup> Často, a dokonce i v odborné literatuře, je používán termín brána místo směrovač. Z pohledu síťových aplikací vlastně není důvodu k rozlišování. My se však v textu budeme snažit oba termíny striktně odlišovat.

<sup>2</sup> Čtyři vrstvy sítě jsou definovány také ve všeobecně uznávaném dokumentu Ministerstva obrany Spojených států (viz [DoD83]). Vrstvy jsou zde pojmenovány

Network Access Layer (přístupová vrstva k síti) – obsahuje přístupové rutiny k fyzické síti..

Internet Layer (síťová vrstva) – definuje datagram a zajišťuje směrování dat.

Host-to-Host Transport Layer (přenosová vrstva mezi dvěma uzly) – zajišťuje služby přenosu dat ve výchozím a cílovém uzlu.

Application Layer (aplikační vrstva) – aplikace a procesy, které využívají síť.

<sup>3</sup> Později uvidíme, že více adres IP může mít i jediné síťové rozhraní.

<sup>4</sup> Analogie s Berkeley sockets je zřejmá.

<sup>5</sup> Při komerčním používání byla přejmenována na NIS, protože Yellow Pages je ochranná známka firmy British Telecom.

<sup>6</sup> Říká se, že elektronická pošta je jeden z hlavních důvodů, proč UNIX dosáhl takové obliby.

<sup>7</sup> Používány jsou také další poštovní systémy, např. **smail** nebo **mmdf** (Multi-channel Memorandum Distribution Facility – je používán v SCO UNIX, původně vyvinut organizací CSNET).

<sup>8</sup> **cu** nepoužívá pro přenos souborů příkazem `put` nebo `take` žádný zvláštní protokol. Ve vzdáleném systému používá příkazy **cat** a **stty** jako děti pracujícího shellu pro získání a přenos obsahu požadovaného souboru. Případné kontrolní součty nebo jiné zabezpečení zde tedy nejsou.

<sup>9</sup> Uživatel `nuucp` slouží k testování UUCP a má nastaven obvyklý shell.



## 8 X

Dnes si stěží obyčejný uživatel dokáže představit komunikaci s počítači bez grafického prostředí. Je také zjevné, že změna práce s výpočetní technikou z textové podoby na podobu obrázkovou nebyvale rozšířila řady obyčejných uživatelů. Specialisté na vnímání člověka také poukazují na nesrovnatelně vyšší sdělnost obrázku oproti textovému vyjádření<sup>1</sup>. Proto můžeme pokládat grafický způsob práce s počítačem jednoznačně za novou epochu zpracování informací. Pro práci v grafickém režimu je ale potřebné nejen hardwarové vybavení (dobrá barevná obrazovka podporovaná grafickým adaptérem a ukazovací zařízení, jako je myš, trackball, tablet nebo joystick), ale i podpůrný software, který ovládá obrazovku a ukazovací zařízení s klávesnicí a umožňuje obrazové informace odkazovat, přesunovat, vytvářet je a měnit. První, kdo s myšlenkou určitého uspořádání práce s obrázky uspěl, byl vývojový tým firmy Apple, který na svých počítačích Macintosh začátkem 80. let vytvořil prostředí tzv. oken (Windows), kdy každé okno je věnováno určité grafické (ale třeba i textové) aplikaci. Okno je pochopitelně na obrazovce libovolné množství a jejich velikost se může měnit od pokrytí celé obrazovky až po zmenšení do tzv. ikony, tj. do symbolu, který okno pouze zastupuje. Plocha obrazovky, na které jsou okna rozmístěna, je nazývána desktop (pracovní deska), jejíž součástí bývají grafické objekty umožňující vytváření nových oken a jiné úpravy pracovní desky. Uživatel tak může střídát práci v různých oknech (práci na různých aplikacích) a přenášet grafické informace z jednoho okna do druhého, a to především pomocí myši a klávesnice. Pracovní stanice Macintosh však neposkytovaly víceprocesový systém. Do dnešního dne je jejich základní operační systém vybaven pouze jednoprocovým prostředím. Aplikace sice mohou běžet v několika oknech, ale po opuštění okna se aplikace pozastaví a očekává se pokračování v aplikaci podle uživatelského zájmu. S rozvojem osobních počítačů IBM PC a jejich derivátů různých firem na začátku 80. let, a to především s jejich stále výkonnější grafickou podporou, poskytla firma Microsoft uživatelům systém oken MS Windows, který se vzhledem k nesmírné popularitě osobních počítačů rozšířil mezi drobné uživatele nejvíce a je mylně považován za systém určující vývoj. MS Windows byl stejně tak jednoprocový systém a pracoval jako grafická aplikace MS DOS, tedy bez jakékoliv systémové podpory a ochrany. Navíc nebyla jeho koncepce přístupu uživatele tak dokonalá, jako tomu bylo u počítačů Macintosh. Teprve dnešní známá verze MS Windows-NT, která staví na zcela jiných principech, je jedinou verzí firmy Microsoft pro osobní počítače, která se dnešním propracovaným profesionálním systémům zpracování grafických informací pracovních stanic dokáže vyrovnat.

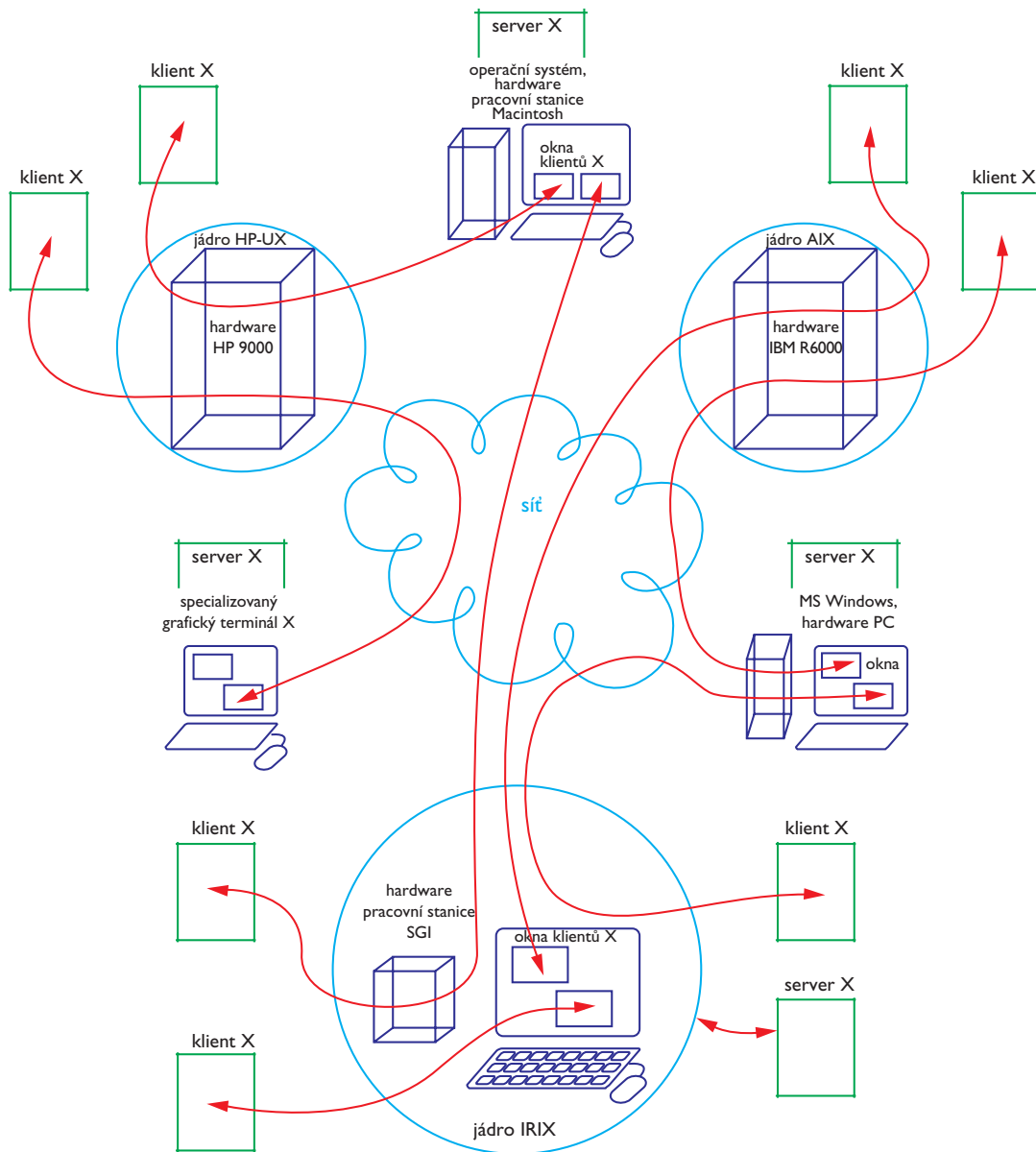
V průběhu let se také objevil termín *grafický operační systém* (Graphics Operating System), který měl označovat operační systémy pracující pouze na bázi grafického přístupu. Ukázalo se však, že grafické zpracování informací je nejlépe koncipovat jako další vrstvu operačního systému, na které jsou dolní vrstvy (jádro, hardware, knihovny) nezávislé, a aby i samotný grafický systém byl nezávislý na použitém operačním systému. Více než grafický operační systém se proto začal používat termín grafické uživatelské rozhraní, GUI (Graphical User Interface), tedy prostředí práce uživatele způsobem obrazové komunikace, která je součástí aplikační úrovně operačního systému. Tento způsob implementace byl také přijat jako směrodatný pro vývoj grafického prostředí pro operační systém UNIX.

UNIX samotný pracuje jako víceprocesový. Současný běh několika aplikací v různých oknech pro něj tedy není nepřekonatelný problém, ale naopak představuje využití jeho přirozených vlastností. Práce na vývoji sítí ale vnesl do grafického prostředí UNIXu nový rozměr. GUI by mělo být koncipováno jako prostředí práce v libovolném uzlu sítě, nikoliv pouze na pracovní stanici, se kterou je GUI hardwarově

spjato. V grafickém podsystému UNIXu s názvem X Window System (zkráceně jen X) byl proto definován termín *displej X* (X Display), což je souhrn nezbytného hardwaru pro zajištění GUI (např. obrazovka, klávesnice, myš). Software pracující na displeji X je nazýván *server X* (X Server) a z pohledu sítě má svoji vlastní adresu IP (!). Uživatel sedící u grafické obrazovky tak za podpory serveru X pracuje v takovém uzlu sítě, který si vybere, ale který s ním pochopitelně umí spolupracovat. Server X komunikuje se zbylou částí X v uzlu formou *protokolu X* (X Protocol), který je (ale nemusí být) obvykle zabalen do paketů TCP/IP. Aplikace v oknech jsou startovány jako procesy ve vzdáleném systému. Jsou přitom označovány termínem *klient X* (X Client). Princip je rozšiřitelný na libovolný počet uzlů sítě. Uživatel jednoho displeje X může pracovat s několika klienty v různých uzlech současně a jejich výsledky kombinovat na pracovní desce svého displeje X. Uživatel tedy není omezen na grafickou podporu práce pouze na fyzicky přítomném stroji, ale využívá pro své potřeby celou síť v rámci jednoho pracovního prostředí. Dostáváme se tak k jednomu z aspektů, kdy uživatel přestává být uživatelem počítače a začíná být uživatelem počítačové sítě. Z pohledu opačného pak může být připojeno k operačnímu systému uzlu více displejů X z různých uzlů sítě. Klienty různých serverů X v různých místech sítě tak běží jako nezávislé procesy uzlu.

Na obr. 8.1 je stručně zobrazeno chování klientů X a serverů X v síti několika uzlů. Server X pracuje tam, kde je displej X. Na pracovní stanici SGI pracuje s klientem běžícím na tomtéž počítači a současně s klientem X, který je procesem v systému AIX počítače IBM. Současně využívá prostředí X také uživatel u PC, kde v prostředí MS Windows běží server X, který má právě rozpracovaný klient X také v operačním systému AIX a současně jiný klient X v IRIXu pracovní stanice SGI. Podobný případ je pro počítač Macintosh, kde jeho operační systém podporuje server X pro využití klientů v IRIXu a v HP-UX firmy Hewlett Packard. Server X je také provozován na hardwaru, který je označen jako specializovaný grafický *terminál X* (X terminal). Jedná se o bezpodmínečně nutnou část bezdiskové pracovní stanice, na které je server X schopen pracovat (kromě obrazovky a jejího řízení, klávesnice a myši musí mít dostatečně velkou operační paměť a síťové rozhraní), kterou výrobci systému X dodávají. Počítače s HP-UX a AIX na obrázku nedisponují hardwarem pro práci s X, a proto v jejich systému není žádný proces server X, ale oba systémy práci klientů X (zde z jiných uzlů sítě) plně podporují.

Grafické prostředí pro operační systém UNIX začal vyvíjet institut MIT (Massachusetts Institute of Technology) v r. 1984. X Window System, jak byl tehdy pojmenován, byl uživatelům k dispozici již následujícího roku v tzv. verzi 9, jejíž označení bylo zkráceně uváděno pouze jako X9. V průběhu následujících dvou let byla vyvinuta verze 10 a 11. Označení X11 pak přetrvávalo od r. 1987 dodnes. Značky produktu X dále totiž pokračovaly rozšířením verze 11 o tzv. vydání (Release). V posledním roce (píše se rok 1998) byla verze X11R5 vystřídána vydáním 6 s označením X11R6. Současná podoba systému oken X se totiž už principiálně nemění a nová vydání jsou rozšiřující především z pohledu způsobů implementace grafických režimů. MIT nezůstal již v 80. letech ve vývoji X osamocen. Na rozsáhlém projektu spolupracovala řada univerzit, renomovaných firem (DEC, Sun Microsystems, IMB, Hewlett Packard, AT&T aj.) a nezávislých odborníků. Spolupracující firmy se také stávají členy skupiny X Consortium, založené pro podporu a vývoj systému X. Dnes je X považován za průmyslový standard grafického prostředí pracovních stanic a je definován v SVID jako X11Window System. SVID jako součást téhož svazku definuje i nadstavbu X s označením NeWS (Network/extensible Window System),



Obr. 8.1 Práce uživatele v prostředí X

integruje distribuované grafické prostředí uživatele a jako mechanismus používá jazyk PostScript. Původním autorem NeWS je firma Sun Microsystems.

Firmy zúčastněné na vývoji X dnes jako součást UNIXu poskytují svou implementaci X, která je podle SVID obecně shodná, ale je lépe přizpůsobená jejich pracovním stanicím. Nejznámější implementace systému X jsou

- MIT Release 6 od Massachusetts Institute of Technology,
- OSF/Motif GUI od sdružení Open System Foundation,
- Sun Open Windows firmy Sun Microsystems,
- DEC Windows firmy Digital Equipment Corp.,
- AIX Windows firmy IBM,
- X Window System firmy Silicon Graphics.

Vyčerpávající literaturu ohledně systému X vydalo jako sadu knih vydavatelství O'Reilly & Associates, Inc. Je to často odkazovaný materiál, který vydavatelství vždy při novém vydání systému X aktualizuje. Jedná se o obecný úplný popis systému X, ale čtenář zde také nalezne výčet odchylek a zvláštností pro jednotlivé implementace, jako je rozložení adresářů nebo pojmenování knihoven či rozdílů v různých GUI. Devět svazků (Volume Zero - Volume Eight) je v seznamu části Literatura pod označením [XOREil].

K nejlepším českým publikacím o principech X patří bezesporu [Macu94].

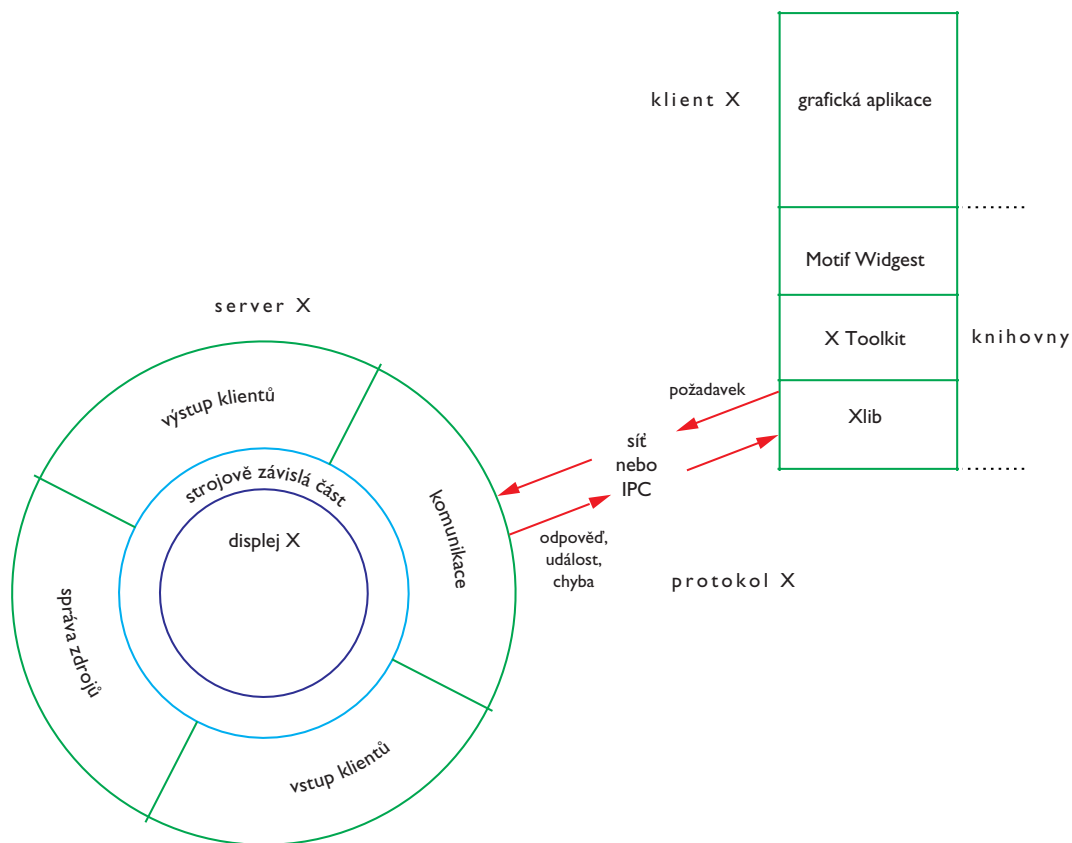
## **8.1 Základní schéma**

Server X je specializovaný program, který na displeji X uživateli poskytuje práci v systému X. Jde o základní software, jenž obaluje displej X, ovládá jeho komponenty a prostřednictvím sítové vrstvy realizuje v/v požadavky klientů ve vzdálených uzlech. Pokud je server X proces operačního systému, ve kterém běží současně jeho klienty X, komunikace mezi nimi a serverem X je namísto prostřednictvím sítě zajišťována prostředky IPC (viz kap. 4). X Window System, jak je již zřejmé, je striktně postaven na technologii klient - server, jejíž implementace v UNIXu je nám již známými prostředky sítě TCP/IP a IPC. Čtenář si pouze musí přivyknout zdánlivě opačnému použití termínů server a klient. Server X zde poskytuje displej X a klient X běží v uzlu s UNIXem. Server X lze vnímat při rozdělení na dvě základní části, a sice část, která přímo ovládá displej X, a část strojově nezávislá, která zajišťuje spolupráci naopak s obecným prostředím X. Princip strojově nezávislé části serveru X (viz obr. 8.2) byl koncipován tak, aby se obecné požadavky klientů X redukovaly na základní pokyny pro grafickou manipulaci a co nejvíce tak odlehčily síti a samotným klientům X. Znamená to, že server X musí zajišťovat zejména údržbu datových struktur popisujících objekty, se kterými klienty manipulují. Jsou to zejména okna, fonty, mapy barev atd. V terminologii X jim říkáme *zdroje X* (resources). Klient X pak protokolárně zadává manipulaci s odpovídajícím objektem a server X ji provádí. S výjimkou správy zdrojů je ovšem server X povinen také zajistit sítovou nebo místní komunikaci s klienty X, tj. akceptovat protokol X. Musí přitom zajišťovat obsluhu více klientů X současně. Také musí akceptovat požadavky uživatele (pohyby myši a stisky klávesnice) a předávat je opět prostřednictvím protokolu klientům X.

Klient X (viz obr. 8.2) je procesem v systému UNIX, který komunikuje s jemu určeným serverem X prostřednictvím protokolu X. Klient X může spolupracovat pouze s jedním serverem X, který je identifikován adresou displeje X. Programátor klientu X využívá knihovnu systému X s názvem **Xlib**.



Funkce této knihovny lze rozdělit do skupin, jako jsou např. komunikační funkce se serverem X, vstupní a výstupní pokyny atd. `Xlib` je definována v SVID. K příjemnější práci má programátor k dispozici knihovnu obecně nazvanou X Toolkit (konkrétně např. `Xt_Intrinsics`), která umožňuje snadnější práci při programování grafických objektů (jako je tlačítko nebo menu). Grafický styl určitým grafickým aplikacím pak dodává používání balíku funkcí grafické podpory na úrovni již definovaných tvarů objektů a stylů, kterým se říká widgets (na obr. 8.2 je uvedena dnes nejpoužívanější knihovna `Motif Widgets`, ale jsou stále podporovány např. `Athena Widgets Set` od MIT nebo `OLIT` prostředí `Open Look`) a které pak prakticky určují grafický styl GUI. Je ovšem zřejmé, že implementace `Xlib` je nejnižší vrstva pro komunikaci se serverem X a vzhledem k definovanému protokolu X, kterého využívá, je libovolný klient X nezávislý na typu a výrobci serveru X.



Obr. 8.2 Schéma práce X

Protokol X je definovaný způsob komunikace procesové vrstvy. Přesná definice protokolu X ještě neznámá, že musí být jednoznačně přenášen sítí typu TCP/IP. Firma DEC např. používá jako sítové rozhraní X také vlastní síť DECNET. Klient X a server X si vzájemně vyměňují zprávy čtyř různých typů. Jednak je to *požadavek* (request), který posílá klient X a kterým sděluje serveru X, co má vykreslit. Zprávu typu *odpověď* (reply) může klient X požadovat jako odezvu na zprávu typu požadavek, ale v mnoha případech toho není potřeba. Zprávu typu *událost* (event) oznamuje server X klientům změny na displeji X, např. když uživatel klikne myší do okna klientu X. Konečně zprávy typu *chyba* (error) jsou posílány opět serverem X jako události. Na straně klientu X pak záleží, zda chybu proces dokáže ošetřit. Chyby mohou být klientem X odstranitelné, nebo fatální. Zprávy typu událost jsou po příchodu ke klientu X frontovány a postupně zpracovávány. Některé požadavky vyžadují odpovědi, proto je nutno tuto frontu klientu X porušit a odpověď přijmout mimo pořadí. Které odpovědi mají právo frontu obejít, je definováno.

Zmíněná adresa displeje X, kterou klient X jednoznačně identifikuje spolupracující server X, je dána sítovou adresou IP, ale vzhledem k tomu, že lze použít adresu IP pro více serverů (na silných systémech bývá běžně instalováno několik grafických podsystémů), pokračuje adresa displeje X číselným označením pořadí odpovídajícího serveru X a dále ještě číselným označením obrazovky (displej může mít více obrazovek). Adresa displeje X je v uživatelské sezení nastavena v obsahu proměnné **DISPLAY** v shellu, jak uvidíme v následujícím čl. 8.2.

Zdroje X jsou objekty, se kterými klienty X manipulují, ale jsou ve správě serveru X. Klient X tak např. požadavkem „vykresli okno“ sděluje serveru X manipulaci se zdrojem okno. Každý vytvořený zdroj je identifikovatelný (má přiřazeno ID, které klient X používá). Zdroje lze sdílet navzájem mezi různými klienty X, takže lze programovat klient X, který jím definovanou množinu zdrojů poskytuje ostatním klientům. Takový klient bývá označen jako manažer oken (window manager) a vytváří tak grafickou podobu práce uživatele u displeje X. Spolupráce klientů při odkazech na zdroje, ale i další komunikace, je předmětem popisu ICCCM (Inter-Client Communication Conventions Manual), protokolu komunikace klientů prostředí X, kterou přijalo také SVID. Přestože mohou klienty X dorozumění s manažerem oken vynechat a používat své vlastní definované zdroje, vzhledem k tomu, že manažer oken je takřka vždy součástí uživatelské sezení, kterým se uživatel připojuje k uzlu sítě, vyplatí se klienty X programovat za využití ICCCM. Je to ostatně jednodušší, jak lze zjistit nahlédnutím do provozní dokumentace. Widgets ICCCM dodržují zcela jednoznačně. Zdroje X jsou rozděleny do několika skupin: *okna* (windows), *rastrové obrázky* (pixmap), *mapy barev* (colormaps), *kurzory* (cursors), *fonty* (fonts) a *grafické kontexty* (graphic contexts). Každý zdroj má své hodnoty, kterým říkáme *atributy* (angl. pouze resources), které klient nastavuje (např. velikost okna) v požadavku na vytvoření zdroje, ale nemůže je rozšiřovat nebo ignorovat. Naproti tomu *vlastnosti* zdrojů (properties), pokud mohou být se zdrojem spojeny, může klient X i rozšiřovat (ale server X nemusí akceptovat). Je to např. barva nebo popis rámečku okna. Právě vlastnosti jsou viditelné ostatními klienty X.

## 8.2 X z pohledu uživatele

Uživatel v operačním systému UNIX otevírá své sezení tak, že se prokáže svým jménem a heslem (přihlásí se, viz kap. 5). Na alfanumerických terminálech je aktivita operačního systému patrná z textu končícího na `login:`. Grafický terminál, který je součástí uzlu s UNIXem (např. pracovní stanice),

může být nastaven také v textovém režimu a situace je stejná, jenom obrazovka je větší a textový režim pokračuje i po přihlášení. Uživatel pak ale může použít příkaz

## § **xinit**

kterým startuje grafický systém X na grafickém terminálu, kde je přihlášen. **xinit** nejprve startuje server X uživatelova displeje jako jeden z procesů UNIXu (server X i klienty jsou v tomto případě procesy stejného uzlu). Program server X je uložen v souboru `/usr/bin/X11/X`. Pokud vyžaduje uživatel použití jiného serveru X pro displej X, může do souboru `.xserverrc` jeho domovského adresáře vložit cestu k souboru jiného serveru X, např. `/usr/bin/X11/Xmono` by mohla být varianta pro displej s monochromatickou obrazovkou. Uvedený soubor X je odkazem (jen další jméno) na soubor s typickým serverem X pro používaný displej X (u pracovních stanic firmy Silicon Graphics je to např. odkaz na soubor `Xsgi`).

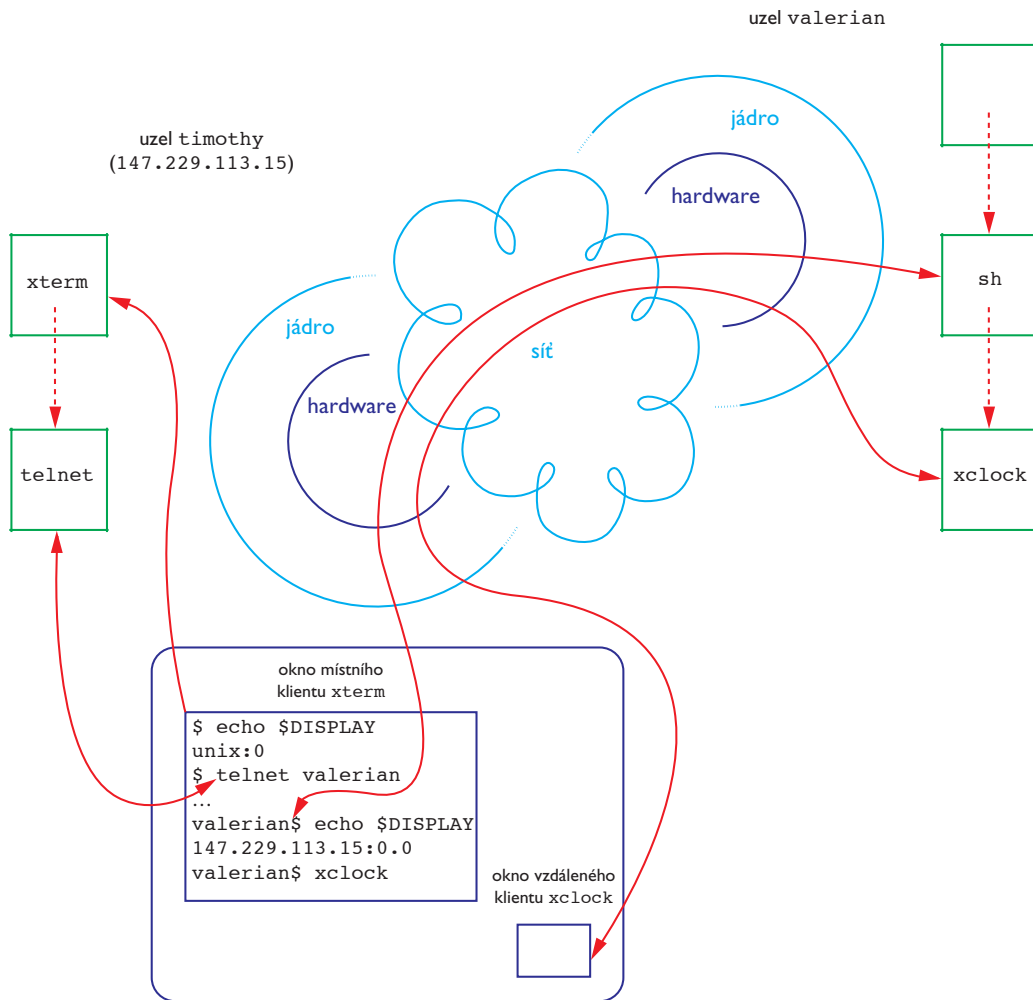
Po rozběhu serveru X se **xinit** zajímá o soubor `.xinitrc`, který obsahuje nastavení pracovní desky přihlášeného uživatele. Jedná se o scénář shellu obvykle pro start klientu X, který plní funkci manažera oken a případných dalších klientů. Pokud **xinit** nenalezne soubor `.xinitrc`, použije soubor `/usr/bin/X11/xinit/xinitrc`, který je obecným nastavením. Pokud ani ten není nalezen, startuje v sezení pouze jeden klient X, kterým je **xterm** jako emulace alfanumerického terminálu v okně. **xterm** přitom v okně startuje výchozí shell uživatele podle obsahu `/etc/passwd`. **xinit** pak nekončí, ale je zablokován do skončení práce svého dítěte, kterým je např. klient **xterm** nebo poslední klient souboru `.xinitrc` (nebo `xinitrc`), což bývá manažer oken. Znamená to, že v případě ukončení základního klientu manažera oken uživatel ukončí práci v grafickém prostředí X.

Namísto **xinit** se také používá program **startx** (nebo **startX**, **x11**, **x11start** z adresáře `/usr/bin` nebo adresáře klientů `/usr/bin/X11`), což je vždy scénář pro shell, ve kterém je nastaveno nezbytné prostředí pro práci X (cesta ke klientům a další proměnné shellu) a jako poslední příkazový řádek scénář obsahuje příkaz pro **xinit**. Jméno takového scénáře záleží na výrobci a je dostupné v provozní dokumentaci. X bývá také implementováno tak, že o obsah uvedených souborů `.xinitrc` nebo `xinitrc` se zajímá pouze scénář **startx**. **xinit** pouze spustí server X a jediný klient **xterm**.

Pro vstup uživatele do systému X se používá také klient **xdm** (**X Display Manager**). Příchozí uživatel objeví na terminálu nabídku k přihlášení prezentovanou již v grafické podobě. Po zadání svého jména a hesla **xdm** uživatele přihlásí do systému podobným postupem, jako to vykoná **getty** pro alfanumerické terminály. **xdm** je tedy procesem (klientem X) v UNIXu, který spolupracuje se serverem X zadaného displeje. O který server X se **xdm** zajímá, je věcí konfigurace procesu **xdm** v době jeho startu. Procesů **xdm** je přitom v systému přítomno tolik, kolik uzlů podporuje grafických terminálů. Pro uživatele je pak příjemné, že se po přihlášení objeví jeho pracovní deska s jeho oblíbeným manažerem oken a klienty ponechanými při posledním sezení. Ukončením prvního klientu svého sezení po **xdm** (manažer oken nebo **xterm**) dojde k nové inicializaci procesu **xdm**, který nabízí sezení dalšímu uživateli. Vzhledem k tomu, že **xdm** je již klientem X, musí být na displeji X již také rozběhnut server X a v případě terminálů X musí dojít k jejich síťové inicializaci. Tyto systémové podrobnosti včetně konfigurace klientu **xdm** si uvedeme v čl. 8.3. Uživatel ovšem podobně jako u procesu **xinit** i zde může start svého oblíbeného klientu manažera oken a dalších klientů zapsat do scénáře pro shell svého domovského adresáře, který systém X provede jako poslední krok jeho přihlášení. Jeho jméno je zde `.xsession`, a pokud v domovském adresáři uživatele není nalezen, použije se scénář v souboru

/usr/lib/X11/xdm/sys.xsession. Za to odpovídá obsah výchozího systémového scénáře /usr/lib/X11/xdm/Xsession, který může správce měnit. V něm může jména jak uživatelova, tak systémového scénáře zaměnit za jiná, ale nedělá se to. Sezení uživatele končí i zde ukončením scénáře Xsession. Ve scénáři naposledy spouštěným klientem X proto obvykle bývá manažer oken.

Každý klient X jako proces UNIXu dokáže získat obsah určité proměnné prostředí procesu podle jejího jména (viz kap. 2). Pro klient X je směrodatný obsah proměnné **DISPLAY**, protože určuje displej X, na



Obr. 8.3 Proměnná **DISPLAY** v místním a vzdáleném uzlu

kterém běží server X, s nímž má klient X komunikovat. Proměnná **DISPLAY** obsahuje proto adresu IP, server a obrazovku, např.

```
$ echo $DISPLAY
147.229.112.15:0.0
```

je typický případ. Adresa IP uzlu je následována znakem **:**, za kterým je uvedeno číselné pořadí serveru (zde 0, tedy první v pořadí), tečkou oddělené číslo pořadí obrazovky (rovněž 0, první obrazovka). Část adresy IP i označení serveru je povinné, označení obrazovky je nepovinné. Pokud proměnná **DISPLAY** není součástí prostředí procesu klientu X, odmítne tento pracovat. Jsou-li server X a klient X procesy téhož systému, je na místě adresy IP běžně používán řetězec **unix**; např. **unix:0** je běžný případ pro pracovní stanici grafické konzoly. Podle řetězce **unix** totiž klient pozná, že nepoužívá síťové prostředky, ale IPC.

Klient X je obvykle startován buďto z jiného klientu nebo je startován v době startu X (**xinit**). Klient X proto získává správný obsah exportované proměnné **DISPLAY**. Jakou hodnotu proměnné **DISPLAY** např. proces **init** v době startu dětí **xdm** nastaví, je předmětem tabulky v **/usr/lib/X11/xdm/Xservers**. Podrobněji tyto výchozí hodnoty proměnné **DISPLAY** probereme v čl. 8.3. Obsah proměnné **DISPLAY** se také přenáší v případě vzdáleného přihlášení do jiného uzlu. Jestliže v okně klientu **xterm** použijeme příkaz **telnet** (nebo **rlogin**), je hodnota proměnné **DISPLAY** vyvezena do vzdáleného procesu shellu a klienty X, které jsou v takto zpřístupněném vzdáleném uzlu spouštěny, budou její obsah používat pro adresaci displeje X, odkud bylo přihlášení provedeno, viz obr. 8.3 (**xclock** je klient, který v okně zobrazí probíhající čas v podobě digitálních nebo analogových hodin). Případná změna řetězce **unix** za adresu IP uzlu v okamžiku přihlášení ve vzdáleném uzlu je pochopitelně nutná.

Práce uživatele v X začíná vždy komunikací s některým klientem X, který má proměnnou **DISPLAY** nastavenou z informací v uzlu, kde běží jako proces. V nejjednodušším případě je to klient **xterm**, ale obvykle to bývá manažer oken. Vzdáleným přihlášením s přenosem identifikace displeje X v proměnné **DISPLAY** pak uživatel může využívat klienty vzdálených uzlů, jejichž v/v zajišťuje stále tentýž server X.

Manažer oken je zpříjemnění práce uživatele s jeho pracovní deskou. Tento klient X uživateli umožňuje vytvářet další místní klienty X v nových oknech a běžně slučovat obsahy oken různých běžících klientů X, měnit velikost a umístění oken, ikonizovat je atd. Obvykle se startuje jako poslední klient X úvodních scénářů **xdm** nebo **xinit**. Ukončením manažeru oken proto také pro uživatele končí práce v X. Pokud tomu tak není, může být spuštěn např. z klientu **xterm** z příkazového řádku, např.

```
$ mwm
```

je spuštění manažeru oken prostředí Motif (Motif Window Manager). Manažery oken může podobně jako jiné klienty uživatel používat podle toho, který je mu příjemnější. Běžně bývá také implementován manažer **twm** (tab window manager), manažer pro OPEN LOOK má příkaz **olwm**. Výrobci X také dodávají své vlastní manažery, např. **4Dwm** firmy Silicon Graphics nebo **rtl** firmy Siemens (podle laboratoří Siemens Research Technology Laboratories). Uvedený **mwm** je doporučený sdružením OSF a je jeden z nejpoužívanějších; mnohé manažery z něj principiálně vycházejí (např. **4Dwm**).

Jak již bylo uvedeno, každý klient je programován tak, že respektuje manažer oken. Jednotlivé manažery se respektují navzájem. Pokusíte-li se pod řízením některého manažeru oken spustit další,

takový nový klient X běh odmítne, protože rozpoznal přítomnost klientu jiného manažeru oken. Popis komunikace klientů mezi sebou na takové úrovni je v ICCCM.

Grafická aplikace běží v okně na obrazovce displeje X jako klient v místním nebo vzdáleném uzlu. Klienty lze programovat a vytvářet tak grafické aplikace. Přesto je řada potřebných aplikací pro pracovní desku a práci v X naprogramována. Označují se jako standardní klienty X a jejich základní seznam podle definice v X je uveden v příloze E.

Každý klient X používá při své práci určité zdroje X (resources). Jejich hodnoty jsou uživatelsky nastavitelné, takže můžeme např. stanovit, jakou barvu bude mít pozadí okna toho kterého klientu. Hodnotu každého zdroje je možné stanovit v příkazovém řádku startu klientu X (odpovídající volbou), což má nejvyšší prioritu. Nejčastěji jsou hodnoty zdrojů definovány v souboru se jménem podle obsahu proměnné `XENVIRONMENT`. Konfigurační soubor si klient X prohlédne při startu a doplní si hodnoty zdrojů, které nebyly nastaveny z příkazového řádku. Je-li spouštěný klient X procesem ve vzdáleném uzlu, akceptuje se pro doplnění dalších hodnot zdrojů místní soubor `.Xdefaults-host` (`host` je jméno vzdáleného uzlu) domovského adresáře uživatele. V dalším kroku je vyhledáván soubor `.Xdefaults` domovského adresáře uživatele. Pokud takový soubor klient X nenalezne, použije soubor `/usr/lib/X11/sys.Xdefaults`. Vždy jsou ale dále prohledávány soubory adresáře `/usr/lib/X11/app-defaults`, jejichž jména jsou dána třídou, do které klient X patří (jméno tohoto adresáře lze nastavit obsahem proměnné `XAPPLRESDIR` v shellu). Pokud i pak nejsou všechny atributy ohodnoceny, klient X si nastaví ostatní podle svých vlastních definic (klient X přitom může sám zakázat nastavovat hodnoty některých zdrojů).

Server X ovšem poskytuje tzv. databázi zdrojů, která je pro klient na konkrétním displeji X závazná před obsahem konfiguračního souboru `.Xdefaults` (zkuste si promyslet proč). Databázi zdrojů lze pomocí klientu **xrdb** (**X** Resource **D**atabase) naplnit obsahem některého konfiguračního souboru. Použitím **xrdb** dochází k výměně hodnot zdrojů databáze serveru X, takže lze v sezení přepínat mezi jednotlivými grafickými styly. Databáze zdrojů je navíc součástí serveru X, práce klientů je tak výkonnější. Databázi zdrojů aktualizujeme klientem **xrdb**. V argumentu přitom zadáváme jméno konfiguračního souboru (`.Xdefaults` nebo kterýkoliv jiný). Soubor s nastavením zdrojů můžeme k databázi zdrojů připojit (volba **-merge**), obsah databáze jím přepsat (**-load**) nebo databázi odstranit (**-remove**), tj. zrušíme její využívání. Dotázat se na aktuální stav zdrojů databáze můžeme volbou **-query**. V knihovně `Xlib` je databáze zdrojů dostupná programátorovi klientů prostřednictvím struktury typu vlastnost (property) se jménem `RESOURCE_MANAGER`.

Hodnota zdroje je v konfiguračním souboru definována logickou (např. rámeček ano nebo ne), číselnou (velikost okna) nebo textovou (barva pozadí, barva písma) hodnotou. Každou hodnotu stanovujeme na samostatném řádku. Hodnota se např. může vztahovat na konkrétní klient (každý klient **xterm** bude mít barvu pozadí okna červenou)

```
xterm*background: red
```

nebo obecně stanovuje hodnotu zdroje pro každý klient (barva pozadí okna každého klientu X, který využívá zdroj pozadí okna, bude červená)

```
*background: red
```

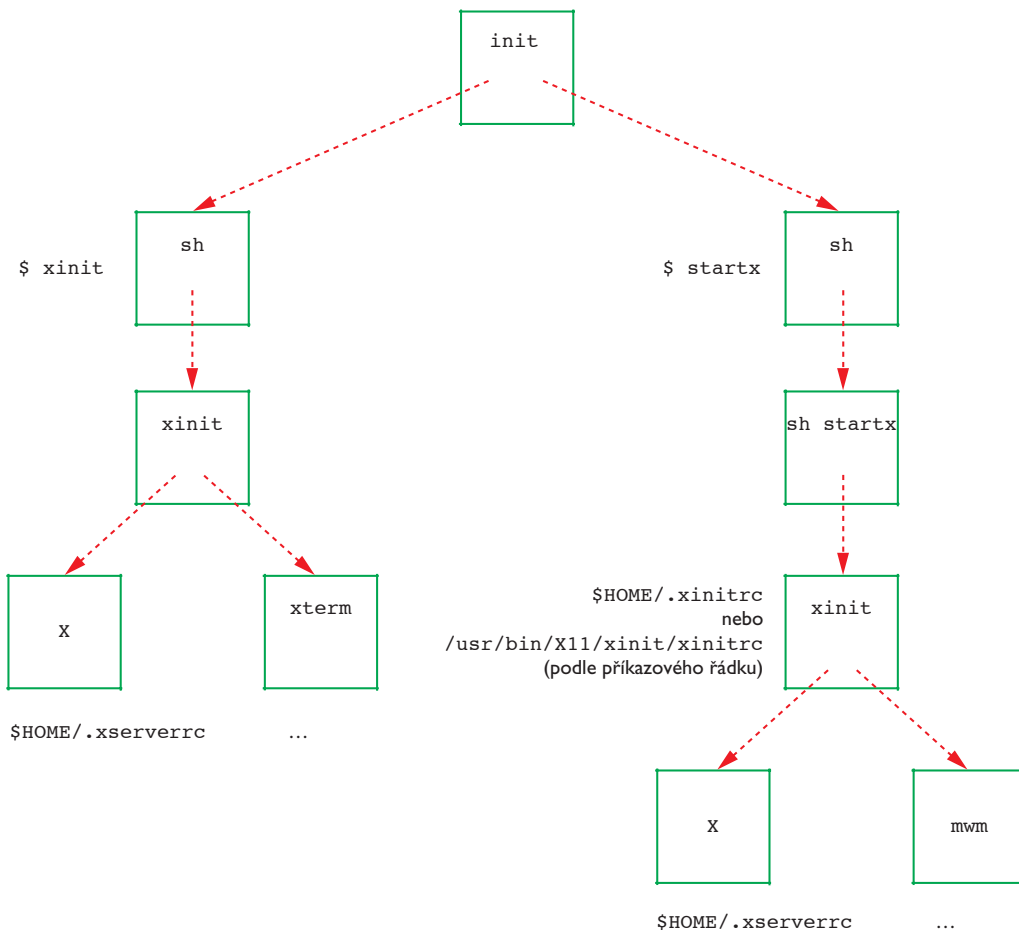
Použitý oddělovač `*` určuje tzv. *volnou vazbu* (loose binding). Jedná se o nastavení zdroje všech objektů (klientu nebo klientů), které jsou označeny uvedeným jménem. V X je totiž zavedena tzv. hierarchická

struktura zdrojů, která vznikla při programování X Toolkit. Jde o zavedení jmen zdrojů podle jména objektu děleného na další podobjekty. Podobjekty se oddělují znakem `.` a jeho použití ve stanovení hodnoty zdroje v konfiguračním souboru je označováno termínem *těsná vazba* (tight binding), např.

```
xprog.helpButton.background: red
```

nastaví červené pozadí pouze pro tlačítka nápovědy klientu **xprog**, zatímco

```
xprog*background: red
```



Obr. 8.4 Procesy a scénáře při použití **xinit** nebo **startx**

nastaví barvu pozadí červenou pro všechny zdroje podle jména **background** u klientu **xprog**. Nejvíce se ovšem používá volná vazba, protože uživatel nemusí znát strukturu zdrojů klientu X a šmahem tak vyjádří svoji barevnou orientaci.

Z objektového programování si X přineslo i náležitost zdrojů (tj. objektů) do určité *třídy* (class). Konkrétní realizaci třídy říkáme *instance*. Třídy jsou označovány texty začínajícími velkým písmenem a závisí na použité objektové knihovně (X Toolkit). Termín instance se používá také pro konkrétní označení zdroje, např.

```
xprog*Buttons*background: red
xprog*help*background: blue
```

Tlačítka (třída Buttons) klientu **xprog** budou mít červené pozadí, pouze tlačítko nápovědy (za předpokladu, že **help** patří do třídy Buttons) bude modré. Jména tříd a zdrojů jsou předmětem provozní dokumentace každého klientu X.

Nastavení hodnoty zdroje z příkazového řádku klientu pro jedno spuštění se dosáhne pomocí obecné volby **-xrm**, např.

```
$ xterm -xrm "xterm*background: red"
```

nastaví programu emulace terminálu červené pozadí.

V prostředí uživatelského sezení je obvyklé startovat místní klienty X z některého menu manažeru oken. Uživatel ovšem může využívat příkazového řádku každého emulátoru terminálu **xterm** pro start všech klientů (místních i vzdálených). Při vývoji klientů X se vytvořila konvence formátu jejich příkazového řádku. Obecně tak může uživatel v příkazovém řádku zadávat adresu displeje X (volbou **-display**), stanovit velikost a umístění hlavního okna po startu (**-geometry**), používaný font (**-font**), barvu pozadí (**-background**), písma (**-foreground**) a barvu rámečku (**-bordercolor**) nebo lze klientu X ihned po spuštění ikonizovat (**-iconic**), změnit nápis na rámečku (**-title**). Volbou příkazového řádku každého klientu **-xrm** můžeme pro jeho běh nastavit jinou hodnotu některého zdroje.

## 8.3 X z pohledu operačního systému

V úvodu čl. 8.2 jsme uvedli dva možné přístupy uživatele k systému X. Jednak to bylo prostřednictvím scénáře **xinit** a jednak pomocí klientu **xdm**. V případě **xinit** jsou skutečně postupně provedeny všechny akce pro start grafického rozhraní (start serveru X, spuštění manažeru oken a základních klientů). Vstup do systému pomocí **xdm** je již za podpory běžícího serveru X na displeji X. **xdm** je klient X, který uživatele do systému teprve přihlásí a provede postupně svou proměnu na několik procesů, z nichž poslední je manažer oken.

