

# PRINCIPY A PROBLÉMY OPERAČNÍHO SYSTÉMU UNIX

LUDEK SKOČOVSKÝ

## Poznámka ke 2. vydání

*I když vím, že jako autor bych to neměl dovolit a jako vydavatel technické publikace bych to už vůbec neměl dopustit, přesto tato kniha není upgrade prvního vydání z roku 1993. Uvažoval jsem nejdřív alespoň o doplnění a obměně některých příloh, ale znáte to, najednou zjišťujete, že píšete novou knihu. Musím také říct, že jsem podlehl odborné poezii doby, ve které kniha vznikla, a oživil si tehdejší stav nejenom světa UNIXu, ale i výpočetního světa vůbec.*

*Hlavním důvodem, proč tato kniha znova vychází, je k mému velkému podivu neustálý zájem o ni. Pokud vím, je stále citována a odkazována v literatuře a uváděna v osnovách a seznamech doporučené literatury u studijních oborů našich vysokých škol. Vzhledem k tomu, že se její elektronická sazba ztratila již těsně po jejím vydání před 15 lety, že vyšla pouze v tehdy běžné papírové podobě a že v knihkupectvích je už rozebraná a já nemohu sehnat původního vydavatele, rozhodl jsem se vydat ji nově v omezeném nákladu i na papíře a především zveřejnit v Internetu. Digitální podoba je zdarma, za výtisk se platí.*

*Vyjma nového rejstříku se tedy vlastní obsah knihy nezměnil. Redakce nového vydání se ujala Daniela Veškrnová, knihu zpracovalo studio petit. Tam se původní vydání nejprve snímalo skenerem a analyzovalo potřebným softwarem, sazba byla provedena znova a novým způsobem. Vzhledem k mnohým překlepům a jazykovým i odborným chybám bylo nutno také zapracovat nové korektury. Novou obálku vytvořil Vladimír Merta.*

*Zdá se mi k nevíře, jak je většina odborného textu stále aktuální. Přičítám to své snaze vždy se orientovat na standardy. Také jsem se tehdy celé roky zabýval přenositelností softwaru, což pohled na koncept knihy hodně ovlivnilo. Knihu jsem psal z potřeby sdělit ostatním zájemcům, co vím a co by se jim mohlo hodit. Snad se to tehdy podařilo a snad i digitální vydání této publikace dnes bude užitečné.*

*Luděk Škočovský, Brno, leden 2008*

# **PRINCIPY A PROBLÉMY OPERAČNÍHO SYSTÉMU UNIX**

Luděk Skočovský

## PRINCIPY A PROBLÉMY OPERAČNÍHO SYSTÉMU UNIX

© Luděk Skočovský, 1993, 2008

redakce	Daniela Veškrnová
sazba	Ivo Pecl
kresby	Luděk Skočovský, Ivo Pecl
obálka	© Vladimír Merta 2008, © Pavel Brabec 2008
tisk	Arch-polygrafické práce, spol. s r.o.
vydal	Luděk Skočovský, Brno 2008

vydání druhé, náklad 200 kusů tiskem, digitálně zdarma na [www.skocovsky.cz](http://www.skocovsky.cz)



UNIX je chráněná značka AT&T,  
DEC, PDP, VAX, VMS, ULTRIX jsou chráněné značky Digital Equipment Corp.,  
HP-UX je chráněná značka HEWLETT PACKARD,  
PERSONAL COMPUTER AT, AIX jsou chráněné značky International Business Machines Corp.,  
XENIX, MS-DOS jsou chráněné značky Microsoft Corp.,  
LynxOS je chráněná značka Lynx Real-Time Systems, Inc.,  
X Window System je chráněná značka Massachusetts Institute of Technology

ISBN 80-902612-5-6

## Obsah

<b>Předmluva</b>	8
<b>1 Úvod</b>	9
1.1 Obecně	9
1.2 Historie	10
1.3 Dokumentace	12
1.4 Poznámky	14
<b>2 Základní schéma</b>	15
2.1 Vstup uživatele do systému	15
2.2 Základní komunikace	15
2.3 Soubory	17
2.4 Procesy	31
2.5 Jádro	37
2.6 Uživatelé	38
<b>3 Hierarchie adresářů</b>	40
3.1 Jméno souboru	40
3.2 Strom adresářů	42
3.3 Hierarchie systémových adresářů	44
<b>4 Bourne shell</b>	49
4.1 Příkazový řádek	49
4.2 Standardní vstup a výstup	52
4.3 Roura	54
4.4 Expanzní znaky jmen souborů	55
4.5 Další znaky zvláštního významu	56
4.6 Popředí a pozadí, procesy	57
4.7 Znaky výluky	59
4.8 Proměnné a prostředí uživatele u terminálu	61
4.9 Příkazové soubory – scénáře	63
4.10 Programování	64
4.11 Vnoření, rekurze, funkce	71
4.12 Ladění scénářů	73
4.13 Vstup z textu scénáře	73
4.14 Několik poznámek k C-shell	73
<b>5 Volání jádra</b>	77
5.1 Uživatel	77
5.2 Procesy	78
5.3 Systém souborů	86
5.4 Komunikace mezi procesy	95
5.5 Ostatní volání jádra	105