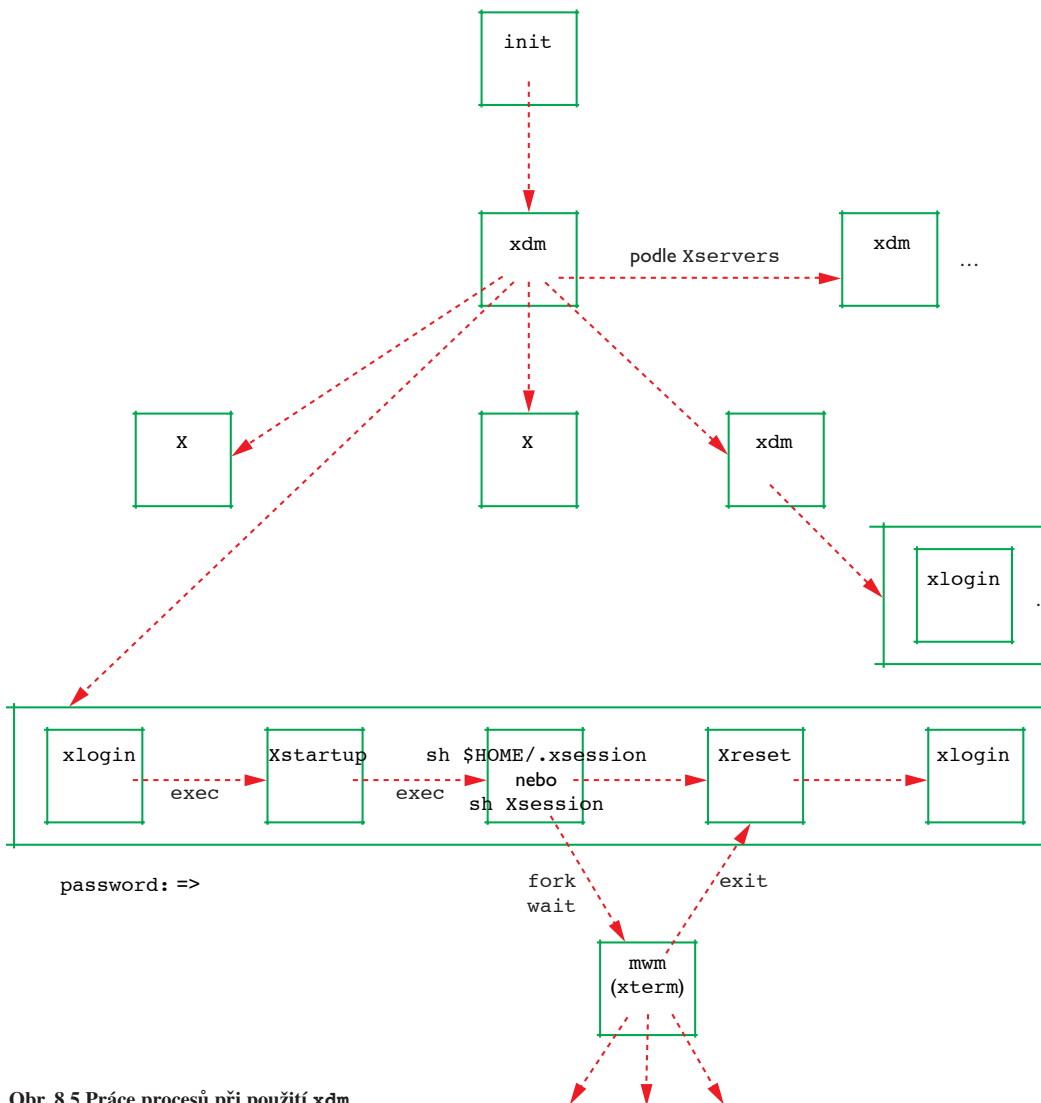
Schéma použití **xinit** nebo **startx** v rámci sezení uživatele ukazuje obr. 8.4.

Proces shell **sh** je v tomto případě výchozí, **xinit** nebo **startx** jsou jeho děti a shell čeká na jejich dokončení (v případě **startx** zůstávají v sezení čekající procesy shellu dokonce dva). Samotný **xinit** použije pro displej X, definovaný buďto z exportované proměnné **DISPLAY** v prostředí shellu nebo z parametru příkazového řádku, server X ze souboru `/usr/bin/X11/X` nebo ze souboru podle obsahu `.xserverrc` v uživatelské domovské adresáři. Po jeho startu pak dále vytvoří první klient X tohoto serveru, kterým je buďto **xterm** nebo manažer oken (např. **mwm**). Všimněte si, že procesy serveru X a klientu X jsou závislé na svém rodiči **xinit**, ale navzájem jsou rovnoprávné děti. **xinit**



přítom čeká na ukončení dítěte prvního klientu X. Pokud skončí, ukončí i proces server X, který obnoví alfanumerický režim grafického terminálu a uživatel pokračuje v sezení dalším příkazovým řádkem shellu.

V příkazovém řádku **xinit** lze určit všechny důležité výpočetní zdroje pro rozběh serveru X a prvního klientu X. Je to hodnota displeje X, parametry prvního klientu, soubor s programem



Obr. 8.5 Práce procesů při použití xdm

serveru X a parametry jeho voleb. Nejlépe je přitom vycházet z provozní dokumentace. Scénář **startx** je po instalaci systému X ve výchozí podobě a správce systému jeho obsah takřka vždy mění podle konfigurace výpočetního systému. Scénáře **.xinitrc** a **xinitrc** pak obsahují start prvních klientů prostředí uživatele a uživatel v **.xinitrc** má nastavenou svoji soukromou pracovní desku. V případě, že začíná, nabízí mu systém pro něj neměnný obsah **xinitrc**. Většina dnešních implementací udržuje obsah **.xinitrc** podle současného stavu pracovní desky uživatele.

Přístup uživatele ke své pracovní ploše prostřednictvím klientu **xdm** je dnes používán více. Grafická aplikace **xdm** podobně jako **xinit** inicializuje prostředí X, ale na libovolně definovaném počtu displejů X. Po úspěšném startu všech potřebných serverů X pak vytvoří tolik kopií samy sebe (voláním jádra **fork**), až počet procesů **xdm** dosáhne počtu displejů X. Každý **xdm** pracuje pro komunikaci s jedním serverem. **xdm** přitom řídí přihlašování a odhlašování uživatelů na displeji X. Vzhledem k tomu, že servery X mohou být produktem různých výrobců, pro sjednocení byl v rámci X11R4 definován komunikační protokol mezi serverem a **xdm** s označením XDMCP (X Display Manager Control Protocol).

Seznam všech displejů X pro **xdm** je definovaný v souboru **/usr/lib/X11/xdm/Xservers**. Na každém řádku je definován server pro odpovídající displej X. Tabulka má 4 položky oddělené mezerami nebo tabulátory. Nejprve je definován displej X, pak je řetězcem **local** nebo **foreign** definováno, zda se jedná o místní displej nebo zda je připojen sítí. Další položka je cesta k souboru se serverem X a zbytek řádku je komentář, např.

```
:0.0          local          /usr/bin/X11/X      místní server X
trmX:0.0      foreign      terminál X připojený sítí
```

je definice jednak místního serveru grafické konzoly a jednak grafického terminálu X připojeného sítí s označením **trmX**. U řádků označených jako **foreign** není třetí položka jménem programu serveru X, ale začíná zde komentář. Server X je totiž zaveden jinými prostředky (např. pomocí **ttftp**, viz další výklad). Vzdálené servery X jsou v **Xservers** uvedeny pouze v případě, že mezi uzlem a terminálem X nelze zajistit protokol XDMCP (viz další text).

Výchozí proces **xdm** je startován při zavádění operačního systému (je uveden v tabulce **/etc/inittab** nebo v některém ze scénářů **rc**, viz kap. 10). Další schéma práce tohoto procesu a jeho dětí ukazuje obr. 8.5.

Práce každého dítěte **xdm** je podobná jako u procesu **getty** (viz čl. 2.2). Proces **xdm** se svým serverem X komunikuje dále jako klient s oknem pro přihlášení. V literatuře je tato fáze **xdm** označována jako **xlogin**. Přestože klient takového jména neexistuje (ale záleží na implementaci), jde o fázi procesu **xdm** (instance pro zdroje X). V případě, že se uživatel prokáže správným heslem, **xdm** interpretuje nejprve scénář **Xstartup**, pak systémový scénář prvních klientů X **Xsession**, a pokud je v domovském adresáři uživatele přítomen soubor **.xsession**, je rovněž interpretován. Jak bylo uvedeno v předchozím článku, úvodní scénáře uživatele jsou věci editace odpovídajících souborů v **/usr/lib/X11/xdm** a také implementace. Posledním příkazovým řádkem úvodního scénáře je první klient X, buďto manažer oken (např. **wm**) nebo klient emulace terminálu **xterm**. Procesy klientů X jsou dále děti prvního klientu X, po jehož skončení čekající **xdm** provede výchozí nastavení pomocí scénáře **Xreset**, přejde do fáze **xlogin** a umožní tak přihlásit se dalšímu uživateli.

Základní konfigurační soubor **xdm** je `/usr/lib/X11/xdm/xdm-config`, který obsahuje seznam jmen dalších konfiguračních souborů, a to ve tvaru zdrojů X. Zdroje X, které jsou již při přihlašování uživatele součástí serveru X (ty, které modifikujeme pomocí **xrdb**), jsou tak např. obsahem souboru **Xresources**.

Pro práci v grafickém podsystému X se používají také specializované počítače, kterým říkáme terminál X (X Terminal). Jde o bezdiskovou stanici se silnou podporou grafických operací (velká obrazovka, klávesnice, ukazovací zařízení). Její součástí je hardware síťového rozhraní pro zajištění sítě TCP/IP. Takový hardware výrobci vyvinuli speciálně pro práci v X Window System a splňuje tedy především požadavky provozu X. Připojení je prostřednictvím sítě. Server X je zde vlastně operačním systémem takové stanice. Pro jeho start je po zapnutí terminálu X používáno několik způsobů. Jednak je to možnost využít lokálního serveru X z ROM paměti (EPROM) dodaných výrobcem terminálu. Také je používán server X ze vzdáleného uzlu, který je nejprve přenesen sítí a poté startován na terminálu X. Často jsou tyto varianty dostupné obě, což je výhodné z pohledu nových verzí, ale původní server X v ROM bývá spolehlivější. Přenos serveru X probíhá sítí přenosovým protokolem **ftp** nebo **tftp**, kterým musí terminál X disponovat (musí být rovněž v ROM). Lokálně bývá také k dispozici protokol **telnet**, prostřednictvím kterého získáme přístup k uzlu s UNIXem v podobě alfanumerického terminálu. Tímto způsobem se také startuje první klient X, např. **xterm**. Metoda startu prostředí X příkazem **xinit** zde není možná, protože **xinit** je specializován na práci v X s vynecháním sítě. Prostředí práce uživatele pro start serveru X je zde závislé na výrobci, ale obvykle se dá konfigurovat vybraná možnost, která uživateli zpřístupní první klient vzdáleného uzlu automatizovaně. Úzké místo nastává v okamžiku přidělování adresy IP terminálu X. Jak ale bylo řečeno, terminál X je vlastně bezdisková stanice. Dynamicky lze její adresu IP konfigurovat v souboru `/etc/ethers` a software ROM terminálu X využije modulu RARP vrstvy IP sítě pro získání odpovídající adresy IP. Struktura souboru `/etc/ethers` je bijekce adresy Ethernetu a adresy IP (případně jména podle seznamu uzlů, viz kap. 7). Pro získání síťových informací používá terminál X také tzv. protokol BOOTP (viz RFC951 a RFC1048). Zatímco pomocí RARP lze obdržet pouze adresu IP, BOOTP nabídne více informací, jako je např. síťová maska, adresa IP serveru DNS atp. Konfigurační soubor je `/etc/bootptab`, který má formát podobný souboru `/etc/termcap` systémů BSD.

Pro přístup uživatele je ovšem nejlepší i zde využívat prostředí klientu **xdm**. Nejvhodnější je využívat XDMCP (tj. uzel s UNIXem i server X jej ovládají), ale není to podmínkou. **xdm** pro server X přitom nemusí pracovat přímo v osloveném uzlu. V uzlu, se kterým terminál X spolupracuje ve výchozím nastavení, totiž lze definovat **xdm** přímo (direct), nepřímo (indirect) nebo pomocí zveřejnění (broadcast). U přímého nastavení terminálu X pracuje **xdm** ve výchozím uzlu, který je jednoznačně dán výchozí adresou IP (nebo jménem uzlu) z úvodní inicializace terminálu X. Výchozí uzel tak po přijetí požadavku XDMCP startuje **xdm**. Je-li terminál X nastaven jako nepřímý, jeho požadavek XDMCP výchozí uzel přesměruje na jiný uzel nebo na skupinu uzlů, kde má **xdm** pracovat. Zveřejnění je pak případ obelstání celé sítě, tj. uzlem přijatý požadavek XDMCP terminálu X je rozeslán do všech definovaných uzlů podsítě. Obvykle první uzel podsítě, který na požadavek reaguje, je ten, na kterém poběží **xdm**. Na rozdíl od přímého nebo nepřímého nastavení je zveřejnění omezeno na masku podsítě, a proto je nelze konfigurovat na uzly za směrovačem nebo bránou. Způsob nastavení příjmu požadavků XDMCP terminálů X je registrován v souboru `/usr/lib/X11/xaccess`. Přímá konfigurace nebo zveřejnění jsou v tomto souboru uvedeny jménem uzlu nebo podsítě, např.

```
*.vic.cz
!timothy.vic.cz
!gagarin.vic.cz
```

znamená, že je zpráva XDMCP očekávána ode všech uzlů podsítě **vic.cz**. Nepřijímají se pouze požadavky XDMCP serverů X (tj. zde terminálů X) **timothy** a **gagarin** (ale maskovat část podsítě můžeme také znakem **?**, který nahrazuje libovolný znak textu – podrobnosti viz provozní dokumentace). Nepřímá konfigurace (předání požadavku XDMCP jinému uzlu nebo několika jiným uzlům) je v souboru **Xaccess** uvedeno převodním vztahem na jiný uzel, např.

```
trm*.vic.cz valerian.vic.cz
```

definuje, že po příchodu požadavku XDMCP od libovolného uzlu sítě **vic.cz** začínajícího na **trm** bude předán uzlu **valerian**, kde se rozběhne klient **xdm**. Chceme-li definovat skupinu uzlů, zadáváme např.

```
trm*.vic.cz CHOOSER valerian.vic.cz timothy.vic.cz
```

Vzhledem k tomu, že uzel s takovým nastavením předává požadavek XDMCP jednoznačně, rozběhne v tomto případě klient X nazývaný **chooser**, který v okně uživatelského displeje X zobrazí za slovem **CHOOSER** nabídku definovaných uzlů. Uživatel na terminálu X si vybere uzel, a v něm se rozbíhá jeho **xdm** (zdroje klientů jsou definovány v **Xresources**).

Uvedená metoda práce terminálů X je výchozí pro funkce emulačního softwaru v grafickém prostředí počítačů, které nedisponují přímo prostředím X (počítače Macintosh, PC v prostředí MS Windows, MS Windows-NT atd.). Součástí je software serveru X, který neběží na holém stroji, ale je aplikací místního operačního systému. Start serveru X je vždy spojen s řadou místních konfigurací, které vycházejí z uvedeného principu. Po zadání adresy IP uzlu pro start prvního klientu proběhne (obvykle formou XDMCP) úvodní komunikace. Je možné nastavit start **xdm**, ale většinou se využívá místní aplikace, která nahrazuje klient manažeru oken, který s uzly klientů X komunikuje.

Implementace grafického systému X jednotlivých výrobců vychází z uvedených předpokladů. Přesto tato systémová podpora může vykazovat mírné odchylky, protože není standardizována určitým dokumentem. Definice v **SVID** se totiž zatím vztahují pouze na funkce knihovny **Xlib**. Na závěr si ale uvedeme základní, obecně shodně používané adresáře, ve kterých je X instalován:

<b>/usr/bin/X11</b>	programy X, jako jsou klienty X, démoni, servery X,
<b>/usr/lib/X11</b>	software využívaný servery X, tj. fonty, databáze barev, konfigurační soubory,
<b>/usr/lib</b>	knihovny programování v X,
<b>/usr/include/X11</b>	hlavičkové soubory X a bitové mapy,
<b>/usr/man</b>	provozní dokumentace,
<b>\$HOME</b>	domovský adresář uživatele, který obsahuje soubory s uživatelem definovanými zdroji X a startovacím scénářem.

<sup>1</sup> Např.-li text *kočka*, představíte si určitě každý jiné zvíře (barva, rasa, stáří atd.), ale kočka může být přeneseně také žena nebo malý jeřáb a k jejímu podrobnějšímu popisu potřebuji další text a více času. Pokud vám ji však ukáží na obrázku nebo lépe přehrají na videu se zvukem, nemůže být pochyb, že mluvíme o tomtéž zvířeti, a navíc seznámení zabere mnohem méně času.

## 9 BEZPEČNOST

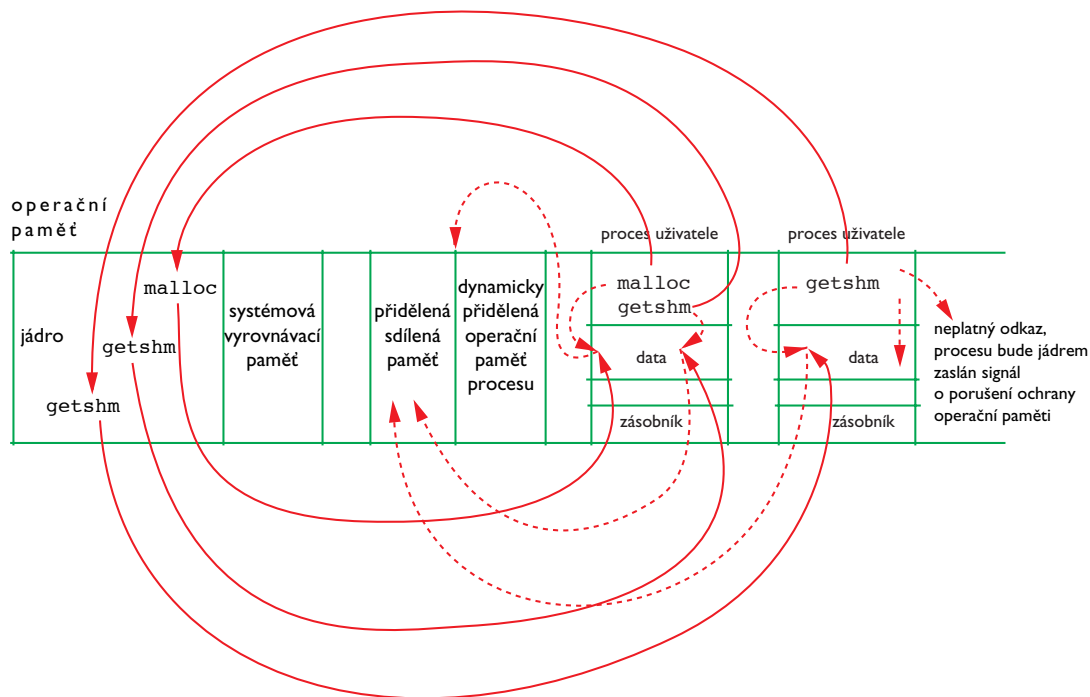
Dříve nebo později uživatel, jenž využívá výpočetní systém jako jeden z prostředků své pracovní činnosti, začne požadovat bezpečnost svých dat. Je pro něj důležité najít všechna svá data tak, jak je v minulém sezení zanechal. Mnohdy je pro něj důležité i jejich soukromí, tj. jistota, že jeho data jsou nejenom nezměněna, ale také jiným uživatelem nepřečtena (tj. kopírována). Tento zdánlivě samozřejmý předpoklad však nebyl v dosavadní historii výpočetní techniky nikdy dokonale zajištěn. Správce výpočetního systému je totiž vystaven nikoliv pouze nedokonalosti ochrany operačního systému, ale také neznalosti uživatelů a možnosti fyzického zneužití stroje. Uživatelé si musí být např. vědomi povinnosti ukončit své sezení (odhlásit se) po skončení své práce. Počítač s daty by měl být v zamykatelné místnosti nebo by měl být zbaven možnosti startovat na holém stroji jiné programy, než je používaný operační systém (např. fyzickým odpojením zařízení pro zavádění takových programů, jako je disketa, CD, páska atd.). Nebezpečí zničení nebo zneužití dat uživatele vystoupilo do popředí jako problém při sdílení dat na tomtéž stroji různými uživateli s interaktivním přístupem. Vznik rozsáhlých počítačových sítí a Internetu pak problém bezpečnosti dat odkryl jako jeden z nejpalčivějších. Např. finanční operace bankéřů nelze provádět přes Internet, pokud nebudou jednoznačně spojeny s konkrétní osobou. Proto další rozvoj využití výpočetní techniky není možný bez dokonalého zabezpečení ochrany dat. Je ale dokonalá ochrana vůbec možná? Teoreticky jistě ne, protože algoritmy praktického světa jsou popsatelné mechanismy až na úrovni Turingových nebo Postových strojů a tedy, jak víme ze studia matematické informatiky (viz např. [Mann74]), nedokazatelné. Žádný operační systém není vzhledem ke své složitosti vyplývající z praktických požadavků dokazatelný, tedy nelze matematicky dokázat jeho bezchybnost. O matematicky dokazatelné reakci uživatele nebo fyzického rozmístění výpočetní techniky a organizace přístupu k ní už ani nemluvě. S vědomím těchto nedostatků při modelování skutečného světa výpočetní technikou je nutné přistupovat k řešení bezpečnosti dat. V podobě dnešních počítačů bezpečnost dat nebude nikdy dokonalá, v mnoha případech však může být pro potřeby dostačující.

Požadavky na bezpečnost dat, kterou zajišťuje výpočetní systém, byly přehledným způsobem postulovány již v r. 1983. Jde o dokument Ministerstva obrany vlády Spojených států (Department of Defense, DoD) s označením TCSEC (Trusted Computer System Evaluation Criteria, Kritéria hodnocení důvěry výpočetního systému), viz [TCSEC83], známý také pod označením Oranžová kniha (Orange book). TCSEC zavedl různé skupiny či stupně bezpečnosti. Každá skupina určuje míru požadované bezpečnosti dat. Skupiny byly označeny písmeny D, C, B a A. Za písmenem obvykle následuje cifra, která rozděluje skupinu na úrovně (např. C2 nebo B1). Nejméně požadované zabezpečení je skupina D, které např. odpovídá způsob ochrany dat v UNIX version 7. Úrovně C, B a A postupně zvyšují nároky, přitom vyšší úroveň (směrem k písmenu A a nižšímu číslu) vždy zahrnuje požadavky úrovně nižší. Reálně jsou implementovány úrovně C2 a B1, skupina A je na rozhraní reálných možností současné výpočetní techniky. Zadavatel výpočetního systému pak může stanovit, nakolik je nutné jeho data ochránit, což určí odkazem na odpovídající úroveň. Každý současný výpočetní systém podporuje uvedený systém kategorií bezpečnosti dat, přestože se běžně používá úroveň C2 a B1. SVID respektuje TCSEC a definuje chování UNIXu v úrovních C2, B1 a B2. POSIX TCSEC necituje, ale z jeho způsobu ochrany dat vyplývá úroveň C1. POSIX dále o bezpečnosti uvádí pouze drobné úvahy v kontextu ochrany dat a definici bezpečnosti v operačních systémech teprve připravuje. Jak uvidíme u SVID, bezpečnost operačního systému úzce souvisí s prací správce systému a se sítěmi. Obě oblasti však

POSIX zatím nedefinuje. V dalším textu kapitoly si uvedeme obsah úrovní podle TCSEC a také jejich konkrétní podobu v SVID.<sup>1</sup>

V kapitole ovšem nejprve ukážeme úzká místa bezpečnosti dat v UNIXu, která nesmí správce systému zanedbávat, zvláště je-li uzel součástí sítě Internet. Ukážeme také, jak lze v těchto případech zabránit zneužití dat i v bezpečnostní úrovni D a jak je bezpečnost posílena instalací vyšší bezpečnostní úrovně. Text kapitoly bude obecný a nemůže zahrnout nedostatky vzniklé chybami v jednotlivých verzích UNIXu, kdy některá ochrana např. není omylem zajištěna. Samozřejmě nezůstane pouze u místního operačního systému, ale uvedeme také bezpečnostní potřeby sítí a jejich praktická zajištění, jako je podsystém Kerberos nebo ochranné zdi (firewalls).

UNIX není koncepčně systém, který by si zasloužil tak špatný věhlas, který z pohledu bezpečnosti dnes má. Jak uvidíme v následujícím čl. 9.1, pokud je UNIX programován pečlivě, nemusí k úmyslným ani neúmyslným poškozením dat docházet. Velká část systémových služeb nebo i součástí jádra UNIXu byla ale programována v akademickém prostředí, kde bezpečnost dat není vyžadována tak přísně jako v průmyslu. Je proto věcí každého výrobce průmyslové verze UNIXu, aby převzatý zdrojový kód analyzoval a ladil také z pohledu bezpečnosti. Nedostatky, které správce systému v konkrétní verzi odhalí, by



Obr. 9.1 Adresace operační paměti procesem uživatele

měl spíše než zveřejňovat v časopisech oznamovat výrobci, který se postará o odstranění. Také může kontaktovat některou z institucí, která se bezpečností výpočetních systémů seriózně zabývá, jako je např. DARPA CERT (Computer Emergency Response Team, Skupina pro řešení neočekávaných stavů počítačů, [cert@sei.cmu.edu](mailto:cert@sei.cmu.edu)), která úzce spolupracuje s výrobcí, nebo DoE's CIAC (Department of Energy's Computer Incident Advisory Capability, Poradenská služba pro výpadky počítačů Ministerstva energie, [ciac@tiger.llnl.gov](mailto:ciac@tiger.llnl.gov)), která je i telefonicky dostupná 24 hodin denně. Obě organizace působí ve Spojených státech.

Povinností správce systému je neustále sledovat chování operačního systému, používat systém sledování vznikajících sezení uživatelů (auditing) nebo vstupů do operačního systému prostřednictvím sítě. Rovněž se vyplatí procházet statistiku chování jádra a jeho parametrů. Jedná se o využívání prostředku **sar** a dalších podpůrných programů, které probereme v kap. 10. Tyto a další z bezpečnosti vycházející aktivity budeme v kapitole probírat, poukazovat na nedostatky, které se v UNIXu mohou vyskytovat, a doporučovat vhodná řešení.

## 9.1 Bezpečnost v úrovni C1

Jak je patrné z předchozích kapitol, UNIX zajišťuje vstup uživatele do systému podle jeho jmenné identifikace, která je podle tabulky `/etc/passwd` převáděna na identifikaci číselnou (UID). Po přihlášení je uživatelova pracovní činnost v systému interpretována skupinou procesů, které jádro eviduje v jeho vlastnictví právě podle UID. Každý takový proces je prováděn za dozoru jádra. Odkazy prováděných instrukcí procesu musí být v rámci jeho adresovatelného prostoru, tj. uvnitř jeho datového nebo textového segmentu, případně zásobníku. To je omezující z pohledu přístupu k dalším částem operační paměti. Operační paměť je přidělována pouze jádrem a o zvětšení datové oblasti o další část operační paměti musí proces požádat např. voláním jádra `malloc`. Odkazy mimo oblast procesu v operační paměti jsou vždy prováděny jádrem. Schéma je uvedeno na obr. 9.1.

Procesu je tedy dovolena pouze manipulace s daty v jemu přidělené oblasti. Tak jsou procesy navzájem mezi sebou chráněny před náhodným nebo záměrným poškozením. Tento mechanismus je přítom u procesů zachován i v případě, že jejich vlastníkem je tentýž uživatel. I takové procesy si navzájem nemohou zpřístupňovat data svým přímým přístupem do datových segmentů. Pro komunikaci procesů byly proto zavedeny prostředky datové komunikace, jako je např. sdílená paměť (viz kap. 4). Každý neplatný odkaz procesu jádro zpracuje tak, že přístup zamítne a posílá procesu signál o porušení přístupu k operační paměti (např. `SIGBUS` nebo `SIGSEGV`), který znamená ukončení procesu, pokud jej proces neošetří maskovací funkcí.

I ostatní výpočetní zdroje jsou procesu přístupné, pouze pokud je zdroj označen pro proces (tj. uživatele) jako přístupný. Označování výpočetních zdrojů je typické trojicí přístupových práv, jak je tomu u souborů. Informace o přístupu k souboru uložená v i-uzlu obsahuje majitele (vlastníka) souboru, tj. jeho UID, skupinu vlastníka (g, group) a trojice přístupu pro čtení, zápis a provádění (rwx) pro vlastníka (u, user), skupinu vlastníka (g, group) a ostatní uživatele (o, others). Proces, který usiluje o přístup k souboru, k tomu používá volání jádra `open`. Podle uživatele, kterému proces patří, jádro určí, zda požadovaný přístup v argumentech `open` k danému souboru je možný. UID a GID z i-uzlu porovnává s UID a GID procesu a pak se zaměří na zkoumání odpovídající trojice přístupových práv. V případě shody UID i GID se jedná o vlastníka, shody pouze GID o skupinu vlastníka a není-li shodné ani UID



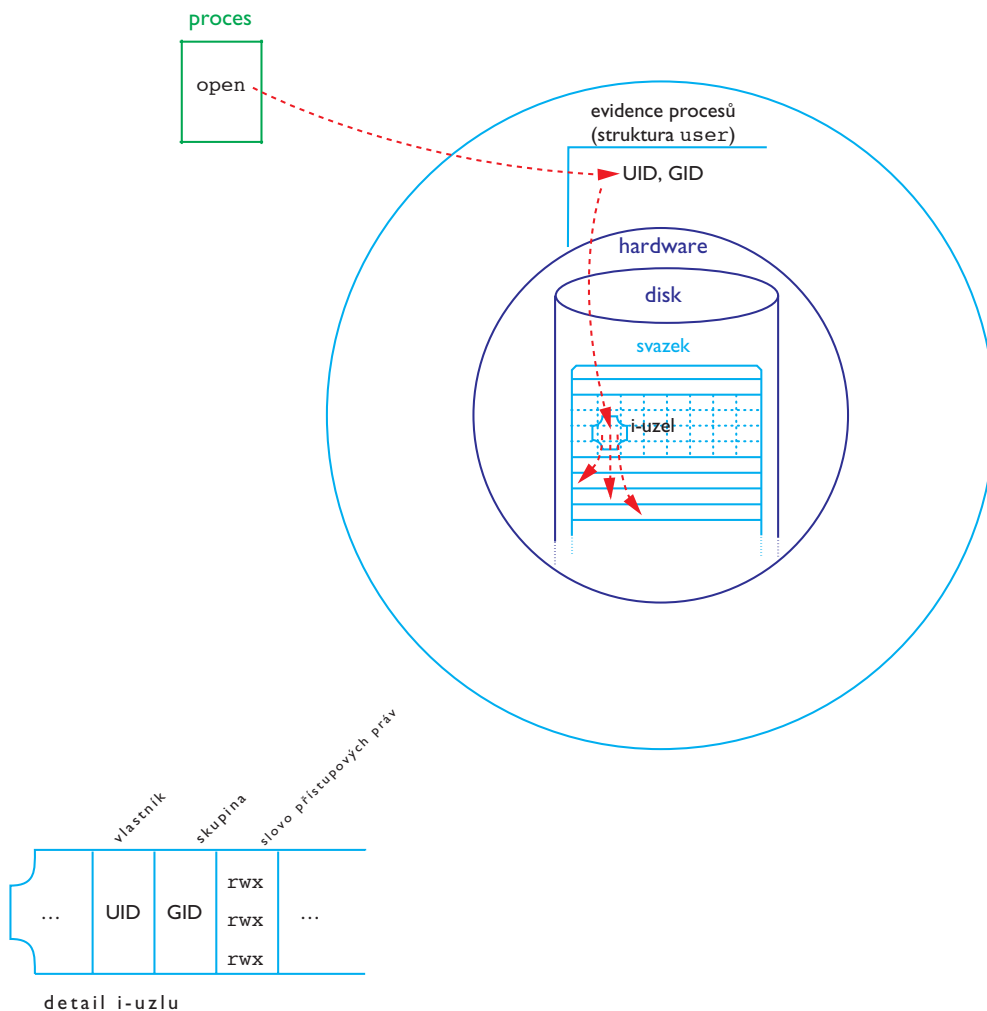
ani GID, je testována trojice určená pro ostatní uživatele. Zvyk zobrazovat přístupová práva z i-uzlu je podle výpisu příkazu **ls** (**rwX** v případě platnosti nebo znak **-**, pokud je způsob přístupu nepovoleno) a je použit také na obr. 9.2.

Na obrázku opět vidíme sledování jádrem. Proces se k souboru nedostane, pokud nepoužije volání jádra **open**. Je-li použito, jádro, které dříve proces vytvořilo a registruje jej (ve strukturách **proc** a **user**, viz kap. 2), testuje, zda má proces požadované oprávnění soubor číst (**r**), zapisovat do něj (**w**) nebo jeho obsah provádět (**x**), a to podle obsahu i-uzlu a UID a GID procesu. Vzniká-li soubor (voláním jádra **creat** nebo také **open**), jádro po testu přístupu k adresáři, kde má být soubor vytvořen do nově alokovaného i-uzlu, vkládá informace získané z tabulky registrace procesu, tj. UID a GID. Přístupová práva přitom přebírá z masky vytváření souborů, kterou má každý proces nastavenou opět v tabulkách jádra (proces ji může měnit voláním jádra **umask**). Současně může proces v parametrech volání jádra, které soubor vytváří, formou bitového doplňku k masce výchozí přístupová práva měnit.

Systém souborů v UNIXu je využíván také pro přístup k dalším výpočetním zdrojům, jako jsou periférie (adresář **/dev**), schránky v sítích BSD nebo pojmenovaná roura při komunikaci více procesů. Informacemi o vlastnictví a přístupových právech odpovídajícího i-uzlu se opět zabývá jádro při pokusu procesu výpočetní zdroj využít. Prováděcí algoritmus je přitom shodný s testem u obyčejných souborů nebo adresářů. Způsob přístupu k výpočetním zdrojům prostřednictvím algoritmů systému souborů se tak ukazuje výhodný i z pohledu soustředění ochrany proti zneužití do jednotné metody. V UNIXu však také najdeme výpočetní zdroje, které se systémem souborů spojeny nejsou. Jde např. o další prostředky komunikace mezi procesy, sdílenou paměť, semaforey nebo fronty zpráv, které jsou procesům přístupny prostřednictvím klíčů, ne jmen souborů. Přesto i zde je zaveden mechanismus přístupových práv obdobný systému souborů (viz kap. 4). Odpovídajícím voláním jádra (**getshm**, **getsem**, **getmsg**) lze číst vlastníka, skupinu vlastníka a odpovídající trojici přístupových práv. Tyto informace jsou uloženy do evidence jádra podle informací o procesu, který odpovídající komunikační oblast vytváří. Další vývoj zjevně směřuje opět ke spojování výpočetních zdrojů se systémem souborů. Je to zřejmé především ze studia standardu POSIX.

Přístupová práva k výpočetním zdrojům a jejich změna jsou v moci majitele výpočetního zdroje (u souborů lze využít volání jádra **chmod**, **chown** nebo příkazů **chmod**, **chown** a **chgrp**). Proto je i pro obyčejného uživatele důležité vědomí uvedených souvislostí pro vlastní ochranu dat a správce systému pak nesmí žádný důležitý systémový zdroj ponechat ve vlastnictví obyčejných uživatelů. Zde se dostáváme k prvnímu z úzkých míst bezpečnosti operačního systému UNIX. Je věcí správce systému, jak výpočetní zdroje přidělí. Přístup ke chráněným výpočetním zdrojům je totiž pro obyčejné uživatele poskytován prostřednictvím procesu se svěřeními přístupovými právy majitele souboru s programem (efektivní vlastník procesu je na dobu provádění majitel souboru). Toho se dosahuje nastavením s-bitu ve slově přístupových práv v i-uzlu. Správce systému pak přebírá odpovědnost za akce všech nových aplikací, které programátor vytváří a pro jejichž provoz je nutné získat přístup k systémovému zdroji. Používání s-bitu je častá příčina porušení bezpečnosti, a to zejména je-li s-bit použit pro interpretované soubory, jako jsou scénáře pro shell, prostředek **perl** atp. Proces, který scénář interpretuje, pak totiž pracuje jako privilegovaný. Text scénáře lze přitom obratným manévrem zaměnit za jiný a získat tak privilegovaný přístup (např. ukončením dětského procesu shellu přerušením z klávesnice). Správce systému by proto neměl podporovat programování důležitých veřejných aplikací v interpretech. Nebezpečné jsou také scénáře startu operačního systému uváděné v **/etc/inittab** (viz kap. 10), které lze





Obr. 9.2 Přístup k souborům

v případě nevhodného naprogramování přerušit a získat tak privilegovaný přístup prostřednictvím shellu, který scénář interpretoval.

Bijekce jmenné a číselné identifikace každého uživatele je obsahem tabulky `/etc/passwd`. Je to současně tabulka, která jako jediná registruje všechny atributy každého uživatele, jak bylo uvedeno v kap. 5. Je veřejně čitelná, protože každý proces musí mít možnost prověřovat existenci uživatele, převádět jmennou identifikaci na UID (a naopak) nebo prověřovat přístup porovnáním hesla (viz

další text kapitoly). Jisté proto je, že je to klíčová tabulka operačního systému, bez jejíž existence nelze výpočetní systém provozovat. Proto je na ni zaměřen i zájem některých osob, tzv. *hackerů*<sup>2</sup>, kteří usilují o neoprávněný přístup k datům a jejich znehodnocení. Vzhledem k tomu, že každý přihlášený uživatel může `/etc/passwd` číst, nestandardní aktivity se zaměřují na uživatele, kteří nemají registrováno heslo nebo jejichž heslo je příliš jednoduché. Dobře navržené heslo každého uživatele je základem ochrany celého systému před nežádoucím přístupem.

V úrovni D může být heslo prakticky libovolné. Úrovně vyšší nutí uživatele kombinovat v textu hesla také číslice a speciální znaky (jako je např. #, \$ atd.). Heslo je také ve vyšších úrovních přesunuto do jiných souborů a je čitelné pouze privilegovanými procesy. Rovněž tak mechanismus kontroly hesla ve vyšších úrovních nutí uživatele heslo v určitých časových úsecích (např. jeden měsíc) měnit na jiné. Šifrovací algoritmus je používán podle čl. 9.4.

Tabulka `/etc/passwd` obsahuje také privilegovaného uživatele (superuživatele) se jménem `root`, jehož číselná identifikace je 0. Procesy s tímto UID pak mají oprávnění prakticky neomezené manipulace se všemi výpočetními zdroji. Heslo uživatele `root` je proto jedna z nejdůležitějších informací instalace operačního systému a uvedený komentář k heslu uživatele o něm platí dvojnásob.

Tabulka `/etc/group` registruje používané skupiny uživatelů (opět viz kap. 5). Jak jsme již uvedli, uživatel může být účastníkem několika skupin a podle toho má přístup k dalším výpočetním zdrojům. Vytvořením skupin uživatelů, kterým zpřístupníme některé systémové zdroje, lze takto stupňovat přístupová práva systémové úrovně pro manipulaci např. s tiskárnou nebo síťovými službami. Běžné takovou skupinou uživatelů bývá `sys` nebo `bin`.

Podle konstrukce přístupových práv je výpočetní zdroj přístupný majiteli, skupině a ostatním. Nelze ovšem definovat přístup současně několika skupinám nebo všem uživatelům, a pouze několika vyjmenovaným přístup zamezit. Bezpečnostní úrovně vyšší to umožňují, jak poznáme v čl. 9.3.

Při sledování práce operačního systému je z hlediska bezpečnosti důležité evidovat všechna sezení, která proběhla. Vzhledem k tomu, že vstup do systému nemusí být provázen přihlášením některého uživatele (např. u sítí je to proces serveru, jehož majitel je označený místní uživatel), sledování aktivit uživatelů v systému musí být evidováno složitějšími mechanismy, které dále musí být rozšiřovány s vývojem nových součástí systému. Základní systémové soubory s evidencí vstupu uživatelů do systému jsou uloženy v adresáři `/usr/adm`. Soubor `lastlog` obsahuje poslední vstup každého uživatele do systému, `utmp` pak záznamy o každém přihlášení uživatelů a `wtmp` záznam o každém přihlášení a odhlášení uživatelů. Monitorování všech příkazů, které uživatel používá, je evidováno v souboru `acct`. Adresář `/usr/adm` je dobré prozkoumat s provozní dokumentací v ruce, protože výrobci sem často ukládají další užitečné výsledky práce operačního systému. Navíc, jak je zřejmé z kap. 5, různé verze sledování a finanční poměrování spotřebovaného strojového času uživatelem ukládají výsledky v různých formátech a v různých jménech souborů. Síťové aplikace, jako je např. WWW, mají své vlastní sledovací mechanismy. Umístění souborů jejich výsledků je ovšem vždy závislé na používané verzi a s vývojem síťové aplikace se různí. Každopádně při poskytování takových služeb je důležitá kontrola, komu a za jakým účelem je poskytnuta. Ohrožení datové základny ze strany síťových služeb uvedeme ve čl. 9.3.

## 9.2 Škůdci

Záměrný průnik do operačního systému s cílem paralyzovat nebo zcizit data výpočetního systému je běžný prostřednictvím programových fragmentů (tzv. threats), které hacker umísťuje do nechráněných míst v systému tak, aby byl fragment prováděn při běhu některého obvyklého (nejlépe systémového) programu. Pokud má vzniklý proces napadený fragmentem oprávnění superuživatele, může fragment obsahovat instrukce k provedení jakékoliv systémové akce. Současně je mu také přístupná celá datová základna, a to pro všechny typy operací. V současné době rozeznáváme několik typů takových škodlivých programových fragmentů.

Využití *zadních vrátek* (back door nebo také trap door) je jedna z cest, jak použít škodlivý fragment, aniž musí hacker cokoli programovat. Často totiž programátor systémové aplikace ponechá v kódu funkce, které slouží k ladění, monitorování nebo testování aplikace při její instalaci. Při takových akcích je obvykle potřeba mít nastavenou prioritu superuživatele. Také má význam používat interaktivní způsob ladění nebo testování s možností startovat další procesy. Interaktivní přístup k procesu, který má nastavenou prioritu superuživatele, jsou sice zadní vrátka programátora aplikace, ale současně otevřené dveře pro hackera, který tak získává neautorizovaný přístup k operačnímu systému. Příklad známého průniku zadními vrátky je verze síťového serveru **sendmail** z r. 1988 při použití volby **DEBUG** (ladění). Ochrana proti zneužití zadních vrátek je obtížná. Důležité je mít jistotu v serverech síťových aplikací, jako je **telnetd**, **ftpd**, ale i v programu zajišťujícím vstup do systému, jako je **login**. Tu by měl poskytnout seriózní výrobce. Pokud používáte volně šiřitelný software, podrobně prostudujte jeho zdrojový kód. Kontrolujte pravidelně soubor **.rhosts**, kterým uživatel (nebo superuživatel) umožňuje vstup z jiných uzlů sítě pomocí serveru **rlogind** bez kontroly hesla. Dbejte na to, aby tabulky exportu svazků v NFS vždy obsahovaly seznam uzlů, které svazky mohou využívat. Pravidelně prohlížejte tabulku pro záměny (aliases) v poštovním podsystému **sendmail**, kde může hacker zanechat odkaz na start programu, který **sendmail** provede. Kontrolujte přístupová práva k adresáři **/etc** a speciálními souborům, jako je **/dev/kmem**, speciálními souborům disků atd. Scénářem pro shell pravidelně procházejte všechny svazky systému souborů a hledejte všechny soubory, které mají nastaven s-bit. Pravidelně kontrolujte, zda se neobjevila nová síťová služba umožňující spouštět vzdáleně shell pod uživatelem **root**.

Termínem *logická bomba* (logic bomb) nebo také skrytá vlastnost, jsou označovány škodlivé fragmenty v programech, které se projeví s výskytem určitých podmínek. Takovými podmínkami může být ztráta určitého důležitého souboru, dosažení určitého počtu startů aplikace nebo známá podmínka vypršení časového intervalu (např. několikaměsíční prodlení splatnosti faktury za aplikaci). Logická bomba je do aplikace vkládána obvykle programátorem, a to do částí kódu, který vyžaduje nastavený s-bit při běžném provozu. V průběhu své nečisté práce může pak škůdce paralyzovat nejenom data aplikace, do které byl vložen, ale i ostatní, obvykle systémová data. Obranou proti logické bombě je dodržování postupů hlídání zdaních vrátek programů. Velmi důležitá je také pravidelná archivace provozních a systémových dat.

*Viry* (viruses) jsou programy, které mění obsah souborů s jinými programy tak, že do nich vkládají samy sebe. Přestože první viry byly napsány pro UNIX, dnes jsou rozšířeny především na platformě PC a Macintosh. Tam je totiž jejich šíření jednodušší, protože jde o systémově prakticky zcela nechráněné osobní počítače. Modifikace programů s nastaveným s-bitem v UNIXu je totiž obtížná a vyžaduje předtím ještě použití např. metody zadních vrátek. Přesto je dobré v UNIXu pravidelně provádět

kontroly konzistence svazků, prověřovat nové adresáře před jejich zařazením do proměnné `PATH` v shellu a všechny již zařazené adresáře (`/bin`, `/usr/bin` atd.) nenechávat dostupné pro zápis, o programech v nich uvedených a používaných už ani nemluvě; nejlépe je mít nastavena přístupová práva souborů s programy na `r-sr-sr-x` nebo ještě lépe `r-s--s--x` (pokud nestačí na místech s pouhým `x`), neinstalovat programy dostupné pouze v binárním kódu z neproověřených zdrojů – autorizace výrobcem se obvykle vyplatí. Každý uživatel by také neměl ponechávat své privátní adresáře otevřené pro zápis jiným uživatelům, ani stejné skupiny.

*Červíci* (worms) jsou zajímavou podobou škůdců. Jde o malé programy, které se šíří sítí mezi jednotlivými uzly, pronikají do operačních systémů, sbírají informace, ale obvykle přitom nemění obsah souborů s programy, ani neparalyzují data (přestože to není pravidlem). Programátor takového škůdce musí mít zkušenosti nejenom s UNIXem, ale i se sítěmi a jejich programováním. Zaznamenat červíka je mnohdy obtížné, nicméně se obvykle projevuje zvláštním chováním v systému procesů. Správci systému se proto doporučuje při takovém zvláštním chování prohlédnout stránky WWW veřejných organizací pro škůdce, případně zaslat na jejich adresu poštou popis zvláštního chování. A pokud se červík již někde projevil (což je pravděpodobné), je nutné řídit se ihned doporučením, které organizace zveřejňují, především archivovat kritická data.

*Trójský kůň* (Trojan horse) je program, který má zdánlivě provádět určitou funkci, ale ve skutečnosti vykonává něco zcela jiného. Trójské koně jsou obvykle volně šířitelné programy, především hry, nové údajně dobré editory atd. Obvykle již v průběhu úvodní interakce relace uživatele s programem (např. jakou klávesnici budete používat) přijde uživatel o svá data. Než zjistí, že program vlastně vůbec nefunguje, je pozdě. Pověstné je použití příkazu `rm -rf $HOME` ve scénáři pro shell. Ve stovce příkazů nějakého volně šířitelného scénáře, který vám má ulehčit práci s daty, jej snadno přehlédnete, pokud vůbec takový scénář studujete (a to byste měli!). A pokud jste dokonce privilegovaný uživatel, padá odpovědnost na vaši hlavu.

Konečně *bakterie* (bacteria nebo rabbit) je program, který vytváří kopie sama sebe s cílem paralyzovat operační systém vyčerpáním jeho zdrojů, jako je diskový prostor nebo operační paměť. Bakterie obvykle neničí data, pouze má za úkol provádět kopie samy sebe, a to dvakrát. Pak dochází k exponenciálnímu růstu jejich počtu a v okamžiku, kdy si správce systému povšimne, že mu rychle ubývá prostor na disku, je pozdě a svazky jsou rychle zahlceny. Bakterie je jeden z nejstarších typů škůdců a jeho průnik do systému je obvykle způsoben opět nepozorností při instalaci nových aplikací nebo používáním veřejně dostupných her a jiných programů.

Předmětem zájmu škůdců je hlavně datová základna. Žertovní škůdci mohou pouze odstranit důležité systémové tabulky (např. `/etc/passwd` nebo `/bin/sh` atd.). Mnohdy jsou ovšem škůdci neomalení, a když se jim podaří do systému vniknout, paralyzují celou datovou základnu. UNIX naštěstí není nechráněný systém a autoři škůdců jej málo znají, a proto jejich pokusy končí obvykle poškozením jenom určité části dat. Pokud se však objeví škůdce kvalitní, neštěstí je hotovo. Nejdůležitější přitom zůstává průnik do systému. Co největší zúžení možností průniku je věcí správce systému, jeho mravenčí práce s testováním programů, kterými uživatelé nebo procesy z jiných systémů vstupují do uzlu. Pro samotného uživatele je také důležitá kázeň provozu, protože tak chrání svá vlastní data. Provozování hry z neznámého zdroje v neprivilegovaném režimu sice neohrozí funkci operačního systému, ohrožuje však bezpochyby data uživatele, který si hraje. Systematicky se možností snížení průniků zabývají dokumenty uvedené v úvodu kapitoly. Jak jsou jejich požadavky implementovány v UNIXu, si uvedeme v následujícím čl. 9.3.

### 9.3 TCSEC v SVID

TCSEC definuje čtyři skupiny bezpečnosti dat s označením D, C, B a A. Každá skupina (s výjimkou D) přitom obsahuje několik číselně rozlišených úrovní. Bezpečnostní úrovně jsou tak od nejslabší po nejsilnější definovány s označením D, C1, C2, B1, B2, B3 a A1. Každá nižší úroveň je přitom podmnožinou úrovně vyšší, jak vidíme na obr. 9.3, kde jsou také uvedena odpovídající označení bezpečnostních úrovní podle evropských dokumentů ITESC a ZSIEC.

Skupina A, v rámci pouze jedné úrovně A1, definuje výpočetní systémy s přísnou kontrolou výhradního přístupu, a to za použití formálních metod. Operační systém musí být demonstrativně dokazatelný podle všech vlastností definovaných ve skupině B.

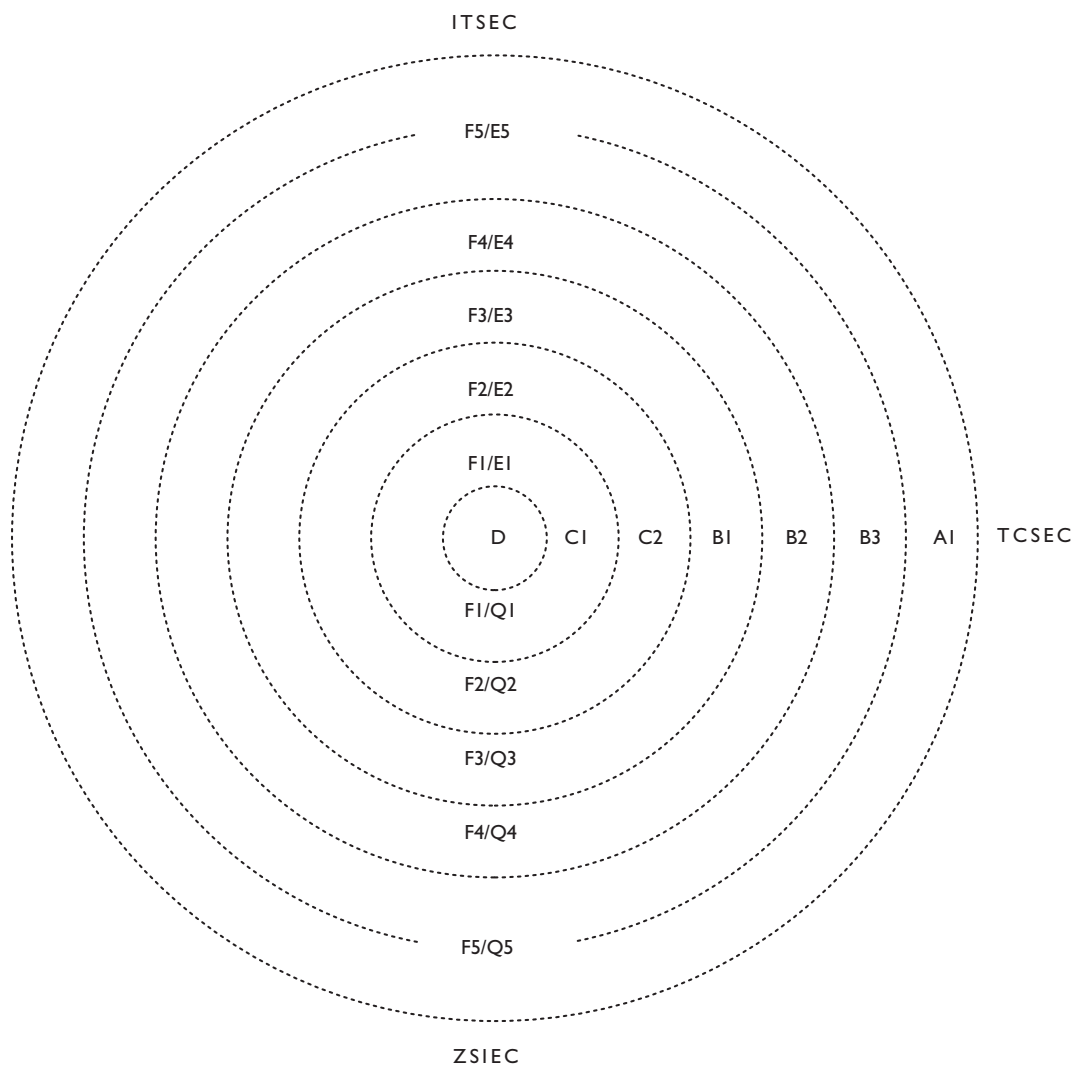
Skupina B obsahuje úrovně B1 až B3. Je definován tzv. *nařízený přístup* (MAC, Mandatory Access Control), který zajišťuje ochranu označených výpočetních zdrojů, a to souborů, zařízení a procesů pro práci uživatele a jemu odpovídajících činností v daném stupni ochrany. V B1 jde o úvodní nastavení ochranných přístupů, B2 podporuje práci s těmito přístupy definicí bezpečnostního podsystému a B3 uvádí bezpečnostní domény pro definici přístupu k systému ochrany. Součástí skupiny B je také *obezřetná kontrola přístupu* (DAC, Discretionary Access Control), která umožňuje nastavit přístupová práva k souborům, jejichž obsah je důležitý pro běh procesů.

Slabší je bezpečnostní skupina C. Úroveň C1 a C2 obsahuje přísné sledování vstupů uživatelů do systému. C1 je vhodná pro použití v běžné komerční činnosti. C2 je další zostření přístupu k datům. Definuje sledování (trasování) uživatelů a zajištění integrity jejich dat.

Skupina D (bez číselné přípony) zahrnuje nechráněné systémy. Patří sem sice také systémy s určitou ochranou, ale podle definice jde o operační systémy, které nevyhovují definicím uvedených silnějších skupin.

Operační systémy, které odpovídají doporučení SVID (libovolného vydání), odpovídají bezpečnostní úrovni C1. Pátý svazek třetího vydání SVID dále zavádí rozšiřující bezpečnost (enhanced security) a definuje (a to až na úroveň příkazů a funkcí) bezpečnostní podsystémy, které odpovídají úrovním C2, B1 a B2.

Změna podle SVID oproti dosavadnímu UNIXu je zavedení více privilegovaných uživatelů. Na rozdíl od původní koncepce, kde uživatel s UID 0, tzv. *root*, má přístup k systémovým akcím s neomezenými přístupovými právy, je zde možné vytvářet více uživatelů pro různé systémové akce, ale s omezenými přístupovými právy k dalším systémovým akcím. Takovým uživatelům říkáme administrativní uživatele (administrative users, „trusted“ users). *Databáze administrativních uživatelů*, tzv. databáze TFM (Trusted Facility Management), je uložena v podstromu adresáře */etc/security/tfm* a je při provozu otevírána obyčejným uživatelem pomocí příkazu **tfadmin**. Příkaz má přístup do databáze TFM, ve které má uživatel definovány jemu přiřazené systémové příkazy (např. je vyhrazen uživatel, který může provádět zálohu dat). Požadovanou systémovou akci zadává uživatel formou systémového příkazu v argumentech **tfadmin**, který prověří jeho práva a pomocí **exec** se promění na požadovaný systémový příkaz. Správa databáze TFM je pak zajišťována privilegovanými příkazy **adminuser** pro přidání, změnu, zrušení nebo jenom zobrazení administrativního uživatele, a **adminrole**, který spravuje databázi TFM z pohledu používaných systémových příkazů (i zde zavede nový, zruší, změní nebo jen zobrazí systémový



Obr. 9.3 Hierarchie bezpečnosti v americkém dokumentu a evropských dokumentech

příkaz). Správně řečeno spravuje **adminrole** část databáze TFM tzv. rolí (role). *Role* obsahuje seznam příkazů a každému příkazu v roli je přidělen (třeba i prázdný) seznam jmenných priorit. Priorita je pak použita procesem **tfadmin**. Např. můžeme zavádět novou roli pro připojování svazků takto

```
adminrole -a mount:/etc/mount:pripojeni:mount
```

kde znakem **:** rozdělený argument obsahuje postupně jméno role, cestu k příkazu a jmenné priority. Role běžně obsahuje více příkazů. Pomocí **adminuser** pak vytvoříme uživatele a přiřadíme mu požadovanou roli nebo jen příkaz role. Uživatelé v databázi TFM musí být přitom dříve evidováni v tabulce souboru `/etc/passwd`.

Druhou možností, jak může obyčejný uživatel použít privilegovaný příkaz, je použitím proveditelného souboru, který má nastavenou tzv. pevnou prioritu (fixed privilege) podle databáze TFM její jmenných priorit. Jde o obdobu původního s-bitu. Pevnou prioritu nastavujeme privilegovaným příkazem **filepriv**, pomocí kterého také získáme informace o nastavených prioritách každého souboru (soubor může mít nastaveno několik různých jmenných priorit). Soubor přitom může mít nastavenou také prioritu dědičnou (inheritable), což znamená, že je přístupný procesem s odpovídající jmennou prioritou. Znamená to, že dědičná priorita je platná pouze tehdy, prokáže-li se jí také již existující proces (tj. **tfadmin** před svou proměnou). Tatáž pevná i dědičná jmenná priorita nemůže být pro tentýž soubor nastavena současně. SVID tak zavádí definice minimálních priorit (least privilege), které musí být nastaveny pro vykonání určité systémové akce. Proces se pak pohybuje v rozpětí dvou priorit: *pracovní* (working), tj. množiny všech jmenných priorit, které musí mít nejméně nastaveny, a *maximální* (maximum), což je množina jmenných priorit, které může proces dosáhnout vůbec.

TFM je jedna z úprav UNIXu, která jej umožňuje certifikovat až na úrovni B2. Samotné TFM k tomu ale ještě zdaleka nestačí. Úroveň B2 musí být totiž plně zajištěna podle MAC. Tento nařízený přístup je v SVID aplikován na přístup k souborům a prostředkům komunikace mezi procesy IPC. MAC je založeno na rozdělení práce s výpočetními zdroji do různých citlivých úrovní (sensitivity levels). Přiřazení takové citlivé úrovně souborům a prostředkům IPC (zde souhrnně nazývaných objekty) je základ realizace MAC. Běžící proces se také prokazuje určitou úrovní MAC a podle ní je mu přístup k objektům dovolen nebo zakázán. Platí přitom, že procesu je určitý objekt přístupný pro čtení a provádění, pokud úroveň MAC procesu je minimálně shodná s úrovní MAC objektu nebo vyšší. Pro zápis je přitom vyžadována shoda úrovní MAC objektu i procesu. (Politiku MAC lze charakterizovat jako zákaz byť pouze čtení informací z vyšších úrovní a pouze zápis v úrovni vlastní.) Zůstává však možnost, kdy proces může mít zvláštní prioritu, a pak je mu dovoleno vše. Úrovně MAC mohou být v hierarchickém vztahu, ale také nemusí. Přístup ke speciálním souborům není v žádném případě veřejný. Zařízení musí být nejprve zpřístupněno v bezpečnostní databázi zařízení, a pak teprve s ním může být zahájena manipulace.

Základní příkaz pro stanovení různých úrovní MAC je **lvlname**. Je vyhrazen pro správce systému, který jím definuje úrovně MAC. Zavádí je s číselným označením LID (level identifier) volbou **-l**, zařazuje do hierarchie nebo kategorie (**-c** a **-h**) a přiřazuje jim jmenné přezdívkami (aliases, **-a**). Např.

```
$ lvlname -l 150::Guru
```

zavádí úroveň MAC s číselným označením 150. Jméno, které lze používat pro tuto úroveň MAC, je **Guru**. V adresáři `/etc/security/mac` je pak v souboru `lid.internal` uveden seznam kvalifikovaných úrovní MAC v jejich LID. Adresář obsahuje i další soubory (s přezdívkami atd.).



Historie změn v úrovních MAC je přitom zachycena v souborech začínajících na `hist..` Výpis historie změn úrovní MAC lze získat použitím volby `-p`. Mechanismus přitom ponechává systému určité úrovně rezervovány (viz provozní dokumentace), přestože volbou `-r` je lze také z rezervace vyjmout. Zrušit úroveň MAC může správce pouze příkazem **lvldelete**. Příkazem ruší nejenom LID, správce musí příkaz použít i na zrušení přezdívky nebo jmenného označení úrovně atp. Ale i obyčejní uživatelé mohou požádat o výpis všech vytvořených úrovní MAC, které byly definovány. Příkaz **lvlprt** takový seznam nabídne, a to včetně jmen, přezdívek, hierarchie, případně kategorie každé úrovně MAC. Souborům přiřazuje (opět pouze správce systému) úroveň MAC příkaz **chlvl**. Proces získává úroveň MAC podle úrovně proveditelného souboru. Od úrovně MAC procesu se také odvozuje úroveň MAC při vytváření prostředku IPC. Problémy nastávají u tzv. veřejných adresářů (např. `/tmp`), protože přiřazenou úroveň MAC tomuto adresáři nebude splňovat každý proces. Proto byla zavedena možnost používání tzv. *víceúrovňových adresářů* (multilevel directory). Do adresářů takto označených příkazem **mlmode** mohou zapisovat procesy různých úrovní MAC. U speciálních souborů přitom nestačí pouze pomocí **chlvl** přiřadit určitou úroveň MAC. Zařízení v bezpečnostní skupině B musí být přiřazeny k použití konkrétním uživatelům podle databáze zařízení (DDB, Device Database). DDB vzhledem k uživatelům spravuje správce příkazem **admalloc**. Databáze je evidována v souborech `/etc/device.tab`, `/etc/security/ddb/ddb_dsf_map` a `/etc/security/ddsb/ddb_sec`. Uživatelé lze přitom přiřadit přístup k zařízení v určité úrovni MAC nebo v rozsahu několika úrovní (od - do). Stejně tak lze uživateli přístup k zařízení odebrat. Samotnou databázi vytváří a o další zařízení doplňuje správce příkazem **putdev**, tj. přidává nebo odebírá zařízení a nastavuje atributy zařízení. Atributy lze nastavovat ve velké škále. Pomocí nich lze zařízení prakticky vyčerpávajícím způsobem popsat z pohledu použití uživatelem, jako jsou atributy obsahující kapacitu zařízení, jméno speciálního souboru, příkazy pro formátování atd. Podstatná je ale skupina *bezpečnostních atributů* (security attributes), jako je např. rozsah úrovní MAC, ve kterých zařízení může být uživateli přiřazeno, zda může být zařízení používáno jako veřejné nebo přísně soukromé atd. Při používání doporučujeme pracovat s provozní dokumentací. Seznam zařízení z DDB lze získat příkazem **getdev**, atributy zařízení příkazem **devattr**. Konečně současné využití zařízení (kterým uživatelem a za jakým účelem) z DDB může správce získat příkazem **devstat**.

Součástí úrovně B2 je také *seznam řízeného přístupu* ACL (Access Control List), který realizuje DAC z Oranžové knihy. Jde o rozšiřující přístupová práva k souborům a prostředkům IPC. ACL je doplňující bezpečnostní podsystém skupiny B, který bývá často používán i bez nasazení MAC (pak ovšem není zajištěna skupina B). Přístup k datům je zde organizován podle uživatelů a skupin (soubory `/etc/passwd`, `/etc/group`), přístupová práva jsou také čtení (read), zápis (write) a provádění (execute). Původní metoda přístupových práv v UNIXu však zůstávala na úrovni tří přístupových způsobů, tj. pro vlastníka, skupinu a ostatní. V ACL lze vyjmenovat uživatele nebo skupiny, jejichž přístup je povolen nebo zakázán (vždy pro `r`, `w` nebo `x`). Prakticky tak lze např. nastavit přístupová práva pro čtení pro všechny uživatele kromě uživatelů patřících do určité skupiny, nebo ještě dále, přístup pro všechny s výjimkou několika (asi zlobivých) uživatelů. Přístupová práva souborů podle ACL zajišťuje v UNIXu příkaz **setacl** a **getacl** (volání jádra `acl`, `aclipc` a `aclsort`, viz obr. 9.4). Příkazem **setacl** může uživatel s odpovídajícím oprávněním měnit obsah seznamu ACL požadovaného souboru. Výchozí jsou údaje z obsahu `i`-uzlu, a to vlastník souboru, skupina a jejich přístupová práva. Znamená to, že přestože je používán ochranný podsystém ACL, původní přístupová práva



souborů zůstávají respektována (včetně masky vytváření souborů nebo změny přístupových práv podle volání jádra `chmod`). Práce s ACL přitom dále vychází z přiřazení práv konkrétním uživatelům (`user`), skupinám (`group`), třídám skupin (`class`, jde o nově zaváděnou kategorii) a ostatním (`other`). Přístupová práva souboru jsou tedy dána přístupovými právy v i-uzlu vztahujícími se k vlastníkově a skupině vlastníka a ostatním a další omezení jsou dána informacemi v ACL. Označení `default` lze využít pouze pro adresáře a definujeme pomocí něj uživatele, skupinu a ostatní, kteří mohou (nebo jsou naopak omezeni) v daném adresáři vytvářet soubory. `setacl` přístupová práva nastavuje, mění nebo odebírá. Např.

```
$ setacl -m user:lenkam:r-- text
```

přidává k ACL souboru `text` přístup uživatele `lenkam` pro čtení. Ostatní práva jsou tomuto uživateli odeprána, jiní uživatelé přitom mohou mít přístup zcela jiný. Získat seznam ACL pak uživatel může pomocí příkazu `getacl`, který na standardní výstup vypisuje úplný (nebo podle použitých voleb pouze částečný) obsah ACL. Studujte ACL a používejte jej! I bez instalace MAC se vám vyplatí zvláště v průmyslových provozech.

TFM, MAC a DAC (ACL) je nutnou podmínkou úrovně B2. Pro úroveň B1 je dostačující pouze MAC (jak TFM, tak DAC není vyžadováno, viz obr. 9.4). Celá skupina B však ještě vyžaduje ochranný podsystém *bezpečné cesty* (TP, Trusted Path) a *bezpečnostní formát přenosu dat* (Trusted Import Export).

Bezpečnou cestou je myšlena zajištěná komunikace mezi uživatelem a systémovou akcí (systémovým procesem, který musí být zabezpečen), kdy uživatel mnohdy komunikuje prostřednictvím sítě nebo telefonického spojení. Součástí bezpečné cesty je *klávesa zvláštní pozornosti* (SAK, Secure Attention Key), kterou uživatel nastavuje pomocí příkazu `defsak`. Stiskem definované klávesy (nebo kombinace kláves) uživatel nastaví bezpečnou cestu mezi ním a systémem. Tak je např. zajištěno výhradní právo požadovat heslo uživatele pouze systémovými (jako je `login`), nikoliv neoprávněnými procesy. Příkaz `defsak` je prostředek pro definici, zobrazení, změnu nebo zrušení SAK pro terminál (připojeného sériovým rozhraním nebo sítí). Při nastavení bezpečné cesty v úrovních B2 a B1 se nemůže uživatel bez předchozího zadání SAK do systému přihlásit. Bez parametrů vypisuje seznam všech nastavených SAK pro všechny připojené terminály. Vždy na jednom řádku je pro každé jméno speciálního souboru za dvojtečkou uveden odpovídající SAK. Pomocí volby `-d` následované kódem (osmičkově nebo mnemonicky, např. 001 nebo ^A) definuje uživatel SAK. Pokud je za `-d` uveden text `none`, bezpečná cesta je odstavena (používá se např. u podsystému UUCP). Pro místní síť se doporučuje používat pro všechny terminály tentýž SAK, odlišný od běžných řídicích znaků terminálového rozhraní (např. pro řízení protokolu XON/XOFF atp.). Doporučuje se naopak používat klávesu současně definovanou pro signál přerušení (tzv. `break`) nebo zrušení řádku linkové disciplíny (tzv. `drop`, line drop signal). Nahlédnutím do kap. 6 byste měli umět odvodit proč. Za účelem takového nastavení slouží při definici SAK doplňující volba `-x`. Zrušení SAK se dosáhne volbou `-r`. Při nastavení využívání podsystému bezpečné cesty je takto terminál odpojen a uživateli je znemožněno se prostřednictvím něj přihlásit do operačního systému. Na rozdíl od `-d none`, kdy je pouze SAK nedefinován, je možné kdykoliv zabezpečený vstup obnovit novou definicí SAK.

Jako doplněk k zajištění bezpečnosti dat skupiny B je archivní formát chráněných dat. Jde o tzv. *bezpečnostní formát přenosu dat* (Trusted Import Export) a je zajišťován příkazem `tcpio` (trusted `cpio`). Jde

o prostředek archivace dat ve smyslu zachování jejich informací bezpečnostního charakteru. Při použití volby **-o** (podobně jako u **cpio**, viz 3.5.2) je podle seznamu jmen souborů na standardním vstupu vytvořen jejich archivní formát. Archivovány jsou přitom také bezpečnostní informace, jako je UID, GID a LID z MAC a informace ACL. Volbou **-i** zadáváme programu **tcpio** obnovení dat z dříve vytvořeného archivu. Pokud odpovídající identifikace bezpečnostního charakteru z archivu není v systému definována (byla např. zrušena), pak odpovídající soubor vyžadující takovou ochranu není z archivu vyjmut.

Úroveň C2 je v SVID zastoupena zesílenou kontrolou a sledováním vstupu a práce uživatelů v operačním systému. Na metody, jakými je možné vstup uživatele do systému sledovat, jsme se zaměřili již v kap. 5 a z pohledu bezpečnosti se jednalo o úroveň typu C1. C2 je dále v SVID definována zajištěním systému účtování s dalšími bezpečnostními prvky v tzv. Auditing Extension. *Bezpečnostní účtovací systém* je založen na evidenci bezpečnostních operací, které mohou v systému proběhnout. Bezpečnostní operace se přitom vztahují k používaným bezpečnostním metodám obvykle vyšší úrovně, ale proto je nutno takovou evidenci definovat již v úrovni C2. Typy bezpečnostních operací jsou určovány tzv. *příznaky účtování* (audit events). Nastavení příznaku účtování definuje správce systému a jejich množina pak určuje registrované bezpečnostní události. Pro zajištění určité minimální integrity hlídaného bezpečnostního podsystému je nutno mít nastavenou minimální množinu příznaků. Takovým příznakům, bez jejichž nastavení nelze podsystém provozovat, říkáme *pevné* (fixed) příznaky na rozdíl od ostatních, volně správcem systému dostupných, kterým říkáme *nastavitelné* (selectable) příznaky. Vznikají tak různé *třídy účtovacích příznaků* (audit event class), které jsou definovány vždy podle skupiny nastavených příznaků. Správce systému nastavuje účtovací příznaky příkazem **auditset**, a to pro jednotlivé uživatele. Uživatelé ale předtím musí mít definovanu tzv. účtovací masku pro bezpečnostní příznaky (user-specific audit mask), kterou definuje správce systému prostřednictvím příkazu **auditcnv**. Příkaz respektuje uživatele registrované pomocí **useradd** nebo **usermod** (viz. kap. 5). **auditcnv** nemá žádné parametry. Uvažuje soubory **/etc/passwd** a **/etc/default/useradd**, masku pro každého uživatele pak vytváří v souboru **/etc/security/ia/audit**. Po definici třídy účtovacích příznaků pomocí **auditset** správce systému startuje účtovací bezpečnostní podsystém příkazem **auditon**. Vypnout jej může pomocí **auditoff**, ale před opětným startem musí znova nastavit třídu účtovacích příznaků pomocí **auditset**. Běžící účtovací podsystém vytváří záznamy v souborech **/var/audit/MMDD###**, kde MMDD je číselné vyjádření měsíce a dne v měsíci a ### je sekvenční číslo. Takový soubor se záznamy (audit event log file) je správcem systému nastavitelný příkazem **auditlog**. Bezpečnostní účtovací podsystém pracuje také s tzv. *účtovací mapou* (audit map), která je uložena v souboru **/etc/default/audit**. Mapa vzniká z kolekce masek uživatelů jako produkt příkazu **auditset**. Správce systému příkazem **auditmap** převádí účtovací mapu do několika souborů (v adresáři **/var/audit/auditmap**), které jsou společně se souborem se záznamy používány příkazem **auditrpt** pro převod zaznamenaných číselných údajů na textové řetězce lidem srozumitelné. Doplnující je příkaz **auditfltr**, který převádí nasbírané údaje souboru s účtovacími záznamy do formátu dat XDR (viz kap. 7), která se tak stávají systémově nezávislá.

Přehled uvedených ochranných podsystémů SVID a seznam realizujících příkazů a volání jádra v souvislosti s odpovídající úrovní podle TCSEC ukazuje následující tabulka obr. 9.4.

Pro přehled si ještě uvedme seznam příkazů, které musí být změněny pro zajištění bezpečnosti ve všech uvedených ochranných podsystémech, protože obsahují odpovídající souvislosti. Jsou to **at**, **batch**,

ochranný podsystém		příkazy	volání jádra
B2	DICRETIONARY ACCESS CONTROL	<b>setacl, getacl</b>	acl, aclipc, aclsort
B2	TRUSTED FACILITY MANAGEMENT	<b>adminrole, adminuser, filepriv, tfadmin</b>	filepriv, procpriv, procprivil
-----			
B1	MANDATORY ACCESS CONTROL	<b>admalloc, chlvl, devattr, devstat, getdev, lvlname, lvldelate, lvlprt, mldmode, putdev</b>	devalloc, devdealloc, devstat, lvldom, lvlequal, lvlfile, lvlin, lvlipc, lvlout, lvlvalid, lvlproc, lvlvfs, mkmlld, mldmode
B1	TRUSTED IMPORT EXPORT	<b>tcpio</b>	
B1	TRUSTED PATH	<b>defsak</b>	
-----			
C2	AUDIT	<b>auditcnv, auditlog, auditmap, auditoff, auditon, auditrpt, auditfltr, auditset</b>	auditbuf, auditdmp, auditctl, auditevt, auditlog

Obr. 9.4 Realizace TCSEC v SVID

**cpio, cron, crontab, find, fsck, ipcs, listusers, logins, lp, lpstat, ls, mkdir, mkfs, mount, passwd, ps, useradd, userdel, usermod, volcopy** a **whodo**. Potřebné úpravy jsou přitom s odkazem na odpovídající bezpečnostní úroveň vždy komentovány v provozní dokumentaci.

## 9.4 Bezpečné sítě

Dnes nejrozšířenější metoda práce v sítích je založena na technologii klient - server. Připojení výpočetního systému k síti lze pak v operačních systémech s touto technologií pro podporu sítí (a UNIX k nim patří) vnímat ze dvou pohledů. Jednak je to zpřístupnění služeb jiných uzlů místním uživatelům a dále je to nabídka vlastních služeb pro vzdálené uživatele. Ošetření (výstupních) požadavků místních uživatelů do vzdálených uzlů znamená instalace a nastavení přenosových protokolů a zpřístupnění programů pro síťové klienty. Podpora pouze odcházejících síťových požadavků je sice sobecká, ale proti běžným průnikům ze sítě takřka ideální, protože především poskytování síťových

služeb vzdáleným uživatelům je nebezpečné. Jak jsme viděli v kap. 7, vždy totiž znamená vytvoření procesu serveru, který pracuje podle požadavků vzdáleného klientu. Je pochopitelně věcí autora (výrobce) síťové služby, zda klientu umožní od serveru požadovat nebezpečné aktivity. Bezpečí datové základny síťového uzlu správce systému posílí, pokud nabízené síťové služby omezí pouze na nejpotřebnější. Postupné omezování nabízených síťových služeb sice znamená zvyšování jeho izolace vůči síti, ale rozumná redukce tabulky v souboru `/etc/inetd.conf` a startovacích scénářů podle tabulky v souboru `/etc/inittab` je důležitou činností správce systému. Instalace síťových služeb je totiž výrobcem obvykle dodávána v maximalizované podobě, tj. uzel do sítě nabízí prakticky všechny služby, které výrobce v operačním systému implementoval. Rozvaha správce systému, které síťové služby ponechá pro síť dostupné a které odpojí, je tedy důležitá činnost, která souvisí s celkovou koncepcí místní sítě a jejího připojení na síť veřejné. Jisté je, že práce uživatelů v místní síti nepřináší tak velká rizika poškození dat, jako je vstup uživatelů z veřejných sítí. Instalace např. služby anonymního **ftp** nebo serveru WWW by proto měl správce pečlivě zvažovat a instalovat pouze na vyhrazeném uzlu místní sítě, a to navíc tak, aby i v případě průniku neměl škůdce možnost pokračovat jinými síťovými aktivitami na další uzly místní sítě (např. odstraněním serverů **rlogin**, **telnet**, **ftp** a další z tohoto uzlu). Dobře programovaný škůdce ale i tak dokáže síť pokračovat, pokud umí pracovat přímo s přenosovými protokoly (vyžaduje to ovšem znalosti síťového programování), protože přenosové protokoly jsou trvale přítomny v jádru každého uzlu sítě, připraveny plnohodnotně sloužit pro příchozí nebo odcházející síťové požadavky procesů. Navržené a implementované metody ochrany uzlů sítě, jako jsou firewalls (ochranné zdi), se proto snaží izolovat a sledovat vstupy a výstupy sítě na úrovni protokolární, jak uvidíme v dalším textu tohoto článku. Ještě předtím si ale ukážeme stručný přehled nebezpečných míst síťových služeb, se kterými musí správce systému přistupovat k návrhu zajištění bezpečnosti místní sítě.

### 9.4.1 Modemy

Propojení dvou výpočetních uzlů lze realizovat sériovým rozhraním RS-232. Pomocí něj lze systémy propojit podsystémem UUCP, případně použít nadstavbu PPP nebo SLIP. Ošetřením serverem PPP nebo třeba pouze procesem **login** tak ale vzniká přístup vzdáleného uživatele do místního uzlu, a to jak přímým připojením (tzv. null-modem), kdy na konci kabelu sériového rozhraní je namísto terminálu připojen jiný uzel (kabel ovšem musí odesílaná data převádět na druhé straně na přijímaná), modemovým spojením průchodem telefonní ústřednou. Rozdíl mezi přímým propojením a propojením prostřednictvím vytáčené telefonní linky (dialup line) je ve větším ohrožení bezpečnosti dat uzlu, protože vytáčená telefonní linka prochází veřejným územím, kde jsou možnosti zcizení procházejících dat odposlechem zcela mimo kontrolu správce systému. Sériové rozhraní, které je připojeno modemem, je také nabízeno prostřednictvím veřejného telefonního čísla vlastně komukoliv na světě, kdo se může o vstup do systému pokusit. Telefonní společnosti také poskytují tzv. pevné (pronajaté) telefonické linky (leased line), které zajišťují trvalé spojení dvou míst bez určení opačné strany telefonním číslem (ústředna pak nenabízí oznamovací tón). Tím je uzel přijímající uživatele více ochráněn, protože jde o trvalé spojení dvou lokalit, podobně jako je to u spojení přímým kabelem, přestože zde navíc spojení prochází veřejným územím. Správce systému při využití ať už vytáčené nebo pronajaté telefonní linky proto musí dbát na zvýšenou pozornost přihlašovaného uživatele. Problém zneužití připojeného uzlu

veřejnou telefonní linkou se navíc rozšíří, pokud je sériovým rozhraním realizováno plnohodnotné spojení sítě IP prostřednictvím PPP nebo SLIP. Uzel nebo celá síť připojená telefonní linkou je tak viditelná právě prostřednictvím jednoho sériově probíhajícího spojení, kterým prochází všechny síťové pakety. Tento dnes nejrozšířenější způsob připojování uzlů a sítí k Internetu je tedy nejméně bezpečný, a správce systému proto musí vyvinout úsilí k co největší možné ochraně úzkého místa průchodu všech síťových dat.

Jak jsme uvedli v kap. 7, sériové rozhraní je ošetřováno podsystémem UUCP. Správce systému by měl z pohledu bezpečnosti prozkoumat po správném nastavení UUCP obsluhu speciálního souboru sériového rozhraní a jeho přístupová práva. Sériové rozhraní podléhá stejným principům jako u připojení terminálu (viz. kap. 6). Procesy přihlášeného vzdáleného uživatele mají tento speciální soubor nastaven jako řídicí terminál, proto jádro přístupová práva a vlastnictví speciálního souboru mění podle současné situace.

V souboru `USERFILE` adresáře `/usr/lib/uucp` může v rámci UUCP správce systému vyjmenovat uživatele, kteří mohou vzdálené přihlášení využívat, a v souboru `L.cmds` nastavit procesy, které jsou vzdálení uživatelé oprávněni spouštět. Obsahem těchto souborů se zamezí např. vstupu uživatele `root` vzdáleně nebo jeho simulaci pomocí `su` (ale tím také znemožní vzdálenou správu uzlu). Je důležité prohlédnout všechny soubory programů UUCP a jejich přístupová práva, protože mnohé soubory vyžadují nastavení `s-bitu`. `S-bit` by pak měl zpřístupnit práva uživatele `uucp`, v jehož vlastnictví se podsystém nachází a jen zcela výjimečně (pokud vůbec) práva uživatele `root`. Pro správce je také užitečný soubor `LOGFILE` adresáře `/usr/spool/uucp`, kde jsou evidovány nejenom všechny vstupy vzdálených uživatelů, ale i jejich důležité akce, jako je např. seznam zpřístupněných adresářů nebo jména přenášených souborů (podrobněji viz čl. 7.5 nebo [OreDouTod96]). Palčivý problém v průběhu vzdáleného přihlašování je cesta hesla přihlašujícího uživatele ze vzdáleného místa do operačního systému. Přestože jej ovladač jádra neopisuje, nešifrované heslo prochází celou cestou sériového rozhraní. Obecně je řeší Trusted Path v úrovni B1 (viz předchozí článek). Průchod hesla vzdáleného uživatele sítí je ale problém obecný, který probereme v popisu podsystému Kerberos u plnohodnotných sítí. Při nastavení PPP nebo SLIP je podsystém UUCP uzpůsoben pro vstup pouze jednoho uživatele (např. `pppc1i`), při jehož přihlášení se namísto procesu některého z shellů rozbíhá proces serveru (např. `ppp`) a další vstup uživatelů probíhá prostřednictvím síťových služeb, jako je **telnet**, **rlogin**, **ftp** aj. Bezpečnost pak musí být zajištěna navíc i prostředky, které uvedeme v dalším textu.

## 9.4.2 Síťové servery

Jak již bylo uvedeno, úzké místo bezpečnosti datové základny určitého uzlu sítě dlí v práci procesu serveru, který ošetřuje příchozí síťový požadavek klientu. Také jsme se zmínili, že kvalita serveru je dána kvalitou jeho programátora a seriózností prodejce nebo distributora, což správce uzlu ovlivní pouze uvážlivým výběrem konkrétních síťových serverů služeb, které se rozhodne do sítě poskytovat. Na straně druhé použití technologie klient - server pro síťové služby umožňuje znalému programátorovi všechny požadavky klientů registrovat v určitém textovém souboru a zanechávat tak pro správce uzlu historii práce síťového serveru. Součástí práce serveru **telnetd** nebo **rlogind** je například registrace přihlašovaných uživatelů v souborech adresáře `/var/adm`, ve kterých zanechává tutéž informaci každý **login** nebo **xlogin** (viz kap. 5). Stejně tak ale mohou být doprovodné akce serverů nebezpečné, jak jsme se zmínili u typu průniku zadními vratky v čl. 9.2. Nahlédneme-li do souboru

/etc/inetd.conf, kde jsou určeny procesy serverů, jejich příkazový řádek startu a vlastník, ve většině případů v poloze uživatele, pod jehož identifikací má server pracovat, nalezneme **root**. To je nutné u síťových serverů, které jsou aktivovány na portech v rozmezí 0 - 1023 (rezervovaná čísla portů). Tato konvence operačního systému UNIX se nazývá práce na *bezpečných portech* (trusted ports). Pouze privilegovaný proces totiž může požadovat od jádra práci na těchto portech. Pokud by tomu tak nebylo, libovolný uživatel by mohl spustit program, který naslouchá na portu např. serveru **rlogind** a od vzdáleně se přihlašujících uživatelů získávat v převlečení za tento server hesla jejich sezení<sup>3</sup>. Ovšem v případě průniku narušitelem, kterému se podaří síťový server využít, jde o ideální prostředek paralyzy operačního systému. Testovat naslouchající proces na určitém portu může kdokoli (a zejména správce) klientem **telnet**. Jeho druhý argument příkazového řádku může nepovinně obsahovat číslo portu, jehož odpovídající server na straně vzdáleného uzlu, který je dán prvním argumentem příkazového řádku, bude startován. Např.

### \$ telnet localhost 110

je test odezvy serveru místního uzlu na portu 110 (je rezervován pro poštovní protokol POP). Správce tak může v dávce příkazů testovat odezvu na rezervovaných portech a zvažovat, zda je naslouchající server v rámci koncepce jeho sítě.

Dobrá metoda posílení bezpečnosti síťových serverů, které jsou startovány konkurentně superserverem **inetd**, je instalace volně šířeného softwaru TCP Wrappers (v Internetu jej lze vyhledat v serverech Archie pod jménem **tcp\_wrappers**). Základní myšlenka je vkládání dalšího procesu se jménem **tcpd** pro ošetření daného čísla portu jako serveru. **tcpd** kontroluje a registruje příchozí síťový požadavek a teprve poté se promění (voláním jádra **exec**) na proces požadovaného serveru. Hlavní úprava proto spočívá v modifikaci souboru /etc/inetd.conf. Např. je-li v souboru přítomen řádek

```
ftp stream tcp nowait root /usr/sbin/ftpd ftpd
```

nahradíme jej za

```
ftp stream tcp nowait root /usr/sbin/tcpd ftpd
```

V souborech /etc/hosts.allow a /etc/hosts.denny pak určíme uzly sítě, jejímž klientům je síťová služba poskytnuta, a naopak uzly, jejímž klientům je odepřena. V souborech vždy na jednotlivých řádcích za jménem síťového serveru (nebo několika jmen oddělných čárkou) následuje za znakem : seznam jmen uzlů oddělených znakem čárka, na které se povolení (nebo odmítnutí) služby vztahuje. Možné jsou ale i kombinace, jako např. v souboru **hosts.allow**

```
ftpd, telnetd : LOCAL EXCEPT web
```

což je povolení startu serverů **ftpd** a **telnetd** pouze pro uzly místní sítě (dané síťovou maskou), s výjimkou uzlu se jménem **web**, jehož klientům budou služby **ftp** a **telnetd** odmítnuty. Prohledávání obou souborů procesem **tcpd** pak začíná v **hosts.allow**, a teprve není-li služba nalezena, pokračuje se v **hosts.denny**. Ve druhém souboru na závěr může být použito klíčové slovo **ALL** takto:

```
ALL : ALL
```

což znamená, že všechny ostatní požadavky ze všech jiných uzlů, než byly v obou souborech vyjmenovány, jsou zamítnuty. Pokud tato direktiva není použita, je požadovaná služba pro vstupní požadavek poskytnuta. Prázdné soubory tedy znamenají poskytnutí síťové služby bez omezení. TCP Wrappers tak



umožňují správci definovat omezení síťových serverů podle jeho bezpečnostního návrhu místní sítě, jak jsme uvedli v úvodu článku.

Volně šířený je také software známý pod jménem **xinetd** (jako zdroj posledních verzí bývá citován uzel `mystique.cs.colorado.edu` dostupný anonymním `ftp`). Myšlenka je podobná jako u TCP Wrappers, ale systémové změny jsou hlubší. **xinetd** je nový proces, kterým je nahrazen dosavadní **inetd**. Výsledkem je sledovaný vstup síťových požadavků. Správce i zde omezuje vstup uživatelů z určitých míst sítě. Omezení lze zadávat v podobě jmen uzlů nebo adres IP (uzlů nebo celých sítí). Dále lze vyjmenovat uživatele, pouze kterým bude vstup do uzlu povolen. Tento bezpečnostní podsystém disponuje při instalaci programem **itox**, který převádí dosavadní soubor `/etc/inetd.conf` na `/etc/xinetd.conf`. Ten je pak novým procesem **xinetd** akceptován. Obsah souboru **xinetd.conf** je složitější a je dobré se orientovat v doprovodné dokumentaci.

Nebezpečná místa síťových serverů se liší případ od případu a správce uzlu by se měl každou novou poskytovanou síťovou službou zabývat z pohledu uvedených bezpečnostních hledisek. Uvedme si příklady síťových aplikací Internetu a jejich slabých míst.

Bezpečnost provozu příkazů **r** (**r**-commands) síťových aplikací systémů BSD, jejichž klienty jsou např. **rlogin**, **rcp** nebo **remsh**, je zeslabena při jejich nastavení pro tzv. bezpečné uzly (trusted hosts) či bezpečné uživatele (trusted users). Správce systému může v souboru `/etc/hosts.equiv` vyjmenovat jména uzlů (jméno uzlu vždy samostatně na řádku), odkud přichází požadavek klientu nebude podmíněn kontrolou jména uživatele a jeho hesla. Klient se prokazuje jménem uživatele vzdáleného uzlu, a pokud je toto jméno nalezeno v tabulce uživatelů `/etc/passwd`, uživatel je přihlášen, přestože heslo jeho sezení může být v uzlu klientu i serveru rozdílné. Pokud navíc je soubor `hosts.equiv` obohacen znakem `+`, je uživatel jména shodného s některým z `/etc/passwd` systémem akceptován z kteréhokoliv uzlu všech připojených sítí. Za bezpečný v uvedeném slova smyslu se nepovažuje žádný uzel pro uživatele **root**. Nastavení obsluhy klientů příkazů **r** může ovlivnit také každý uživatel evidovaný v uzlu serveru, a to tak, že naplní soubor `.rhosts` svého domovského adresáře seznamem jmen uzlů a uživatelů, kteří mohou používat sezení vlastníka souboru. Jde tedy o pokračování souboru obecné platnosti `/etc/hosts.equiv`. Např.

```
gagarin      petr  lenkak
```

může být obsahem souboru `.rhosts`. Klienty příkazů **r** z uzlu `gagarin`, které budou ve vlastnictví tamních uživatelů `petr` nebo `lenkak`, pak mohou používat sezení majitele takového souboru.

V případě použití `.rhosts` pak výjimka pro uživatele **root** neplatí. I privilegovaný uživatel může mít v domovském adresáři (`/`) umístěn tento soubor. Pokud vás ale zajímá bezpečí vašeho uzlu a dočteli jste kapitolu až sem, jistě to nikdy neuděláte. Naopak, pokud uzel disponuje servery příkazů **r**, rozumný správce pravidelně prochází domovské adresáře uživatelů a zvažuje obsah každého souboru `.rhosts`.

Příkazy **r** mají původ v systémech BSD, stejně jako démon tisku **lpd**, který je také používán jako síťový server pro vzdálený tisk. Uživatelé přihlášení v jiných uzlech mohou server **lpd** používat, jsou-li jejich uzly uvedeny v seznamu bezpečných uzlů v `/etc/hosts.equiv`, což v nás vzbudí nevoli. Protože si bezpečnostního rizika byli vědomi i programátoři **lpd**, seznam uzlů pro příjem požadavků vzdálených klientů lze také umísťovat do souboru `/etc/hosts.lpd`. Jeho formát je i zde textový soupis jmen uzlů oddělených znaky nového řádku.

Příkaz zjišťující informace o evidovaném uživateli je **finger** a uvedli jsme jej v kap. 5. **finger** je i síťovou aplikací. Jeho použití tedy překračuje místní uživatele a pokud v uzlu, který je připojen k síti, jako je Internet, tuto službu prostřednictvím serveru **fingerd** poskytujete, zveřejňujete tak informace o svých uživateli do celého světa. Samotný **finger** sice vyžaduje v parametru jméno uživatele následované znakem @ a identifikací uzlu v síti, ale při použití bez jména uživatele, např.

**\$ finger @gagarin.vic.cz**

poskytuje server **fingerd** seznam právě přihlášených uživatelů, což je nebezpečné. Další nedůvěru si **fingerd** vysloužil odhalením bezpečnostní díry, kdy pomocí dlouhého příkazového řádku bylo možné dosáhnout přepisu zásobníku procesu **fingerd**. Červík R.T. Morris z r. 1988 pak dokázal podstrčit procesu spuštění procesu shellu. Na bezpečném portu tedy běžel proces s neomezenými přístupovými právy. Verze síťové služby **finger**, které jsou staršího data, než je 5. listopad 1988, tuto díru již neobsahují, ale nedůvěra byla zaseta. **fingerd** většinou správci systémů odpojují.

Podobnou ztrátu důvěry si vysloužil i poštovní agent síťové pošty **sendmail**, který implementuje poštovní přenosový standard Internetu SMTP. Reputace služby **sendmail** byla navíc v historii ohrožena několikrát, naposledy odhalením zadních vrátek ladicího režimu serveru. Co víc, mnozí výrobci do nedávné doby stále prodávali své systémy s bezpečnostními dírami. Test přítomnosti bezpečnostních děr lze prověřit např. klientem **telnet** takto

**\$ telnet localhost smtp**

Trying 127.0.0.1...

Connected to localhost.vic.cz.

Escape character is '^['.

220-valerian.vic.cz Sendmail 950413.SGI.8.6.12/1.08 ready at Sat, 1

1 Jul 1998 19:57:03 GMT

220 ESMTP spoken here

**wiz**

500 Command unrecognized

**debug**

500 Command unrecognized

**kill**

500 Command unrecognized

**quit**

221 valerian.vic.cz closing connection

Connection closed by foreign host.

**\$**

V příkladu jsme oslovili server v aplikačním protokolu postupně příkazy **wiz** (možnost spuštění privilegovaného shellu), **debug** (vstup do režimu ladění a možnost získání neomezených přístupových práv) a **kill** (lze provést přepis systémových souborů, jako je např. `/etc/passwd`). Pokud bude reakce serveru jiná, než je v příkladu uvedena, tj. odmítavá, kontaktujte dodavatele operačního systému a požadujte novou verzi serveru (verze musí být nejméně 5.65 nebo vyšší).

Síťová služba přenosu souborů **ftp** (používejte verze od prosince roku 1988) disponuje možností tzv. anonymního **ftp** (anonymous **ftp**). Po jeho nastavení může do uzlu prostřednictvím služby **ftp**



vstoupit kdokoliv, nejen uživatel v uzlu evidovaný. Server **ftpd** při oslovení klientem vždy požaduje jméno uživatele, pod jehož právy přistupuje k datům uzlu (implicitně bývá také server nastaven na domovský adresář takového uživatele). Vstup takového uživatele je registrován jako vzdálené přihlášení v odpovídajících souborech (např. v souboru `/var/adm/wtmp`, podrobněji viz kap. 5). Pokud však klient použije namísto jména uživatele text `anonymous`, je namísto hesla požadován pouze řetězec další textové identifikace (jako je např. civilní jméno nebo poštovní adresa). Tento anonymní přístup není odmítnut tehdy, je-li zde evidován uživatel se jménem `ftp`. Jeho domovským adresářem pak začíná podstrom adresářů, který je k dispozici klientu, a to jak pro čtení, tak pro zápis. Tento domovský adresář server **ftpd** zobrazuje klientu pod jménem `/` a nelze jej překročit použitím odkazu `...`. Je věcí správce uzlu, zda použije další omezení pro práci anonymního **ftp**. Pokud totiž např. vytvoříme v domovském adresáři uživatele `ftp` podadresář `bin` a umístíme do něj kopii souboru `/bin/ls` (pod jménem opět `ls`), bude pak **ftpd** používat pro výpis obsahu adresáře tuto kopii. Rovněž tak jsou-li v podadresáři `etc` uloženy kopie souborů `/etc/passwd` a `/etc/group`, vlastnictví souborů při výpisu je pak zobrazováno podle těchto souborů (doporučujeme odstranit i šifrovaná hesla z těchto kopií a nahradit je např. znakem `*`). Konečně se běžně používá podadresář `pub` domovského adresáře uživatele `ftp` pro ukládání veřejně dostupných dat uzlu. Server **ftpd** anonymního klientu pracuje v sezení uživatele `ftp` (jako takový je tedy i registrován při vstupu do systému). Podstrom jeho domovského adresáře podléhá přístupovým právům, která nastaví správce uzlu. Je např. důležité ponechat podadresáře `bin` a `etc` ve vlastnictví uživatele `root` a zamezit jinému uživateli nejenom jejich přepis, ale třeba i čtení. Zdali umožníte uživateli `ftp` používat podadresář `pub` také pro zápis souborů odesílaných klienty, vychází z vašich potřeb. Minimální ochranu, kterou však doporučujeme, představuje omezení čerpání diskového prostoru uživatele `ftp`.

Obecně je server **ftpd** z bezpečnostních důvodů posílen také obsahem souboru `/etc/ftpusers`. Jeho obsahem je seznam uživatelů, kterým je zakázáno tuto síťovou službu používat. Každý uživatel je uveden samostatně na řádku tohoto souboru a znevýhodňují se uživatelé, jako je `root`, `uucp`, `bin`, `daemon` a další.

Přestože X Window System umožňuje uživateli transparentní práci v prostředí celé sítě, jeho koncepce (koneckonců z poloviny 80. let) je z hlediska bezpečnosti skutečný hazard. Server X je pouhý služebníků všech klientů X v různých uzlech. Přitom každý klient má možnosti takřka neomezené z pohledu direktiv protokolu X, kterými určuje práci serveru X. Je samozřejmě věcí návrhu celé sítě, zda bude server X provozován pouze na terminálech X, nebo i v uzlech s důležitým obsahem datové základny. Nic ovšem nezabrání uživateli vyzkoušet klientu X, který pod rouškou půvabného barevného obrázku bude monitorovat váš displej X. X navíc nemá vyhrazen žádný z bezpečných portů. Jako základní bezpečnostní vybavení, kterým prostředí X disponuje, je program **xhost**, pomocí něhož může privilegovaný uživatel stanovit jména uzlů, pouze kterým je možné síťovou aplikaci X poskytnout. Pro ostatní uzly je síťové spojení zamítnuto. Bez parametrů vypisuje **xhost** seznam povolených uzlů. Argument v příkazovém řádku je jméno dalšího uzlu, který je povolen. Pokud argument začíná znakem `+`, následující jméno patří uzlu, který naopak má být ze seznamu vyjmut. Použijete-li jako argument pouze znak `+`, zrušíte omezení a podsystém X je dostupný komukoliv.