<b>6</b>	<b>Programátor</b>	107
6.1	Metodika programování	107
6.2	Vývoj programu	109
6.3	Nástroje	115
<b>7</b>	<b>Terminál</b>	119
7.1	Změna způsobu komunikace	119
7.2	Nastavení výchozích charakteristik	120
7.3	Ovladač	124
7.4	Národní prostředí	124
7.5	Řízení obrazovky	125
7.6	X-WINDOW SYSTEM	133
<b>8</b>	<b>Sítě</b>	135
8.1	Základní prostředky komunikace	135
8.2	PROUDY	139
8.3	Podsystém UUCP	142
8.4	Model OSI	146
8.5	Přenosový protokol TCP/IP	149
8.6	Síťové aplikace	
<b>9</b>	<b>Údržba</b>	157
9.1	Spuštění a zastavení operačního systému	157
9.2	Proces init	162
9.3	Svazky, údržba, opravy	165
9.4	Instalace	178
9.5	Generace jádra	187
9.6	Ovladače periférií	190

<b>Příloha A – Tabulka kódu ASCII</b> .....	192
<b>Příloha B – Textové editory</b> .....	194
B.1 <code>ed(1)</code> .....	194
B.2 <code>ex(1)</code> .....	198
B.3 <code>vi(1)</code> .....	205
<b>Příloha C – Bourne shell</b> .....	212
C.1 Znaky zvláštního významu .....	212
C.2 Vnitřní příkazy .....	215
C.3 Řízení prací (platí pouze pro <code>jsh(1)</code> ) .....	217
<b>Příloha D – Volání jádra</b> .....	219
<b>Příloha E – Vnější příkazy</b> .....	235
<b>Příloha F – Vnější příkazy správce systému</b> .....	254
<b>Příloha G – Knihovna CURSES</b> .....	271
<b>Literatura</b> .....	282
<b>Rejstřík</b> .....	284

## Předmluva

*Jsem zastáncem dorozumění. Lidé se vzájemně lépe pochopí, budou-li si své myšlenky a zkušenosti sdělovat srozumitelně a jednoduše. Protože computer science je značně složitá, povinností těch, kteří se rozhodli o ní psát knihy, je snažit se o co největší srozumitelnost. Tato kniha mi umožňuje splatit dluh, který z tohoto hlediska pociťuji vůči uživatelům operačního systému UNIX.*

*V uplynulých dvou letech jsem se lektorsky podílel na různých školeních, která se vztahovala k operačnímu systému UNIX. Vždy jsem byl znovu a znovu překvapován názory a znalostmi programátorů – účastníků školení, z nichž mnozí neznali funkci a význam operačního systému. Je to šokující, protože kdo chce programovat v operačním systému UNIX, bez těchto znalostí se neobejde, a pokud to přesto zkusí, bude programovat špatně. V následujících devíti kapitolách jsem se pokusil při zachování jasnosti a názornosti postupovat v objasňování úkonů a důsledků úkonů programátora do co největší možné hloubky operačního systému.*

*autor, leden 1993*

*Rád bych poděkoval své ženě a dětem za trpělivost, Zdeňku Vincencovi a jeho vydavatelství SCIENCE, bez něhož by kniha nevznikla, majiteli firmy SA&S Zdeňku Jirkovcovi za vstřícnost a všem pracovníkům jeho brněnské pobočky, zejména Vlastimilu Smíškovi a Janu Schillerovi, za realizaci prvních tisků a podporu při testování některých partií knihy.*



# 1. Úvod

## 1.1 OBECNĚ

Snaha využívat co nejefektivněji drahý sálový počítač nutila výrobce vyvinout operační systém. Běžící úlohu ve chvíli, kdy čekala na příchod dat od některé z periférií, procesor přestal provádět a věnoval se jiné. Úlohy pracovaly zdánlivě současně na témže počítači. Program, který řízení práce jednotlivých úloh zajišťuje, je operační systém. Protože i on potřebuje být procesorem prováděn, je sám také jednou z úloh, ale na vyšší úrovni, takže k jeho provádění se procesor vrací nejčastěji. Říkáme, že je prováděn v supervizorovém režimu. Výpočetním systémem rozumíme technické vybavení, tj. vlastní počítač (hardware) a základní programové vybavení neboli operační systém (Operating System). Zdrojem výpočetního systému je některá jeho část, kterou úloha potřebuje pro plnění své funkce. Je jím např. tiskárna, a to jak fyzická periferie, tak program, který ji řídí (ovladač, driver), a program, který k ní zajišťuje v daném okamžiku výlučný přístup, tj. program, který vytváří a spravuje frontu požadavků na tisk.