X Window System je ovšem v současné době bezpečnostním mechanismem doplňován a o jeho koncepci (ale i současné implementaci) se lze dočíst v provozní literatuře ve verzi X11R6. Na stránce **Xsecurity** se lze seznámit s možnými způsoby ochrany. Návrh ochrany v X vychází z používaných

ochranných podsystémů, jako je využívání šifrovacího mechanismu DES (zde tzv. přístup typu XDM-AUTHORIZATION-1), který je podrobněji rozpracován v použitém bezpečném RPC v tzv. přístupu SUN-DES-1. Je ovšem možné používat i bezpečnostní podsystém na bázi Kerbera, tzv. přístup MIT-KERBEROS-5. Nejslabší způsob ochrany, tzv. MIT-MAGIC-COOKIE-1, je pak postaven na prověřování tajného lístku o velikosti 128 bitů. Lístek je generován podsystémem X a sítí je poslán nijak nešifrován. V silnějším přístupu XDM-AUTHORIZATION-1 je princip podobný, ale lístky jsou posílány sítí šifrovány. Šifrovací mechanismus je typu DES. Klientem X odeslaný lístek přitom obsahuje jak šifrovací klíč, tak část identifikace, tj. údaj současného data a času, u spojení TCP/IP pak adresu IP, číslo portu a u místního spojení PID a klíč komunikace procesů. Výsledný paket (o velikosti 192 bitů) je šifrován a odeslán serveru X, který z informací po fázi rozšifrování může prověřit oprávnění procesu klientu X pracovat za podpory osloveného serveru X. Podsystém Kerberos i bezpečný RPC bude předmětem odst. 9.4.3. I zde se pracuje s tzv. bezpečnostními lístky. Bezpečnostní lístek kteréhokoliv používaného bezpečnostního přístupu k X pak je v X11R6 (s omezenou platností) uložen v souboru `.xauth`, jehož obsah je viditelný příkazem `xauth`.

V případě podsystému X je ovšem v dnešní době při zajišťování bezpečnosti stále ještě doporučován i jiný, obecný způsob zajištění bezpečnosti, jako jsou ochranné zdi nebo omezení daná každým správcem systému v rámci koncepce sítě.

Procesy **httpd**, které zajišťují přístup k datům zveřejňovaným jako stránky WWW, nepřinášejí žádné zvláštní nebezpečí, pokud tvůrce stránek nezačne používat prostředky interaktivní komunikace klientu (např. **netscape**) a **httpd**. Interaktivní komunikace je totiž zajišťována pomocí formulářů, jejichž obsahovou stránku (tj. akce, které se s daty odeslanými klientem provedou, jako je např. jejich zápis do databáze) zajišťuje další proces, který je dítětem **httpd**. Takové programy, které jsou nazývány scénáře CGI (CGI scripts, Common Gateway Interface scripts), jsou programy v UNIXu běžně spustitelné a jejich chování se přitom odvíjí od požadavků klientu, který v argumentech příkazového řádku nebo na standardním vstupu tokem dat formuluje své požadavky. Scénáře CGI, umístěvané v adresáři `cgi-bin` výchozího adresáře serverů **httpd**, přitom navrhují často programátoři, kteří mnoho o bezpečnosti neví. Přesto často požadují nastavení s-bitu. Jejich programovací jazyky jsou přitom především některý shell nebo jiné interprety (např. **perl**). Úkolem správce systému (nebo správce bezpečnosti, chcete-li) je pak před samotným nastavením s-bitu pečlivě pročíst program, jehož chování ovlivňuje kdokoli ze světa, a odstranit v něm zřejmá nebezpečí, jako jsou zadní vrátka nebo spouštění dalších procesů shellu. Taková práce ovšem nemusí být nijak lehká, protože scénářů CGI mohou být desítky i stovky. Správce systému se tak stává odpovědným za síťové servery, které doposud nebyly vyzkoušeny ve výzkumných laboratořích, veřejných sítích a které programovali většinou nadšení amatéři. Není tedy divu, že jsou scénáře CGI hlavním předmětem útoku hackerů, ale i jiných uživatelů Internetu, kteří se pokoušejí průnikem získat další neveřejné informace, jež jsou často také obsahem datové základny stránek WWW (např. v rámci Intranetu). V případě serverů WWW je doporučení jednoznačné. Pro zajištění funkce WWW používat pouze vyhrazený uzel místní sítě bez přítomnosti dalších síťových služeb, a to jak klientů, tak serverů, zbytek sítě pak pečlivě vůči takovému uzlu ochránit uváděnými prostředky jednotlivých síťových služeb, doplňované databáze pak cyklicky archivovat a jejich další zpracování provádět odděleně v jiných uzlech. Vysvobození nepřináší ani programování v jazyce JAVA, protože jeho bezpečnostní předpoklady zatím nebyly definovány a současné implementace žádným jednotným bezpečnostním mechanismem nedisponují.

Vzhledem k tomu, že aplikace síťového systému souborů NFS (nebo RFS) a její doplněk NIS jsou programovány v RPC, jejich bezpečnost je úzce spojena s bezpečností samotného RPC. O takový bezpečnostní mechanismus je skutečně RPC posílen. Mluví se pak o bezpečném RPC (secure RPC) a vzhledem k tomu, že jde o analogii bezpečnostního podsystému Kerberos, uvedeme si základní chování bezpečného RPC jako součást následujícího odstavce. Pro bezpečný export diskových periférií do sítě prostřednictvím serverů **nfsd** a **biod** si alespoň uvedme základní pravidla, která musí správce systému dodržovat. Do sítě nabízejme co nejmenší počet stromů systému souborů a vždy v jejich seznamu v souboru `/etc/exports` uvádějme jména uzlů, pro které je export prováděn. Připojení nabízených svazků je jedna z prvních akcí, o kterou se hacker při zájmu o data uzlu pokusí. Data, která nemusí být exportována s možností přepisu, exportujte opět v uvedeném souboru pouze pro čtení. Dbejte na jediné UID uživatelů v uzlech, kam jsou svazky exportovány. Pokud je síť větší, používejte pro sjednocení službu NIS. Do sítě exportujte soubory s proveditelnými programy pouze pro čtení. Pokud váš systém disponuje programem **fsirand**, používejte jej. Tento program provede náhodné rozložení i-uzlů v jejich oblasti. Pravděpodobnost průniku do systému, kdy hacker odhaduje důležitost obsahu i-uzlů podle jejich malých čísel, se tak sníží. V distribuci NFS hledejte program obdobného významu a postupujte podle provozní dokumentace. Při používání síťové služby NIS pečlivě rozvažte umístění systémových souborů pro servery NIS. Po nastavení pak znovu prohledejte obsah souborů `/etc/passwd` a `/etc/group` systémů se servery, zda neobsahují znaky + v prvním sloupci na místě registrace uživatele. Řádek souboru `/etc/passwd`

```
+:0:0:::
```

je zcela nepřijatelný jak v uzlu serveru, tak klientu! Nejlepší je vždy koncipovat místní síť pouze s jedním uzlem, který nabízí diskové periferie do sítě, získáte tak větší přehled. Jako test bezpečného serveru NIS vyzkoušejte, zda se vám nepodaří přihlásit se jako uživatel se jménem +. Pokud se tak stane, konfigurovali jste NIS špatně. Pečlivě také stanovujte síťové skupiny (odkazová jména) v souboru `/etc/netgroup`, protože tak omezujete uživatelům z různých uzlů přístup k informacím serverů.

### 9.4.3 Kontrola autenticity, Kerberos, bezpečný RPC

Jedním ze základních problémů bezpečnosti sítí v UNIXu je způsob prověřování autenticity uživatele. Ať už jde o mechanismus přihlašování nebo jinou situaci požádání serveru o heslo, vždy je odpovědí klientu textový řetězec, jehož správnost analyzuje teprve proces serveru ve vzdáleném systému (šifrovacími algoritmy a porovnáním se šifrovaným heslem v uzlu). Cesta od klientu k serveru může být poměrně daleká. Její součástí je průchod různými uzly různých sítí. Celou cestu je přitom textový řetězec hesla umístěn v síťovém paketu v podobě, jaké jej klient odeslal, v nechráněných sítích tedy v nešifrované podobě. Tuto bezpečnostní díru lze zacetit jenom velmi obtížně. Správci uzlů znali tohoto problému tedy zásadně nepracují v sezení uživatele `root` ze vzdálených sítí, avšak omezit vzdálené prokazování se heslem znamená odstavit nejenom možnost vzdáleného přihlašování, ale i další síťové služby. Pro běžný provoz síťových služeb je to prakticky nemožné.

V rámci projektu Athena v M.I.T. byl v 80. letech vyvinut bezpečnostní systém Kerberos<sup>4</sup>. Jeho mateřským operačním systémem byl 4.3BSD. Vychází se zde opět z pojetí sítě jako celku. Uživatel se přihlašuje v určitém uzlu sítě, ale jeho autenticita je v tomto okamžiku stanovena i pro jeho pohyb sítí, tj. využívání služeb vzdálených síťových serverů. Jde o mechanismus, kdy proces **login** v okamžiku

přihlašování uživatele požádá některý smluvený uzel sítě o získání takové autenticity. Pokud přihlášení uživatele proběhne úspěšně, při využívání vzdálených serverů se každý klient uživatele na základě své autenticity opět obrací na službu poskytování autenticity, která mu poskytne určitý šifrovaný text jako průkaz pro práci se serverem. Server při kontaktu s klientem obdrží takový průkaz a o jeho prověření požádá uzel, který tento průkaz vydal. Uzel, který takto určuje autenticitu klientů a jejich oprávněnost používat odpovídající servery, je považován za bezpečný a spolehlivý. Takový uzel neposkytuje žádné jiné síťové služby a musí být dostatečně zajištěn před nežádoucím průnikem. Síťové servery, které zde pracují, tvoří základ bezpečnostního systému Kerberos. Pro svoji práci vystavování a prověřování autenticity pak používají bezpečnostní databázi evidovaných uživatelů sítě v uzlech sítě i jejich hesla a dále evidovaných síťových služeb sítě v uzlech sítě a rovněž jejich hesla. Práce Kerbera je založena především na dvou hlavních serverech. Server pro kontrolu autentičnosti, tj. proces, který na požádání procesu **login** vzdáleného uzlu prověřuje uživatele, a server TGS (**T**icket **G**ranting **S**erver, server zajišťující lístky), který vydává tzv. *lístek* (ticket), jehož správnost opět prověřuje zase on sám. Lístky jsou pak základním komunikačním prostředkem prověřování oprávněnosti mezi klientem a serverem. Pro získání spolupráce s určitým serverem jsou ale lístky klientu poskytnuty pouze na velmi krátkou časově omezenou dobu, a to z důvodu možnosti jejich zcizení. Při používání Kerbera je proto naprosto nezbytná dobrá časová synchronizace uzlů takto zabezpečené sítě. Základní myšlenky Kerbera tedy jsou nikdy neposílat hesla sítí nešifrovaná a jediné místo, kde jsou hesla jak uživatelů tak služeb uložena, je databáze Kerbera. Důležitý je také tichý chod systému, tj. uživatel nesmí být zbytečně obtěžován neustálými požadavky na zadávání hesla.

Práci systému Kerberos ukazují obrázky 9.5 a 9.6. Na obr. 9.5 je uvedena základní kontrola autenticity při vstupu uživatele do systému (nebo sítě, chcete-li). Uživatel zadává své jméno. Proces **login** aktivuje síťové spojení se serverem Kerberos pro kontrolu autentičnosti a posílá mu toto jméno uživatele. Kerberos generuje pro sezení uživatele jednorázový klíč, který zůstává v uzlu Kerbera. Na základě jména uživatele a jeho hesla pak vytvoří úvodní lístek, který zajistí opětovný přístup uživatele k serveru TGS. Lístek obsahuje jméno uživatele, jméno TGS, síťovou adresu uzlu uživatele a klíč sezení na serveru TGS. Lístek je *zapečetěný* (sealed), tj. šifrovaný jednorázovým klíčem sezení uživatele na TGS, a procesu **login** je odeslán v síťové zprávě, ve které je kromě tohoto zapečetěného lístku ještě nešifrovaně uveden opět klíč sezení na serveru TGS. Tato zpráva je ale ještě před odesláním opět šifrována, zde heslem uživatele. Proces **login** převezme ze sítě zprávu a požádá uživatele o jeho heslo. Heslo proběhne běžným šifrovacím mechanismem a pomocí výsledku se rozšiřuje zpráva přijatá od Kerbera. Textová podoba uživatelem zapsaného hesla se zapomene a uživateli pro využívání síťových služeb zůstává zapečetěný lístek a klíč jeho sezení na TGS. Způsob odpečetění lístku přitom zná pouze Kerberos. Zapečetěný lístek je pochopitelně nutné dobře utajit, ale implementace je často ukládají v oblasti /tmp místních systémů, kde hrozí možnost jejich zneužití. Tyto lístky mají také omezenou platnost, přestože uživatel již v dalším trvání svého sezení není nucen zadávat své heslo. Je běžné, že platnost jeho sezení TGS vyprší cca do 8 hodin. Pokračovat lze novým přihlášením nebo příkazem **kinit**, kdy se vytvoří nové sezení TGS tak, jak bylo právě popsáno.

Obr. 9.6 ukazuje práci síťových klientů a kontrolu jejich oprávněnosti síťovými servery. Klient nejprve požádá server Kerberos TGS o získání lístku pro přístup k serveru. Kerberu se přitom identifikuje svým přiděleným zapečetěným lístkem a dále určuje jméno serveru. Součástí takové zprávy pro Kerberos je ještě prvek autentičnosti, který se skládá ze jména uživatele, síťové adresy uzlu klientu a údaje

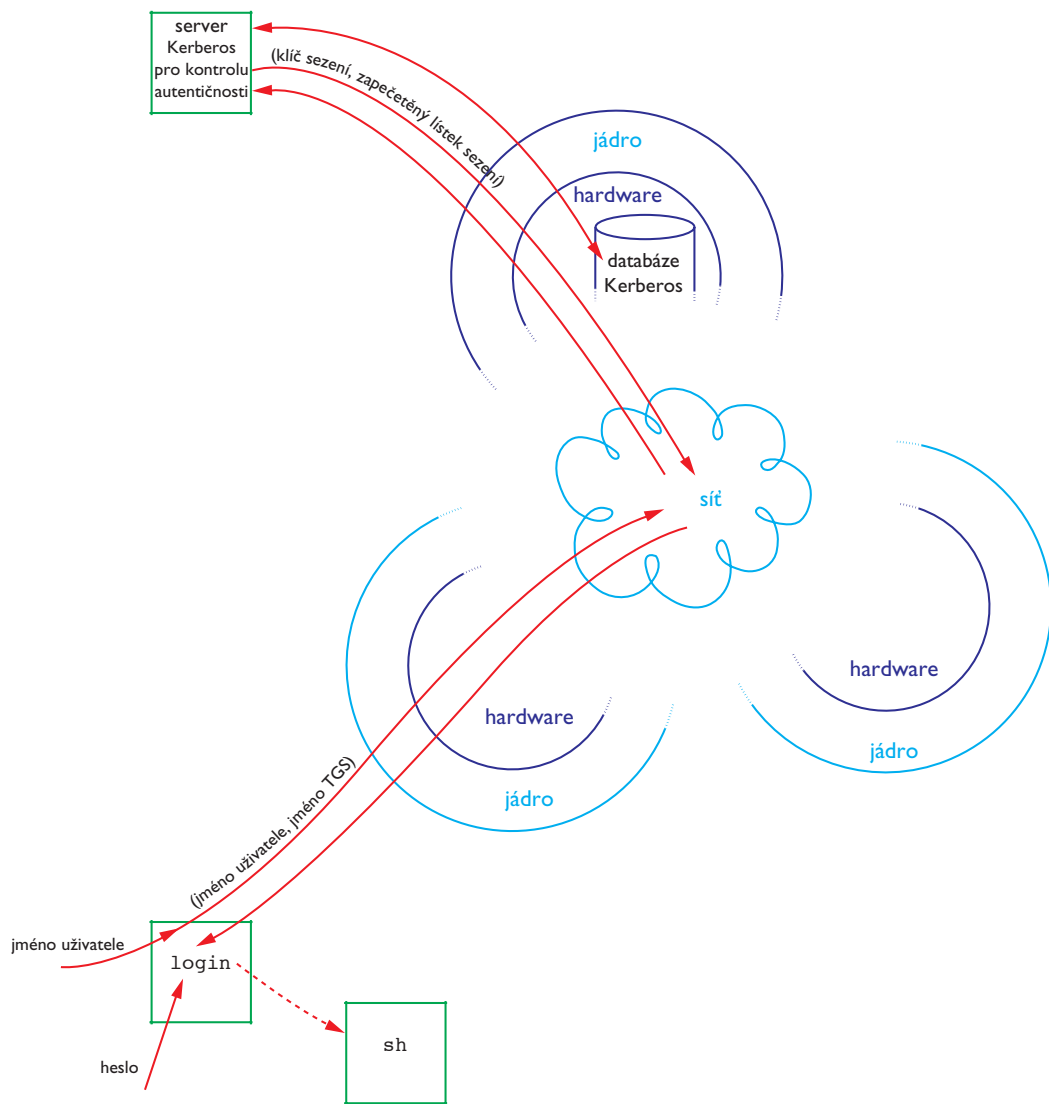
o současném čase. Prvek autentičnosti je zapečetěn klíčem relace TGS a zpráva je odeslána serveru TGS. Server TGS rozšifruje zapečetěný lístek, z něj získá klíč pro označení relace a rozšifruje jím prvek autentičnosti. Kerberos pak kontroluje na shodu všechno, co se dá, tj. jméno uživatele, síťovou adresu lístku atd. Nakonec také porovná časový údaj z prvku autentičnosti s aktuálním časem. Pokud všechno souhlasí, generuje nový klíč pro určení relace klientu a serveru a vytvoří lístek jejich komunikace, který obsahuje jméno uživatele, jméno požadovaného serveru, síťovou adresu uzlu uživatele a klíč relace (nový klíč sezení TGS). Zprávu, kterou šifrovaně (pomocí výchozího klíče sezení uživatele na TGS) posílá klientu, pak vytvoří z nového klíče TGS a zapečetěného lístku pro komunikaci se serverem. Klient rozšifruje zprávu, vytvoří prvek autentičnosti (jméno uživatele, síťová adresa uzlu, současný čas), zapečetí jej pomocí klíče relace se serverem a pošle zprávu serveru. Zpráva serveru obsahuje tytéž položky jako zpráva posílaná serveru TGS (zapečetěný lístek, zapečetěný prvek autentičnosti a jméno serveru). Zpráva není šifrována (jsou šifrovány její položky), takže server se v jejím obsahu vyzná. Server zná svůj šifrovací klíč (tedy zná jej on a Kerberos). Provede rozšifrování a ověří obsah zprávy, stejně jako to učinil klient, když o spojení Kerbera žádal.

Kerberos není jednoduchá cesta získání bezpečné sítě, ale určitě se vyplatí. Případný narušitel totiž musí znát dokonale samotný Kerberos, musí zcizit několik lístků současně, a to ještě v omezeně krátkém časovém intervalu.

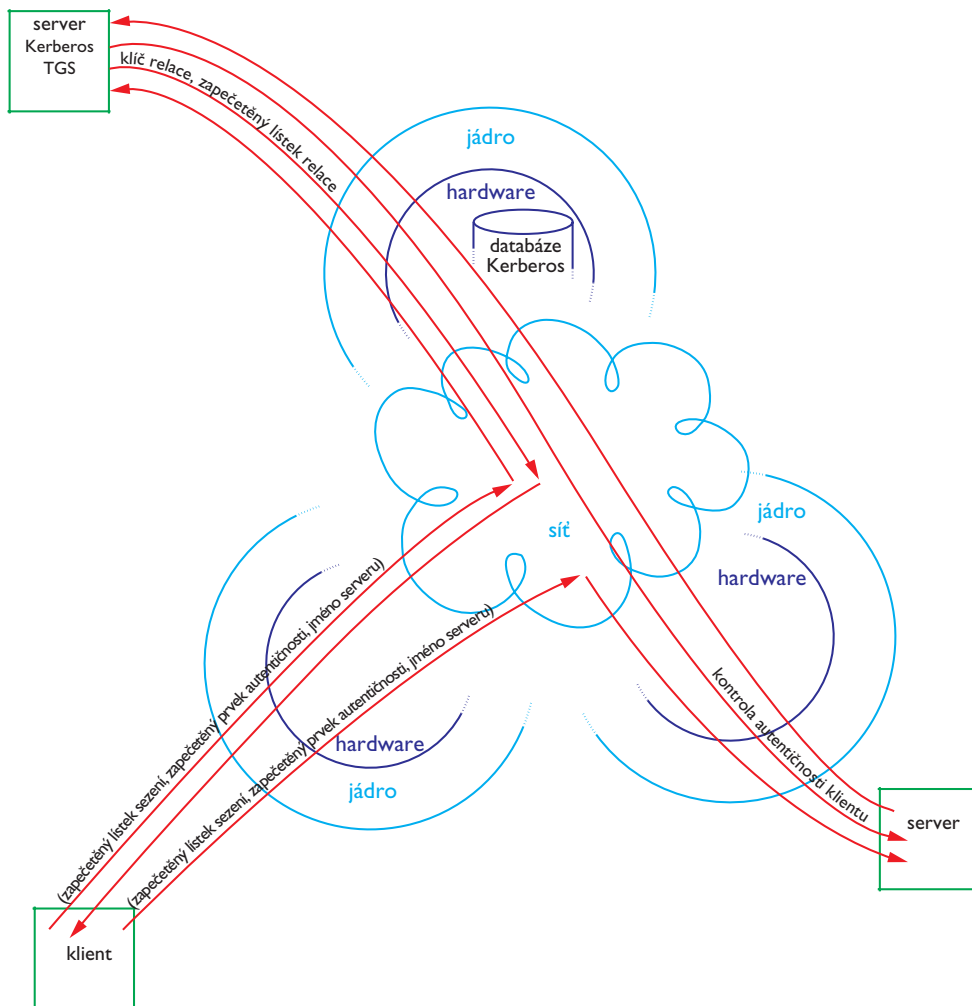
Podobné prostředky pro zajištění bezpečné práce uživatele v síti jako Kerberos poskytuje také *bezpečný RPC* (secure RPC). Bezpečnost do síťového programovacího jazyka RPC byla implementována firmou Sun Microsystems v polovině 80. let jako součást systému SunOS4.0. Na rozdíl od Kerbera však bezpečný RPC nevyžaduje zvláštní třetí uzel prověřování autenticity. Pro distribuci tajných klíčů pro uživatele a síťové služby totiž používá distribuovanou informační databázi NIS. Jeho výhoda je také v posílení základního RPC. Síťové aplikace programované v RPC mají navíc zajištěnu kompatibilitu podle definovaných standardů. Při používání Kerbera je nutné obohatit síťové aplikace o nezbytnou komunikaci prostřednictvím nových funkcí. Reprezentativní síťová aplikace, která byla v bezpečném RPC uvedena na trh, je *bezpečný NFS* (secure NFS), bezpečný způsob sdílení diskových svazků.

Pověření požadavku klientu serverem je provedeno kontrolou identifikace uživatele vzhledem k jeho heslu. Server od klientu jako potvrzení své autenticity získává tzv. *konverzační klíč* (conversation key). Uzel s klientem jej generoval jako náhodný 56 bitový klíč a šifroval jej pomocí *klíče sezení* (session key). Klíč sezení přitom klient odvodil od uživatele *tajného klíče* (user's secret key) a *veřejného klíče* serveru (server's public key). Veřejné a tajné klíče uživatelů jsou přitom součástí databáze NIS. Uživatel je v databázi identifikován síťově. Obvykle má formát odvozený podle pravidla *uživatel.doména* (dříve *UID.UNIX@doména*). Tajný klíč je tajný, protože je uložen jako šifrovaný, a to heslem uživatele *sezení*. Server rozpozná, zda je uživatel oprávněn jej používat postupným rozšifrováním. Je přitom zřejmé, že paket od klientu byl šifrován konverzačním klíčem. Klient jej přitom může získat pouze generací z veřejného klíče serveru a uživatele *tajného klíče*. Ke znalosti uživatele *tajného klíče* však musí mít klient oprávnění vstupu do databáze NIS a rozšifrovat jej pomocí hesla uživatele *sezení*.

Podobně jako Kerberos má i bezpečný RPC časové omezení platnosti paketů s konverzačními klíči. Platný časový interval je ale součástí jádra místního operačního systému, proto je jeho uzpůsobení pro síťovou bezpečnost nutné provádět regenerací jádra (viz kap. 10).



Obr. 9.5 Kerberos – vstup uživatele do sítě



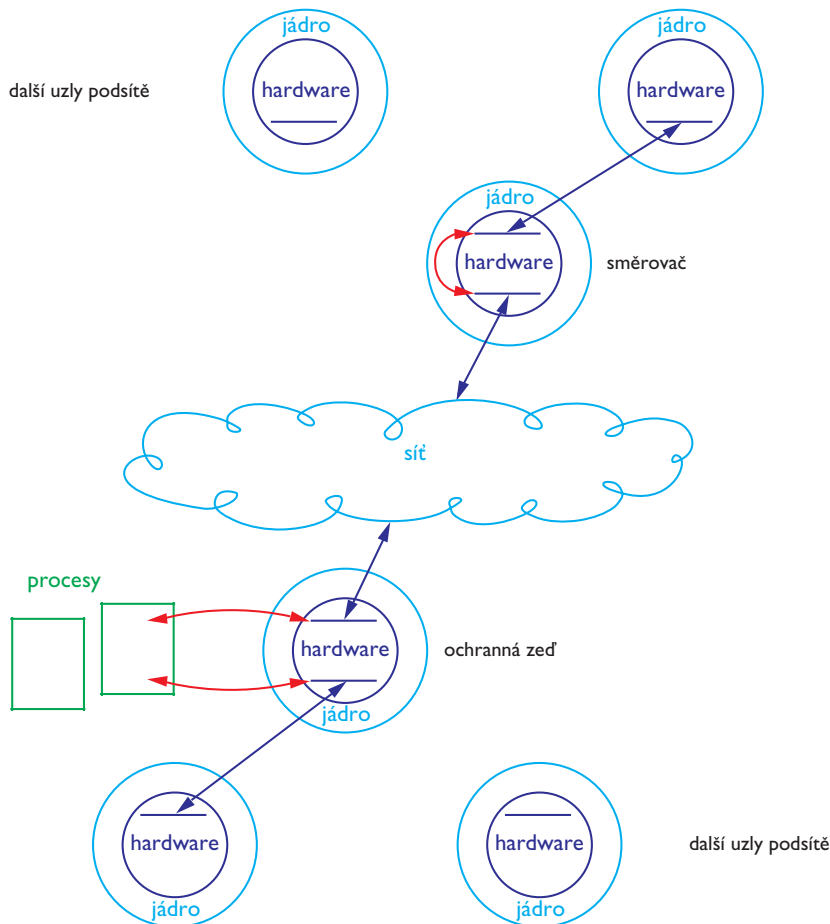
Obr. 9.6 Kerberos – práce uživatele v síti

#### 9.4.4 Ochranné zdi (Firewalls)

I s odkazem na úvod čl. 9.4. lze jistě souhlasit s názorem, že úplně bezpečný uzel z pohledu síťových služeb je ten, který k síti není vůbec připojen. To je jistě v přímém rozporu s vývojem a s požadavky uživatelů. V úvodní části článku jsme ukázali dvě strany síťové komunikace v UNIXu jako stranu klientu a stranu serveru. Obě přinášejí nebezpečné situace, které je nutné akceptovat při plánování síťového provozu. Uvedené metody posilování bezpečnosti dat síťových uzlů se doposud vztahovaly



výhradně k aplikační vrstvě sítě. Technologie *ochranné zdi* (firewall) sestupuje na úroveň nižší, a to síťovou. Síť IP (viz kap. 7) umožňuje oslovit libovolný uzel její libovolné podsítě. Znamená to, že každý uzel může komunikovat s každým, tj. komunikující klient a server mohou být principiálně účastníky libovolných uzlů sítě IP. Směrování paketů je zajišťováno směrovači (routers), což jsou plnohodnotné uzly sítě IP, jejichž software plní navíc úlohu propouštění paketů daným směrem. Pomocí směrování jsou přitom propojeny jednotlivé podsítě vzájemně mezi sebou. Ochranná zeď je software, jehož umístění v síti IP je podobné jako u směrování. Je instalován v uzlu, který spojuje podsít' se zbytkem sítě IP. Pakety IP ale nesměruje, síť IP v uzlu s jeho instalací vlastně končí. Všechny pakety, které tento uzel obdrží, jsou totiž zpracovány aplikační vrstvou a do podsítě jsou následně odeslány opět vrstvou IP



Obr. 9.7 Směrovač a ochranná zeď

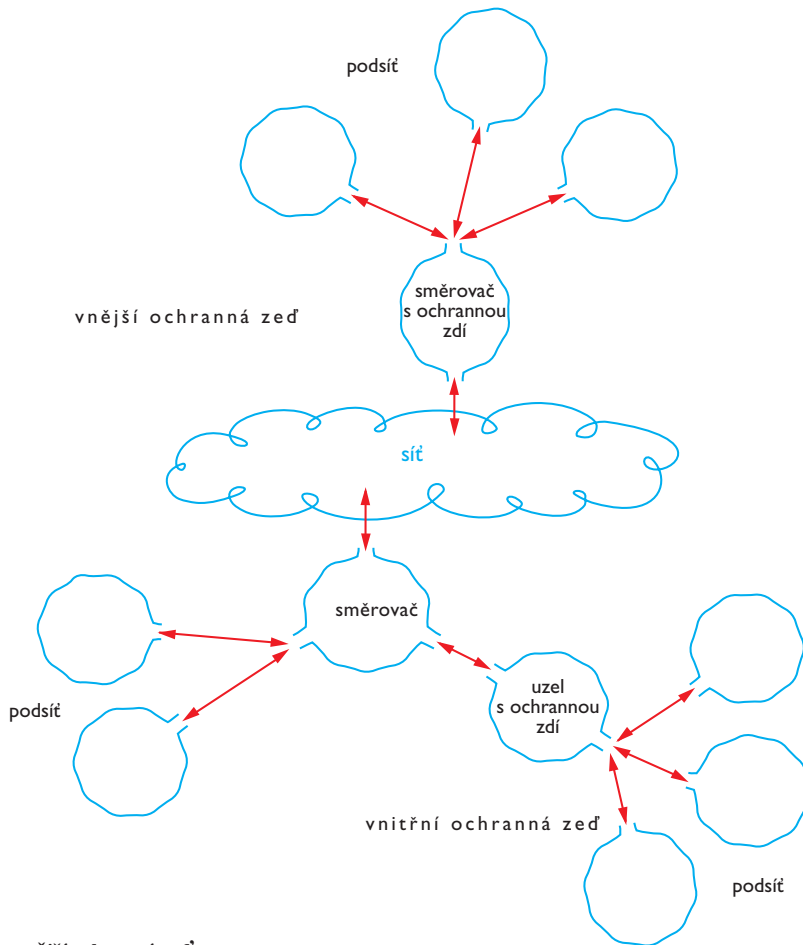


jenom některé. Pakety, o kterých je rozhodnuto nepropouštět je do podsítě, jsou odmítnuty, pakety, které nejsou určeny pro podsít, jsou ignorovány. Vzhledem k tomu, že z vnějšího pohledu ochranná zeď některé prověřené pakety propouští, je často používán termín bezpečný směrovač nebo bezpečná brána. Rozdíl mezi pouhým směrovačem a ochrannou zdí vidíme na obr. 9.7.

Pakety IP i směrovače jsou pouze předány na úrovni síťové vrstvy odpovídajícímu uzlu podsítě.

Ochranná zeď plní funkci kontroly obsahu procházejících paketů, o kterých rozhoduje, zda budou do podsítě (nebo z podsítě ven) odeslány až na úrovni aplikační.

Vzhledem k tomu, že použití ochranné zdi je opatření, které výrazně omezuje přístup k místní síti, ale také pohyb uživatelů místní sítě v ostatních sítích, odděluje se často pomocí ní pouze určitá část sítě



Obr. 9.8 Vnitřní a vnější ochranná zeď

(z pohledu sítě IP opět podsít), která zahrnuje důvěrná interní data firmy. Zbytek místní sítě je pak dostupný prostřednictvím směrovače klasickým způsobem. Hovoříme o vnitřním použití ochranné zdi, tj. o interní ochranné zdi (internal firewall) na rozdíl od vnějšího použití (external firewall), kdy je od sítě IP takovou externí ochrannou zdí oddělena celá místní síť IP. Obě použití ukazují obr. 9.8.

Zdánlivá jednoduchost ochranné zdi je pouze v její myšlence. Uvědomíme-li si, co všechno musí aplikační vrstva zajistit, ztratíme odvalu něco takového programovat ve vlastní režii. Vrstva procesová musí totiž např. zajistit viditelnost jmen uzlů do celého světa. Některé uzly místní sítě jsou ze světa neviditelné, ale v místní síti viditelné jsou. Elektronická pošta musí být rozesílána podle požadované konfigurace místní sítě. Jména poštovních schránek pro uživatele místní sítě se liší od jejich přihlašovacích jmen a ochranná zeď musí zajistit odesílání a příjem pošty tak, že se sama tváří jako agent transportu. Jména uživatelů místní sítě jsou přes ochrannou zeď nedostupná. Vzdálené přihlašování (**telnet**, **rlogin**) nebo prokazování identifikace u služby **ftp** proběhne vždy ve dvou fázích. Uživatel se nejprve přihlašuje do ochranné zdi a teprve potom do vzdáleného uzlu. Přenos souborů proto probíhá s dočasným uložením v ochranné zdi, nebo lépe řečeno, každý přenos dat, ať už jako obsah souboru nebo interaktivní komunikace, prochází aplikační vrstvou v ochranné zdi a lze jej proto kontrolovat. K prokazování identifikace pro účely takové kontroly se často v ochranné zdi vytváří jeden uživatel, jehož sezení využívá celá skupina uživatelů místní sítě. Není to zcela bezpečné, a pokud to lze, měl by každý uživatel mít vlastní sezení v ochranné zdi.

Používání ochranných zdí se doporučuje v přísně střeženém prostředí, a to vnitřně. Jisté je, že každá ochranná zeď musí mít vlastního správce bezpečnosti, který ochrannou zeď navrhuje a sleduje. Dnes jsou na trhu k dispozici různá řešení ochranných zdí od různých výrobců UNIXu. Velmi často jsou závislé na konfiguraci hardwaru, je proto dobré zajímat se u výrobce konkrétního UNIXu, který používáte v místní síti.

## 9.5 Šifrování, fyzická bezpečnost

Z toho toho, co bylo v této kapitole prozatím napsáno, vyplývá, že bezpečnost dat uživatele v UNIXu není příliš růžová (ačkoli Oranžová je dozajista). Zajištění většího bezpečí dat navíc stojí provozovatele uzlu více peněz a uživatele pak nepříjemná omezení. Důvěra v soukromí vlastních dat je přitom pro uživatele jeden ze základních předpokladů, se kterými začíná s počítačem pracovat. Teprve po čase zjišťuje, jak snadno si jeho data může přečíst třetí strana. Pokud mu přitom správce výpočetního systému vysvětlí, jak obtížné je zajistit dobrou bezpečnost systémově, uvědomí si, že se musí postarat o svá data sám. UNIX mu přitom vychází vstříc, obsahuje totiž možnost dobrého šifrování dat.

První šifrovací (nebo kryptografický, chcete-li) program v UNIXu, je známý **crypt**, jehož popis a použití jsme už uvedli v čl. 5.3. Od dob druhé světové války, kdy byl algoritmus německého šifrovacího stroje Enigma (Enigma encryption machine) používán v prostředku **crypt** vymyšlen, se však vývoj v šifrování posunul výrazně kupředu. Přestože jsou dobré šifrovací algoritmy vždy pod dohledem odpovídajícího ministerstva obrany, i do veřejných implementací se časem dostávají programy kvalitního šifrování dat. Ty nejlepší jsou obvykle na příkaz vlády (Spojených států, které jsou stále určující země vývoje softwaru) záměrně paralyzovány nebo je vládou vydán výnos, na základě kterého je nelze šířit při distribuci operačních systémů za hranice země. Dnes se takovému zájmu těší zejména šifrovací algoritmy s označením RC2, RC4 a RC5 (výrobce a vlastníkem prozatím nezveřejněného algoritmu je

firma RSA Data Security). Při vývozu je jejich implementace omezována na kratší délku šifrovacího klíče. V UNIXu je ale hodně používaný také algoritmus DES (Data Encryption Standard, v r. 1970 byl vyvinut firmou IBM a National Institute of Standards and Technology) nebo jeho silnější bráška 3-DES s delším šifrovacím klíčem. V systémech UNIX (ale ne v SVID ani POSIXu) jej lze používat pomocí příkazu **des**. Volbou **-e** šifrujeme a **-d** získáváme původní čitelný text, např.

```
$ des -e -k r4$a6g -f soubor sifsoubor
```

použijeme klíč **r4\$a6g** pro šifrování obsahu souboru **soubor**. Šifrovaná data budou uložena do souboru **sifsoubor**.

Při použití ochrany dat jejich šifrováním jsou pak důležité dvě složky celého procesu. Jednak je to kvalita odpovídajícího algoritmu a jednak šifrovací klíč, který uživatel používá. Oba uvedené programy **crypt** i **des** přitom pracují se stejným klíčem jak při šifrování, tak i při zpětném získání dříve šifrovaných dat. Jiný uživatel pak data přečte za znalosti takového klíče bez problémů. V případě předávání dat mezi různými právními subjekty počítačovou sítí je ale taková ochrana dat nedostatečná. Pokud obě komunikující strany požívají vzájemné důvěry a jejich společný šifrovací klíč nevyzradí třetí straně, je bezpečí přenášených dat takto sice výrazně posíleno, ale pokud je potřeba jednoznačně prokázat svoji totožnost, a to i v případné soudní pře mezi oběma subjekty, takový mechanismus nedostačuje. Proto se používají šifrovací algoritmy se dvěma klíči, jeden je tzv. *tajný* a používá jej vlastník dat při šifrování, a druhý je tzv. *veřejný*, pomocí kterého je možné data pouze rozšifrovat. Veřejný klíč vlastník tajného klíče zveřejní (např. na svých stránkách WWW). Jím odeslaná pošta nebo jinak poskytované dokumenty sítí je možné rozšifrovat veřejným klíčem, ale šifrovat data může pouze majitel klíče tajného. Velmi používaný algoritmus s veřejným a tajným klíčem je např. RSA (Rivest Shamir Adleman) vyvinutý v MIT. Identifikace uživatele počítačové sítě tak může být jednoznačně potvrzena jeho elektronickým podpisem. Samotný text dat nebo zprávy nemusí být šifrovány, je k nim pouze přiložen šifrovaný podpis, který lze veřejně rozpoznat na základě znalosti veřejného klíče. Elektronický podpis autor textu přitom vytváří zadáním svého soukromého klíče a podepisovaných dat. Podepisovaná data jsou uvažována jako jeden z podkladů pro generaci šifrovaného podpisu. Obvykle se z podepisovaných dat vytváří tzv. kontrolní součet, který je pak použit jako jedna složka při šifrování podpisu. Elektronický podpis je tedy úzce s daty spojen, ale veřejným klíčem je možné jej při čtení textu kdykoliv potvrdit. Změna jediného bitu v podepsaných datech znamená zánik shody s šifrovaným podpisem. DSA (Digital Signature Algorithm) je jeden z nejpoužívanějších, byl vyvinut státní správou Spojených států. Pro účely elektronického podpisu se ale také používá již zmíněný RSA. Čtenáře, který se o šifrování a elektronickou autenticitu zajímá více, pak odkazují na [BerMacHan97].

Přestože je řada algoritmů pro šifrování s veřejným a tajným klíčem nebo pro elektronický podpis v různých verzích implementována, ani POSIX ani SVID je prozatím nedefinují. Mnohé lze získat prostřednictvím školních nebo veřejných uzlů sítě Internet, ale ty kvalitní jsou obvykle touto metodou nedostupné (viz výše). V zemi, kde se rozhodne určitá organizace, firma nebo banka algoritmy šifrování a elektronického podpisu používat, se musí ještě předtím zajímat o právní podklad elektronického potvrzení autenticity (tzv. neodmítnutelnost zodpovědnosti). Pokud právní řád termíny, jako je elektronický podpis, důvěryhodnost veřejného a tajného klíče atp., nedefinuje (což je případ i České republiky), elektronická komunikace veřejnými datovými sítěmi pak nemá v obchodním styku praktický smysl.

Smysl pak má jen zajišťování bezpečnost elektronického přenosu dat soukromými prostředky. To je cesta velmi složitá a velmi nákladná. Složitost je dána především nutností fyzicky zajistit cestu mezi různými uzly sítě vzájemně komunikujících subjektů veřejnými prostory nebo prostory, které jsou v soukromých rukou třetí strany. Zákony na vlastnictví veřejných médií (jako je např. vzduch) soukromými firmami jsou pak v konfliktu se zájmy zabezpečení přenosu dat. Zajištění bezpečné cesty veřejnými datovými spoji pak nutně musí ze zákona vyplývat pro provozovatele datových spojů, ale mnohdy tomu tak není a cesta k zajištění bezpečných spojů např. telekomunikační společností je podle právního řádu velmi složitá a mnohdy pro požadující firmu nebo organizaci finančně neproveditelná (zvláště v chudé zemi). Tyto úvahy, neboť jinak předchozích pár vět chápat nelze, úzce souvisí s fyzickou bezpečností dat uchovávaných v počítačích. Do dnešního dne byla napsána řada dokumentů a knih o zajištění počítačů a jejich periférií proti krádežím nebo vzájemnému odpojení či zneužití prostřednictvím startovacího média s přineseným operačním systémem atp. Nebývá opomenuto ani nebezpečí prachu, kouře či stárnutí materiálů, ze kterých jsou počítačové sestavy vyrobeny. Tato problematika je ale mimo rámec našeho textu. Zájemce lze pak odkázat na publikace, jako je např. [GarfSpaf94], a na zdravý rozum.

<sup>1</sup> Evropská snaha o definici bezpečnosti dat začíná rokem 1989, kdy vzniká dokument ZSIEC německého Federálního úřadu pro bezpečnost v informační technologii (BSI), který principálně vychází z TCSEC, ale označování úrovní má odlišné (viz čl. 9.2). V roce 1990 pak vzniká společný dokument zemí Francie, Německa, Nizozemí a Velké Británie s názvem ITSEC (Information Technology Security Evaluation Criteria, Kritéria hodnocení bezpečnosti informační technologie, tzv. Bílá kniha, viz [ITSEC90]), který navazuje na TCSEC a který opět zavádí jiné značení úrovní bezpečnosti (viz opět čl. 9.2). Vzhledem k tomu, že kolébkou informačních technologií jsou Spojené státy, v drtivé většině případů je citována Oranžová kniha a i my se v dalším textu podle toho zachováváme.

<sup>2</sup> Anglický termín, který se ujal i v českém žargonu. Člověk zabývající se hledáním cest k poškození výpočetních systémů je námět pro psychology a sociology. Každopádně je zřejmé, že jde o skupinu lidí, kteří nerozumí příliš principům computer science a snaží se porozumět pouze některým částem operačních systémů, jejichž vedlejší efekt uniknul výrobci. Naopak každý erudovaný programátor má snahu vytvářet, nikoliv ničit, což je obecný princip vývoje lidstva, bez něhož by počítače nikdy nevznikly.

<sup>3</sup> Tato konvence je dodržována u výrobců UNIXu, ale není akceptována výrobci protokolů TCP/IP pro jiné operační systémy, zejména tam, kde není zajištěna jiná ochrana, jako jsou např. programy na IBM PC. Není ani závazný pro Internet, jak vyplývá z dokumentů RFC.

<sup>4</sup> Název je z řecké mytologie. Tříhlavý pes Kerberos byl pověstným strážcem podsvětí.

# 10 SPRÁVA

Správný chod operačního systému znamená jeho dobrou a bezchybnou podporu používaného informačního systému. Operační systém UNIX slouží pro širokou podporu aplikačních programů. Je používán jak na pracovních stanicích pro podporu práce především grafických programů, tak i pro výkonné databázové aplikace v místních nebo metropolitních sítích. Toto obecné použití a jeho jednoduché uživatelské a programátorské prostředí jej vyneslo vysoko na žebříčku popularity posledních desetiletí. Práce každého operačního systému je ale vždy podmíněna požadavky uživatelů, přesněji způsobu využití výpočetního systému pro dané potřeby. Nastavit operační systém podle takových požadavků a trvalé sledování jeho chodu je práce pro správce operačního systému, která nemusí být nijak snadná. Instalace operačního systému dnes probíhá poměrně automatizovaně, je přitom zohledněn výkon používaného hardwaru již výrobcem. Nejpriznivější je situace, kdy je výrobce hardwaru i UNIXu tentýž. Rozhraní mezi holým strojem a operačním systémem je pak velmi efektivní. I tak je ale takový celek výrobcem dodáván v obecné podobě a je připraven na podporu různých typů různě výkonných aplikací. Teprve s nasazením požadovaného aplikačního softwaru se jednoznačně potvrdí požadavky na výpočetní zdroje. Po jejich rozpoznání pak správce operačního systému může přizpůsobit operační systém pro jejich dobré zabezpečení, a tak zvýšit výkon celého výpočetního systému. V mnoha případech tak i předejde provozním problémům, které mohou vzniknout při náhlém vyčerpání výpočetních zdrojů (např. počet procesů, které mohou současně pracovat s přístupem k databázi).

Správný provoz operačního systému UNIX vychází ze znalosti předchozích kapitol této knihy. Pochoopení principů implementace jednotlivých částí UNIXu umožňuje správci provádět kvalifikované nastavení nabízených výpočetních zdrojů. Metody a postupy, které jsou při této práci v UNIXu běžně používány, uvedeme v této kapitole. Zaměříme se přitom na principy, protože z pohledu příkazů jsou jak jména, tak formát systémových příkazů nejednotné a POSIX je zatím nedefinuje (přestože se rozšíření tohoto standardu pro správu operačního systému připravuje). Příklady pak použijeme jak podle SVID, tak i z nejvíce používaných a obecně prakticky ustálených systémových příkazů.

## 10.1 Život operačního systému

Jádro operačního systému UNIX je uloženo na diskové paměti v souboru `/unix1`. Do operační paměti jej musí kopírovat zaváděcí programy, které jsou z části specializované výrobcem a z části obecné pro každý UNIX (obecně je používán program v souboru `/boot`). Teprve po tomto zavedení a po rozběhu jádra v operační paměti můžeme říct, že operační systém ožívá. Jádro je supervizorem všech dějů v operačním systému. Jak jsme ukázali v předchozích částech knihy, v jádru se evidují všechna potřebná nastavení práce s periferiemi, síťovými protokoly, práce se systémem souborů atd. tak, aby jádro zajišťovalo požadavky jednotlivých procesů. Tato složitá činnost jádra vyžaduje, aby při jeho provozu pokud možno nedocházelo k chybám. UNIX je koncipován tak, že jádro je jeden program sestavený z modulů práce systému souborů, procesů, síťových protokolů atd. (viz obr. 1.3). Součástí jádra jsou i ovladače periferií a strojově závislá část. Změny v jádru, které správce systému musí zvládnout, je např. přidávání dalších ovladačů do jádra. Pokud i ovladač sám programuje (v jazyce C), musí pečlivě dbát na jeho správné chování, protože programuje nový modul jádra, který bude k proveditelnému programu `/unix` připojen a jeho chybování způsobuje i chybování celého jádra (uvážnutí v operaci nad periferií, havárie,

přepisy tabulek jiných modulů atd.). Tato křehká implementace je základním principem provozu UNIXu a po celá desetiletí jeho vývoje nebyla závažným způsobem změněna. Všechny změny, které správce v jádru může provádět pro jeho lepší činnost, podléhají tzv. regeneraci jádra. Regenerace jádra bude předmětem článku 10.3.

Jádro `/unix` je po zavedení do operační paměti startováno a obaluje hardware. Skutečnost, že jádro bylo zavedeno do operační paměti a spuštěno, již předpokládá určitou minimální konfiguraci hardwaru, na níž je možné jádro provozovat. Je to přítomnost procesoru, dostatečná velikost operační paměti a disk se systémovým svazkem. Obalení hardwaru pak znamená rozpoznání úplné konfigurace hardwaru, její evidence a převzetí těchto výpočetních zdrojů pod supervizorový režim, tj. umožnit k nim přístup pouze pod dohledem samotného jádra (jak již víme, přístup procesů k nim je dále možný pouze pomocí volání jádra). Jádro za účelem rozpoznání jednotlivých periférií volá funkce podle seznamu všech ovladačů v tabulce `cdevsw` a `bdevsw` (viz kap. 6, obr. 6.2), a to ty, jejichž jména jsou zakončena textem `_init` (např. `xx_init`). Jedná se vždy o inicializační funkci periférie. Přítomnost takové funkce v jádru je povinnou součástí ovladače. Její činnost musí být programována tak, aby rozpoznala přítomnost periférie, rozhraní nastavení periférie a po určité, pro periférii typické reakci na pokyn pro výchozí nastavení (reset), to oznámila jádru. Jádro přebírá informaci o správné nebo chybné funkci periférie. Pro jeho další chod to pak znamená zpřístupnění periférie (prostřednictvím dalších funkcí ovladače) procesům nebo její odpojení. Inicializační funkce ovladačů mají také možnost oznámit zjištěný stav periférie zápisem zjištěných skutečností na systémovou konzolu. Úzus je jednořádkový výpis o správně se chovajícím rozhraní připojené periférie s informacemi o adrese hardwaru, typu přerušení, kanálu DMA (Direct Memory Access), případně výrobci atd. Nevypisuje nic v případě, že periférii vůbec neobjeví, a vypisuje informaci o kolizi, pokud při komunikaci s periférií nastala určitá chyba. Výsledkem vypisovaného textu na systémovou konzolu je tedy seznam přítomného hardwaru, který jádro rozpoznalo a je schopno prostřednictvím svých ovladačů jej v další činnosti využívat. Periférie, která v seznamu není nebo je označena za chybující, je jádrem v další činnosti odstavena a procesům nepřístupná. Jádro přitom nijak neudrzuje souvislost právě uvedeného postupu rozlišení hardwaru s množinou speciálních souborů v podstromu adresáře `/dev`. Speciální soubory vznikají a zanikají příkazy `mknod` a `rm` a jejich vazba hlavního a vedlejšího čísla na jádro není jádrem nijak kontrolována. Dobrou výjimkou jsou dnes implementace např. systémů Digital UNIX, AIX nebo IRIX, ve kterých je v okamžiku rozpoznávání hardwaru jádrem také aktualizována přítomnost speciálních souborů na systémovém svazku v `/dev`. Znamená to, že odpojená periférie nebude jednak po dalším startu systému rozpoznána, ale bude také zrušen její speciální soubor (což nemusí být vždy výhodné) a naopak, v případě rozšíření o nový hardware, pokud tento bude jádrem rozpoznán, bude mu i vytvořen odpovídající speciální soubor, a to podle aktuálního umístění ovladače v `bdevsw` nebo `cdevsw` (hlavní číslo) a podle možností ovladače (vedlejší číslo).

V další práci jádro rozpozná velikost operační paměti. Operační paměť jádro rozpoznává ve třech hlavních částech. Jednak je to část, kterou obsadí samo jádro (kernel memory). Její velikost je odvozena od velikosti textového segmentu, datového segmentu a zásobníku jádra podobně jako u procesů. Kolik to je, je možné zjistit použitím příkazu `size` na soubor `/unix` např. takto

```
$ size /unix
```

text	data	bss	dec	hex
995328	61440	128156	1184924	12149c

Výpis takového příkazu obvykle jádro uvádí ve fázi po zavedení a rozpoznání hardwaru. Velikost jádra v operační paměti závisí na velikosti samotného jádra. Jádro není po celou dobu práce operačního systému nijak z paměti uvolněno. Druhá část operační paměti, která je rovněž pevně přidělena, je jádrem využívána pro systémovou vyrovnávací paměť a dosahuje velikosti stovek diskových bloků. Konkrétní velikost je pevně zadána v jádru a je měnitelná regenerací jádra. Konečně třetí část operační paměti je používána pro umísťování procesů a jádro ji označuje jako uživatelskou (user memory). Je to zbylá část operační paměti a její požadovaná velikost je odvozena od používaného aplikačního softwaru.

Pokud část operační paměti pro umísťování procesů provozně nestačí pro všechny existující procesy v systému, je využívána odkládací oblast (swap area) na vyznačeném diskovém prostoru (viz obr. 3.16). Jádro primární odkládací oblast registruje ve speciálním souboru `/dev/swap`, který je dalším jménem některé sekce disku, na kterém je umístěn také systémový svazek. Umístění a velikost sekce disku primární odkládací oblasti je určena v době instalace operačního systému a jádro ji při každém startu již jenom rozpoznává. Informaci o takovém rozpoznání pak uvádí výpisem na systémovou konzolu za výpis o rozdělení operační paměti.

V případě, že jádro nenalezne principiální konflikt v průběhu rozpoznávání konfigurace hardwaru, který by znemožňoval jeho další běh, z vlastní iniciativy vytváří první proces s číselnou identifikací (PID) 0. Říkáme, že proces je vytvořen nestandardně, protože je to jediný proces, o jehož vznik nikdo nepožádá voláním jádra `fork`. Jméno prvního procesu bývá různé, např. **swapper**. Je odvozeno od předmětu jeho hlavní činnosti, tj. práce pro odkládání procesů z operační paměti na disk a zpět. Ještě dříve, než se své činnosti začne věnovat, však vytváří další proces, jehož jádrem přidělené PID je vždy 1 a jehož jméno je vždy **init**. Proces **init** je důležitý systémový proces, který zodpovídá za další správný chod přístupu uživatelů k operačnímu systému, a to tak, že kontroluje existenci systémových procesů, které zajišťují správný provoz operačního systému. **init** vytváří další takové procesy a po jejich zániku rozhoduje o dalším pokračování startem dalších procesů. Literatura, např. [Bach87], jej uvádí jako prapředka všech dalších procesů. Proces **init** je ovšem pouze nástrojem správce systému. Přestože je jeho chování po instalaci UNIXu určitým způsobem definováno, je věcí správce systému jeho chování přizpůsobovat potřebám provozu celého výpočetního systému. Základní tabulka, se kterou **init** pracuje, je uložena v souboru `/etc/inittab`. Operační systém UNIX je podle jejího obsahu procesem **init** nastaven do jednoho z 8 možných stavů (neboli úrovní, angl. level). Stav operačního systému je definován množinou procesů, které proces **init** vytváří a případně dále udržuje při životě. Původní UNIX ze začátku 80. let rozlišoval dvě základní úrovně práce operačního systému. Byly označovány jako režim *jednouživatelský* (single user mode) a *víceuživatelský* (multi user mode). Práce jádra v obou stavech byla stejná, ale proces **init** v jednouživatelském stavu vytvořil pouze několik základních procesů, které jsou nutné pro chod systému. Vedle procesů práce s organizací operační paměti (stránkování, segmentace) to byl pak již pouze proces shell, který umožňoval privilegovanému uživateli (s UID rovno 0 a pod jmenným označením `root`) pracovat v běžném příkazovém řádku na systémové konzole (`/dev/console`). Všechny další procesy, které podporovaly vstup uživatelů a jejich práci v systému, nebyly startovány. Tento režim byl určen pro údržbu a nerušenou práci správce systému. Přechodem do režimu víceuživatelského (odmítnutím nebo ukončením režimu jednouživatelského na systémové konzole) pak proces **init** startoval všechny procesy nutné pro uživatelsky provozuschopnou práci celého výpočetního systému. Proces **init** toto zajišťoval tak, že vytvořil proces **sh**, jehož činnost byla interpretace scénáře souboru `/etc/rc`. Obsah tohoto souboru byl běžně správcem systému rozši-

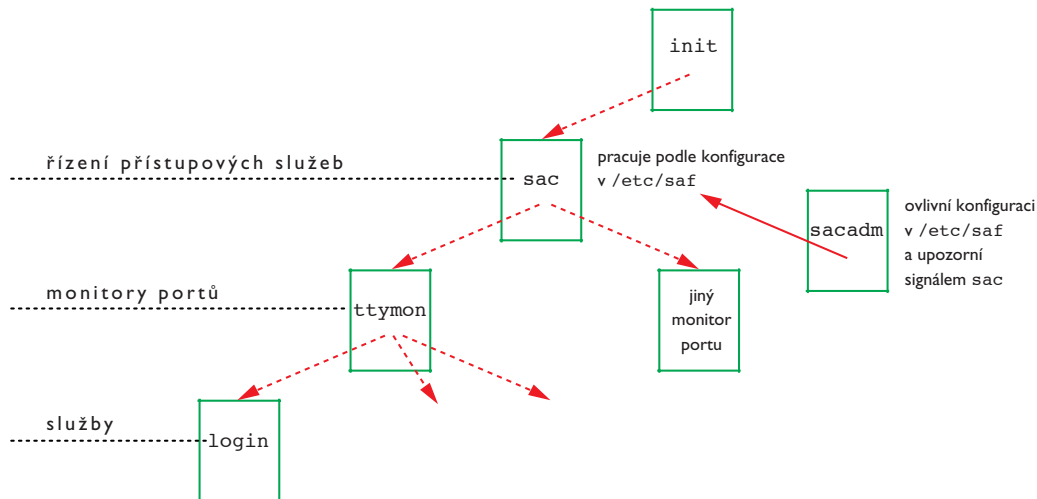


řován nebo jinak modifikován při potřebě změn podpory víceuživatelského režimu. Obvykle zde byly umísťovány příkazy kontroly a připojování diskových svazků (**fsck**, **mount** atd.), interaktivní kontrola data a času, start démonů jako např. **update**, **cron** a dalších. Vzhledem k tomu, že proces **sh** pracoval ve vlastnictví uživatele **root**, pracoval s odpojeným řídicím terminálem. Procesy z **/etc/rc**, které vyžadovaly interakci s operátorem na systémové konzole, pak musely mít svůj standardní vstup i výstup explicitně přesměrován na zařízení **/dev/console**, např.

```
/etc/fsck -y /dev/rdsk/ls7 >> /usr/local/adm/startsys
```

Po provedení scénáře **/etc/rc** proces **sh** ukončil svoji činnost a **init** podle obsahu tabulky **/etc/ttys** vytvořil procesy **getty** pro všechny terminály, na kterých byl požadován uživatelský přístup. Tak byl zahájen uživatelský provoz výpočetního systému.

Uvedený popis práce procesu **init** byl v polovině 80. let vystřídán tzv. novou verzí, která umožňuje nastavit až 8 různých stavů operačního systému. Pouze dvouúrovňový provoz UNIXu je popsáným způsobem stále využíván např. u systémů BSD (a ještě některých jeho dožívajících klonů, jako je ULTRIX). Nový **init** jako jediný v SVID definovaný způsob organizace chodu UNIXu vychází z definice 8 stavů v tabulce **/etc/inittab**. Každý řádek tabulky je popis startu určitého procesu. Jeho příkazový řádek je uveden teprve jako poslední část. Předcházející informace, které jsou vzájemně odděleny znakem **:**, jsou informace pro **init**. Jejich význam je postupně následující. První položkou je pouhé textové označení řádku tabulky (přestože je text proměnné délky, často bývá omezen na 4 znaky). Druhá položka je výčet všech stavů, ve kterých má být proces odpovídajícím způsobem startován. Označení stavů je přitom číselné (0, 1, 2, 3, 4, 5, 6) nebo písmenem (S). Následující položka zadává



Obr. 10.1 Příklad použití SAF



vztah procesu **init** ke startu definovaného nového procesu. Položkou je text, každé použité slovo je přitom jednoznačně definováno v provozní dokumentaci a může být použito pouze jedno na řádku. Např. při použití textu `wait proces init` čeká na dokončení vytvořeného dětského procesu (**init** totiž implicitně startuje v tabulce popsané procesy a ihned pokračuje). V tabulce jsou popsána také všechna uživatelská připojení, tj. start procesů **getty** a **xdm**. Za tímto účelem je v tabulce používán text **respawn**. **init** pak takový proces sleduje a po jeho zániku jej znovu startuje (implicitně po ukončení dětského procesu **init** proces neobnoví). Použití textů **sysinit** a **bootwait** (nebo jen **boot**) souvisí se stavem S. Pokud totiž **init** v době startu převádí systém do úrovně S, před tímto převedením provádí příkazy označené textem **sysinit**. Po ukončení prvního stavu S pak před převedením do jiné úrovně provádí příkazy označené textem **bootwait**. Poslední položka každého řádku tabulky je příkazový řádek startu procesu. Následující příklad ukazuje připojení alfanumerického terminálu na terminálovém rozhraní.

```
tt01:234:respawn:/etc/getty /dev/tty01 9600
```

V příkladu je na speciálním souboru `/dev/tty01` jako řídicím terminálu udržován proces `/etc/getty`, a to ve stavech 2, 3 a 4. Popis všech procesů, které určují každý stav, takto vyplývá z celkového obsahu tabulky. Proces **init** se proto při svém vytvoření zajímá o obsah tabulky a v prvním průchodu v ní hledá řádek, který má ve způsobu manipulace s procesem text **initdefault**, např.

```
is:4:initdefault:
```

Jde o označení stavu, do kterého má být operační systém po startu jádra uveden (v příkladu je to stav 4). Takový řádek tabulky dále nemá využitou položku příkazového řádku startu procesu. Ve druhém průchodu tabulkou **init** vyhledává všechny řádky, které v seznamu stavů obsahují stav takto zadaný. Jejich příkazové řádky (v poslední položce) pak používá pro vytvoření dětských procesů. Pro řízení jejich činnosti **init** používá text ve třetí položce řádku tabulky. V libovolném stavu operačního systému může privilegovaný uživatel proces **init** požádat o převedení operačního systému do stavu jiného, např.

```
# init 3
```

je požadavek převedení do stavu 3. Proces **init** po takovém příkazu projde tabulku a všechny procesy, které do nového stavu nepatří, zastavuje, naopak ty, které nejsou aktivní a jsou požadovány v novém stavu, vytváří. Procesy zastavuje odesláním signálu **SIGTERM**, pokud proces v průběhu následujících 15 až 20 vteřin sám neskončí, pak mu posílá signál **SIGKILL**. Obecně se **init** o obsah tabulky zajímá vždy po startu a pak vždy při přechodu do jiného stavu (žádostí privilegovaného uživatele). Pokud je proto tabulka v průběhu práce systémem změněna (editorem), na provedenou změnu je nutno **init** upozornit, a sice příkazem

```
# init q
```

Nedojde pak ke změně stavu, ale **init** aktualizuje současný stav podle obsahu tabulky (výměnou textu **respawn** za **off** např. odstavíme odpovídající proces **getty**). Stav, ve kterém se systém právě nachází, lze pak získat příkazem **who** takto:

```
$ who -r
```

```
.      run-level 2 Apr   4 15:53      2      0      S
```

§

(současný stav operačního systému je 2 a do tohoto stavu byl systém převeden 4. dubna v 15 hod. 53 min.)

Přestože význam stavů 0 - 6 a S je zcela v kompetenci správce systému, vždy po instalaci je obsah tabulky `/etc/inittab` již v určitém významu definován. Provozní dokumentace toto nastavení také popisuje. Obecně se používá pro práci systému analogického s jednouživatelským režimem stav s označením 1 a S (SVID definuje úroveň S jako přísně jednouživatelskou, která nesmí být předefinována na úrovni s jiným zaměřením). Správce systému tak může využívat dva různé způsoby práce údržby. Víceuživatelský režim je možné si vybrat ze stavů s označením 2, 3 a 4. Stav 0, 5 a 6 bývá využíván pro zastavení a opětný start systému. Takže např.

```
# init 0
```

je zastavení operačního systému. Skutečný obsah tabulky `/etc/inittab` je vždy věcí správce systému. Ten by ji měl ihned po instalaci dobře prozkoumat a dobře pochopit všechny její řádky. První pohled do tabulky zaujme její stručností. Teprve pečlivým čtením odhalíme místa, kudy tabulka pokračuje v definicích akcí pro nastavení odpovídajícího stavu operačního systému. Tabulka např. může obsahovat řádek

```
rc2:2:wait:/bin/sh /etc/rc2 </dev/console >/dev/console
```

což je využití scénáře pro shell ze souboru `/etc/rc2` pro provedení systémových akcí při uvedení operačního systému do stavu 2.

K zastavení operačního systému je používán příkaz **shutdown** privilegovaným uživatelem. Jeho volbou **-g** lze ovlivnit časový interval, po jehož uplynutí bude uživatelům znemožněno pokračovat v práci a operační systém se zastaví. Implicitní hodnota časového intervalu je obvykle 15 minut.

**shutdown** přitom v jeho průběhu opakovaně upozorňuje přihlášené uživatele na blížící se konec jejich práce a vyzývá je k odhlášení. Po pozorném prozkoumání programu **shutdown** zjistíme, že odpočítávání časového intervalu je jeho hlavní funkce, protože na závěr své činnosti provádí příkaz **init 0**. Přestože jsou dnes programy **shutdown** v různých systémech implementovány jako binární proveditelné soubory (víte proč?), **shutdown** lze realizovat pomocí jednoduchého scénáře pro shell (prostřednictvím **wall**, **sleep** atd.). SVID příkaz **shutdown** nedefinuje. Samotné zastavení operačního systému lze v konkrétní implementaci studovat v tabulce `/etc/inittab` a v ní použité příkazy pro přechod do úrovně 0. Jde o ukončení práce všech uživatelských i systémových procesů. **init** toho dosáhne odesláním signálu **SIGTERM** všem svým potomkům. Poté měří časový interval 15 až 20 vteřin a opět zjišťuje přítomnost svých dětí. Během časového intervalu každý správně programovaný proces po přijetí signálu provedl závěrečné práce a skončil. Pokud tomu tak není, **init** jej ukončí zasláním signálu **SIGKILL** (přítomnost takových procesů přitom oznamuje na systémové konzole). Analogie s převedením do jiné úrovně procesem **init** je zřejmá. Po ukončení práce potomků procesu **init** jsou odpojeny všechny svazky a **init** ukončí svoji činnost.

Vstup uživatele do systému (přihlašování) se v UNIXu uskutečňuje prostřednictvím procesů **getty** a **login** a jako příklad proměny procesů jsme jej popsali v čl. 2.2 (viz obr. 2.6). Uvedená metoda se vztahuje na přístup alfanumerických terminálů, ale analogická situace je i u přihlašování v rámci grafického rozhraní prostřednictvím procesu **xdm** (viz čl. 8.3, obr. 8.5). UNIX SYSTEM V ale pro řízení vstupu uživatelů do systému používá (a SVID definuje) také tzv. *podporu přístupových služeb SAF*

(Service Access Facility). Se známým přístupem prostřednictvím např. procesu **login** ovšem není ve sporu. SAF jej zahrnuje jako jednu ze služeb tak, že celková koncepce SAF vyhovuje dokumentům OSI. Hlavní komponenty SAF jsou *řízení přístupových služeb* SAC (Service Access Controller), *monitor portů* (port monitors) a *služby* (services). Služba je chápána jako existující proces. Základní proces SAC je démon **sac**, který je startován procesem **init** v době přechodu do víceuživatelského stavu (jeho start je popsán v `/etc/inittab`). Démon pracuje podle definice *systémové konfigurace* souboru `/etc/saf/_sysconfig`, odkud získá informace o programovém prostředí, ve kterém poběží a které dědí jeho dětské procesy. Hlavní funkcí **sac** je řízení práce monitorů portů, jejichž způsob startu a konfigurace přebírá **sac** ze souboru `/etc/saf/_sactab`. Každý monitor portu má pak svoji vlastní konfiguraci (po volání jádra **fork** je před **exec** u každého monitoru portu nastaveno prostředí podle souboru `/etc/saf/pmtag/_config`), řídí práci služeb portu a každá služba portu má také svoji vlastní konfiguraci. Příkladem monitoru portu je proces **ttymon**, který je v nových systémech používán namísto procesu **getty**. Pro manipulaci s démonem **sac** správce používá příkaz **sacadm**, který je definován jako jediný prostředek pro práci SAF v SVID. Pomocí něj lze provádět změny konfiguračních tabulek démonu **sac**, tedy je možné do konfigurace přidat nebo naopak z ní odebrat monitor portu, odpojit, připojit, zastavit nebo obnovit běh monitoru portu atd. Na obr. 10.1 je příklad hierarchie procesů za podpory SAF.

Při použití SAF je tedy zodpovědnost za správný start a obnovu procesů pro vznik a zánik uživatelské komunikace s operačním systémem odpovědný příslušný monitor portu za dohledu démonu SAC, řízení přístupových služeb. V případě uvažnutí monitoru portu nebo jiné kolize démon SAC řeší vzniklé situace, případně je může ovlivňovat správce systému prostřednictvím **sacadm**.

Monitor portů **ttymon**, který je dnes používán namísto **getty** a **uugetty**, je v SVID definován s odkazem na širší možnosti řízení vstupu uživatelů do systému. Patří sem např. možnost ovládání více portů jedním procesem, běžné využívání modulů jádra technologií STREAMS a také možnost práce i s jinou službou, než je implicitní **login**. V praxi je ale stále čtenější výskyt použití jeho předchůdců typu **getty**.

## 10.2 Disková paměť

Důležitou součástí údržby správného chodu operačního systému je dohled nad diskovou oblastí. V UNIXu jsou všechna data, ať už uživatelská nebo systémová, součástí diskových pamětí, jejichž obsah se dynamicky proměňuje podle principů, které jsme uvedli v kap. 3. Tamtéž jsme také uvedli příkazy pro základní manipulace správce systému se svazky a dalšími částmi disků (jako je např. odkládací oblast). Z pohledu údržby se u datových částí disků jedná o jejich inicializaci (příkazem **mkfs**), připojování k aktivnímu systémovému stromu adresářů (příkazem **mount**), odpojování (**umount**), o kontrolu konzistence dat (**fsck**, **fsdb** aj.) a konečně o zálohování obsahu dat (**tar**, **cpio**, **pax**, **backup**, **restore** atd.). Důležitá je přitom pravidelná kontrola a pořizování (nejlépe automatizované opakované) zálohy dat. Z obecného pohledu podle kap. 3 lze operace nad svazkem chápat ze dvou pohledů. Jednak jsou to všechny operace procesů s obsahem datové základny za připojeného svazku k systémovému stromu adresářů, tj. uživatelské (ale i systémové) zpracování dat běžícího systému, a jednak jsou to operace nad speciálním souborem daného svazku, tedy globální manipulace se svazkem. Pohled na disk prostřednictvím speciálního souboru je přístup k jeho vnitřní organizaci dat, což je často nebez-

pečné. Proto jsou správci systému k dispozici programy, které mu tento pohled a případnou manipulaci s vnitřní strukturou svazku usnadňují. Je důležité také opět upozornit na nebezpečí takové manipulace v případě připojeného svazku, a to z důvodu práce algoritmů jádra pro systémovou vyrovnávací paměť (buffer cache). Přístupem ke speciálnímu souboru (byť blokového) riskujeme takové operace jiných procesů se svazkem, které přivodí změnu jeho vnitřních vazeb.

Vytvořit strukturu svazku můžeme na každém blokovém zařízení. Používáme k tomu hlavně disky, ale nic nebrání vytvářet svazky na magnetických páskách s přístupem po blocích nebo na disketách. Disketu jako pokusné médium při různých pokusech je naopak výhodné používat, protože médium je levné a rychle vyměnitelné. Jak již bylo uvedeno v kap. 3, svazek vytváříme příkazem **mkfs**. Jeho parametrem je speciální soubor sekce disku, kde svazek vytváříme. Např. příkazem

```
$ mkfs -F xfs /dev/dsk/2s7 2000000
```

vznikne v sedmé sekci třetího disku svazek o velikosti 2 milionů diskových bloků. Typ svazku jsme zadali **xfs**; **-F** je nepovinná volba, program v případě jejího neuvedení vychází z informací o typu svazku v souboru `/etc/vfstab` (v praxi často jen `/etc/fstab`). Typ svazku je přitom věcí implementace UNIXu a jak jsme již uvedli v kap. 3, výrobci na vývoji nových typů svazků stále pracují z důvodů posilování průchodnosti a bezpečnosti. Jednotná je tak pouze úroveň podle kap. 3, ale možnost připojení svazku vytvořeného v jiném typu UNIXu je mizivá. Výjimkou jsou pak disky CD ve formátu podle ISO9660 nebo média vytvořená na počítačích PC ve formátu MS-DOS, ale tyto formáty nelze přijmout jako provozně výhodné. Přenos dat mezi jednotlivými systémy je pak zajišťován na logické úrovni, tj. vytvořením archivního balíku dat typu např. `tar`, a jeho přenosu na fyzicky čitelném médium v obou systémech nebo sítí. Volby příkazu **mkfs** podle SVID jsou dále ještě **-m**, při jejímž použití svazek nevytváříme, ale naopak zjišťujeme, jakého je typu a zadáním jakého příkazového řádku byl vytvořen. Příkazový řádek totiž může pokračovat dalšími parametry, které se vztahují např. k prokládání fyzických bloků na disku vzhledem k otáčkám disku a rychlosti zpracování přečtených dat dalším hardwarem (tzv. gap nebo interleave factor), případně další, pro typ svazku specifické informace. Volba **-V** má význam pouze kontroly příkazového řádku. Příkazový řádek je prozkoumán a je-li správně formulován, je opsán na standardní výstup bez jeho vykonání. Pokud **mkfs** vytváří nový svazek, po zadání příkazu následuje prodleva 10 vteřin, kdy může správce systému klávesou přerušení (obvykle DEL) přepis obsahu sekce disku ještě zastavit. Pokud se tak stane, data v sekci zůstávají neporušena.

Typ vytvořeného svazku můžeme zjišťovat nejenom volbou **-m** příkazu **mkfs**. SVID uvádí příkaz **fstyp**, jehož parametrem je speciální soubor sekce, na které je umístěn svazek. Oba způsoby vedou správce systému ke správné manipulaci se svazkem, ve většině případů to není nutné. Svazky jsou totiž připojovány voláním jádra `mount` (které využívá uváděný příkaz `mount`) a jádro při této manipulaci samo typ svazku rozpoznává. Jádro tak musí disponovat algoritmy, které rozumí organizaci různých typů svazků. Poskytnutím svazku prostřednictvím síťové služby NFS se ovšem data svazku stávají viditelná v každém operačním systému připojeném k síti.

Vytvořený svazek připojujeme ke stromu adresářů příkazem **mount** a odpojujeme příkazem **umount**. Jejich formát je jednoduchý: prvním parametrem je jméno speciálního souboru svazku a v případě **mount** je dalším parametrem cesta adresáře, se kterým bude svazek spojen (tzv. mount point). Např.

```
# mount /dev/dsk/2s7 /mnt
# cd /mnt; ls -l
```

```
# umount /dev/dsk/2s7
device is busy
```

Je připojení svazku speciálního souboru `/dev/dsk/2s7` k adresáři `/mnt` a prohlížení jeho obsahu. V příkladu nám odpojení svazku selže, protože po výpisu obsahu adresáře náš shell neopustil kořenový adresář odpojovaného svazku. Jako pracovní adresář tohoto procesu jádro eviduje svazek jako využívaný, a proto jej odmítne odpojit.

Příkaz **mount** bez parametrů vypisuje současné připojení svazků k adresářům (vyzkoušejte sami) a je přístupný i neprivilegovanému uživateli (program je ovšem umístěn v adresáři `/etc`, a proto obyčejný uživatel, který jej nemá běžně nastaven v proměnné `PATH` v prostředí shellu, jej musí zadávat celou cestou). I příkaz **mount** využívá informace souboru `/etc/vfstab`, jako je např. typ svazku. Typ svazku může být explicitně uveden v příkazovém řádku za volbou **-F**. Teprve pokud tomu tak není, **mount** prohlíží `/etc/vfstab`, a teprve v případě neúspěchu testuje svazek sám. Příkaz **mount** v souvislosti s vyšším stupněm bezpečnosti (viz kap. 9) disponuje také volbou **-l**, pomocí které lze stanovit úroveň MAC podle obsahu bezpečnostní databáze zařízení DDB. Za volbou musí být uvedena bezpečnostní úroveň ve formátu hierarchické klasifikace, případně (za dvojtečkou) následovaná jménem kategorie, tak jak byla zavedena použitím příkazu **lvlname**. Při používání zvýšené bezpečnosti dat operačního systému se jedná o důležitou funkci příkazu **mount**, kterou nesmí správce systému opomenout.

Adresář, který používáme pro připojení svazku, musí být předem vytvořen. Po připojení svazku je jeho výchozí adresář (kořenový adresář svazku) v jádru evidován pod jménem, na které svazek připojíme. Všechny atributy jsou ale uvažovány podle výchozího adresáře svazku, neboli směrodatný je obsah i-uzlu na připojovaném svazku. Jádro v další práci procesů eviduje tabulku připojených svazků a v případě odkazu na odpovídající adresář uvažuje data připojeného svazku. Tato metoda nebrání umístění dat do adresáře, pokud na něj není svazek připojen. Pouze je jeho obsah nepřístupný po připojení svazku a po odpojení se zpřístupní opět jeho původní obsah. Obecně jádro na skutečnost, že adresář, na který svazek připojujeme, není prázdný, neupozorňuje. Správce systému si proto do adresářů, které jsou používány pro připojení svazků, ukládá soubor prázdné délky pod jménem např. `not_mounted`. Přesto je důležité obsah těchto adresářů pravidelně procházet. Pokud totiž **mount** při automatizovaném volání z nějakého scénáře v době startu systému selže, procesy mohou v adresáři hromadit data, která později při správném připojení svazku nemusí být viditelná a při větších objemech dat mohou odčerpat kritické množství diskového prostoru výchozího svazku.

Automatizované volání příkazů **mount** a **umount** je vkládáno do scénářů volaných procesem **init** podle tabulky v `/etc/inittab`. Obvykle jsou scénáře se jmény `/etc/rc2`, `/etc/rc4` atd. (případně scénáře v adresářích `/etc/rc2.d` atd.) připraveny na připojování svazků tak, že akceptují obsah tabulky `/etc/vfstab`, postačí tedy obvykle tabulku rozšířit o připojované svazky. Před připojením svazků při startu operačního systému je také důležitá kontrola vnitřní struktury svazků. Tabulky správce systému proto také upravuje tak, aby před vlastním připojením byl svazek kontrolován např. příkazem **fsck**, který komentujeme dál. Svazek totiž může být odpojen od adresáře násilně bez použití příkazu **umount** např. výpadkem el. proudu. V obsahu superbloku pak není značka odpojení svazku a příští **mount** selže. Použitím **fsck** je po kontrole značka provedena, **mount** proto připojení provede.

Nutnost odpojovat svazek pouze použitím příkazu **mount** je dána průchodem dat systémovou vyrovnávací pamětí všech připojených svazků. Tento způsob urychlení práce procesů s daty má slabinu v případě náhlých havárií hardwaru nebo jádra. Kromě toho, že v okamžiku havárie pracující procesy o svá poslední data obvykle přicházejí, je také porušena vnitřní organizace připojených svazků, protože superblok připojeného svazku je uložen do vyrovnávací paměti a změny ve struktuře svazku jsou prováděny zde. Teprve volání jádra **sync** (automatizované démonem **update** každých 30 vteřin) obsah superbloku ve vyrovnávací paměti přepisuje na disk. Mezitím ale mohlo dojít k promítnutí změn v jiných blocích svazku i na disk (např. pokud je vyrovnávací paměť plně obsazena), a to bez přepisu nových dat superbloku. Vnitřní strukturu svazku je po havárii proto nutno prohlédnout a opravit.

Prohlížení a opravy svazků provádíme obecně příkazem **fsck**. Jeho formát není složitý, v parametru postačí uvést speciální soubor odpojeného svazku, nad kterým proběhne kontrola. Doporučuje se používat znakový speciální soubor, protože kontrola je pak rychlejší. Výjimkou je kořenový systémový svazek, který naopak nesmí být kontrolován jinak než přes blokový speciální soubor. Např.

```
# umount /dev/dsk/2s7
# fsck /dev/rdisk/2s7
** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Count
** Phase 5 - Check Free List
...
```

je kontrola svazku speciálního souboru `/dev/dsk/2s7`. Program provádí výpis statistiky jednotlivých fází průběhu kontroly. V ukázce jde o typický případ, který ovšem nemusí být totožný s výpisy v současných systémech. V rámci implementace nových typů svazků bývá kontrola pozměněna podle jejich potřeb údržby a jednotlivé fáze práce **fsck** se tak mohou lišit. Každopádně se jedná vždy o interaktivní práci. V případě nalezení nedostatků totiž **fsck** upozorňuje na chybu a ze standardního vstupu požaduje potvrzení k provedení opravy. Program **fsck** vznikl začátkem 80. let jako integrovaná kontrola dat svazků. Do té doby byly dnes jednotlivé fáze **fsck** prováděny vždy určitým programem. Tyto programy byly **icheck** (kontrola i-uzlů) a **dcheck** (kontrola struktury adresářů) a následovaly další doplňující programy, jako např. **clri** (nulování obsahu i-uzlu) nebo **ncheck** (výpis všech jmen i-uzlu). Tyto programy měly řadu zajímavých voleb a bylo dobré příkazy provádět v doporučeném pořadí. S příchodem **fsck** se z implementací UNIXu postupně vytratily, přestože je v některých systémech nalezneme. **fsck** má ve výpisu obvykle komentář pouze několika proběhlých fází kontroly, ale ta se provádí celkem deseti způsoby (které jsou uvedeny i v SVID):

alokované datové bloky nesmí být odkazovány z více i-uzlů, nesmí také být součástí seznamu volných bloků,

odkazy na datové bloky z i-uzlů nebo ze seznamů volné oblasti nesmí být vedeny mimo rozsah svazku,

pro každý i-uzel je propočítán počet odkazů na něj skutečně vedených a je porovnán s počtem takových odkazů v jeho položce,



velikost datové části každého adresáře musí být násobkem velikosti definované položky (tzv. `DIRSIZE`), velikost datové části obyčejných souborů musí být v souladu s počtem pro ni alokovaných bloků,

formát každého i-uzlu musí odpovídat definici,

na alokované diskové bloky musí být odněkud odkazováno,

položka adresáře nesmí ukazovat na nealokovaný i-uzel, odkaz na i-uzel nesmí být mimo rozsah oblasti i-uzlů,

v superbloku nesmí být evidováno více i-uzlů, než je definovaná horní hranice svazku (tzv. `INODE_MAX`), velikost oblasti i-uzlů nesmí přesáhnout velikost svazku,

seznam volných bloků musí odpovídat definovanému formátu,

součet alokovaných a volných diskových bloků musí být shodný s velikostí svazku, rovněž tak součet alokovaných a volných i-uzlů musí být shodný s velikostí oblasti i-uzlů.

Uvedené kontroly souvisejí s algoritmy práce jádra se svazkem. Jsou zaměřeny na situace, kdy k porušení svazku odpovídajícím způsobem může dojít, pokud je algoritmus náhle neočekávaně ukončen havárií výpočetního systému. Uvedené kontroly jsou proto v souladu s principy organizace svazků podle kap. 3. Svazky, které výrobci realizují, mají často pro zvýšení průchodu dat svazkem další algoritmická rizika, na která se jejich implementace **fsck** také zaměřuje.

V případě, že **fsck** nalezne alokovaná, ale uživateli nedostupná data (je alokován i-uzel, jeho obsah je korektní, ale není odnikud odkazován), po vyžádaném souhlasu z klávesnice je připojuje do adresáře `lost+found`. Vzhledem k tomu, že obvykle nezná původní ztracené jméno spojené v adresáři s číslem i-uzlu, pojmenuje přístup k i-uzlu podle jeho číselné hodnoty. Při prohlédávání tohoto adresáře je pak nutné prozkoumat obsah rekonstruovaných dat.

Svazek, který **fsck** prohlíží, musí být odpojen. Pokud svazek odpojit nelze (kořenový svazek), musí být **fsck** spuštěn s parametrem blokového speciálního souboru (data procházejí vyrovnávací pamětí) a systém musí být bez dalších diskových aktivit (musí být nastaven režim jednovýsadový). Další svazky musí být odpojeny, protože po případných změnách ve svazku musí být svazek nově připojen bez odpojení (původní suprblok ze systémové vyrovnávací paměti je zapomenut), což dnešní verze **fsck** obvykle vynutí. V některých případech je jednoduše systém nově startován bez odpojení tohoto svazku.

**fsck** disponuje řadou voleb, které mohou být závislé na typu svazku. Obecně jsou v SVID definovány volby pro explicitní určení typu kontrolovaného svazku (volba **-F**) a volba **-m**, při jejímž použití **fsck** pouze kvalifikuje stav svazku. Návrátový status **fsck** je pak 0, pokud je svazek schopen být připojen. Návrátový status 32 znamená nutnost běžné kontroly svazku a 33 je indikace již připojeného svazku. Hodnoty vyšší než 33 označují velmi poškozené svazky. Volby, které správci systému často používají pro neinteraktivní použití **fsck**, jako je **-y** (upozornění na nedostatky je sice vypisováno, program ale ihned pokračuje s okamžitou nápravou) nebo **-n** (opravy nejsou prováděny) a další, SVID uvádí, ale do budoucna s nimi nepočítá.

Prostředek opravy svazků **fsck** je velmi výhodný při náhlých haváriích pro rychlé obnovení provozu. Jako takový ale dává málo možností záchrany dat z poškozené oblasti, přestože se této možnosti nevyhýbá (jak jsme uvedli). Možnost citlivého zásahu do vnitřní struktury svazku poskytuje na začátku

90. let vzniklý (a SVID podporovaný) program manipulace s vnitřní strukturou svazku **fsdb**. Je mimo jiné také řešením obnovy omylem smazaných dat, přestože je tento problém složitější a souvisí s umístěním dat v systémové vyrovnávací paměti. I **fsdb** má v parametru příkazového řádku jméno speciálního souboru prohlíženého (a odpojeného) svazku. Součástí příkazového řádku může i zde být volba **-F** pro explicitní určený typu svazku, případně další parametry dané typem svazku (SVID za tímto účelem definuje jednotné volbu **-o**). Ladění svazku je interaktivní a pro správce, který program používá, je nutná znalost obecné vnitřní struktury svazků v UNIXu alespoň podle kap. 3. Příkazy, které správce svazku při ladění používá, SVID sice obecně nedefinuje, ale jsou v různých implementacích UNIXu velmi podobné.

Jak bylo uvedeno, současné typy svazků, které jádro UNIXu různých výrobců podporuje, mají často vnitřní strukturu natolik odlišnou od původního pojetí, že program **fsck** nepokrývá potřeby jejich kontroly. Výrobci proto ve své verzi systému dodávají také nové kontrolní programy. Příkladem takového typu svazku je **xfs** operačního systému IRIX. Při spuštění **fsck** s parametrem speciálního souboru takového svazku se objeví zpráva

```
# fsck /dev/dsk/3s7
/dev/dsk/3s7 contains an XFS filesystem; running fsck is unnecessary.
#
```

Podle provozní dokumentace **xfs** můžeme používat program **xfs\_check** pro kvalifikovanou kontrolu tohoto typu svazku. Provozní dokumentace pak také uvádí další programy podpory svazku typu **xfs**, jako je např. **mkfs\_xfs**, **xfsdump**, **xfsrestore**, případně další.

Pro správce systému jsou v UNIXu obecně dostupné i další systémové nástroje práce se speciálními soubory svazků. Zdánlivě nenápadný program **dd** je v dokumentaci popisován jako prostředek konverze a kopie souboru a je starý jako UNIX sám. Je to silný prostředek správce systému, protože souborem je myšlen jak soubor obyčejný, adresář, tak i speciální soubor. **dd** dokáže kopírovat data tak, že vstupní i výstupní soubor je speciální. Dosáhne se tak úplné kopie dat sekce disku včetně vnitřní struktury uspořádání svazku. Např.

```
# dd if=/dev/rdisk/2s7 of=/dev/rdisk/3s7
```

kde parametry **if=** a **of=** uvádějí jména speciálních souborů, mezi kterými je přenos dat (kopie) pak prováděn. Obsah sekce **2s7** je v našem příkladu kopírován do sekce **3s7**, a to podle sekvenčního uspořádání dat výchozí diskové sekce. Jde tedy o přenos dat na úrovni ovladače disků (viz obr. 3.7). Cílová sekce disku po kopii pak obsahuje svazek, jehož velikost logického bloku je dána výchozím svazkem a i ostatní logické vazby svazku jsou tytéž (včetně případných chyb vnitřní struktury svazku). Jde tedy o věrnou kopii všech dat svazku a pro zálohu před úpravou např. programem **fsdb** se může hodit, ale lze ji používat i jako zálohu dat svazku, protože výstupní speciální soubor může být speciálním souborem magnetické pásky. Kopie diskové sekce programem **dd** je časově náročná operace, protože jsou přenášena všechna alokovaná i nealokovaná data svazku, a v případě poloprázdného svazku je to kopie neefektivní. Hlavní použití programu **dd** je proto v oblasti manipulace s částí svazku. Např.

```
# dd if=/dev/dsk/2s7 of=superblok skip=1 count=1
```

je ukázka kopie v pořadí druhého logického bloku svazku **2s7** do obyčejného souboru se jménem **superblok** (pracovního adresáře). Po úpravě obsahu souboru **superblok** můžeme příkazem



```
# dd if=superblok of=/dev/dsk/2s7 seek=1 count=1
```

provést naopak přepis druhého logického bloku svazku obsahem obyčejného souboru **superblok**.

Uvedená manipulace se svazkem ovšem předpokládá velmi dobrou znalost struktury superbloku.

V případě svazků typu **bsd** je navíc příklad neúplný, protože kopie superbloku jsou umístěny vždy také na začátku oblasti skupiny cylindrů. Správná úprava v tomto případě by proto byla komplikovanější (ale nikoliv nemožná).

Pro účely práce s binárními daty je v UNIXu uživatelům (a správci především) k dispozici program **od**, který obsah binárního souboru (obyčejného, speciálního, adresáře atd.) zobrazuje na standardní výstup v osmičkové nebo šestnáctkové reprezentaci obsahu jednotlivých bytů. Možný je i tzv. znakový výpis, kdy je obsah každého bytu vypisován v konvencích jazyka C, např. znak nového řádku jako `\n`, tabulátor jako `\t` atd. V případě, že pro obsah byte není v C zaveden symbol, je výpis v konvenci osmičkové konstanty jazyka C, např. znak Esc je `\033`. Např.

```
# od -c superblok
```

```
0000000  \0 021  w 10 \0 001 030  \0  \0  {  L 003  >  \0  1
0000020  0  \n  \0  $  w  w  \0  \0  5 250 234 037  \0 007  )  Y
0000040  n  o  n  a  m  e  n  o  p  a  c  k  \0 002  * 371
0000060  \0  \f  ~ 025  \0 001 313 361  \0  \0  \0  \0  \0 021  w 347
0000100  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000120  \0  \0  \0  \0  \0  \0  \0  \0  6 327 002  k  \0  \0  \0  \0
0000140 210  P 316  (  \0  \0  \0  \0  \0  \0  \0  \0 002  \0  \0  0
0000160  \0  \0  \0  \0  \0  \0  \b  \0 002  \0 024  \0  \0  \f 370
0000200  \0 001 322 337  \0 017 001 026 213 035  \0  \0  \0  \0 001 030
0000220  \0  \0 004  v  \0  \0  \0  \0  \0  \0 006 244  \0  \0 002  z
0000240  \0  \0 022  E  \0  \0 005  +  \0  \0  \0  \0  \0  \0  \0  \0
0000260  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0001000
#
nebo
```

```
# od -c /dev/dsk/2s7 | more
```

```
0000000  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0001000  \0 021  w 310  \0  \0 001 030  \0  \0  {  L 003  >  \0  1
0001020  \0  \n  \0  $  w  w  \0  \0  5 250 234 037  \0 007  )  Y
0001040  n  o  n  a  m  e  n  o  p  a  c  k  \0 002  * 371
0001060  \0  \f  ~ 025  \0 001 313 361  \0  \0  \0  \0  \0 021  w 347
0001100  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0001120  \0  \0  \0  \0  \0  \0  \0  \0  6 327 002  k  \0  \0  \0  \0
0001140 210  P 316  (  \0  \0  \0  \0  \0  \0  \0  \0 002  \0  \0  0
0001160  \0  \0  \0  \0  \0  \0  \b  \0 002  \0 024  \0  \0  \f 370
0001200  \0 001 322 337  \0 017 001 026 213 035  \0  \0  \0  \0 001 030
```

...

Jako editor binárních dat používají správci systému od nepaměti prostředek **adb**. Původně jej D. Ritchie programoval jako prostředek dynamického ladění programů v UNIXu na úrovni assembleru a také byl celá desetiletí za tímto účelem používán. Jedna z jeho možností je ale také přepis bytu nebo dvojice bytů zadanou hodnotou. Jde o instrukci **w** (nebo **W** pro dva byte) v rámci interaktivní komunikace s **adb**. Užitečná je také instrukce **l**, pomocí které lze vyhledat byte daného obsahu, nebo **L** pro dva po sobě jdoucí byty. Různé implementace UNIXu nabízejí i daleko mocnější binární editory, ale žádný z nich není uveden v základních dokumentech SVID nebo POSIX. **adb** rovněž ne, je ovšem přítomen prakticky vždy.

Pravidelné provádění zálohy dat diskových svazků je základní povinnost osoby správce operačního systému. UNIX vždy nabízel různé možnosti zálohy dat, jejichž přehled jsme uvedli v kap. 3, a to v odst. 3.2.5 a čl. 3.5.

V principu můžeme pořizování zálohy dat rozdělit na dva způsoby.

Za prvé je to záloha celého obsahu svazku. S odkazem na speciální soubor svazku použitý systémový program archivuje data včetně informací o způsobu jejich umístění na disku (a obsahu zaváděcího bloku). Takto pracují např. původní archivační programy UNIXu **dump** a **restor**. Při jejich použití nebo při používání jejich dnešních variant (mnohdy pod jiným jménem) se pracuje vždy s odpojenými svazky, jejichž data se zálohují. Záloha dat celého informačního systému je časově náročný systémový úkol, zvláště u velkých objemů dat používaných svazků, což je způsob v praxi běžný. Proto je úkol archivace obvykle plánován na dobu mimo hlavní provoz používaného informačního systému (např. v pozdních nočních hodinách). Využívá se přitom služeb démonu **cron** a příkazu **crontab** (viz čl. 5.2). Jako archivační médium musí být použita periferie, která je schopna takový objem dat přijmout. V tomto případě se proto stále používají magnetické pásky, jejichž rychlost je vždy výrazně menší než rychlost disků. Bohatá organizace nebo firma si může dovolit pohodlí archivace dat celého svazku do jiné sekce disku a záloha dat pak probíhá výrazně rychleji. Přestože programy **dump** a **restor** nejsou definovány v jednotných dokumentech UNIXu, jsou stále v systémech často přítomny a správci využívány. Umožňují totiž zálohovat diskový svazek s rozpoznáním jeho minulé archivace, tj. podle současného stavu obsahu svazku a podle jeho poslední archivace ukládá **dump** do archivu pouze modifikovanou část (tzv. inkrementální neboli přírůstková záloha). **dump** i **restor** umožňují pracovat na deseti úrovních archivace. Např.

```
# umount /dev/dsk/1s7
```

```
# dump 0uf /dev/rmt0 /dev/dsk/1s7
```

je archivace svazku v sekci **1s7** na magnetickou pásku podle speciálního souboru **rmt0** (archivní médium může být implicitní, zde jej explicitně určujeme pomocí volby **f**). **dump** má povinný argument jednak jméno speciálního souboru archivovaného svazku a jednak úroveň archivace, která je v příkladu použita jako **0** (**dump** i **restor** nepoužívají u volby pomlčku). Úroveň **0** znamená úplnou zálohu na archivační médium. Pokud použijeme jako úroveň číslo vyšší (v rozmezí do **9**), **dump** na založeném archivním médiu vyhledává předchozí úroveň archivace svazku a po prozkoumání rozdílů provede úschovu vyšší úrovně podle nalezených změn. Na magnetické pásce je pak uložen obsah svazku sekvenčně např. za posledních 10 dnů. Je proto možné obnovovat i data, která uživatel zrušil omylem a po několika dnech je potřebuje. Výhoda takové inkrementální zálohy dat je v optimálním využívání

zálohovacích médií a v případě pásek i v rychlejším přístupu k posledním archivům. Nutné je ovšem přísné dodržování organizace úschovy dat (každý svazek má své archivační médium). Použitá volba `u` v našem příkladu znamená zápis o provedeném archivu do historie archivu v souboru `/etc/dump-dates`, což je pohodlné, protože podle speciálního souboru svazku při příští archivaci **dump** dokáže sám určit současnou úroveň archivu. Obsah archivačního média pak lze chápat jako sekvenci archivů jednotlivých úrovní, jak ukazuje obr. 10.2.

Příkazem **restor** provádíme obnovu dat archivu. Jeho hlavní možností je rekonstrukce celého obsahu svazku podle zadané úrovně pořízeného archivu, např.

```
# restor rf /dev/rmt0 /dev/rdisk/1s7
```

je rekonstrukce celého svazku z archivní pásky **rmt0** (je-li použita volba **f**) do svazku speciálního souboru **1s7**. Připomeňme, že případný předchozí obsah svazku bude ztracen. Je také nesmyslné (a nebezpečné) provádět rekonstrukci do speciálního souboru některého připojeného svazku. Volba **r** je základní volba pro takové použití. Úroveň je použita podle posledního zápisu v `/etc/dumpdates` nebo ji lze explicitně stanovit v příkazovém řádku. Je zřejmé, že obnova dat určité úrovně musí probíhat postupnou rekonstrukcí předchozích úrovní. Archivní médium je proto postupně čteno od začátku až po místo požadované úrovně včetně. Volba **x** dále umožňuje pouze částečnou rekonstrukci archivovaných dat. Požadované soubory nebo adresáře jsou uváděny v seznamu argumentů, ve zbytku příkazového řádku. V tomto případě jsou soubory a adresáře (včetně celého podstromu) z archivu umísťovány do pracovního adresáře procesu **restor**. Oblíbená je možnost interaktivního prohlížení archivu a rekonstrukce jeho dat. Lze ji použít volbou **i**, např.

```
# restor if /dev/rmt0
```

...

Následuje pak interaktivní rozhovor správce systému, který většinou také nabízí pomocný text (příkazem **help**). Částečná nebo interaktivní rekonstrukce dat programem **restor** se většinou používá málo. Obvykle se rekonstruuje obsah celého svazku a pro účely úschovy pouze některých dat uživatelé používají archivy **tar** nebo **cpio**, jak uvedeme dále.

Dříve uvedený program **dd** může také posloužit pro archivní účely. Příkazem

```
# dd if=/dev/dsk/0s7 of=/dev/dsk/1s7
```

dosáhneme kopie veškerých dat svazku speciálního souboru **0s7** do speciálního souboru **1s7**. Předpokladem je stejná velikost obou diskových sekcí (cílová sekce může být větší, ale zbytek diskového prostoru je pak nevyužit). Kopie je prováděna podle pořadí diskových bloků (v případě použití znakových speciálních souborů v souvislosti po znacích) svazku, znamená to, že cílová sekce musí být pouze fyzicky dostupná a formátovaná, ale nemusí na ní být vytvořen svazek pomocí **mkfs**, protože vnitřní struktura svazku je dána organizací kopírovaných diskových bloků a je přenášena také. Výhodou provedené kopie pomocí **dd** je možnost okamžitého použití okopírovaného svazku. V případě zálohy uživatelských dat nemá tato výhoda takový význam a je lépe používat externí výměnná média, ale možnost okamžitého využití kopie svazku ocení každý správce systému pro potřebu zálohy systémového svazku. Mnohdy náročné úpravy tohoto svazku po původní instalaci systému (viz např. čl. 10.3) jsou časově náročné a jsou v nich uloženy hodiny a dny náročné práce, které pokračují dalšími týdny a měsíci sledování správně provedené úpravy. Po zhroutení disku se systémovým svazkem je potřeba provoz celého systému co nejdříve obnovit. V případě využití zálohy na externím médiu to není jednoduchá operace.

Pokud není možné použít verzi archivačního programu, která je schopna běžet na počítači bez přítomnosti operačního systému (standalone program), je nutné nejprve instalovat verzi systémového svazku dodaného výrobcem z distribučního média a teprve pak archivovaný svazek obnovit. Čas výpadku celého systému je pro uživatele počítán na hodiny, což dnes u mnoha aplikací není možné. Kopie provozního systémového svazku je možné použít prakticky ihned. Výpadek se počítá v minutách, v jeho průběhu se pouze startuje operační systém z jiného svazku. Důležité je kopii pořízovat na jiném, stejným způsobem rozděleném disku, který v době výpadku buďto fyzicky vyměníme za poškozený disk, nebo jej pro konfiguraci hardwaru označíme jako disk se systémem. Jsou možné i jiné varianty, ale ty jsou dány opět možností zavádění systému z jiného disku. Komplikace pak přináší speciální soubor systémového svazku a odkládací oblasti (swap area), kterým je nutno vyměnit hlavní a vedlejší číslo za odpovídající konfiguraci v jádru, a to ihned po provedení kopie. V každém případě je nutné kopii systémového svazku provádět vždy při změně systémové konfigurace. Současně je také provozně nepoužíván další disk s takovou kopií; při nasazení aplikace, která vyžaduje prakticky okamžité pokračování práce systémem, je cena jednoho disku navíc zanedbatelná. Popsanou situaci ukazují obr. 10.3.

Při kopii systémového svazku programem **dd** je obtížné dodržet podmínku odpojení svazku pro jeho jednoznačnou kopii. Pokud není možnost program **dd** spustit mimo operační systém kopírovaného svazku, je nutné pracovat v jednouživatelském režimu a snížit přitom všechny aktivity operačního systému na minimum. Po provedení kopie pak start operačního systému z kopie odzkoušet. Povedení kopie bez účasti aktivního systémového svazku odpovídá obecným principům práce se systémovým svazkem. Správce systému má u dobrých implementací obvykle možnost startovat záložní jádro z distribučního média. Systémový svazek je pak buďto samotné distribuční médium (CD, disketa, výměnný disk) nebo je systémový svazek dočasně vytvořen v odkládací oblasti (swap) disku nebo přímo ve zbytku nevyužité operační paměti. Provozní (nyní neaktivní) systémový svazek je pak k takovému (velmi omezenému) aktivnímu svazku připojen k některému adresáři (např. `/root` nebo `/mnt`, `/tmp` atd.) a jeho data jsou takto dostupná uživateli `root` běžícího systému<sup>2</sup>. Připojovat nebo odpojovat lze takto i jiné sekce ostatních disků. Při odpojení je pak možné provádět všechny globální manipulace se systémovým svazkem (kontrolu, ladění, kopii atd.). Další možnost je využívat systémové programy, které jsou uzpůsobeny pro běh na holém stroji, tzn. bez účasti operačního systému. Součástí větších výpočetních systémů obvykle bývá systém programové manipulace na úrovni hardwaru (tzv. firmware), který prozatím není sjednocen, ale který mimo specializované manipulace s hardwarem umožňuje i start programů schopných běžet na holém stroji. Již v prvních verzích UNIXu se takový přístup k systémovému svazku běžně podporoval. Správce systému disponoval adresářem `/stand`, který obsahoval verze programů proveditelných na holém stroji (**fsck**, **dd**, **dump**, **restor**, **mkfs**...). UNIX dále obsahuje principiální možnost spouštění takových programů nejen z prostředí firmware, ale i z prostředí programu **boot**, který je jednou z fází zavádění jádra UNIXu, jak bylo popsáno v odst. 3.2.3. Program **boot** a celý princip vícefázového zavádění jádra UNIXu byl implementován pro platformy hardwaru, který nedisponuje inteligentním prostředím firmware.

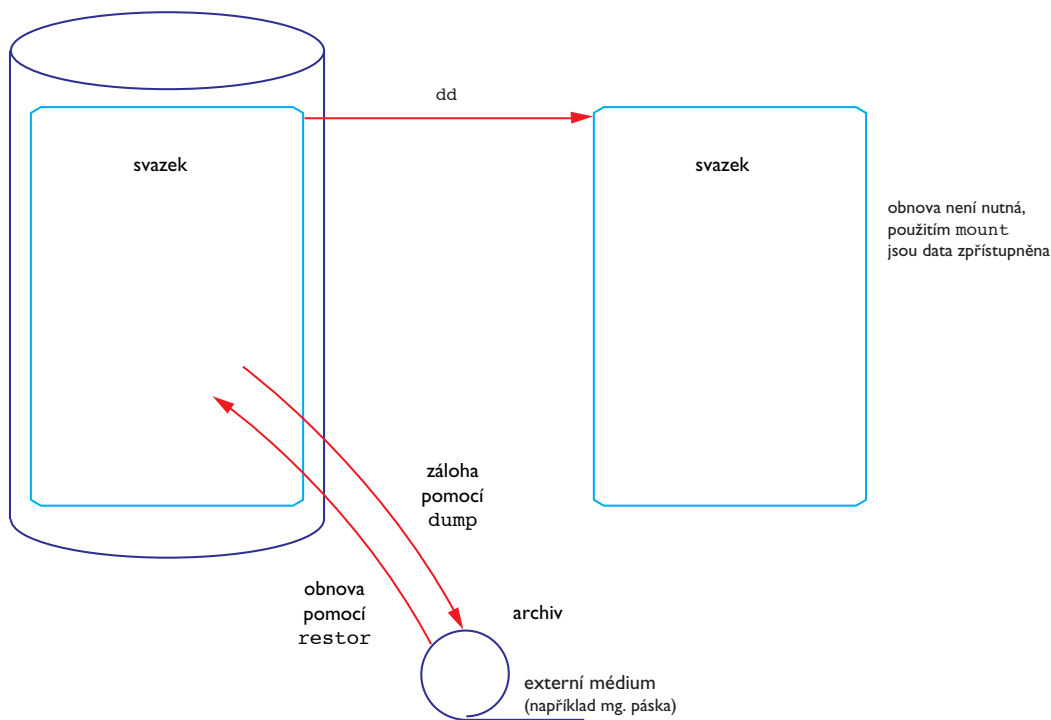
Druhý způsob zálohy dat má charakter zálohy dat pouze určité části svazku, a to na logické úrovni, to znamená, že je archivován obsah každého souboru, jeho jméno a atributy podle obsahu i-uzlu, ale nikoliv informace o jeho umístění na svazku. Vznikají tak archivy dat, které jsou pouhými částmi provozních dat, ale mají výhodu, že nejsou nijak rozměrné. Používá se především program **tar**, **cpio** nebo **pax**, jejichž vnitřní formát je jednoznačně definován. Archivy jsou tedy přenositelné mezi různými

magnetická páska

archiv úrovně 0	archiv úrovně 1 (vychází z archivu úrovně 0)	archiv úrovně 2 (vychází z archivů úrovně 0 a 1)	archiv úrovně 3 (vychází z archivů úrovně 0,1 a 2)	...
-----------------	--	--	--	-----

Obr. 10.2 Sekvence dat archivu dump/restor

implementacemi UNIXu. Tento druhý způsob je zejména uživatelsky dostupný a programátoři jej používají pro zálohu své práce, případně k distribuci výsledných programů, kdy je archiv dat přenášen na výměnných médiích nebo sítí. Vzhledem k tomu, že výsledkem programů je archiv v podobě obyčejného souboru, je možné jeho obsah navíc zhustit některým komprimačním programem, jako je běžně používaný **compress** (a **uncompress** nebo **zcat**, podle SVID) nebo **gzip** (a **gunzip**, případně



Obr. 10.3 Záloha svazku

**gzcat**). Příkladem nabídky nové verze programu pro distribuci prostřednictvím sítové služby anonymního **ftp** pak může např. být

```
# tar cvf progvl01.tar prog man
# compress progvl01.tar
# cp prog101.tar.Z /usr2/people/ftp/pub; rm prog101.tar.Z
```

kdy v pracovním adresáři předpokládáme přítomnost podadresáře **prog** a **man**, jejichž obsah včetně celého podstromu adresářů je uložen do souboru se jménem **progvl01.tar**. Tento soubor komprimujeme a výsledný soubor (s příponou **.Z**) kopírujeme do veřejně dostupného adresáře. Na tomtéž řádku zrušíme nadbytečný soubor. Za domácí úkol prostudujte variantu (podle provozní dokumentace) použití všech akcí na jednom příkazovém řádku a bez nutnosti vytvářet pomocný soubor **prog101.tar.Z**. Programy **tar**, **cpio** nebo **pax** není vhodné používat pro systémovou zálohu dat, protože nepřenášejí systémové informace svazku. Jimi pořízená záloha je pouze zálohou souborů a jejich atributů.

Programy celkové úschovy dat definované v SVID jsou **backup** a **restore**. Jako většina současných implementací programů globální úschovy pracují na úrovni logické. Obvykle jde o programy řízení archivních balíků dat a pro vlastní úschovu jsou používány prostředky **cpio** nebo **tar**, ale obecně podle SVID tomu zdaleka tak nemusí být. Takový archivační program lze programovat také jako scénář pro shell a v některých implementacích UNIXu tomu tak doopravdy je. Principálně jsou programy **backup** a **restore** vybaveny možností archivovat data podle daného soupisu datových oblastí (ve formě adresářů, jejichž celý podstrom bude archivován, případně jmen konkrétních souborů). V termínech SVID pracuje archivační program **backup** s archivační tabulkou. Jedná se o soupis archivovaných datových oblastí v **/etc/bkup/bkreg.tab** (nebo **/var/sadm/bkup/default/bkreg.tab**). Způsob vlastní archivace je dán tzv. metodou (backup method), což je odkaz na proveditelný soubor v adresáři **/etc/bkup/method**. Správce však může používat metody implicitní (default metod), což bývá obvykle využití formátu **cpio** (nebo jemu podobnému, SVID definuje program **ffile**). Archivační metodou jsou ale také v SVID definovány programy **fdisk** a **fdp**, jejichž způsob archivace je po diskových blocích, je tedy archivována i vnitřní struktura dat svazků. Tamtéž se uvádí další metoda **fimage**, která rovněž pracuje s archivem celých svazků, a to jejich kopií pomocí programu **volcopy**, což je program ve své funkci velmi podobný programu **dd**. Příkazový řádek programu **backup** má řadu voleb, jejichž použití závisí na způsobu využívání archivační tabulky, archivačních metod a samotné práce programu. **backup** totiž může být spouštěn správcem systému přímo, nebo automatizovaně. Může být pozastaven, např. v případě, že ve stojanu není založeno archivační médium, operátor pak je vyzván buďto interakcí na terminál nebo odesláním požadavku ve formátu elektronické pošty; opět příkazem **backup** odpovědný operátor z pozastaveného režimu pokračuje v další archivaci. Archivační podsystém může správce využívat ve smyslu archivační tabulky, výjimky z ní ovšem může uvádět v tzv. seznamu výjimek (exception list) v souboru **/etc/bkup/bkexcept.tab**. Seznam výjimek spravuje prostřednictvím příkazu **bkexcept**. Se seznamem výjimek také pracuje příkaz **incfile**, pomocí kterého lze archivovat (nebo naopak vybírat z archivu) samostatné soubory. Ten se musí také orientovat podle tabulky záznamu o provedených archivech (backup history log), která je uložena v souboru **/etc/bkup/bkhist.tab** a jejíž obsah je přístupný pomocí příkazu **bkhistory**. Velmi důležitá je pochopitelně také správa archivační tabulky. Její obsah je totiž podstatný pro proces archivace. Poměrně široké možnosti této tabulky podporuje příkaz **bkreg**, jehož formát je stejně košatý jako vlastní obsah tabulky, prozkoumání jeho možností se ovšem vyplatí. Tabulka obsahuje jak soupis oblastí

pro archivaci, tak jména speciálních souborů archivovaných svazků, metody archivace, intervaly archivace atd.

Obnovu dat z pořízených (a v `bkhist.tab` zaznamenaných) archivů lze podle SVID vykonat prostřednictvím programu **restore**, kdy se jedná o obnovu celého svazku (využívá se metoda **fdisk**), nebo programu **urestore** pro vyjmutí jednotlivých souborů. Podobně jako u pořizování archivu se i zde využívají uvedené archivační tabulky a metody. Tabulka v souboru `/etc/bkup/rsstatus.tab` je evidence všech uživatelských požadavků pro obnovu dat. Požadavek na obnovu jednotlivých souborů totiž mohou využívat i obyčejní uživatelé (běžně prostřednictvím **urestore**). Jejich požadavek je ovšem obvykle pouze evidován, a to formou odeslání zprávy elektronickou poštou uživateli, který má oprávnění obnovu dat vykonat. Seznam elektronických adres takových uživatelů je dán v tabulce `/etc/bkup/rsnotify.tab`.

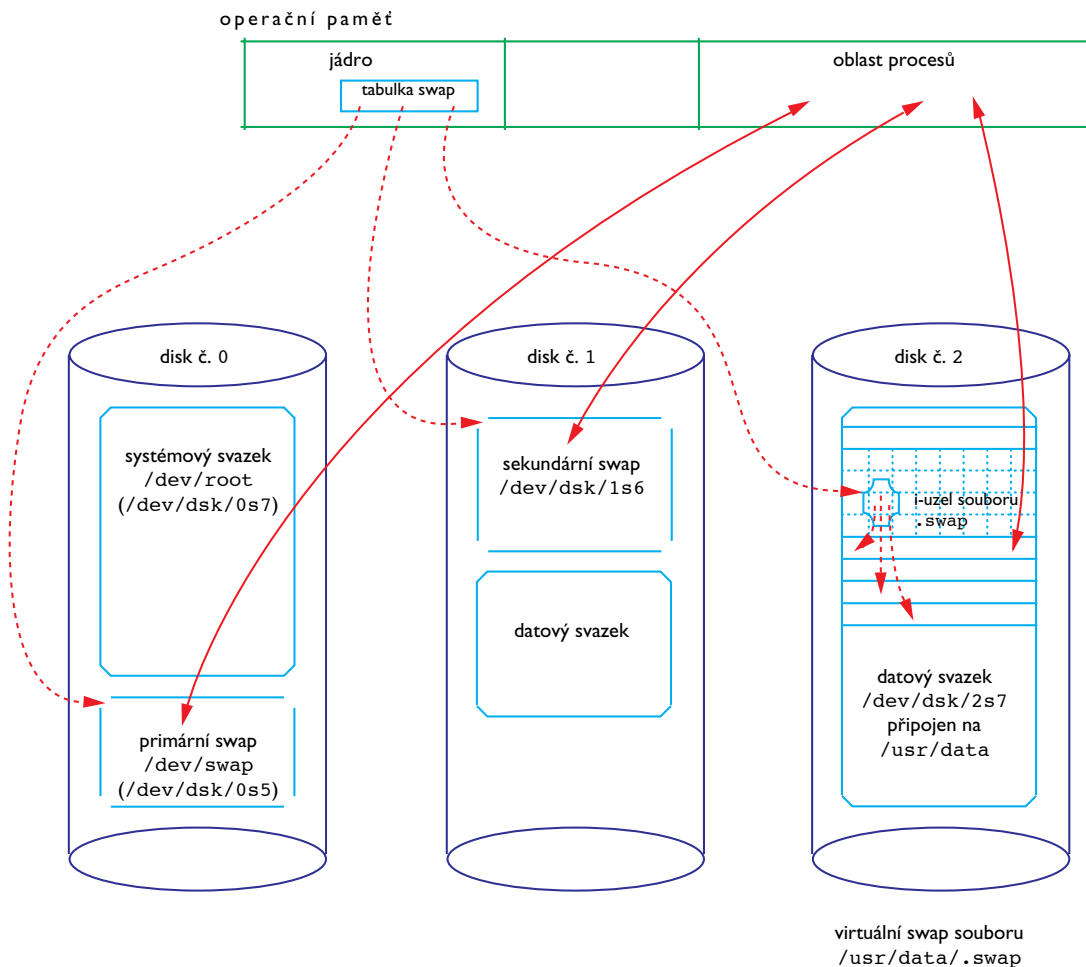
Správa dat logických svazků (logical volumes) nebo pruhovaných svazků (striped disks), jak jsme jejich princip uvedli ve čl. 3.2, není zdaleka ujednocena, jejich podporu v POSIXu nebo SVID prozatím nenalezneme. Ve většině systémů, které logické svazky používají, je základním příkazem pro vytvoření svazku **mklv**. Příkazový řádek je bohužel většinou v různých systémech zcela odlišný. Principiálně příkaz umožňuje sdružit několik sekcí různých disků pod jméno jednoho speciálního souboru (**mklv** jej sám vytváří v adresáři `/dev`, a to jak jeho blokovou, tak znakovou podobu). Jméno (tedy jméno logického svazku) obvykle zadává správce systému v příkazovém řádku. Tamtéž také zadává jména speciálních souborů sekcí disků, ze kterých bude logický svazek vytvořen. Po vytvoření logického svazku je s jeho jménem manipulováno jako s každým jiným speciálním souborem, na kterém lze vytvořit a udržovat diskový svazek, tj. správce systému použije **mkfs**, Vytvoří tak diskový svazek, připojí jej k adresáři příkazem **mount**, odpojí **umount** a kontroluje pomocí **fsck** a **fsdb**. Systémy dále nabízejí příkazy další správy logických svazků, a to pro možnost jejich dalšího rozšiřování (tj. přidávání dalších diskových sekcí, např. **extendlv**) změny jeho charakteristik (např. **chlv**) nebo zrušení logického svazku (např. **rmlv**). Využívání logických svazků je výhodné zvláště tam, kde je potřeba poskytovat rozsáhlý diskový prostor velkému množství uživatelů. Správce tak předchází problémům, které nastávají při vyčerpání rozsahu sekce disku a tím i kapacity určitého adresáře. V takovém případě totiž musí následovat pečlivé přidělování dalších sekcí disků podadresářům vyčerpaného svazku. Připojení další sekce disku k logickému svazku jej naopak poskytne všem uživatelům pracujícím na logickém svazku rovnoměrně. Pro správu diskového prostoru jsou logické svazky určité daným směrem vývoje a předpokládá se brzké ujednocení příkazů jejich správy.

Vždy po instalaci UNIXu je kromě sekce pro systémový svazek alokována také sekce disku pro odkládací oblast (swap). Operační systém ji využívá k odkládání existujících procesů, které již nedokáže umístit v operační paměti. Principy odkládací oblasti jsme uvedli v kap. 2 a 3. Operační systém bez odkládací oblasti prakticky nelze používat. V průběhu instalace vytvořená odkládací oblast má označení primární a často nebývá velká. Její umístění je zpravidla na tomtéž disku jako systémový svazek, v následující sekci. Umístění na tomtéž disku je pro instalaci výhodné, protože operační systém je tak kompaktní a při přemístění disku do jiného počítače ihned provozuschopný. Nevýhoda je ve snížené průchodnosti, protože tentýž hardware zajišťuje jak přístup k systémovému svazku, tak k odkládací oblasti současně. Velikost odkládací oblasti se stanovuje vždy podle celkové zátěže výpočetního systému, tj. podle používaného aplikačního softwaru. Určující je jak počet současně pracujících uživatelů, tak nároky jejich procesů na operační paměť. Základní odhad vychází z dvojnásobku velikosti



operační paměti, ale i její velikost vždy vychází ze zátěže operačního systému (viz následující článek 10.3). Velikost odkládací oblasti není ovšem definitivně stanovena určením velikosti primární odkládací oblasti v průběhu instalace. Správce systému totiž může definovat odkládací oblast sekundární a případně i virtuální, které jádro nevyužívá teprve až po vyčerpání možností primární odkládací oblasti, ale jejich prostor pro odkládání procesů využívá současně (viz obr. 10.4).

Sekundární odkládací oblast je dobré umísťovat do některé ze sekcí dalších disků. Na rozdíl od primární je obvykle její velikost odpovídající požadavkům provozu a její disková kapacita je při využívání jádrem podstatná. Primární i sekundární odkládací oblast je pro správce systému pouze sekce disku bez



Obr. 10.4 Typy odkládací oblasti



vnitřní organizace, kterou si v okamžiku startu jádra inicializuje a dále spravuje samo jádro (jakým způsobem lze studovat v [Bach87]). Po instalaci operačního systému je obvykle proveden nový odkaz na speciální soubor odpovídající sekce disku, takže primární odkládací oblast je pak jádru dostupná pod jménem `/dev/swap` (nebo `/dev/rswap`). Možnosti manipulace s primární odkládací oblastí pro správce systému nejsou nijak velké. Změnu jejího umístění nebo její velikosti dosáhne změnami velikosti sekce disku, což není v UNIXu nijak jednoduché (ale nikoliv nemožné) bez nové instalace. Větší prostor je mu dopřán v případě sekundární oblasti, kterou dynamicky připojuje, odpojuje, mění její umístění, velikost atd. podle současných požadavků provozu. V SVID (ale nikoliv v POSIXu) je pro práci s odkládací oblastí definováno pouze volání jádra `swapctl` (viz 2.4.1). Toto volání jádra využívá příkaz správce systému, jehož jméno bývá různé, snad nepoužívanější je **swap** (nebo **swapon**). Příkaz **swap** poskytuje tři základní možnosti. Jednak je to výpis současné tabulky všech odkládacích oblastí, se kterými jádro pracuje (volba **-l**), možnost připojení další (sekundární nebo virtuální) oblasti (volbou **-a**) a možnost jejího odpojení (**-d**). Např.

```
# swap -l
path          dev          swaplo      blocks      free
/dev/swap     128,188      0          200000      60346
# swap -a /dev/dsk/1s6
# cp /dev/null /usr/data/.swap
# swap -a /usr/data/.swap
# swap -l
path          dev          swaplo      blocks      free
/dev/swap     128,188      0          200000      60346
/dev/dsk/1s6  128,38      0          100000      100000
/usr/data/.swap 128,183      0          0           0
```

Z prvního příkazu vidíme způsob připojení primární odkládací oblasti. Je identifikována speciálním souborem (pod záhlavím `path`) a vedlejším a hlavním číslem (pod záhlavím `dev`). Pod záhlavím `swaplo` je v dnešních implementacích uváděna vždy 0, je to označení začátku odkládací oblasti v sekci disku. Dnes věnujeme obvykle odkládací oblasti celou sekci, ale v případě, že disk nelze rozdělit na několik sekcí a musí být v UNIXu uvažován jako jedna disková plocha, svazek se na disku pomocí **mkfs** vytvoří menší, než je kapacita disku. Zbytek se pak využije na odkládací oblast. Odkládací oblast a svazek mají takto speciální soubor shodný, ale odkládací oblast začíná až za koncem svazku, tj. od určité vzdálenosti od začátku disku. Tato vzdálenost je pak dána parametrem `swaplo`. Jiným příkazem připojujeme sekundární oblast sekce `1s6` a dalším virtuální odkládací oblast v souboru `/usr/data/.swap`, jejíž soubor předtím vytvoříme. Správnost připojení prověříme posledním příkazem výpisu všech používaných odkládacích oblastí.

Virtuální odkládací oblast je odkládání procesů z operační paměti do postupně zvětšovaného a zmenšovaného datového souboru, který je za tímto účelem definován (jak jsme v příkladu uvedli). Výhodou virtuální odkládací oblasti je její dynamická velikost, která se mění podle aktuální potřeby provozu operačního systému, její velikost je pak limitována pouze velikostí volného datového prostoru daného svazku. Nevýhodou je pak větší režie, protože přístup k odkládací oblasti je zatížen průchodem strukturou svazku. Virtuální paměť by měla být využívána pouze pro nouzové stavy provozu výpočetního systému a neměla by být nikdy definována v souboru systémového svazku, jehož volná kapacita nebyvá

velká (je z důvodů bezpečnosti obvykle oddělen od datové části) a při vyšších požadavcích na odkládací oblast hrozí nebezpečí vyčerpání volné oblasti systémového svazku a následná havárie operačního systému.

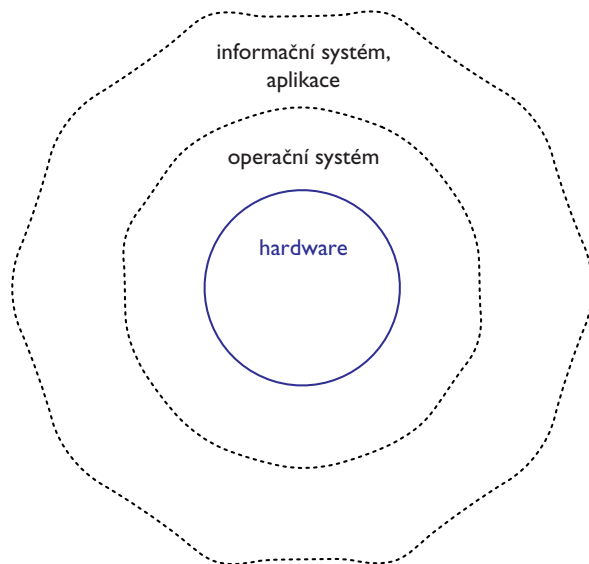
Primární odkládací oblast je připojována v době startu jádra operačního systému. Která disková sekce to je a jakým způsobem má být připojena, lze definovat také v tabulce práce s disky `/etc/fstab`. Pokud definována není, je implicitně využíván soubor se jménem `/dev/swap`. Připojení a odpojení sekundární a virtuální odkládací oblasti je možné uvádět tamtéž, ale správce systému může příkaz **swap** použít jako součást odpovídajících inicializačních scénářů pro přechod do stavu víceuživatelského nebo při zastavování systému. V jednouživatelském režimu totiž správce mnohdy vedle systémového svazku a primární odkládací oblasti další připojené diskové sekce nevyžaduje, právě naopak.

Uvedená tabulka `/etc/fstab` je jednou z klíčových tabulek práce správce systému s obsahem diskových sekcí. Správce systému by měl formát této tabulky studovat a využívat ji, protože je obecně jádrem využívána v době, kdy jádro její informace potřebuje. Podle této tabulky pracují implicitně také příkazy připojování, odpojování a automatizované kontroly konzistence datových svazků. Tabulka není složitá, přestože se její formát v různých implementacích liší. Správce její formát nalezne ve 4. oddíle provozní dokumentace (File Formats) pod klíčovým slovem **fstab**.

### 10.3 Výkon operačního systému (stavba a generace jádra, periferie)

Současný mimořádně rychlý rozvoj masové výroby hardwaru se pro neodborníka jeví jako jediný ukazatel vývoje computer science. Výroba jednotlivých komponent hardwaru dnes opravdu není hlavní zábranou dalšího vývoje. Problémy nastávají ve spojování dílčích komponent v jednotný celek. Dobrá architektura hardwaru počítače velmi záleží na způsobu komunikace procesoru a operační paměti, způsobu připojení periférií. Dominantním problémem je dnes způsob komunikace jednotlivých procesorů víceprocesorových architektur a modularita jejich rozšiřování. Současně je důležitý způsob a rychlost připojení do místní počítačové sítě, která dnes sama o sobě musí být koncipována na optimální výkon používaného informačního systému. V dobře provedeném návrhu architektury hardwaru proto není např. ani tak důležitý maximální výkon samotného procesoru, ale výkon počítače jako celku. Takový návrh není jednoduchý a do finální podoby konkrétního typu celého stroje jej dokáže dovést pouze tým zkušených odborníků. Atributy takového výsledku jsou ale kvalita, spolehlivost a stabilita provozu.

Za předpokladu, že používáme takto dobře koncipovaný hardware, je mezi uživatele ještě vložen operační systém (viz obr. 10.5). Obvykle je operační systém ovšem navržen natolik obecně, aby pokryl různé oblasti využití hardwaru. Pro možnosti mnohdy protichůdných požadavků různých používaných aplikací a informačních systémů nabízí operační systém možnosti své vnitřní modifikace tak, že lze maximalizovat nebo minimalizovat možnosti jeho výpočetních zdrojů a výkon hardwaru společně s operačním systémem kvalifikovat pro konkrétní provoz. Jisté přitom je, že při pořizování výpočetní techniky nelze postupovat směrem od nákupu hardwaru k přizpůsobení požadavků na výpočetní výkon. Prvotní zájem řešitele požadavků zpracování dat je vždy návrh informačního systému, který zpracování dat řeší. Podle jeho náročnosti se pak stanovuje počet potřebných počítačů potřebného výkonu, způsob jejich vzájemného propojení a možnosti operačního systému při provozním zatížení informačním



**Obr. 10.5** Vrstvy výpočetního systému

systémem. Pro řešitele to není úkol lehký, samotný řešitel přitom musí disponovat dobrou erudicí a zkušenostmi a jeho výše platu se musí pohybovat vysoko nad průměrnou mzdou v dané organizaci.

UNIX je operační systém, který byl navržen pro potřeby různých typů aplikací. Je používán na pracovních stanicích pro provoz grafického návrhového systému, ale i jako operační systém silných počítačů centrálního zpracování a toku dat velkých firem, případně jako součást sítí pro provoz služeb různých serverů. Optimální výkon UNIXu se odvíjí od fáze jeho instalace na hardware. Určující je rozhodnutí o velikosti sekce disku systémového svazku, protože data operačního systému by měla být oddělena od svazků uživatelských dat. Velikost sekce systémového svazku je ale také závislá na instalovaném aplikačním softwaru, protože jejich část souvisí s operačním systémem. Jak je velká, určuje výrobce aplikačního softwaru, proto je dobré již při instalaci UNIXu znát používaný aplikační software a jeho požadavky na výpočetní zdroje. Systémový svazek také obsahuje základní oblast pro dočasné soubory, které aplikace běžně používají, tj. adresář `/tmp` nebo `/usr/tmp`. Potřebná volná kapacita systémového svazku je pak vystavena požadavkům používaných aplikací na dočasné soubory. Další část systémového svazku, kde se může náhle kapacitně rozšířit, začíná adresářem `/var/adm` (ve starších systémech `/usr/adm`). Je to oblast, kde je např. ukládán opis chybových zpráv jádra, které jsou vypisovány na systémovou konzolu, účtovací informace uživatelů nebo výpis operační paměti v případě havárie jádra (v UNIXu používaný výraz `core file`) a další časově závislé informace provozu systému. Vzhledem k tomu, že požadovaný neobsazený prostor systémového svazku se dá takto těžko stanovit, správce systému obvykle řeší situaci tak, že na kritické adresáře (zejména pak `/tmp`) připojuje jiné svazky jiných disko-

vých sekcí. Obvyklé je také vytvoření symbolického odkazu (symbolic link) na adresář jiného svazku, jehož kapacita je obecně dostatečná. Např. v jednouživatelském režimu

```
# cd /tmp
# rm -r *
# cd /
# ln -s /tmp /usr3/tmp
```

je zrušení všech souborů a případných podadresářů oblasti dočasného ukládání dat a její spojení na adresář `/usr3/tmp`. Uvedený příklad dostává smysl v případě, že je `/usr3/tmp` součástí jiného datového svazku.

Jak vyplývá z předchozího článku, organizace sekcí disků, a to jak systémového svazku, odkládací oblasti ale i datových svazků, je věcí zralé úvahy požadavků provozu. Po instalaci a rozběhu aplikačního softwaru přichází ovšem fáze optimalizace operačního systému. Je označovaná termínem *system performance tuning*, který je překládán jako *nastavování výkonu operačního systému*. To je v UNIXu předmětem změn číselných konstant jádra, které zadávají pevně maximální velikost tabulek jádra sloužící pro evidenci systémových zdrojů. Tyto konstanty jsou označovány termínem parametry jádra a jejich změn se dosahuje tzv. regenerací jádra. Parametrem jádra je např. velikost tabulky všech současně existujících procesů, počet takových procesů registrovaných pro jednoho z právě přihlášených uživatelů, počet právě přihlášených uživatelů, počet současně připojených svazků, velikost systémové vyrovnávací paměti, počet současně využitelných sdílených pamětí pro komunikaci mezi procesy, počet semaforů atd.

Dříve než začne správce systému měnit hodnoty parametrů jádra, musí se zajímat o sledování jejich skutečných a podle zátěže operačního systému dynamicky se měnících hodnot. Sjednocený a základní (definovaný také v SVID) způsob takového sledování je v UNIXu realizován časosběrnou metodou pomocí programů **sa1**, **sa2** a **sadc**. Jejich procesy vznikají na základě požadavků v tabulkách démonu **cron** opakovaně vždy s odstupem několika minut. Příklad použití opakovaně prováděné akce démonem **cron** jsme uvedli v kap. 5, ale správce nemusí explicitně tyto údaje vkládat, protože je běžný zvyk distribuovat systém se zavedenými záznamy sběru hodnot parametrů jádra, a to v rozmezí od 8.00 do 18.00 každý všední den, vždy po uplynutí 20 minut. Čtenář se může o tom ve svém systému přesvědčit použitím příkazu **crontab**. Stejným příkazem pak může správce systému zadaný interval měnit, což je v případě nestandardního chování jádra běžná metoda. Pořízená data je pak možné prohlížet programem **sar**. Hodnoty získané z jádra (dnes již obvykle ze svazku `/proc`) procesy **sa1** a **sa2** ukládají do souboru `/var/adm/sa/sadd`, kde `dd` je den v měsíci. Data pořízená minulý měsíc jsou takto přepisována, a pokud o ně správce nechce přijít, musí je zálohovat jinou metodou (např. vždy první den v měsíci programem **tar**). Proces **sadc** má doprovodný charakter, jeho úkolem je označovat pořízená data v časovém kontextu. Program **sar** slouží k prohlížení naměřených parametrů jádra. Údaje naměřené pomocí **sa1** a **sa2** jsou totiž všechny, které lze pořídit pomocí **sar**; z nich vybíráme ta data, která pro nás mají současný význam. **sar** vypisuje data ve formě tabulky, v jejímž záhlaví jsou uvedena označení pro parametry jádra; naměřené hodnoty jsou pak zobrazovány vždy s uvedením časového údaje, kdy sběr dat proběhl. Selektce se zaměřením na určité parametry jádra se určuje volbou v příkazovém řádku. SVID definuje následující volby, které určují:

- u parametry jádra související s vytížením procesoru,

- b aktivitu systémové vyrovnávací paměti,
- d parametry související s každým blokovým zařízením (disky, pásky...),
- y aktivitu terminálových zařízení,
- c aktivitu volání jádra,
- w parametry jádra související s odkládací oblastí (swap) a stránkováním procesů,
- a parametry jádra přístupu k souborům,
- q naplnění front procesů při přechodu do různých stavů při jejich existenci,
- v parametry jádra evidovaných procesů, i-uzlů, souborů a zámků používaných souborů,
- m parametry jádra komunikace mezi procesy (semafore, fronty zpráv, sdílená paměť),
- A je výpis všech hodnot nasbíraných parametrů jádra.

Např.

```
# sar -b
```

je vytížení systémové vyrovnávací paměti ve statistice dnes naměřených hodnot (často úzké hrdlo průchodnosti systému). Nebo

```
# sar
```

je výpis aktivit dnešního dne. Ohraničit požadovaný interval pro daný den a čas můžeme pak volbou **-s** (začátek) a **-e** (konec), také je možné volbou **-f** definovat i soubor s nasbíranými daty namísto implicitně uvažovaného `/var/adm/sa/sadd`.

Měření parametrů jádra má pochopitelně význam při zatížení operačního systému aplikacemi běžného provozu, kdy správce systému získá přehled o využití výpočetních zdrojů, případné dosahování jejich limitních hodnot nebo naopak jejich malého využití. Po prozkoumání takto pořízené statistiky se správce systému rozhoduje, na jaké mezní hodnoty je potřeba parametry jádra předdefinovat tak, aby výkon jádra operačního systému pro daný provoz byl optimální. Statistika naměřených hodnot je také pro správce často výchozí při zjišťování kolizí provozu, které souvisí s náhlým pozastavením nebo výpadkem některých služeb. Výsledkem úvahy může být i výměna hardwaru za výkonnější, ale této radikální změně obvykle vždy předchází modifikace limitních hodnot parametrů jádra.

Změna limitních hodnot parametrů jádra probíhá jejich přepisem v základní tabulce modulů jádra a jeho novým sestavením, tj. vytvořením nové verze souboru `/unix`. Adresář, ve kterém jsou uloženy moduly jádra, ze kterých je možno jádro sestavit a také definice tabulek parametrů jádra, případně definice periférií, se v jednotlivých systémech různí. Ve výchozích verzích UNIXu z konce 70. let totiž vycházela generace jádra z jeho zdrojových textů, které byly součástí distribuce operačního systému. Adresářem `/usr/src` začínala oblast umístění zdrojových textů, zdrojové texty jádra pak byly umístěny v adresáři `/usr/src/sys`. Další struktura adresáře `sys` v jednotlivých podadresářích oddělovala soubory zdrojových textů ovladačů, strojově závislé části a samotných modulů jádra. V adresářích byl vždy přítomen soubor se jménem `makefile`, v němž byl definován správný postup stavby binárních modulů z jejich zdrojových textů a vytvoření odpovídající knihovny této části modulů. Překlad a údržba knihovny modulů tak probíhala za pomoci prostředku **make**. Prostředek **make** byl také využíván pro sestavení souboru se jménem `unix` v podadresáři `config`. Soubor `makefile` tohoto adresáře byl výchozí pro generaci jádra a obsahoval odkazy na ostatní soubory `makefile` potřebných knihoven modulů jádra, ovladačů nebo strojově závislé části. Vytvoření souboru se jménem `unix`, tj. generace nového jádra v tomto adresáři probíhalo příkazem

**# make unix**

Nástroj **make** přitom prozkoumal odpovídající vazby na knihovny všech potřebných modulů jádra a zohlednil také soubory **cc.c** a **config.h**, ve kterých byly definovány parametry jádra a tabulky periférií **bdevsw** a **cdevsw**. Verze jádra v souboru **unix** ovšem zůstávala v pracovním adresáři a teprve jejím přenosem do kořenového adresáře systémového svazku se stávala výchozí při zavádění systému. Změny v parametrech jádra prováděl správce systému editací souboru **config.h**, připojování nové periférie pak přidáním odpovídajícího ovladače do podadresáře ovladačů a jeho popisem v souboru **cc.c**. Správce systému současně také mohl číst nebo dokonce měnit zdrojové texty i jiných modulů jádra a získávat tak stále větší kontrolu nad provozem operačního systému. Bylo tak možné operační systém UNIX prozkoumat do úplných detailů a případně se podílet na jeho dalším vývoji, což zřejmě způsobilo jeho velkou oblibu v akademickém prostředí a u každého správce, kterého zajímá vývoj jako obecný trend a který se snaží porozumět používaným prostředkům také z hlediska jejich funkce.

Se zvyšováním obliby se ale UNIX stával stále vhodnějším předmětem obchodu a velmi brzy přestaly být zdrojové texty UNIXu součástí distribuce a staly se součástí poskytované licence při implementaci UNIXu na určitý hardware. Univerzitní verze UNIXu (jako je např. BSD) zachovaly původní způsob práce přímo se zdrojovými texty, které zůstávaly součástí distribuce, nebylo je však podle právních kritérií možné používat pro komerční provoz. Komerčně distribuované a výrobci garantované systémy zdrojové texty neobsahovaly a generace jádra byla uskutečňována na základě přítomnosti binární podoby knihoven modulů jádra. Změnou souboru s obsahem podobným souboru **config.h** i zde docházelo ke změně parametrů jádra. Rozsah jejich hodnot byl a je ovšem takto omezen na určitý interval, jehož překročení může způsobit vzájemný přepis některých částí běžícího jádra. Rozsah použitelných hodnot parametrů jádra každý výrobce uvádí v provozní dokumentaci, dnes je většinou navíc pro modifikaci souboru s parametry jádra doporučován interaktivní prostředek jejich modifikace, který správce upozorňuje na interval použitelnosti a v případě jeho překročení označuje konfiguraci jádra za nekorektní. Adresář **/usr/src** v distribuci komerčních systémů přestal existovat a výchozím adresářem generace jádra se stal **/var/sys** (dříve **/usr/sys**) nebo adresář **/etc/conf**. SYSTEM V.4 a jeho klony ovšem využívají adresář **/etc/master.d**. Obsah jednotlivých podadresářů však zůstal principiálně stejný, různí se jen jejich jména. I princip sestavení jádra zůstává zachován. Podstrom adresáře generace jádra je rozdělen na podadresáře, např. **/etc/conf.d/pack.d** je adresář s binárním kódem modulů jádra, adresář s binárními kódy ovladačů periférií je **/etc/conf/sdevice.d**, atd. Vždy je ovšem důležitý podadresář, ve kterém správce systému provádí samotnou generaci nového jádra. Je jím např. **/etc/conf/cf.d** nebo **/usr/sys/conf** (atd). Jeho obsahem jsou soubory s výchozí konfigurací jádra, tj. tabulky seznamu funkcí ovladačů (původně v souboru se jménem **c.c**), konfigurace ovladačů a také soubor s definicemi konstant parametrů jádra. V původních verzích UNIXu to byl soubor se jménem **conf.h**, dnes je jeho jméno určeno výrobcem operačního systému. Editovat číselné hodnoty v tomto textovém souboru znamená měnit hodnoty parametrů jádra. Jak jeho jméno napovídá, jde o hlavičkový soubor jazyka C. Při zadání konfigurace jádra je kompilátorem jazyka přeložen a použit jako jeden z modulů při vytváření souboru se jménem **unix**, což je nově vytvořené jádro. Změny v konfiguračním souboru parametrů jádra dnes ale obvykle správce systému uskutečňuje pomocí interaktivního programu, jak jsme se již zmínili dříve. Přestože je nejlépe se orientovat v provozní dokumentaci, SYSTEM V a jeho klony používají **id tune**. Správce systému pak pracuje s nabídkou několika menu, kdy postupně určuje část jádra, k němuž se

změna vztahuje, a nakonec stanovuje novou hodnotu vybraného parametru. Ta je obvykle vyžadována v určitém předem oznámeném intervalu. Ve výsledném efektu jde ale pouze o změnu v uvedeném textovém souboru definic parametrů jádra. Program **id tune** bývá umístován do adresáře `/etc/conf/bin`, kde lze najít i mnohé další programy, které usnadňují vytváření nového jádra. Namísto původního příkazu **make** tak dnes správce pro generaci jádra používá **idmkunix**. **idconfig** provede další potřebné změny podle požadovaných tabulek v `/etc/conf/cf.d` a příkazem **idcheck** lze požadovanou konfiguraci nového jádra před jeho vlastním vytvoření testovat na konzistenci. Důležitý je příkaz **idbuild**, jehož zadáním se zahajuje postupné provádění všech potřebných příkazů s předponou **id**, nutných pro generaci jádra (některých i dříve neuvedených), a to ve správném pořadí. Jeho poslední fází je obvykle příkaz **idmkunix**, jehož výsledkem je soubor `/etc/conf/cf.d/unix` s novým jádrem. Tento soubor je jednou z alternativ provozuschopného jádra. Pokud jej chceme používat jako provozní verzi, musíme jej přemístit do adresáře `/`. Původní jádro (za jehož chodu jsme nové jádro vytvořili) uschováme pro případ, že nové jádro nebude pracovat správně, což se může projevit někdy až po několikadenním provozu. Klíčová je tedy sekvence příkazů

```
# mkdir /stand /stand/kernel
# cp /unix /stand/kernel/unix.001
# cp /etc/conf/cf.d/unix /unix
# init 6
```

Uschované doposud funkční jádro pak lze za pomoci prostředků zavádění jádra po zapnutí počítače i nadále používat, např. v případě, že se nám nové jádro z nějakého důvodu nepodařilo vytvořit správně. Zavádění operačního systému má totiž dvě fáze, fází programu `/boot` a vlastní jádro, tedy obecně program `/boot`, který je schopen zavést do operační paměti jakýkoliv samostatně proveditelný program a předat mu řízení. Samostatně proveditelný program (standalone program) je např. jádro, ale často se staví i jiné programy tohoto typu, které jsou schopny běžet na hardwaru bez další podpory operačního systému, např. **fsck**, **mkfs** atp., viz předchozí článek. Program `/boot` není obvykle viditelný, a přestože je interaktivní, pro běžný provoz je nastaven tak, aby nevyžadoval interaktivně zadání souboru se samostatně proveditelným programem, ale implicitně uvažuje soubor `/unix`. Pokud změníme toto nastavení (hledejte v adresáři `/etc/default` nebo `/boot` nebo pouze jen `/etc` atd., klíčová je provozní dokumentace), program `/boot` po svém zavedení (zavádí jej prvotní program zaváděcího bloku systémového svazku, viz kap. 3) se ohlásí na konzolu počítače textem `Boot` a očekává zadání jména souboru se samostatně proveditelným souborem od operátora z klávesnice konzoly. Takže např.

```
Boot
: /unix
```

(znak `:` je součástí výzvy) je požadavek zavedení současného jádra, čili je startována současná provozní verze operačního systému,

```
Boot
: /stand/kernel/unix.001
```

je zavedení předchozí verze jádra a

```
Boot
: /etc/conf/cf.d/unix
```



je zavedení nově generovaného jádra. Je to použití, kdy testujeme nové jádro bez toho, že bychom jej zaměnili za provozní. Pracujte pozorně, protože v případě chybného příkazu můžete ztratit provozuschopné jádro a budete muset použít distribuční verzi operačního systému.

Z uvedeného vágního popisu vyplývá stejně tak vágní umístování a používání ovladačů nových periférií. Ovladače jsou v UNIXu vždy součástí jádra. Aby bylo možné periférii v UNIXu používat, je nutné ji proto vložit do jádra. To znamená, že je nutno jádro nově vytvořit s tím, že je v konfiguračních tabulkách uveden odkaz na nový ovladač a přeložený modul ovladače také umístěn jako jeden z modulů nového jádra. Adresář binárních souborů ovladačů jádra je např. `/etc/conf/sdevice.d` a soubory, které popisují způsob připojení periférie, jsou `/etc/conf/cf.d/sdevice` a `/etc/conf/mdevice`. Původní příkaz pro aktualizaci tabulek ovladačů má jméno **configure** (často se dodnes používá), v jehož parametrech je možné zkráceně definovat novou periférii.

**configure** pak provede všechny nutné a rutinní změny v konfiguraci nového jádra. Vytvoření jádra probíhá stejným způsobem, jak již bylo uvedeno. Programování nového ovladače dnes již není věcí správce systému. Přestože se nejedná o příliš komplikované programování, pro podporu slouží řada funkcí jádra (jako je funkce přenosu obsahu jednoho bytu z požadované adresy) a programování probíhá v jazyce C. Výrobci dnes obvykle nedodávají popis svých periférií tak, aby byl dostatečný pro tento účel. Správce systému se proto musí spolehnout na ovladače dodávané výrobcem odpovídajícího hardwaru, kterým je v lepším případě také výrobce používaného UNIXu. Součástí distribučního média s potřebným ovladačem pak také bývá instalační scénář pro shell, který ovladač umístí do požadovaného místa v podstromu generace jádra a novou generaci jádra také provede i s automatizovanou výměnou jádra za současně používanou.

## 10.4 Síť

Jestli operační systém pracuje správně jako síťové uzel, záleží na správném nastavení všech síťových vrstev (viz kap. 7, obr. 7.5). Správce systému pak obvykle nahlíží na síťový provoz ve smyslu čtyřvrstvého modelu (viz obr. 7.7), zajímá jej funkční hardware, vrstva síťová, tj. přiřazení adres IP a správné připojení na síť, vrstva přenosová jako správná funkce sady protokolů TCP a UDP a konečně vrstva aplikační, kde je nutno ošetřit správnou komunikaci procesů klient a server. Při provozu síťového uzlu v UNIXu je běžně zvykem používat síťové aplikace jak strany klientu, tak strany serveru. Nastavení sítě se přitom prakticky pro oba případy liší teprve až ve vrstvě aplikační.

Základní nastavení síťové vrstvy prostředí protokolu IP vychází z přiřazení adresy IP odpovídajícímu hardwaru. K tomu slouží příkaz **ifconfig**, jehož formát a použití jsme uvedli v kap. 7. **ifconfig** pro nastavení vrstvy IP je první příkaz, který správce systému použije, připojuje-li počítač k síti. Po prověření správné funkce vrstvy IP jej správce systému vkládá do některého z výchozích scénářů při startu operačního systému v odpovídající úrovni. **ifconfig** je ale i program, kterým můžeme zjišťovat také současné nastavení síťové vrstvy, např.

```
# ifconfig ef0
ef0:
flags=1c63<UP,BROADCAST,NOTRAILERS,RUNNING,FILTMULTI,MULTICAST,CKSUM>
inet 194.1.1.2 netmask 0xffffffff broadcast 194.1.1.127
#
```



(příklad je použit ze systému IRIX 6.4) nás informuje o síťové periférii **ef0**. Její adresa IP je nastavena na hodnotu **194.1.1.2**. Adresa sítě je **194.1.1**, ovšem za přiřazení podsítě v rozsahu **1 – 127**. Poslední adresa **127** je použita jako broadcast, tj. odkazovací pro všechny uzly této podsítě. Jestli je takové nastavení síťové vrstvy správné pro operační systém, to se prověří ihned po nastavení, protože **ifconfig** ihned rozpozná např. chybně zadanou masku podsítě nebo adresu pro broadcast z obecných zásad sítě IP (viz kap. 7). Je-li však nastavení správné vzhledem k samotné síti IP, musíme prověřit oslovením některého ze vzdálených uzlů, a to nejlépe uzlu téže podsítě; není-li použita, tedy jen téže sítě<sup>3</sup>, např.

```
# ping 194.1.1.10
```

```
...
```

(předpokladem je ovšem prověřená síťová funkce osloveného uzlu). Spojení síťové vrstvy s hardwarem lze prověřit také příkazem **arp**. Je to program, který vypisuje dynamicky se vytvářející tabulku přiřazení adresy IP a adresy Ethernetu. Tabulka se vztahuje jak na vlastní uzel, tak i na ostatní již oslovené uzly sítě. Zkuste si

```
$ arp -a
```

a získáte informace celé tabulky používané modulem ARP protokolu IP.

Složitější případ práce vrstvy IP nastává při použití více síťových rozhraní. Síťová vrstva protokolu IP je totiž vybavena systémem směrování, které musíme v takovém případě nastavit. Jak bylo uvedeno v kap. 7, směrování můžeme používat statické nebo dynamické. V menších sítích používané statické směrování nastavuje správce uzlu příkazem **route** (příklad viz kap. 7). Také jeho příkazový řádek správce vkládá do scénáře startu operačního systému v síťovém režimu. Správně nastavená cesta směrování paketů mimo vlastní síť uzlu je předpokladem spojení uzlu se světem sítě IP, jako je např. Internet. Správa propojování sítí pomocí dynamického směrování je činnost složitější a týká se především spojování více sítí mezi sebou, jako jsou systémy směrování v metropolitních sítích. Současné protokoly dynamického směrování jsme stručně uvedli v kap. 7, výběr odpovídajícího protokolu je pak na správci takových komplikovanějších celků. V UNIXu často používaný je protokol RIP, který je implementován démonem **routed**. Příkazový řádek pro jeho spuštění umístíme do startovacích scénářů síťového rozhraní a při běžném použití nemá příkaz žádné parametry. **routed** spolupracuje s démony okolních uzlů, přijímá od nich jimi prozkoumané trasy a sám jim své trasy také poskytuje. Volbou **-q** stanovíme tichý chod tohoto démonu, tj. prozkoumané trasy nezveřejňuje. To je ale případ, kdy uzel není směrovačem. Pokud tedy dynamické směrování používáme, jde o uzel, který je směrovačem, ostatní uzly místní sítě mají zadáno statické směrování na adresu tohoto směrovače. **routed** pracuje bez výchozího zadání, ale správce pro něj může vytvořit výchozí informace pro jeho směrování v tabulce **/etc/gateways**. Obvykle jde o uvedení prvního směrovače, kterého démon po svém startu osloví, např.

```
net 0.0.0.0 gateway 147.229.1.1 metric 1 active
```

zadáva uzel s uvedenou adresou pro výchozí použití, její vzdálenost je přitom 1. Program testu správného provozu protokolu RIP je pak **ripquery**, který správce obvykle nalezne jako součást distribučního balíku. Podrobnosti lze studovat v provozní dokumentaci protokolu RIP. Vhodná literatura je [Hunt94], kde lze nalézt čtivě podané informace i o jiných protokolech dynamického směrování v tzv. externích protokolech (viz také kap. 7).

Protokol PPP je určen pro spojení dvou sítí na sériovém rozhraní. Takové spojení je (obvykle jediným) směrem toku paketů mezi spojenými sítěmi. Konfigurace protokolu PPP na straně klientu nebo serveru v souboru `/etc/ppp.conf` proto zahrnuje přidělení adresy IP pro sériové rozhraní, které je současně definováno jako směr pro vrstvu IP (PPP pracuje jako aplikace). Obvykle proto není nutno definovat směrování programem **route**. Ostatní uzly sítě mají pak adresu tohoto síťového rozhraní definovanu jako adresu směrovače, ale uzel s démonem procesu **ppp** v případě příchodu paketu i na jinou adresu tohoto uzlu (např. na adresu IP přidělenou síťovému hardwaru pro místní síť) paket směřuje na sériové rozhraní. Víte proč? Samotný uzel má totiž nastaveno směrování démonem **ppp**, obdržené pakety proto převádí stejně jako pakety požadavků vlastních síťových aplikací. Tento případ nebo i jiné, třeba složitější případy směrování můžeme prozkoumat prostřednictvím příkazu **netstat** s volbou **-r**. Volbou **-n** navíc potlačíme převod adres IP na přiřazená jména (např. podle DNS). **netstat** pak vypisuje tabulku používaných směrů uzlu. Analýza obvykle s tužkou v ruce a malým náčrtkem nás zbaví nočních můr o cestách paketů místní sítě. Správnost směrování paketů IP lze sledovat také pomocí programu **traceroute**. Jde o test dostupnosti vzdáleného uzlu s uvedením cesty, kterou testovací pakety procházejí. Podle výpisu lze pak určit, kde nastala chyba, a tato lokalizace závady v síti, nás vede ke správci sítě nebo k poskytovateli sítě sítí. Výsledky cest směrování tohoto programu mohou být v různých časech a situacích různé a někdy dokonce ohromující, ale takový už je svět sítě IP, kdy za správné nastavení směrů je zodpovědný každý místní správce sítě, jehož síť právě vaše pakety prochází.

Důležitý prostředek zjišťování informací o práci síťové vrstvy IP je ovšem již zmiňovaný prostředek **netstat**. Jeho použití je zaměřeno na zjištění informací o každém evidovaném síťovém rozhraní uzlu. Jinými slovy se jedná o výpis dostupných síťových rozhraní známých síťové vrstvě, což nemusí být jednoznačně výpis provozovaného síťového hardwaru (zamyslete se proč, není to těžké). Použití příkazu jsme už uváděli u výkladu směrování. Tehdy jsme výpis omezovali na rozhraní, která zajišťují směrování (byla to volba **-r**). Volbou **-i** získáme všechna konfigurovaná rozhraní, **-a** všechna známá, ale prozatím nekonfigurovaná rozhraní. Často se používá i volba **-n**, kterou vypínáme převod adres IP při výpisu na jejich známá jména. Výpis tak uvádí rozhraní s přidělením jejich adres, nikoliv jmen uzlů.

**# netstat -ain**

Name	Mtu	Network	Address	Ipkts	Ierrs	Opkts	Oerrs	Coll
ec0	1500	194.1.1	194.1.1.2	6530599	24	5565154	4	158746
ec3*	1500	none	none	0	0	0	0	0
lo0	8304	127	127.0.0.1	43036	0	43036	0	0
#								

je případ uzlu s přítomností dvou rozhraní typu Ethernet z prostředí systému IRIX, a to **ec0** a **ec3**. Rozhraní **lo0** je součástí každé síťové vrstvy a slouží pro loopback. V záhlaví uvedeného výpisu jsou uvedeny texty charakterizující síťové rozhraní a jeho atributy. **Mtu** (Maximum Transmission Unit) je délka rámce, následuje síťová adresa a adresa IP přiřazená rozhraní. **Ipkts** je počet přijatých paketů a **Ierrs** počet přijatých paketů, které byly chybné. **Opkts** je počet odeslaných paketů a **Oerrs** počet odeslaných paketů, které skončily kolizí. **Coll** je pak položka vztahující se k činnosti rozhraní typu Ethernet. V příkladu je zřejmé, že síťové rozhraní **ec3** (proto je také následováno znakem **\***) není nijak konfigurované, a tedy ani používané síťovou vrstvou. Položky tabulky jeho řádku se změní v případě konfigurace prostředkem **ifconfig**.

Správce systému může dále ovlivnit správný chod vrstvy procesové. Jedná se převážně o poskytované služby, tj. správný chod síťových serverů uzlu. Každý síťový klient může být také omezován systémem, ve kterém byl spuštěn, ale implicitně může klienty používat libovolný uživatel. Samotný klient je pak síťová aplikace, která je programována ve smyslu oslovení odpovídající nižší vrstvy volání jádra (Berkeley sockets) nebo knihovny poskytovatele síťového spojení (v TLI), a nepodléhá tedy zvláštním systémovým tabulkám. Jak již byl uvedeno v kap. 7, práce síťových serverů je poskytována dvěma možnými způsoby. Buďto jde o samostatný démon, který očekává příchod síťového požadavku na určitém portu, nebo je server startován jako dětský proces síťového superserveru **inetd**. Oba způsoby předpokládají umístění příkazových řádků démonů nebo procesu **inetd** ve scénářích startu operačního systému v určité úrovni podle procesu **init**, jak bylo popsáno v úvodu kapitoly. Použití služeb superserveru **inetd** znamená vyšší režii, ale je integrovanější. V kap. 7 jsme si uvedli formát všech odpovídajících tabulek tohoto démonu, jako je **inted.conf**, **services**, **protocols** atd. v adresáři **/etc**. Přidání nebo naopak odebrání některého serveru je předmětem modifikace především tabulky **inted.conf**, v případě přidávání pak i tabulky **services**, kde stanovujeme jméno služby a přenosový protokol. Při provedení změn je pak nutné nezapomínat na signál č. 1 (**SIGHUP**), kterým proces **inetd** (ostatně jak je u démonů obvyklé) upozorňujeme na provedené změny v konfiguračních tabulkách, které má akceptovat. Provéřit libovolný správně se startující server můžeme klientem síťové aplikace **telnet**, protože v příkazovém řádku můžeme zadávat číslo portu, na kterém se má server startovat. Takže namísto standardního portu 23 můžeme **telnet** spustit s požadavkem oslovit server na námi určeném portu. Obvykle je aplikační protokol klientu a serveru textový výpis; na standardní výstup terminálu je pak text, kterým server oznamuje klientu svoji přítomnost. Pokud komunikační protokol klientu a serveru známe, můžeme v komunikaci se serverem pokračovat., např.

```
$ telnet localhost 110
Trying 127.0.0.1...
Connected to localhost.vic.cz.
Escape character is '^]'.
+OK valerian POP3 Server (Version 1.001) ready.
USER skoc
+OK please send PASS command
Noaco.
-ERR Invalid command; valid commands: PASS, QUIT
QUIT
+OK valerian POP3 Server (Version 1.001) shutdown.
Connection closed by foreign host.
$
```

je test na správnou funkci serveru pro poštovní služby POP3, a sice na místním uzlu. V tomto případě tak navíc testujeme pouze vrstvy od protokolu IP směrem nahoru, protože jde o případ startu běhu klientu i serveru na tomtéž, místním uzlu. V případě, že server není správně nastaven, je jeho start odmítnut způsobem

```
$ telnet localhost 113
Trying 127.0.0.1...
telnet: Unable to connect to remote host: Connection refused
```

\$

Složitější síťové aplikace mají pro test správného chodu obvykle k dispozici určitý specifický nástroj. Jako příklad můžeme uvést program **nslookup** pro testování nastavení DNS. Většinou je používán po instalaci serveru DNS, ale je možné jím testovat i přístupnost a nastavení vzdálených serverů DNS. Pro zjištění správného nastavení uzlu lze příkazový řádek parametrizovat, např.

```
$ nslookup timothy.vic.cz
Server:      valerian.vic.cz
Address:     184.229.10.10

Name:        timothy.vic.cz
Address:     184.229.10.11
```

kdy odkazem na jméno uzlu podle DNS získáváme jednak informaci, který server DNS uzel registruje včetně jeho adresy IP, a jednak adresu IP samotného uzlu, na který je dotaz směrován. Hlavní využívání programu **nslookup** je v interaktivním režimu, pomocí kterého můžeme získávat obsah tabulek zadaných serverů DNS. Vnitřním příkazem **set type** zadáváme typ záznamu, který nás zajímá (implicitně jsou to záznamy typu A). Příkazem **server** zadáváme server DNS, jehož informace nás zajímají (implicitně je server DNS definovaný pro náš uzel), příkazem **exit** interaktivní komunikaci s programem ukončíme. Práce s **nslookup** pochopitelně předpokládá znalosti síťové aplikace DNS nejméně v kontextu uvedeném v kap. 7. Správce sítě se většinou také orientuje podle provozní dokumentace, jak je to ostatně nutné i u ostatních testovacích programů síťových aplikací, protože jejich použití je obvykle převzato z univerzitních systémů a upraveno vždy daným výrobcem. Jednota daná určitým dokumentem není stanovena, když pomineme dokumenty RFC, které obvykle výrobci neberou vážně. Jak už bylo ale řečeno v knize na několika místech, sjednocení správy operačního systému na úrovni příkazů se připravuje na půdě POSIXu a očekává se v brzké době.

## 10.5 Živý operační systém (denní údržba)

Základním cílem této kapitoly bylo seznámit čtenáře se způsobem náhledu na správu operačního systému UNIX.

Správce musí chápat provoz výpočetního systému jako soustavu současně probíhajících procesů, které využívají výpočetní zdroje pro účely uživatelů. Obvykle má výpočetní systém určité zaměření, které definují typy používaných aplikací. Může se jednat o pracovní stanici s provozem grafické návrhové aplikace, kdy je výpočetní systém zaměřen především na grafickou práci jednoho nebo více současně pracujících uživatelů, nebo může výpočetní systém realizovat informační systém firmy nebo organizace, kdy přístup k jeho výpočetním zdrojům je buďto přímo pomocí grafických nebo alfanumerických terminálů, nebo vzdálený prostřednictvím síťového rozhraní. Zvládnutí provozních požadavků různě velkých systémů předpokládá vědomí všech nutně probíhajících akcí a důležitost nebo zanedbatelnost provedení jejich pořadí. Dobrý správce uzlu nebo celé výpočetní sítě rozumí všem takovým akcím, procesům nebo pochodům a dokáže jejich průběh zdůvodnit, registrovat je, případně omezovat nebo opakovat. Je to práce složitá, ale nutná. V kapitole jsme se proto seznámili s obecným chováním provozu operačního systému UNIX.

Výpočetní systém ožívá startem operačního systému a po vstupu uživatelů zajišťuje denní provoz. Správce systému využívá možnosti ovlivňovat chod operačního systému změnami v konfiguračních tabulkách, a to jak samotného operačního systému, tak používaných aplikací. Často jsou tato nastavení úzce spojena. Mechanizmy sledující provoz jsme uváděli postupně v různých částech knihy. Základní nastavení vstupu uživatelů do systému, sledování jejich přihlašování a provádění akcí jsme uvedli v kap. 5, přestože pro správu konkrétního UNIXu nemůže být dostačující. Právě zde v každém určitém UNIXu (snad z důvodu chybějící standardizace) najdeme různá jména příkazů nebo souborů a adresářů s evidencemi proběhlých událostí. Podle uvedeného ale není orientace nemožná, protože základní stavba UNIXu je táž. Čtenáře proto upozorňujeme na nejvariabilnější část provozní dokumentace, kterou je svazek 8 (někdy svazek 1M nebo 1A), kde je uveden úplný popis všech existujících prostředků správce systému. O správu systému se lze také pokusit z grafického prostředí. Obvykle se jedná o prostředí pracovních stanic, které se výrobci snaží přizpůsobovat uživatelům tak, aby provoz systému dokázal spravovat uživatel, jehož hlavní záměr je používat určitou grafickou aplikaci. Proto je snaha předkládat mu v oknech grafické komunikace formuláře pro změny provozu operačního systému, ale dispozice takové podpory jsou většinou omezené, protože při komplikovanějších změnách je musí vždy realizovat odborník. Ten využívá příkazů řádkové komunikace, případně textového editoru pro změny v tabulkách. Význam přítomnosti textové vrstvy ve struktuře operačních systémů je (zdá se) obdobná jako matematická formulace obecných principů. Grafické vyjádření je pak vždy jen aplikace obecného pro konkrétní účely.

Ve zdánlivém rozporu s úvodem kap. 8 tak lze naopak zdůraznit potřebu formalizace chodu výpočetního systému, kterou v UNIXu reprezentuje vrstva textová, přestože je stará více než 20 let, ale trvale se vyvíjí, jak konečně ukazuje následující závěrečná kapitola této knihy.

<sup>1</sup> Nebo /hpunix nebo /vmunix atd. Jméno je ovlivněno výrobcem.

<sup>2</sup> Z pohledu bezpečnosti je taková situace nepřipustná, protože je takto možný jakýkoliv zásah do datové základny provozní verze výpočetního systému. Periferie, ze které je taková možnost zavedení náhradního operačního systému možná, by proto měla být jiné osobě než správci systému fyzicky nepřístupná.

<sup>3</sup> Mezi uzly dvou sítí je totiž ještě vložen směrovač, který by v případě své chybné funkce mohl test zkomplikovat.



## II PLAN 9

Navzdory litaním malověrných vývoj pokračuje dál.

Koncem 80. let se ve výzkumných laboratořích firmy AT&T objevily první úvahy o novém typu operačního systému. Skupina lidí, která s takovou aktivitou přišla, se sdružovala kolem nestorů operačního systému UNIX, jako jsou Ken Thompson, Brian W. Kernighan nebo Rob Pike, přestože jejich řady rozšířili mnozí další, jako jsou Phil Winterbottom, Dave Pressoto nebo Howard Trickey. Hlavním garantem navrhovaného projektu pak je Denis Ritchie. Vzhledem k vazbám na již úspěšně vykonané dílo vychází navržený Plan 9 z původních myšlenek UNIXu. Jeho koncepce a implementace je ovšem úplně jiná. Samotní autoři pak uvádějí, že jde o snahu udělat dnes a jiným způsobem totéž, co měl být UNIX ve své době.

Změna způsobu využívání výpočetní techniky byla totiž v polovině 80. let zřejmá. Namísto centrálního zpracování dat na silných střediskových počítačích se začal více prosazovat distribuovaný způsob práce. S nárůstem výkonu malých stolních pracovních stanic nebo osobních počítačů bylo možné provozovat řadu aplikací na místní stanici. Nezbytné ovšem zůstávalo připojení na sdílená data ostatních účastníků informačního systému, tj. přístup ke sdíleným datům na základních datových serverech firem nebo organizací. Tento trend vedl ke stále intenzivnějšímu rozvoji počítačových sítí. Objevila se potřeba nového způsobu organizace přístupu k datům, či přístupu k výpočetní technice vůbec. Uživatelé pracovní stanice nebo PC dnes už ani nezajímá, zda jsou periferie, které používá, připojeny k jeho stanici nebo ke stanici kolegy u vedlejšího stolu. Obvykle je např. také až při první kolizi překvapen, že data, která běžně používá, nejsou uložena na discích jeho stanice, ale kdoví kde. A pokud je síť, kam je připojen, dobře spravována, nemusí se to dozvědět vůbec. Řada problémů, které nový způsob práce s daty přináší, ovšem není možné řešit na úrovni práce klasického operačního systému, jako je např. UNIX. Klasický operační systém sice dokáže podporovat síťovou práci (jak jsme v dosavadním textu knihy jistě ukázali), ale zbytečná komplikovanost jeho chodu často uživatele omezuje, a to zvláště při správě pracovní stanice, na které pracuje. Je neefektivní, aby každé pracovní místo kromě uživatele spravoval kvalifikovaný odborník na operační systémy. Místo připojení by měl uživatel aktivovat zapnutím monitoru a přihlášením se do výpočetního celku, kterým je dnes nejméně místní síť. Uživatel je uživatelem počítačové sítě, nikoliv pracovní stanice. Pod pojmem počítačová síť přitom uživatel vbrzku přestane rozlišovat typy sítí, jako jsou místní, metropolitní, rozsáhlá nebo celosvětová. Pojem počítačová síť bude vyjadřovat jeden pojem, např. Internet.

Jako obvykle jsou to myšlenky pěkné, vzletné a optimistické, ale praktická realizace takového softwaru je v nedohlednu. Novou koncepci práce uživatele s daty je totiž potřeba koncipovat a implementovat. Finanční úspěch dnes přitom slaví výroba hardwaru a operačního systému pro koncové stanice typu PC, jako je Windows95 nebo Windows98 či Windows-NT. Jejich koncepce ale nepřekračuje sféru koncového uživatele a pokud ano, jde pouze o nedostatečné návrhy a náhodné realizace bez jasné celkové koncepce. Současná doba je naopak zasvěcena vývoji operačního systému nového typu, operačního systému, který koncepčně využívá hardwarové výpočetní zdroje celé počítačové sítě, globalizuje přístup uživatele ke zbytku sítě (celosvětové), pro uživatele je nenáročný na správu, je spolehlivý a ještě ke všemu je bezpečný. Obvykle bývá používán termín *síťový operační systém*. Tento termín si ale začaly přivlastňovat všechny operační systémy, které nabízejí pouze síťové služby strany serverů. Označení je proto obecně zavádějící, ale v našem kontextu správné.

Výsledky práce výzkumných týmů špičkových počítačových laboratoří ve světě lze číst na stránkách Internetu předních počítačových firem. Přestože jich není mnoho, řešení se objevují, implementují a dnes už i prakticky provozují.

Plan 9<sup>1</sup> je jedním z možných řešení. Podmínky při jeho vzniku byly přitom jednoznačné a vycházely z myšlenek UNIXu: otevřenost, přenositelnost, dostupnost. Historicky byla navíc podmínkou kompatibilita s používanými operačními systémy, tj. shoda s definicemi standardu POSIX. Plan 9 je otevřený, protože je principiálně schopný přizpůsobovat se datovým, síťovým a jiným formátům jiných operačních systémů. Je přenositelný, protože je programován v jazyce C. Strojově závislá část je pak programována pro různé platformy různé výpočetní síly, a to od PC typu notebook až po superpočítače typu SGI Origin. Plan 9 je dostupný, protože je možné používat jej včetně zdrojových textů. Jeho cena ve verzi pro PC je přibližně 350\$ (sada obsahuje 2 knihy, CD a 4 diskety). Demonstrační verze je dostupná v Internetu na adrese <http://plan9.att.com>, kde lze najít i další obsáhlou dokumentaci a poznámky.

## 11.1 Základní schéma

Plan 9 je operační systém, jehož instalace je určena pro počítačovou síť. Pro přístup k operačnímu systému používá uživatel počítačové síť *terminál*. Tento výraz zde označuje samostatně provozuschopný výpočetní prostředek pro (samozřejmě grafickou) práci uživatele, jako je osobní počítač nebo pracovní stanice. Hardware, který terminál spojuje se sítí, je např. Ethernet nebo ISDN. Terminál přitom může být umístěn prakticky v libovolné reálné vzdálenosti od ostatní sítě, ať už v kanceláři firmy nebo v domácím prostředí. Základem sítě jsou výkonné počítače se správou sdílených disků nebo poskytování výpočetních cyklů víceprocesorových architektur. Na rozdíl od způsobu v současnosti používaného, kdy je respektován počítač jako jednotka v síti určená trvale pro určitou osobu, Plan 9 (jak bychom od nástupce UNIXu očekávali) personifikuje terminál v okamžiku přihlášení některé osoby, a to pouze na dobu sezení. Přístup uživatele je skutečně síťový. Veškerý hardware (včetně terminálů) je uvažován jako hardware sítě, který operační systém poskytuje uživatelům. V terminologii Planu 9 jsou to výpočetní zdroje, které jsou identifikovány prostřednictvím systému souborů. Výpočetní zdroje tak pro uživatele představují dostupný výpočetní systém, který je takto jednak personifikován a který je současně sdílen všemi právě přihlášenými uživateli sítě.

Plan 9 je založen na třech základních principech, kterými je systém souborů, protokol 9P a spojování služeb serverů do jednoho jmenného prostoru. Výpočetní zdroje sítě jsou reprezentovány soubory v hierarchickém systému souborů. Přístup uživatele k systému souborů je syntakticky shodný s prostředím v UNIXu. Výpočetní zdroje jsou procesům přístupné pomocí jednotného komunikačního protokolu 9P. Vzájemná komunikace procesů vzdálených i místních je prostřednictvím protokolu 9P založena na technologii klient - server. Přihlášený uživatel má k dispozici výpočetní zdroje sítě, které spojí (ručně nebo automatizovaně ihned po přihlášení) s jemu přiřazeným hierarchickým systémem souborů. Uživatel má tak k dispozici vlastní síťové prostředí, které je nezávislé na terminálu, který právě používá.

Operační systém Plan 9 umožňuje pracovat v grafickém prostředí X, ale současně vlastní a prosazuje svůj grafický podsystém prostředí oken s názvem 8", který je jednodušší, pracuje s libovolným bitově mapovaným grafickým rozhraním a práce s ním je jednoduchá (používá např. všechny ustálené funkce



menu typu pop up, funkce cut and paste atd.). Plan 9 ale také zachovává vrstvenou strukturu přístupu uživatele k operačnímu systému (viz kap 1). Shell, který zde reprezentuje příkazový řádek, má název **rc**, jeho definice odpovídá standardu POSIX a jeho prostřednictvím jsou přístupné všechny externí příkazy standardizované v POSIXu (jak je známe z UNIXu). Kódování znaků v operačním systému Plan 9 odpovídá standardu Unicode. Migrace programů z prostředí UNIXu je zajištěna na úrovni zdrojového kódu a je pečlivě dokumentována. Vzhledem k podpoře prostředí X je však mnohdy zbytečná.

Programování v prostředí Planu 9 je možné v několika jazycích. Samotný operační systém je psán v jazyce C (dialektu ANSI C). Programátor musí běžně zvládat principy konkurentního a sítového programování, které je zastoupeno jazykem Alef (v syntaxi podobnému jazyku C). Běžně se také využívá programovacích možností shellu **rc**. Různorodost procesorů a jejich architektur v rámci těžce instalace Planu 9 musí být řešena v kontextu generace proveditelného kódu programů. Prostředí programování v Planu 9 proto obsahuje program **mk**, který generuje kód pro odpovídající procesor, na němž bude program interpretován procesem. Podpora na úrovni programování je rovněž v kontextu používání standardu Unicode.

Protokol komunikace systému souborů (a tím i zdrojů sítě) je 9P. Sítové rozhraní je tedy reprezentováno systémem souborů, který je trvale přítomen v jádru každé výpočetní jednotky sítě. Pro zajištění přístupu do sítě s protokolem IP (tj. Internetu) používá Plan 9 nově navržený protokol IL (Internet Link).

Přístupová práva k souborům, rozlišení uživatelů a celková bezpečnost provozu je na úrovni popsané v této knize v úvodu kap. 9. Podrobnosti jak k bezpečnosti, tak k ostatním částem operačního systému Plan 9 uvedeme v následujících článcích této kapitoly.

Provozní dokumentace operačního systému Plan 9 přebírá strukturu z UNIXu. Je rozdělena na dva svazky, které jsou označeny jako manuály (Manuals) a dokumenty (Documents). Manuály mají 9 sekcí, každá sekce pak obsahuje abecedně seřazené názvy jednotlivých komponent popisovaného tématu.

Sekce jsou označeny:

Příkazy (Commands)

Volání jádra a funkce (System and library calls),

Periferie (Devices),

Souborové servery (File servers),

9P, protokol systému souborů v Planu 9 (Plan 9 File Protocol, 9P),

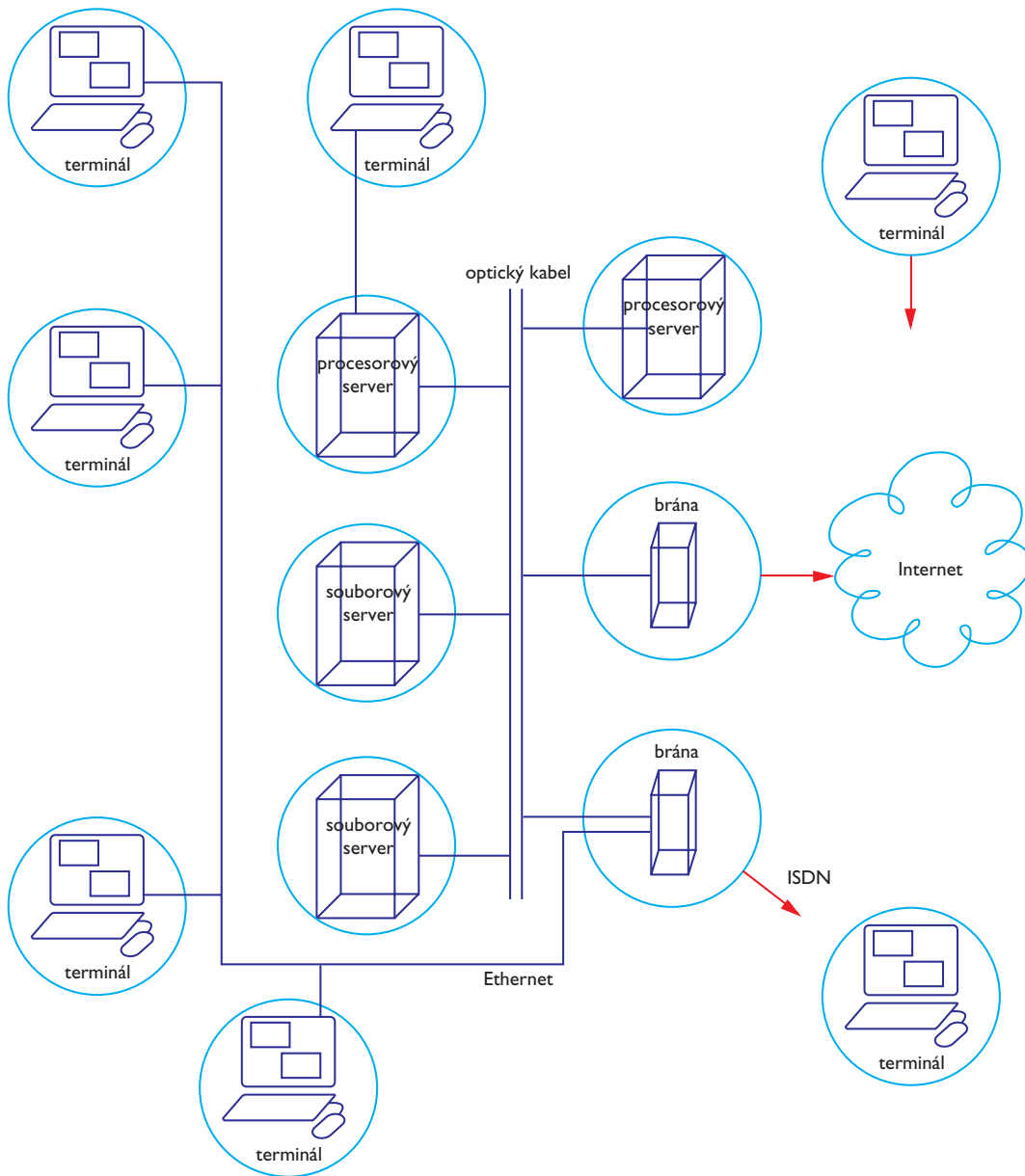
Formáty souborů a další, jako např. systémy maker pro **troff** (File formats and other miscellany such as **troff** macro packages),

Databáze (Databases),

Správa systému (System administration),

Software pro podporu grafiky (Raster image software).

Každá sekce přitom obsahuje stránku **intro**, stručný přehled obsahu sekce. Jako v UNIXu i zde jsou stránky jednotlivých sekcí uživatelům přístupné pomocí příkazu **man**. Dokumenty jsou souborem článků, které obsahují souvislý popis určité charakteristické části operačního systému. V současné době jsou dokumenty rozděleny na šest následujících témat.



Obr. 11.1 Příklad instalace síťového operačního systému Plan 9

**Úvod** (Introduction). Obsahuje např. úvodní přehledový dokument op. systému, používání jmenných prostorů (name spaces) nebo organizaci sítí.

**Programování** (Programming), dokumenty se vztahují k implementovaným programovacím jazykům, jako je C, APE (ANSI/POSIX Environment), Alef, Acid, popis assembleru atd.

**Uživatelská rozhraní** (User Interfaces) obsahuje dokumenty o používání grafického rozhraní 8", popis shellu **rc** nebo vývojového prostředí programátorů Acme.

**Implementace** (Implementation). Dokumenty popisují vnitřní strukturu operačního systému, ale i poznámky k implementaci např. jazyka C.

**Různé** (Miscellany). Např. popis práce návrhu paralelních a distribuovaných aplikací (tzv. SPIN), popis tiskového serveru atd.

**Instalace** (Installation). Dokumentace odlišností instalace na různé platformy hardwaru, nároky na hardware atd.

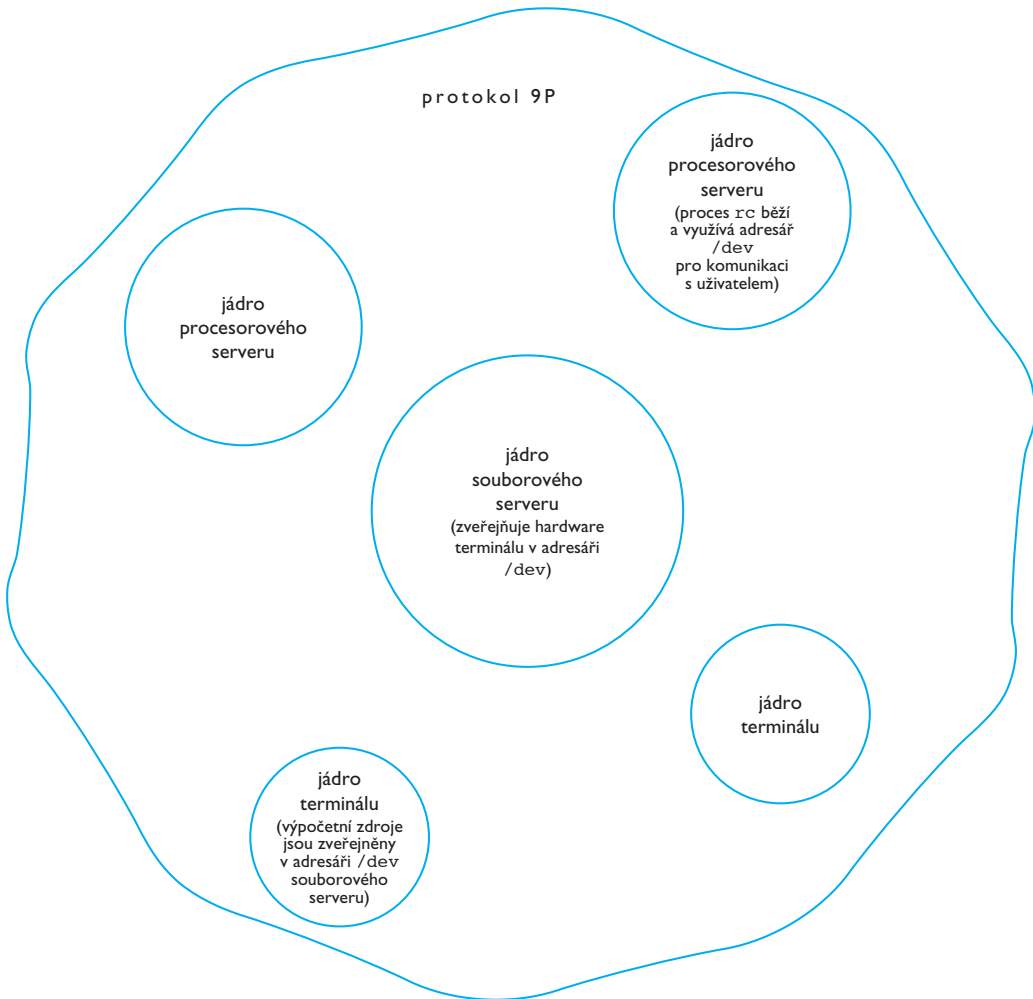
## 11.2 Souborové servery

Typická instalace operačního systému Plan 9 vedle instalovaných terminálů především zahrnuje instalaci výkonných počítačů pro poskytování diskového prostoru, tzv. *souborové servery* (File Servers), a pro poskytování výpočetního výkonu, tzv. *procesorové servery* (CPU Servers). Běžná, ale ne bezpodmínečná je také přítomnost brány do Internetu. Viz obr. 11.1.

Souborový server je výkonný centrální počítač sítě, jehož úkolem je poskytovat hierarchický systém souborů prostřednictvím protokolu 9P. Souborový systém je instalován bez dalších služeb operačního systému. Jeho výkon a poskytované souborové služby jsou dostupné pouze prostřednictvím sítě, a jak je vidět i z obr. 11.1, počítač současně neobsahuje žádný terminál. Při výběru hardwaru pro takový souborový server je kladen důraz především na rychlou obsluhu disků. Kromě jádra s protokolem 9P je spuštěno pouze několik dále trvale přítomných procesů, které zajišťují rychlou realizaci požadavků přístupu uživatelů k datům. Na obr. 11.1 jsou uvedeny takové servery dva, což odpovídá praxi, ale z pohledu údržby a správného chodu sítě se doporučuje používat pro jednu instalaci jeden výkonný souborový server.

Souborový server je implementován ve třech úrovních. První úroveň, k uživatelům nejbližší, je disková vyrovnávací paměť (o velikosti řádově stovek MB). Druhou úroveň jsou pracovní disky (desítky GB) a třetí je rozsáhlá disková paměť se zaměřením na časté čtení a občasný zápis (stovky GB, používají se zásobníky magnetooptických disků, tzv. jukebox s technologií WORM, tj. write-ononce-read-many). Pracovní disky jsou vyrovnávací paměti rozsáhlých disků. Velikost kapacity jednotlivých úrovní diskové paměti je dána vzájemným poměrem a požadovaným počtem současně připojených terminálů a jejich aktivit (analogicky konfigurace systémové vyrovnávací paměti v UNIXu). Důraz je kladen na zálohu dat, která nezávisle na zálohy vytvářené jednotlivými uživateli probíhá automatizovaně (obvykle v časných ranních hodinách). V době vytváření zálohy dat je systém souborů odstaven od provozu operačního systému a záloha je provedena do některého adresáře rozsáhlé diskové paměti. Záloha je tedy prováděna do vlastního systému souborů, používá se např. adresář `/n/dump/1998/0629` pro označení zálohy dat daného dne. Výsledkem práce systému vyrovnávacích pamětí je pak vytvoření zálohy v průběhu několika minut, přestože samotný přepis na disky třetí úrovně probíhá na pozadí

i několik hodin. Data zálohy jsou pak dostupná všem uživatelům v přímo použitelné podobě. Jsou pouze chráněna proti zápisu nastavením odpovídajících přístupových práv. Při současných neustále rostoucích kapacitách diskových pamětí se už dnes odhaduje využitelnost kapacity diskové paměti třetí úrovně na dobu desetiletí. Plan 9, tak jako kdysi UNIX, předpokládá neustále rostoucí kapacitu médií pro záznam informací.



Obr. 11.2 Terminál využívá procesorový server prostřednictvím protokolu 9P

Souborové servery plní také další důležité funkce provozu operačního systému. Jak bylo řečeno, výpočetní zdroje sítě jsou poskytovány prostřednictvím systému souborů. Znamená to, že jako soubor jsou viditelné i všechny periferie, které uživatel potřebuje pro práci v operačním systému a souborový server musí poskytovat i takový přístup. Takovým službám souborových serverů říkáme *neobvyklé* (unusual services), a serverům, které je poskytují, *neobvyklé souborové servery* (unusual file servers). Které jádro operačního systému neobvyklý souborový server realizuje, není podstatné. Může se jednat o místní službu nebo o službu vzdáleného souborového serveru (v případě bezdiskové stanice). Jako dobrý příklad slouží práce s komponentami grafického prostředí terminálu. Proces klientu se odkazuje na soubory v adresáři `/dev`, jako jsou `mouse` (myš), `screen` (obrazovka), `cons` (konzola), `bitblt` (bitově mapované grafické rozhraní), které mu slouží jako vstupní a výstupní zařízení. Grafické prostředí 8" je souborový server, který tyto soubory poskytuje procesům terminálů. Poskytuje tedy klientům soubory pro práci s odpovídajícím grafickým hardwarem, který je síťovým zdrojem. Přestože odkazy klientu jsou na soubory uvedených jmen, každý klient má v adresáři `/dev` pro manipulaci se svým oknem k dispozici vlastní sadu souborů (analogie se speciálním souborem `/dev/tty` v UNIXu). Znova si připomeňme, hardware terminálu je jeden z výpočetních zdrojů celé sítě a tedy (umožní-li to uživatel u terminálu) může být využíván klienty celé sítě, tudíž i klienty jiných terminálů nebo serverů. Každý terminál má možnost uvolnit své výpočetní zdroje do sítě a mnohdy ve vlastním zájmu. Uživatelský proces **exportfs** poskytuje výpočetní zdroje. Pro zpřístupnění zdroje se pak používá příkaz **import** (startuje **exportfs**) ve vzdáleném uzlu. Např.

```
import animal /proc
```

zpřístupní informace o běžících procesech v souborovém serveru `animal`.

Procesorový server je jeden z neobvyklých souborových serverů. Příkaz **cpu** připojí terminál k odpovídajícímu serveru. V tomto případě iniciuje příkaz **cpu** ve vzdáleném uzlu sítě start procesu **exportfs** a místní souborový server pro příchozí terminál vytvoří sadu souborů v adresáři `/dev` pro práci s v/v periferiemi terminálu. Obvykle je také startován proces shellu **rc**, kterým je uživateli zpřístupněna práce na daném procesorovém serveru. Grafické prostředí terminálu (nebo pouze jeho část) je tak přepnuto pro práci procesů v jiné části sítě. Na rozdíl např. od práce síťové aplikace **telnet** je zde jádrem vzdáleného systému využít výpočetní zdroj hardwaru příchozího terminálu, se kterým klienty v tomto uzlu pracují. Obtížně vyjádřitelnou abstrakci se snaží schematizovat obr. 11.2.

### 11.3 Provoz sítě, Internet

Z dosavadního vyplývá, že obecným protokolem práce sítě s instalovaným operačním systémem je 9P, a to vzhledem k tomu, že všechny výpočetní zdroje jsou viditelné jako soubory a 9P je protokol souborových serverů. Přístup k sítím jiného typu, např. s protokolem IP, je zajišťován jako výpočetní zdroj. Spojení typu TCP je tak např. aktivováno manipulací se soubory v adresáři `/net/tcp`. Jeho obsahem je soubor `clone` a podadresáře se jmény postupně 0, 1, 2 atd., jejichž obsah je přiřazen vždy novému síťovému spojení. Každý takový podadresář obsahuje pak soubory se jmény `ctl`, `data`, `listen`, `local`, `remote` a `status`, se kterými síťová aplikace pracuje. Odpovídající adresář je přiřazen novému síťovému spojení otevřením souboru `/net/tcp/clone`. Uživatel aktivuje síťové spojení např. pro aplikaci **telnet** způsobem

```
% connect 147.229.113.10!23
```

(shell **rc** se ohlašuje znakem **%**). V síti IP na uvedené adrese je na portu č. 23 osloven odpovídající server **telnetd**. Po navázání spojení využívá aplikace **telnet** soubor **data** pro přenos dat (zápisem nebo čtením jeho obsahu). Strana serveru je v Planu 9 zajištěna démonem s názvem **listener** a lze jej startovat příkazem

```
% announce 23
```

Zajímavé je také používání protokolu FTP. Plan 9 zpřístupní lokality FTP sítě IP prostřednictvím souborového serveru **ftpts** do adresáře `/n/ftp`. Obsah tohoto adresáře je pak distribuován protokolem 9P.

Obdobným způsobem je implementováno síťové rozhraní jiného typu. Jádru Planu 9 je vybaveno mechanismem používání zásobníku modulů pro vytvoření rozhraní mezi procesem a hardwarem, tzv.

PROUDY (STREAMS), které jsou převzaty z operačního systému UNIX. My jsme se jimi zabývali v kap. 6. Procesy sítí jiných typů je tak možné implementovat způsobem, který je využíván v UNIXu.

Přestože Plan 9 zahrnuje implementaci protokolu TCP a UDP, pro využívání sítě IP nabízí také vlastní síťový protokol IL (Internet Link). Tento protokol byl vyvinut pro průchod paketů protokolu 9P sítě IP. Připojení terminálu k síti s Planem 9 prostřednictvím Internetu je sice možné i pomocí protokolu TCP/IP, ale Plan 9 nevyžaduje tak náročnou režii, kterou disponuje TCP/IP. Protokol IL je tedy menší, rychlejší, celkově výkonnější. Jeho integrace v jádru Planu 9 tak umožňuje používat současné standardy Internetu jako prostředníka k přenosu dat při práci terminálů. Na obr. 11.1 je takový terminál zakreslen.

## 11.4 Programování

Plan 9 dává programátorovi k dispozici několik programovacích jazyků. Jsou to ty, ve kterých byl samotný operační systém programován. Podobně jako v případě UNIXu je možné v shellu **rc** programovat tzv. scénáře (scripts), ve kterých je realizována řada nástrojů Planu 9. Moduly, které je nutné implementovat metodami paralelního programování, jsou programovány v jazyku Alef, který byl pro Plan 9 zvláště vyvinut, přestože takových modulů v Planu 9 není mnoho. Největší část je ale programována i zde v jazyku C a převážná většina programů prostředí uživatele podle POSIXu byla takto převzata z UNIXu a přeložena do prostředí Planu 9.

Jazyk C je dialektem standardu ANSI C, tzv. dialekt Plan 9 C, který je naopak zúžením ANSI C<sup>2</sup>. Zúžení je např. překvapivě provedeno na vynechání konstrukce `#if` při používání preprocesoru.

Konstrukce `#ifdef` je zahrnuta, ale ve vlastním kódu operačního systému je použita jen výjimečně. Programátoři operačního systému to vysvětlují dosažením většího přehledu nad údržbou zdrojových textů, kdy jsou např. funkce grafických knihoven rozděleny na jednotlivé balíky knihoven vždy podle určitého grafického rozhraní. Standardní knihovna je `libc`. Její definice pak programátor získá využitím hlavičkového souboru `<libc.h>`, ale definice některých funkcí nemusí zcela odpovídat ANSI C. Jak standard ANSI C, tak POSIX je ovšem z pohledu zdola nahoru přísně respektován, např. provedené změny v POSIXu (volání jádra `creat` bylo změněno na `create` atp.) jsou plně v Planu 9 respektovány. Výjimkou je používání kódu znakové reprezentace. Na rozdíl od ANSI C používá Plan 9 C pro vnitřní kódování znaků 16 bitový Unicode (ISO 10646, viz [Unicode96]), čímž podporuje lokalizaci operačního systému do všech jazyků a umožňuje programátorům vytvářet aplikace pracující v jejich přirozeném jazyce nebo kombinovat programy pro použití jejich jednotlivých modulů v různých přirozených jazycích (např. u několikajazyčných verzích nebo u softwaru pro překladatelské služby).

Programátor ovšem vytváří programy pro počítačovou síť, která je složena z různých typů architektur CPU. Podobně jako jádro operačního systému i nově programovaný software musí být možné sestavovat s různým binárním kódem, který pak interpretuje vždy jiný procesor. Plan 9 proto programátorům poskytuje kompilátor např. jazyka C, který je uzpůsoben generovat binární mezikód pro různé typy procesorů. Rovněž tak pro sestavení souborů s binárním mezikódem do výsledného proveditelného programu je poskytována řada sestavujících programů pro různé procesorové architektury. Takový křížový kompilátor (cross compiler) programátor spouští v příkazovém řádku vždy podle typu procesoru samostatným příkazem. Např. pro platformu procesorů Intel x86 má kompilátor jazyka C jméno **8c**, kompilátor jazyka Alef má jméno příkazu **8a1**. Přeložený mezikód má příponu **.8** (tedy nikoliv **.o**). Sestavující program má jméno příkazu **81**. Samotné kompilátory jsou pochopitelně proveditelné na různých architekturách a jejich shodný zdrojový kód zajišťuje shodný způsob kompilace. Výsledný mezikód má shodnou strukturu (je vyžadována operačním systémem), ale jiný obsah (je vyžadován typem procesoru). Prostředek podobný nástroji **make** z UNIXu je **mk**, který používá odpovídající kompilátor a sestavující program podle obsahu proměnné prostředí `$cputype` a `$objtype`.

Metody paralelního programování je v Planu 9 možné realizovat jednak vybranou sadou volání jádra a používat jazyk C a jednak je možné využívat nově definovaný programovací jazyk Alef. Požadavky paralelního programování jsou tři: sdílení výpočetních zdrojů mezi procesy, přístup k řízení přidělování procesoru a vzájemná synchronizace procesů. V Planu 9 používá programátor pro vytvoření nového procesu volání jádra `rfork`, v jehož jediném argumentu (je přijímán jádrem jako vektor bitů) rodič zadává, jaký vztah k němu bude vznikající dětský proces mít, tj. zda výpočetní zdroje rodiče jsou sdíleny, kopírovány nebo vytvořeny nově. Výpočetními zdroji se zde rozumí jmenný prostor procesu, prostředí provádění, tabulka deskriptorů, paměťové segmenty a poznámky (notes, obdoba signálů z UNIXu). Speciální volání jádra je také pro sdílenou paměť (`segattach`) a synchronizování procesů (`randezvous`). Stranou nemusí programátor ponechat ani klasické metody používání komunikace procesů, jak jsme je v kontextu normy POSIX uvedli v rámci kap. 4. Programovací jazyk Alef ovšem nabízí více komfortní práci paralelního programování a při realizaci paralelních programových systémů by měl programátor tento prostředek používat.

Více se o metodách programování v operačním systému Plan 9 včetně příkladů lze dočíst v provozní dokumentaci [Plan995].

## II.5 Bezpečnost

Výpočetní zdroje viditelné prostřednictvím souborových serverů jsou uživatelům viditelné v podobné syntaxi jako v UNIXu (nebo podle POSIXu). Prostřednictvím známého příkazu **ls** dostáváme výpis požadovaného adresáře včetně přístupových práv a vlastnictví souborů. I zde je vlastnictví souborů evidováno na jednotlivé uživatele, kteří jsou sdružováni do skupin. Evidence uživatelů a jejich skupin je centrální, v základním souborovém serveru sítě. Skupiny, do kterých jsou uživatelé sdružováni, jsou označovány vždy jménem některého z uživatelů. Takto je skupina ve vlastnictví určitého uživatele, *vedoucího skupiny* (group leader), zatímco ostatní jsou *členové skupiny* (members of that group). Ve výpisu adresáře je i zde viditelný vlastník a skupina souboru a trojice přístupových práv ve tvaru **rwrxrwxrwx**.

Plan 9 nemá superuživatele. Pro údržbu souborových serverů je k dispozici např. uživatel `adm`, jehož práva jsou rozšířena pouze na možnost provádění správcovských operací týkajících se např. zálohování dat atp. Podobně je také ustanoven zvláštní uživatel pro přístup k procesorovým serverům (když je třeba např. některý proces zrušit). Přístup k souborům má vždy pouze majitel souboru a nikdo jiný. Na rozdíl od UNIXu poskytuje Plan 9 naopak uživatele `none`, jehož vstup do systému není chráněn žádným heslem. V dokumentaci se hovoří o analogii s anonymním FTP v Internetu. Protože přístupová práva uživatele `none` jsou podobně zúžena, jsou mu přístupné soubory pouze veřejného charakteru. Uživatel `none` je ovšem používán jako výchozí pro zjišťování autenticity (viz následující text).

Prověřování autenticity v operačním systému Plan 9 probíhá podobně jako při aplikaci bezpečnostního systému Kerberos (viz kap. 9). Uživatel, který vznáší požadavek přístupu k výpočetnímu zdroji, je nazýván *klient* (client), uživatel, který přístup poskytuje, pak *server* (server). Právo přístupu je prověřováno zprávou typu *přístup* (attach) v rámci protokolu 9P. Přístupová práva jsou kvalifikována v rámci uživatelů, ne uzlů sítě. Současně může být také v příštím okamžiku vztah uživatelů opačný, z klientu se stává server a naopak. Pro klíčování se používá šifrovací algoritmus DES (viz kap. 9). Pověření autenticity je schopnost uživatele šifrovat a rozšifrovat zprávy protokolu 9P typu *výzva* (challenge). Pokud server požaduje autentický přístup, obsah každého paketu protokolu 9P pro tento přístup je šifrovaný. Databáze klíčů DES je uložena na serverech a je inicializována v okamžiku startu operačního systému. Terminál přitom generuje klíč podle hesla zadaného uživatelem při vstupu do operačního systému. Protokol při vzniku komunikace klientu a serveru spočívá v kontaktu zvláštního serveru autenticity pro získání šifrovaného lístku (získá se z tajného klíče klientu nebo serveru), který obsahuje nový konverzační klíč. Každá strana pak rozšifruje lístek a získá tak nový klíč pro šifrování zpráv typu *výzva*.

V případě přístupu k procesorovému serveru je praktické provedení následující. Jádro, které zachytí (prostřednictvím naslouchajícího procesu) požadavek přístupu, vytvoří proces, který je ve vlastnictví uživatele `none`. Teprve po prověření požadovaného přístupu si proces může proměnit vlastníka na požadovaného. V opačném případě je přístup k serveru odmítnut.

## 11.6 Konfigurace a správa

Obtížnost instalace operačního systému Plan 9 závisí na velikosti sítě, na kterou je instalován. Přesto je jeho instalace a údržba co nejvíce sjednocena. Všechny uzly sítě totiž pracují pod stejným jádrem. Odlišná je pak konfigurace např. uzlů se souborovými servery a uzly, které jsou terminály (ať už jde o notebook nebo o pracovní stanici). Situaci pochopitelně komplikuje nutnost mít k dispozici varianty jádra pro různé architektury CPU. Možnost generace jádra pro nový typ procesoru však byla jedna z podmínek existence operačního systému Plan 9. Samotná údržba je co nejvíce centralizována, a to do uzlů se souborovými servery. Jeden z uzlů souborových serverů je hlavní. Tento uzel pak distribuuje výchozí strom systému souborů a z jeho aktivity vychází akce pravidelné údržby, jako je např. již zmiňovaná záloha systému souborů. Snaha přenášet všechny akce údržby na centrální souborový server vychází z myšlenky co nejvíce usnadnit provoz terminálům tak, aby se po jejich zapnutí mohl uživatel prokázat svým jménem a heslem a pracovat na svých datech v odpovídající aplikaci. Takto je např. centralizována evidence uživatelů a skupin. Existuje pouze jedna tabulka takové evidence, jejíž modifikace je modifikací platnou rázem pro celou síť (nebo operační systém, chcete-li). Informace síťového provozu jsou centralizovány v adresáři `/lib/ndb` a změny v jeho struktuře a obsahu znamenají změny v konfi-



guraci provozované sítě. Připojení nového terminálu pak probíhá přidáním souboru do tohoto adresáře, který je tak ihned viditelný v celé síti. Nová stanice je pak připojena jednoduše tak, že je hardwarově spojena se sítí, zapnuta a pomocí protokolu BOOTP a TFTP je zavedeno a spuštěno jádro.

Instalace síťového operačního systému, jako je např. Plan 9, je současným trendem dalšího vývoje softwaru operačních systémů. Třeba i slabý prostředek výpočetní techniky (jako je starší PC) může být využit pro práci uživatelů, protože takový terminál je pouze prostředek pro přístup k výpočetní síle celé sítě. Dominantní zůstává výkon souborových serverů, který musí odpovídat požadovanému výkonu provozovaného informačního systému. Hlavní výhoda ovšem stále zůstává ve využití výkonu sítě jako celku, nikoliv pouze výkonu jednotlivých uzlů sítě.

<sup>1</sup> Operační systém Plan 9 byl pojmenován podle jednoho z prvních filmů science fiction signovaného nejhoršího režiséra Hollywoodu Eda Wooda z poloviny 50. let. Celý název filmu je Plan 9 from Outer Space a jeho hlavní myšlenkou je ožívání mrtvol a jejich vystupování z hrobů na pokyn přicházejících mimozemšťanů. Programové celky operačního systému Plan 9 jsou pojmenovány podle děje, postav a dekorací filmu.

<sup>2</sup> Programy psané v Plan 9 C dialekt jsou tedy přeložitelné překladačem podle ANSI C, Plan 9 C dialekt je tak se standardem kompatibilní.



# PŘÍLOHY

## A Volání jádra

Příloha obsahuje všechna volání jádra operačního systému UNIX, která jsou nějakým způsobem použita v knize. Na odpovídající straně textu knihy je obvykle uveden jejich formát podle POSIXu nebo SVID. Úplný seznam všech volání jádra, která konkrétní operační systém nabízí, nalezne čtenář ve svazku provozní dokumentace s označením ( 2 ).

jméno	popis ... str.
access	ověření přístupu procesu k souboru ... 29
accept	aktivace síťového spojení strany serveru v Berkeley sockets ... 237
acct	zapnutí účtování ... 180
acl	DAC, práce s atributy ACL u souboru ... 312
aclipc	DAC, práce s atributy ACL u fronty zpráv, sdílené paměti nebo semaforu ... 312
aclsort	DAC, nastavení správného pořadí položek ACL ... 312
alarm	plánování příchodu signálu SIGALRM ... 136
auditbuf	AUDIT, práce s vyrovnávací pamětí ... 315
auditctl	AUDIT, řízení a výpis protokolů účtování ... 315
audidmp	AUDIT, zápis záznamu účtování do vyrovnávací paměti ... 315
auditevt	AUDIT, nastavení nebo získání příznaků účtování ... 315
auditlog	AUDIT, nastavení nebo získání atributů souboru evidence účtování ... 315
brk	posunutí hranice datového segmentu na danou hodnotu, na úkor zásobníku ... 42
bind	přřazení čísla portu a síťové adresy místní poloviční asociace, v Berkeley sockets ... 237
chdir	změna pracovního adresáře procesu ... 30
chmod	změna přístupových práv souboru ... 29
chown	změna vlastníka a skupiny souboru ... 84
chroot	změna kořenového adresáře procesu ... 30
close	uzavření kanálu procesu (tabulky otevřených souborů) ... 83
connect	inicializace síťového spojení v Berkeley sockets ... 237
creat	vytvoření souboru ... 84
devalloc	MAC, nastavení nebo zjištění bezpečnostních atributů periferie ... 315
devdealloc	MAC, vyjmutí periferie z bezpečnostního podsystemu ... 315
devstat	MAC, práce s bezpečnostními atributy periferie ... 315
dup	vytvoření kopie kanálu (tabulky otevřených souborů) ... 38
exec	výměna řídicího programu běžícího procesu za jiný ... 33
exit	ukončení procesu ... 21
fcntl	řídicí operace nad otevřeným souborem (např. zamykání souboru) ... 91
filepriv	bezpečnostní podsystem TFM, řízení přístupu k souboru ... 315
fork	vytvoření procesu ... 21
free	uvolnění operační paměti získané pomocí malloc ... 42
fstat	získání obsahu i-uzlu otevřeného souboru ... 86
fsync	aktualizace otevřeného souboru v systémové vyrovnávací paměti na disk ... 120
getcwd	pracovní adresář procesu ... 82
getegid	získání identifikace efektivní skupiny procesu ... 29
geteuid	získání identifikace efektivního vlastníka procesu ... 28
getgid	získání identifikace reálné skupiny procesu ... 28
gethostname	získání jmenné identifikace operačního systému ... 249
gethostid	získání číselné identifikace operačního systému ... 249

getitimer	získání hodnot časových intervalů plánování procesu ... 55
getmsg	čtení zprávy z PROUDU ... 200
getpmsg	čtení zprávy z PROUDU ... 200
getpggrp	získání identifikace procesu vedoucího skupiny ... 169
getpid	získání identifikace procesu PID ... 26
getppid	získání identifikace rodiče procesu PPID ... 26
getpriority	získání hodnoty výchozí priority jiného procesu ... 57
getuid	získání identifikace reálného vlastníka procesu ... 28
getsid	získání identifikace vedoucího procesu sezení ... 170
gettimeofday	získání data a času reálných hodin stroje s přesností na mikrosekundy ... 46
getsockopt	získání vlastností schránky v Berkeley sockets ... 239
ioctl	řídící operace nad periferií ... 187
kill	odeslání signálu jinému procesu ... 27
link	vytvoření nového odkazu na i-uzel souboru téhož svazku ... 89
listen	zajištění výhradního přístupu k vytvořené schránce v Berkeley sockets ... 237
lseek	přesun ukazatele otevřeného souboru ... 103
lvdome	MAC, zjištění vztahu dvou bezpečnostních úrovní ... 315
lvlequal	MAC, zjištění rovnosti dvou bezpečnostních úrovní ... 315
lvlfle	MAC, nastavení nebo zjištění bezpečnostní úrovně souboru ... 315
lvlin	MAC, převod textové specifikace úrovně do vnitřního formátu ... 315
lvlipc	MAC, manipulace s prostředky komunikace mezi procesy IPC ... 315
lvhout	MAC, převod specifikace úrovně vnitřního formátu na textovou ... 315
lvlproc	MAC, nastavení nebo zjištění úrovně procesu ... 315
lvlvalid	MAC, test správnosti úrovně ... 315
lvlvfs	MAC, nastavení nebo zjištění úrovně připojeného svazku ... 315
memcntl	práce se stránkami procesu v operační paměti ... 54
malloc	připojení další, procesem adresovatelné, operační paměti ... 42
mkdir	vytvoření adresáře ... 88
mkfifo	vytvoření pojmenované roury ... 140
mknod	vytvoření i-uzlu ... 99
mkmlid	MAC, vytvoření adresáře více úrovní ... 315
mlmode	MAC, nastavení nebo získání režimu adresáře více úrovní pro proces ... 315
mlock	zamknutí regionu operační paměti ... 54
mmmap	mapování regionu operační paměti ... 54
munlock	odemknutí regionu operační paměti ... 54
munmap	ukončení mapování regionu operační paměti ... 54
mount	připojení svazku ... 114
mq_close	odpojení procesu od fronty zpráv ... 151
mq_getattr	získání atributů fronty zpráv ... 151
mq_notify	nastavení procesu na příchod určité zprávy ... 151
mq_open	vytvoření fronty zpráv nebo připojení procesu k frontě zpráv ... 151
mq_receive	odebrání zprávy z fronty zpráv ... 151
mq_send	odeslání zprávy do fronty zpráv ... 151
mq_setattr	nastavení atributů fronty zpráv ... 151
mq_unlink	zrušení fronty zpráv ... 151
msgctl	řídící operace nad frontou zpráv (zrušení fronty zpráv) ... 149
msgget	vytvoření nebo zpřístupnění fronty zpráv ... 148
msgrcv	odebrání zprávy z fronty zpráv ... 149
msgsnd	odeslání zprávy do fronty zpráv ... 148
msync	aktualizace regionu operační paměti v systémové vyrovnávací paměti na disk ... 54

nice	změna výchozí priority procesu ... 57
open	otevření souboru ... 82
pause	pozastavení procesu (do příchodu signálu) ... 136
pipe	vytvoření roury ... 38
plock	zamykání procesu v operační paměti ... 52
poll	řízení toku dat v několika PROUDECH ... 201
prcntl	plánování procesů (reálného času) ... 59
procpriv	TFM, řízení přístupových práv procesu ... 315
procpv1	TFM, řízení přístupových práv procesu ... 315
profil	souhrnné informace o dětském procesu pro účely statického ladění ... 49
ptrace	sledování činnosti dětského procesu pro účely dynamického ladění ... 47
putmsg	zápis zprávy do PROUDU ... 200
putpmsg	zápis zprávy do PROUDU ... 200
read	čtení dat z kanálu (otevřeného souboru) ... 83
readv	alternativa čtení dat síťového spojení v Berkeley sockets ... 237
readlink	čtení symbolického odkazu ... 90
recv	čtení dat ze schránky v Berkeley sockets ... 160
recvmsg	čtení dat ze schránky v Berkeley sockets ... 238
recvfrom	čtení dat ze schránky bez stálého spojení v Berkeley sockets ... 238
rename	změna jména souboru ... 90
rmdir	zrušení prázdného adresáře ... 90
sbrk	posunutí hranice datového segmentu o danou hodnotu, na úkor zásobníku ... 42
semctl	řídící operace nad množinou semaforů (zrušení množiny semaforů) ... 159
semget	vytvoření nebo zpřístupnění množiny semaforů ... 156
sem_close	uzavření pojmenovaného semaforu ... 159
sem_destroy	zrušení bezejmenného semaforu ... 159
sem_getvalue	zjištění hodnoty semaforu ... 159
sem_init	vytvoření bezejmenného semaforu ... 159
sem_open	vytvoření pojmenovaného semaforu nebo otevření již existujícího pojmenovaného semaforu ... 159
sem_post	operace V nad semaforem ... 159
sem_trywait	operace P nad semaforem bez zablokování procesu ... 159
sem_unlink	zrušení pojmenovaného semaforu ... 159
sem_wait	operace P nad semaforem se zablokováním procesu ... 159
semop	operace nad semaforem ... 157
send	zápis dat do schránky v Berkeley sockets ... 160
sendmsg	zápis dat do schránky v Berkeley sockets ... 239
sendto	zápis dat do schránky bez stálého spojení v Berkeley sockets ... 238
setgid	přepnutí na efektivní skupinu procesu ... 29
sethostname	nastavení jmenové identifikace operačního systému ... 249
sethostid	nastavení číselné identifikace operačního systému ... 249
setitimer	nastavování hodnot časových intervalů plánování procesu ... 55
setpgid	změna příslušnosti procesu k jiné skupině ... 170
setpgrp	nastavení procesu jako vedoucího nové skupiny procesů ... 26
setpriority	nastavení výchozí priority jinému procesu ... 57
setsid	nastavení vedoucího procesu skupiny sezení ... 170
setsockopt	manipulace s vlastnostmi schránky v Berkeley sockets ... 239
settimeofday	nastavení data a času reálných hodin stroje s přesností na mikrosekundy ... 47
setuid	přepnutí na efektivního vlastníka procesu ... 29
shm_open	vytvoření sdílené paměti nebo připojení procesu ke sdílené paměti ... 155
shm_unlink	odpojení procesu od sdílené paměti ... 155
shmat	připojení datového segmentu procesu ke sdílené paměti ... 152

shmctl	řídící operace nad sdílenou pamětí (zrušení sdílené paměti) ... 154
shmdt	odpojení datového segmentu procesu od sdílené paměti ... 154
shmget	vytvoření nebo zpřístupnění sdílené paměti ... 151
shutdown	deaktivace síťového spojení v Berkeley sockets (uzavření schránky) ... 160
sigaction	rozšířené řízení reakce na příchod signálu ... 136
signal	řízení reakce na příchod signálu ... 27
sigpending	synchronizace na již doručené, ale nepřijaté signály procesu ... 136
sigqueue	řízení fronty signálů ... 138
sigprocmask	práce s maskou signálů procesu ... 136
sigsuspend	blokování procesu podle nastavené masky signálů ... 136
sigtimedwait	čekání na příchod signálu s časovým omezením ... 138
sigwaitinfo	práce s frontou příšlých, ale nevyřízených signálů ... 138
socket	definice typu síťové domény a síťového spojení v Berkeley sockets ... 235
stat	získání obsahu i-uzlu ... 86
stime	nastavení data a času reálných hodin stroje s přesností na vteřiny ... 45
statvfs	získání podrobných informací o svazku ... 115
swapctl	práce s odkládací oblastí ... 52
sync	aktualizace systémové vyrovnávací paměti na discích ... 120
symlink	vytvoření symbolického odkazu ... 90
time	získání data a času reálných hodin stroje s přesností na vteřiny ... 45
times	měření času života dětských procesů ... 43
umask	nastavení masky přístupových práv při vytváření souboru ... 29
umount	odpojení svazku ... 115
uname	zjištění jména a verze operačního systému ... 249
unlink	zrušení odkazu na i-uzel ... 89
ustat	informace o svazku (volný prostor) ... 115
wait	čekání na dokončení některého dětského procesu ... 36
waitpid	čekání na dokončení určitého dětského procesu ... 36
write	zápis dat do kanálu (otevřeného souboru) ... 83
writew	alternativa zápisu dat síťového spojení v Berkeley sockets ... 237

## B Signály

Definice odpovídají SVID a každý UNIX je obsahuje v `<signal.h>`. Při maskování příchodu signálu lze využít makro `SIG_DFL` pro nastavení implicitního zachování procesu. Pro nastavení ignorování signálu lze použít makro `SIG_IGN`. Následuje seznam možných signálů. Kolonka **implicitně** označuje chování procesu, který signál obdržel v případě, že signál nebyl procesem zachycen nebo ignorován. Signály označené znakem — nejsou uvedeny v POSIXu.

signál	implicitně	důvod
SIGHUP	ukončení	odpojení na řídícím terminálu nebo zánik jeho řídícího procesu
SIGINT	ukončení	přerušení z klávesnice
SIGQUIT	ukončení s <code>core</code>	pokyn ukončení z klávesnice
SIGILL	ukončení s <code>core</code>	neznámá instrukce
– SIGTRAP	ukončení s <code>core</code>	událost určená k ladění
SIGABRT	ukončení s <code>core</code>	výjimečná událost
– SIGEMT	ukončení s <code>core</code>	emulace přerušení, instrukce EMT
SIGFPE	ukončení s <code>core</code>	chyba v pohyblivé řádové čárce
SIGKILL	ukončení	ukončení, nelze zachytit nebo ignorovat
SIGBUS	ukončení s <code>core</code>	chyba sběrnice
SIGSEGV	ukončení s <code>core</code>	zjištěn neoprávněný odkaz do operační paměti
– SIGSYS	ukončení s <code>core</code>	špatný formát volání jádra
SIGPIPE	ukončení	selhání roury, zapisovaná data do roury nikdo nečte
SIGALRM	ukončení	časová synchronizace
SIGTERM	ukončení	pokyn k ukončení činnosti
SIGUSR1	ukončení	první uživatelem definovaný signál
SIGUSR2	ukončení	druhý uživatelem definovaný signál
SIGCHLD	ignorování	dětský proces byl ukončen nebo pozastaven
– SIGPWR	ignorování	selhání zdroje
– SIGWINCH	ignorování	změna velikosti okna
– SIGPOLL	ukončení	signál řízení více PROUDŮ
SIGSTOP	pozastavení	pozastavení, nelze zachytit nebo ignorovat
SIGTSTP	pozastavení	pozastavení z klávesnice
SIGCONT	ignorování	z pozastavení lze pokračovat
SIGTTIN	pozastavení	čekání na standardní vstup
SIGTTOU	pozastavení	čekání na standardní výstup
– SIGXCPU	ukončení s <code>core</code>	překročen omezený časový interval CPU
– SIGXFSZ	ukončení s <code>core</code>	překročena omezená velikost souboru

## C Identifikace

Příloha uvádí identifikace použité v knize. Každá identifikace je v implementaci reprezentována celočíselnou hodnotou.

### označení

### popis

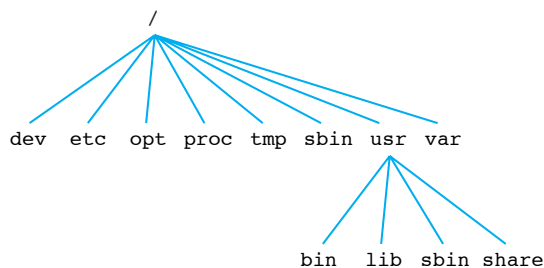
CID	<u>class</u> <u>i</u> dentification, jednoznačná identifikace třídy procesů
EGID	<u>e</u> ffective <u>g</u> roup <u>i</u> dentification, efektivní identifikace skupiny uživatelů
EUID	<u>e</u> ffective <u>u</u> ser <u>i</u> dentification, efektivní identifikace uživatele
GID	<u>g</u> roup <u>i</u> dentification, jednoznačná identifikace skupiny uživatelů
LID	<u>l</u> evel <u>i</u> dentifier, jednoznačná identifikace bezpečnostní úrovně v MAC
PGID	<u>p</u> rocess <u>g</u> roup <u>i</u> dentification, jednoznačná identifikace vedoucího skupiny procesů
PID	<u>p</u> rocess <u>i</u> dentification, jednoznačná identifikace procesu
PPID	<u>p</u> arent <u>p</u> rocess <u>i</u> dentification, jednoznačná identifikace rodičovského procesu
SID	<u>s</u> ession <u>i</u> dentification, jednoznačná identifikace sezení
UID	<u>u</u> ser <u>i</u> dentification, jednoznačná identifikace uživatele



## D Rozložení adresářů

Začátkem 90. let přineslo třetí vydání SVID definici základních jmen adresářů a jejich rozložení v hierarchické struktuře začínající adresářem /. Do té doby používaná jména a umístění adresářů, pokud je výrobci používali v rozporu, byla postupně ve všech implementacích změněna na odpovídající této definici. K následné provozní záměně jmen za jiná dopomohla možnost používat symbolické odkazy. Adresáře původních jmen z důvodů kompatibility byly proměněny na symbolické odkazy vedoucí na adresáře nové. Postupně v dalších verzích jsou pak tyto symbolické odkazy vypouštěny.

Systém adresářů začíná adresářem /, který je také slovně označován jako adresář root, protože je domovským adresářem privilegovaného uživatele root. Na obr. D.1 je uvedena základní struktura, kterou root pokračuje. Výrobci mohou pochopitelně systém adresářů rozšiřovat, přítomnost adresářů uvedených na obr. je ale podmínkou shody s obecnou definicí.



Obr. D.1 Rozložení základních adresářů v UNIX System V

Jednotlivé adresáře pak mají následující význam:

/dev obsahuje speciální soubory,

/etc obsahuje systémové tabulky a některé systémové programy,

/opt je určena pro instalaci aplikací, v rámci instalace, ale i běhu aplikace může pak také být využíván i adresář /var/opt,

/proc slouží k připojení svazku typu proc,

/tmp je adresář s veřejným přístupem pro ukládání dočasných souborů různých aplikací,

/sbin obsahuje základní programy a nástroje systému, které jsou nutné pro práci operačního systému,

/usr je určen pro všechna systémová data, která mohou být sdílena mezi počítači stejné architektury,

`/usr/bin` obsahuje programy systému, aplikace a nástroje, které ale nejsou pro práci systému nezbytné,

`/usr/lib` je adresář knihoven a strojově závislých databází,

`/usr/sbin` je adresářem dalších programů, aplikací a nástrojů,

`/usr/share` je používán pro data, která jsou strojově nezávislá (např. provozní dokumentace) a mohou být sdílena mezi počítači různé architektury,

`/var` je adresářem rozličných systémových dat, jako jsou soubory s evidencí chodu aplikací, vstupu uživatelů do systému, dočasné systémové soubory atp.

Adresáře `/`, `/usr` a `/var` mohou být umístěny na různých svazcích. Adresáře `/dev`, `/etc`, `/proc`, `/tmp`, `/sbin` a `/usr/sbin` jsou určeny pro operační systém. Většina aplikací by nikdy neměla vytvářet soubory v těchto adresářích.

## E Standardní klienty X

Kromě základních klientů manažeru oken jako je **mwm**, **olwm** nebo **twm**, dále klientu přihlášení uživatele do systému **xdm**, nebo emulace terminálu VT102 či Tektronix 4014 **xterm**, jsou v každém systému X přístupny následující klienty X, které jsou rozdělené do uvedených tematických skupin

### Príslušenství pracovní plochy (Desk accessories)

- xbiff** program elektronické pošty,
- xclock**, **oclock** plynoucí časový údaj v podobě analogových nebo digitálních hodin,
- xcalc** kalkulačka,
- xload** monitor vytížení systému,
- xman** zobrazování stránek provozní dokumentace.

### Nastavení klávesnice a obrazovky (Display and keyboard preferences)

- xset** nastavování významu kláves, zvukových upozornění, akcelerace kurzoru nebo úschovy obrazovky,
- xsetroot** nastavování vlastností obecného okna,
- xmodmap** mapování kláves a tlačítek ukazovacího zařízení.

### Manipulace s fonty (Font utilities)

- fs** server fontů, umožňuje přístup k fontům přes síť,
- xlsfonts** výpis seznamu fontů místního uzlu,
- fsfonts** výpis seznamu fontů dostupných prostřednictvím serveru fontů,
- xfd** zobrazí znaky daného fontu,
- xfontsel** zobrazuje sekvenčně jednotlivé fonty a lze si vybrat pro použití v následující aplikaci,
- showfont** zobrazí binární podobu fontu v čitelné podobě ASCII.

### Manipulace s grafickými objekty (Graphics utilities)

- bitmap** editor bitové mapy,
- xmag** editor úpravy vzhledu části displeje X,
- atobm**, **bmtoa** program konverze bitové mapy do formátu ASCII a opačně.

### Tisk (Printing applications)

- xwd** úschova vzhledu obsahu okna do souboru,
- xpr** konverze obsahu souboru získaného pomocí **xwd** do formátu PostScript nebo jiného tiskového formátu,
- xwud** zobrazí obsah souboru získaného **xwd**.

### Odstranění okna (Removing window)

- xkill** ukončení aplikaci klienta X.

### Údržba zdrojů (Resource management)

- xrdb** správce databáze zdrojů,
- appres** výpis seznamu zdrojů, které souvisí s daným klientem X,
- editres** test a editace specifikací zdroje.

### Informace o oknech a displeji (Window and display information utilities)

- xlsclients** výpis seznamu klientů X běžících na displeji X,
- xdpinfo** výpis seznamu obecných charakteristik displeje,
- xwininfo** výpis seznamu obecných charakteristik vybraného okna,
- xprop** výpis seznamu vlastností spojených s daným oknem.

# LITERATURA

[Bach87]

Bach, M. J.

**The Design of the UNIX Operating System.**

Engelwood Cliffs, NJ: Prentice Hall, 1997.

[BerMacHan97]

Berka, M., Macur, J., Hanáček, P.

**WWW informační servery.**

Unis Publishing 1996, Brno.

[BrodSkoc89]

Brodský J., Skočovský L.

**Operační systém Unix a jazyk C.**

SNL 1989, Praha.

[DoD83]

Defense Communications Agency, BBN Communications Corporation (MIL-STD)

**X.25 „Defense Data Network X.25 Host Interface Specification“.**

Volume 1 of the „DDN Protocol Handbook“ (NIC 50004), 1983.

[GarfSpaf94]

Garfinkel, S., Spafford, G.

**Practical UNIX security.**

Sebastopol CA: O'Reilly & Associates, 1994.

[HarbStee87]

Harbison, S. P., Steele, G. L.

**C: A Reference Manual.**

Englewood Cliffs, NJ: Prentice-Hall, 1987.

[Hunt94]

Hunt, C.

**TCP/IP Network Administration.**

Sebastopol CA: O'Reilly & Associates, 1994.

[ISOOSI]

ISO7498.

**Information processing systems – Open Systems Interconnection – Basic Reference Model.**

1984.

[ITSEC90]

**Information Technology Security Evaluation Criteria.**

Luxemburg: Office for Official Publications of the European Communities 1991.

[KernPike84]

Krnighan, B. W, Pike, Rob.

**The UNIX Programming Environment.**

Englewood Cliffs, NJ: Prentice-Hall, 1984.

[KernPlau76]

Kernighan, B. W., Plauger, P.J.

**Software Tools.**

Reading, Massachusetts: Addison-Wesley 1976.

[KernRitch78]

Kernighan, B. W., Ritchie, Dennis M.

**The C Programming Language.**

Englewood Cliffs, NJ: Prentice-Hall, 1978.

[LeffFabr87]

Leffler, S. J., Fabry, R. S., Joy, W. N., Lapsley P., Miller, S., Torek, C.

**An Advanced 4.3BSD Interprocess Communication.**

Berkeley, CA: University of California 1987.

[Macu94]

Macur J.

**X – WINDOW, grafické rozhraní operačního systému UNIX.**

SCIENCE, Veletiny 1994.

[Mann74]

Manna, Z.

**Mathematical Theory of Computation.**

McGraw-Hill 1974.

[OreDouTod96]

O'Reilly, T., Dougherty, D., Todino, G., Ravin, E.

**Using and Managing UUCP.**

Sebastopol CA: O'Reilly & Associates, 1996.

[Plan995] Bell Laboratories.

**Plan 9 Volume 1 – The Manuals, Plan 9 Volume 2 – The Documents.**

AT&T 1995, <http://plan9.bell-labs.com/plan9/>.

[POSIX94]

The Institute of Electrical and Electronics Engineers (IEEE).

**Portable Operating System Interface (POSIX).**

Std 1003.1b-1993 (Formerly known as IEEE P1003.4). New York, NY: IEEE 1994.

[Ritch84]

Ritchie, D. M.

**A Stream Input Output System.**

AT&T Bell Laboratories Technical Journal, vol 8 October 1984, pp 1897-1910.

[Skoc93]

Skočovský L.

**Principy a problémy operačního systému UNIX.**

Veletiny: SCIENCE 1993.

[Stev90] Stevens, R. W.

**UNIX Network Programming.**

Englewood Cliffs, NJ: Prentice-Hall, 1990.

[SVID391]

American Telephone and Telegraph Company.

**System V Interface Definition (SVID), Issue 3.**

Morristown, NJ: UNIX Press, 1989.

[TCSEC83]

**Trusted Computer Systems Evaluation Criteria.**

DoD 5200.28-STD, Department of Defense, U.S.A., 1985.

[Unicode96]

The Unicode Consortium.

**The Unicode Standard.** Version 2.0.

Reading, Massachusetts: Addison-Wesley Developers Press 1996.

[XOREil]

The Definitive Guides to the X Window System. Volume 0 – Volume Eight.

Nye, A. **Volume 0 – X Protocol Reference Manual.**

Nye, A. **Volume One – Xlib Programming Manual.**

Nye, A. **Volume Two – Xlib Reference Manual.**

Quarcia, V., O'Reilly, T. **Volume Three – X Window System User's Guide.**

Nye, A., O'Reilly, T. **Volume Four – X Toolkit Intrinsics Programming Manual.**

Flanagan, D. **Volume Five – X Toolkit Intrinsics Reference Manual.**

Heller, D. **Volume Six – Motif Programming Manual.**

Heller, D. **Volume Seven – XView Programming Manual.**

Mui, L., Pearce, E. **Volume Eight – X Window System Administrator's Guide.**

Sebastopol CA: O'Reilly & Associates, 1993.

# REJSTŘÍK

## A

a.out ... 41, 46, 63, 73  
accounting viz účtování  
Acid ... 370  
ACL (Access Control List) ... 165, 312  
adresa IP ... 219, 223, 229, 240, 255, 282, 293, 299, 361  
AFS (Andrew Filesystem) ... 264, 268  
Alef ... 369, 370, 374, 375  
ANSI C ... 8, 14, 369, 374  
APE (ANSI/POSIX Environment) ... 370  
ARP (Address Resolution Protocol) ... 224, 225, 361  
AS (Autonomous Systems) ... 225, 228  
assembler ... 12, 41  
asociace (association) ... 160, 219, 235, 237, 239  
Athena Widgets ... 289

## B

bakterie (bacteria, rabbit) ... 308  
BBN (Bolt, Beranek, Newman) ... 219, 232, 272  
bdevsw ... 118, 184, 188, 334, 358  
Berkeley sockets ... 132, 160, 197, 215, 219, 232, 234, 235, 360  
bezpečnostní atribut (security attribute) ... 312  
bezpečný port (trusted port) ... 318  
BGP (Border Gateway Protocol) ... 228  
BIND ... 251, 253  
bod přerušení (breakpoint) ... 47  
boot block viz zaváděcí blok  
BOOTP ... 299, 377  
Bourne shell ... 17, 65, 73, 166  
brána (gateway) ... 213, 329  
breakpoint viz bod přerušení  
bridge viz most  
buffer cache viz systémová vyrovnávací paměť  
build-in command viz vnitřní příkaz shellu

## C

c-seznam (c-list) ... 189, 190  
C-shell ... 17, 65, 67, 73, 166  
cc ... 41, 42, 44, 66, 247  
CCITT (Consultative Committee for International Telephony and Telegraphy) ... 220  
cdevsw ... 118, 186, 188, 334, 358  
CERT (Computer Emergency Response Team) ... 303  
CGI (Common Gateway Interface) viz scénář CGI

CIAC (Department of Energy's Computer Incident Advisory Capability) ... 303  
cílový bod transportu (transport endpoint) ... 239  
client-server viz klient-server  
coff (common object file format) ... 41, 64  
controlling terminal viz řídicí terminál  
CPU (Central Processor Unit) viz procesor  
CPU server viz procesorový server  
critical section viz kritická sekce  
cron ... 23, 39, 41, 175, 176, 177, 181, 315, 346, 356  
CURSES ... 195, 205  
cylinder group viz skupina cylindrů  
červík (worm) ... 308, 320

## D

daemon viz démon  
DAC (Discretionary Access Control) ... 309, 312, 313  
DARPA (Defense Advanced Research Project Agency) ... 219, 221  
dd ... 118, 125, 344, 345, 347, 348  
DDB (Device Database) ... 312, 341  
DECNET ... 290  
dědictví (inheritance) ... 26  
default routing viz implicitní směrování  
démon (daemon) ... 25, 39, 40, 41, 79, 189, 248, 258, 300  
DES (Data Encryption Standard) ... 322, 331, 376  
deskriptor (file descriptor) ... 83, 84, 143, 148, 149, 152, 154, 235  
device viz periferie  
DFS (Distributed File System) ... 264, 269  
directory viz adresář  
displej X (X Display) ... 286, 289, 291, 296, 321  
doména (domain) ... 229, 235, 250, 265, 266, 272, 325  
domovský adresář (home directory) ... 77, 82, 166, 300  
DNS (Domain Name System) ... 229, 248 - 252, 362  
driver viz ovladač  
DSA (Digital Signature Algorithm) ... 331

## E

efektivní vlastník a skupina procesu (effective user, effective group) ... 28  
EGP (Exterior Gateway Protocol) ... 228  
elektronická pošta (electronic mail, email) ... 174, 270  
elf (executable and linking format) ... 41, 64  
endian (little endian, big endian) ... 245

environ ... 33  
 errno ... 14, 63, 192, 240

## F

file viz soubor  
 file descriptor viz deskriptor  
 file server viz souborový server  
 firewall viz ochranná zeď  
 firmware ... 11, 348  
 frame viz rámec  
 fronty zpráv (messages) ... 129, 130, 132, 146 - 151, 304, 354  
**fsdb** ... 115, 122, 186, 339, 344, 351  
**fsck** ... 106, 114, 115, 122, 186, 315, 336, 341 - 4  
 FTP (File Transfer Protocol) ... 160, 216, 230, 248, 251, 258, 264, 272, 274, 276, 299, 320, 350, 374, 376

## G

gateway viz brána  
 grafické rozhraní uživatele (graphical user interface, GUI) ... 17, 285  
**getty** ... 23 - 5, 30, 168, 173, 179, 189, 194, 282, 291, 298, 336 - 9  
 getopt ... 35  
 GOPHER ... 274, 276

## H

hacker ... 307, 323  
 hlavní číslo (major number) ... 117, 131, 186, 334  
 home directory viz domovský adresář  
 HTML (Hypertext Markup Language) ... 274 - 7

## I

i-uzel (i-node) ... 86, 88 - 91, 96 - 100, 104 - 116, 342-3  
 i/o streams viz v/v proudy  
 IAB (Internet Architecture Board) ... 274  
 ICCCM (Inter-Client Communication Conventions Manual) ... 290, 294  
 ICMP (Internet Control Message Protocol) ... 223 - 5, 235, 248, 259  
 IEEE (Institute for Electrical and Electronic Engineers) ... 8, 220  
 IL (Internet Link) ... 369, 374  
 implicitní směrování (default routing) ... 222  
 inheritance viz dědictví  
**inetd** viz síťový superserver  
**init** ... 22, 24, 30, 39, 52, 65, 170, 173, 189, 234, 293, 335 - 8  
 inittab ... 22, 23, 39, 52, 180, 194, 258, 298, 304, 316, 335 - 8  
 Internet ... 213, 222, 229, 272 - 4, 301, 373  
 Internet provider viz poskytovatel Internetu  
 internetworking ... 212, 213, 272  
 Intranet ... 272, 278

IP (Internet Protocol) ... 219, 221-30, 250, 259, 272, 278, 328, 329, 360 - 4, 374  
 IP address viz adresa IP  
 IPng ... 219, 229  
 interruption viz systém přerušení

## J

JAVA ... 216, 274, 277, 322  
 jednouživatelský režim (single user mode) ... 335  
 Job Control Shell ... 17, 67

## K

Kerberos ... 302, 317, 323 - 5, 376  
 klient-server (client-server) ... 131 - 4  
 klient X (X Client) ... 286 - 96  
 KornShell ... 17, 65, 67, 73, 74, 166  
 kořenový adresář (root directory) ... 30, 77, 114, 276  
 kořenový svazek (root file system) ... 77, 92, 343  
 kontext procesu ... 63  
 kritická sekce (critical section) ... 15, 16, 130, 135, 137, 144

## L

LAN (Local Area Network) ... 211, 221  
 linková disciplína (line discipline) ... 189 - 194  
 logická bomba (logic bomb) ... 307  
 logické svazky (logical volumes) ... 95, 351  
 LVM (Logical Volume Manager) ... 96

## M

machine dependent code viz strojově závislá část  
 magické číslo (magic number) ... 42, 64  
 major number viz hlavní číslo  
 MAC (Mandatory Access Control) ... 309 - 314, 341  
 MAN (Metropolitan Area Network) ... 213, 225  
 mapování paměti (mapped physical storage, addressable storage) ... 16, 41, 54, 63  
 mapped physical storage viz mapování paměti  
 maska signálů (signal mask) ... 135, 136  
 maska přístupových práv souboru ... 26, 28  
 master viz vládce  
 mátoha (zombie) ... 36, 54, 62  
 messages viz fronty zpráv  
 minor number viz vedlejší číslo  
 MIT (Massachusetts Institute of Technology) ... 286 - 9, 331  
 monitor portů (port monitor) ... 339  
 most (bridge) ... 213  
 Motif ... 17, 289, 293  
 multilevel directory viz víceúrovňový adresář  
 multi user mode viz víceuživatelský režim

## N

neobvyklé souborové servery (unusual file servers) ... 373  
 News ... 174  
 NeWS (Network/extensible Window System) ... 286, 288



netmask viz síťová maska  
 NFS (Network File System) ... 79, 209, 230, 243, 248, 260, 264, 267 - 70, 323, 325, 340  
 NIC (Network Information Center) ... 222, 249 - 54  
 NIS (Network Information Services), také Yellow Pages ... 250, 260, 264 - 6, 323, 325  
 NSF (National Science Foundation) ... 272  
 NSFNet ... 228, 272

## O

odkládací oblast procesů (swap area) ... 22, 51, 93, 115, 116, 339, 351 - 4  
 ochranná zeď (firewall) ... 302, 316, 327 - 30  
 OLIT ... 289  
 opakovač (repeater) ... 213  
 Open Look ... 17, 289, 293  
 OSI (Open System Interconnection) ... 214, 215, 228  
 otrok (slave) ... 261

## P

/proc ... 62, 63, 80, 92, 129, 356, 373  
 9P ... 369 - 70, 373  
 page stealer viz zloděj stránek  
 perl ... 304, 322  
 perror ... 14, 44  
 pipe viz roura  
 port ... 160, 217, 219, 235, 259, 275  
 port monitor viz monitor portů  
 poskytovatel Internetu (Internet provider) ... 222  
 PPP (Point to Point Protocol) ... 196, 221, 278, 282, 283, 316, 362  
 pracovní adresář (working directory, current directory) ... 30, 82, 341  
 prioritizace procesu (process priority) ... 56  
 proc ... 61, 63, 64, 170, 304  
 procesorový server (CPU Server) ... 372 - 3  
 process priority viz prioritizace procesu  
 proměnné shellu ... 32, 70, 72, 280, 291  
 protokol X (X Protocol) ... 288, 290  
 PROUDY viz STREAMS  
 pruhované svazky (striped disks) ... 95, 108, 115, 351  
 přesměrování vstupu a výstupu (input and output redirection) ... 9, 67, 124  
 pseudoperiferie, pseudozařízení (pseudo-device) ... 202, 260  
 pseudoterminál ... 195, 260 - 5

## R

rámeček (frame) ... 217, 230  
 RARP (Reverse Address Resolution Protocol) ... 224, 225, 299  
 rc ... 258, 298, 335, 336  
 rc ... 369  
 RC2, RC4, RC5 ... 330

reálný vlastník a skupina procesu (real user, real group) ... 28  
 repeater viz opakovač  
 resources viz zdroje X  
 RFC (Requests For Comments) ... 221, 229, 243, 245, 274, 364  
 RFS (Remote File Sharing) ... 79, 209, 243, 260, 268, 269  
 RIP (Routing Information Protocol) ... 228, 259, 361  
 role ... 311  
 root directory viz kořenový adresář  
 root file system viz kořenový svazek  
 roura (pipe) ... 37, 38, 79, 98, 129, 130, 138 - 43, 205, 304  
 router viz směrovač  
 RPC (Remote Procedure Calls) ... 215, 228, 247, 322, 323  
 RSA (Rivest Shamir Adleman) ... 331  
 řídicí terminál (controlling terminal) ... 117, 170, 181, 189, 192, 262, 317

## S

s-bit ... 28, 48, 58, 144, 169, 177, 304, 307, 314  
 SAC (Service Access Controller) ... 339  
 SAF (Service Access Facility) ... 336, 338, 339  
 SAK (Secure Attention Key) ... 313  
 scénář (script) ... 72 - 74  
 scénář CGI (Common Gateway Interface script) ... 322  
 sdílená paměť (shared memory) ... 130, 150 - 4, 303  
 security attribute viz bezpečnostní atribut  
 semafore (semaphores) ... 91, 130, 150, 155 - 160, 304  
 server X (X Server) ... 286, 288 - 298, 321  
 shutdown ... 168, 338  
 schránka (socket) ... 197, 205, 219, 232, 258, 302  
 signál mask viz maska signálů  
 signál (signal) ... 12, 26, 27, 55, 129, 133 - 6  
 single user mode viz jednouživatelský režim  
 síťová maska (netmask) ... 222  
 síťový operační systém ... 367  
 síťový superserver **inetd** ... 23, 36, 160, 248, 258  
 skupina cylindrů (cylinder group) ... 107  
 slave viz otrok  
 SLIP (Serial Line Interface Protocol) ... 196, 221, 278, 282, 316  
 slovo přístupových práv (file access mode word) ... 85, 101, 150, 178  
 směrovač (router) ... 213, 220, 282, 299, 328, 361  
 směrování (routing) ... 200, 223, 228, 282, 328, 362  
 SMTP (Simple Mail Transfer Protocol) ... 270, 320  
 socket viz schránka  
 souborový server (File Server) ... 370  
 speciální soubor (special file) ... 270, 320, 185  
 SPIN ... 370  
 správce transportu (transport provider) ... 239 - 241  
 spojka (stub) ... 245, 246  
 standalone program ... 11, 348

STREAMS viz PROUDY  
 striped disks viz pruhované svazky  
 strojově závislá část (machine dependent code) ... 333, 368  
 stub viz spojka  
**su** ... 173, 178, 317  
 superblok ... 104 - 107, 344  
 svazek (file system) ... 77, 94, 340  
 SVR4 (SYSTEM V, Release 4) ... 8, 58  
 swap area viz odkládací oblast procesů  
 symbolický odkaz (symbolic link) ... 90, 98 - 102  
 system call viz volání jádra  
 systémová vyrovnávací paměť (buffer cache) ... 120  
 systém přerušení (interruption) ... 15

## T

/tmp ... 37, 145, 235, 312, 324, 348, 355  
 tabulka otevřených souborů, deskriptorů (file descriptors) ... 26, 30  
 TCP (Transmission Control Protocol) ... 196, 197, 213, 229 - 272, 259  
**tcpio** (trusted **cpio**) ... 313 - 315  
 TCP Wrappers ... 318  
 TCSEC (Trusted Computer System Evaluation Criteria) ... 301, 309, 314, 315  
 terminál X (X terminal) ... 286, 299  
 terminálová skupina ... 30  
 TFM (Trusted Facility Management) ... 309, 311, 313  
 TIMEZONE ... 46  
 TLI (Transport Layer Interface) ... 132, 160, 203, 239 - 242  
 TP (Trusted Path) ... 313  
 transport endpoint viz cílový bod transportu  
 transport provider viz správce transportu  
 trójský kůň (trojan horse) ... 308  
 trusted port viz bezpečný port  
 třída procesů (process class) ... 59

## U

/unix ... 109, 333, 359  
 UDP (User Datagram Protocol) ... 196, 229 - 232, 259  
 Unicode ... 368, 374  
 unusual file servers viz neobvyklé souborové servery  
 URL (Uniform Resource Locator) ... 275 - 7  
**user** ... 61, 82, 111, 189, 304  
 UUCP (UNIX to UNIX Copy) ... 196, 221, 271, 278 - 82, 313, 316  
 účtování (accounting) ... 180 - 3, 314

## V

vedlejší číslo (minor number) ... 117, 131, 186, 348  
 vedoucí skupiny procesů (leader of the group) ... 26  
 víceuživatelský režim (multi user mode) ... 335  
 víceúrovňový adresář (multilevel directory) ... 312  
 virtuální adresa (virtual address) ... 42, 53

virtuální paměť (virtual storage) ... 16, 62  
 virtuální počítač (virtual machine) ... 16  
 virus ... 307  
 vládce (master) ... 261  
 vnitřní příkaz (build-in command) shellu ... 66  
 v/v proudy (i/o streams) ... 84, 139

## W

WAN (Wide Area Network) ... 214  
 WORM ... 207, 370  
 WWW (World Wide Web) ... 248, 274, 306, 322

## X

X/OPEN ... 8  
 X11R6 ... 286, 321  
 XDR (External Data Representation) ... 220, 243, 314  
 X Client viz klient X  
 X Consortium ... 286  
 X Display viz displej X  
**xdm** (X Display Manager) ... 23, 291, 296, 337  
 XDMCP (X Display Manager Control Protocol) ... 298 - 300  
**xinit** ... 291 - 296  
 xlib ... 288  
 X Protocol viz protokol X  
**xrdb** (X Resource Database) ... 294  
 xsecurity ... 321  
 X Server viz server X  
 X terminal viz terminál X  
 X Toolkit ... 289, 294

## Y

Yellow Pages viz NIS

## Z

zadní vrátka (back door, trap door) ... 307, 322  
 zaváděcí blok (boot block) ... 109  
 zdroje X (resources) ... 288, 298  
 zloděj stránek (page stealer) ... 56  
 zombie viz mátoha