Část operačního systému se věnovala výběru několika z úloh dodaných na děrných štítcích nebo magnetických médiích, jejichž současné provádění znamená optimální využití výpočetního systému. Klasická kniha [1] operačních systémů 60. let hovoří o funkci plánování prací. Plánování prací ale časem přestalo být potřebné, protože se s počítačem začalo pracovat přímo, tj. uživatel sedící u terminálu zadával spuštění své úlohy a čekal na její zpracování v nejmenším možném časovém úseku. Protože je možné k počítači připojit více terminálů, operační systém se začal zabývat otázkou, jak nejrychleji vyhovět požadavkům všech uživatelů a ztratilo smysl vybírat jen některé úlohy, přičemž by jiné byly odstaveny třeba i na několik desítek minut. Přímá komunikace uživatele s operačním systémem pomocí terminálu je interaktivní práce a doba rozhovoru člověka s počítačem se označuje jako sezení (session).

UNIX je interaktivní operační systém. Každá interakce uživatele a operačního systému je navíc zvýhodněna oproti již běžícím úlohám. Při interakci je používán určitý komunikační jazyk, příkazový interpret, který obecně nazýváme shell.

Obecná proslulost operačního systému UNIX vyplývá především z jeho přenositelnosti. Jednou z hlavních myšlenek na začátku bylo naprogramovat UNIX ve vyšším programovacím jazyce a zřetelně oddělit (a komentovat) strojově závislou část. Za tím účelem vytvořil Denis Ritchie programovací jazyk **C**. Jedna z prvních verzí – UNIX version 7 oficiálně uvolněná pro veřejnost – se mohla pochlubit naprogramovaným operačním systémem o 10 000 řádcích v jazyce **C** a 1000 řádcích v assembleru. První přenos na jiný typ počítače prokázal, že UNIX přenositelný je, i když ne tak snadno, jak se předpokládalo. Každopádně tím byl ale odstartován neoficiální projekt operačního systému UNIX na různých typech počítačů od různých výrobců. Současně se snahou o přenositelnost operačního systému vznikla myšlenka přenositelnosti programů a aplikací mezi různými počítači. Tuto myšlenku potvrdil návrh Briana Kernighana hovořící o jednotném programovém vybavení programátora (viz [2] a [3]). Znamená to, že programátor může při psaní aplikací využívat programové celky, které jsou součástí operačního systému, a to platí všude, kam bude aplikace přenesena. Vznikla podpora přenositelnosti programů na dvou úrovních. Jednak je to přenositelnost programu, který využívá přímo operační systém pro zajištění základních úkonů (přenos

dat z periferie, vytváření procesů...), a jednak přenositelnost na úrovni sady podpůrných programů (z nichž každý obvykle vykonává jednu funkci, a to bezchybně) a možnosti spojovat programy navzájem a se zbylou částí vlastní aplikace. Říkáme, že používáme volání jádra (System Calls), tj. první úroveň, a nástroje programátora (Tools), druhá úroveň.

Jak je vidět, to, co jsme na začátku označili jako operační systém, přestalo být „jednou z úloh běžících současně s ostatními“. Z tohoto pohledu můžeme operační systém UNIX rozdělit na:

- jádro operačního systému
- základní podpůrné tabulky a programy
- nástroje programátora

Vlastnost, kterou se vyznačuje UNIX a která vychází zejména z přenositelnosti, je otevřenost. UNIX v dnešních návrzích otevřených systémů vyniká snadnou přizpůsobivostí, a to zvláště z pohledu počítačových sítí. Původně byla otevřenost pro UNIX chápána také z hlediska průhlednosti vnitřní struktury. Tato otevřenost byla v době distribuce prvních verzí provázena také tím, že součástí distribuce byly všechny zdrojové texty jádra a nástrojů. Navíc na magnetických páskách uživatel obdržel i podrobnou a kvalitní dokumentaci. To bylo v 80. letech trnem v oku všem obchodníkům s operačními systémy a přestože původní myšlenka „nelíbí-li se ti funkce tohoto nástroje, máš k dispozici zdrojové texty pro modifikaci“ byla jednou z hlavních, prodávají se dnes zdrojové texty operačního systému UNIX pouze za velké peníze. Avšak na začátku 80. let právě tato vlastnost znamenala bouřlivý vývoj pro UNIX, jak také ukazuje následující článek.

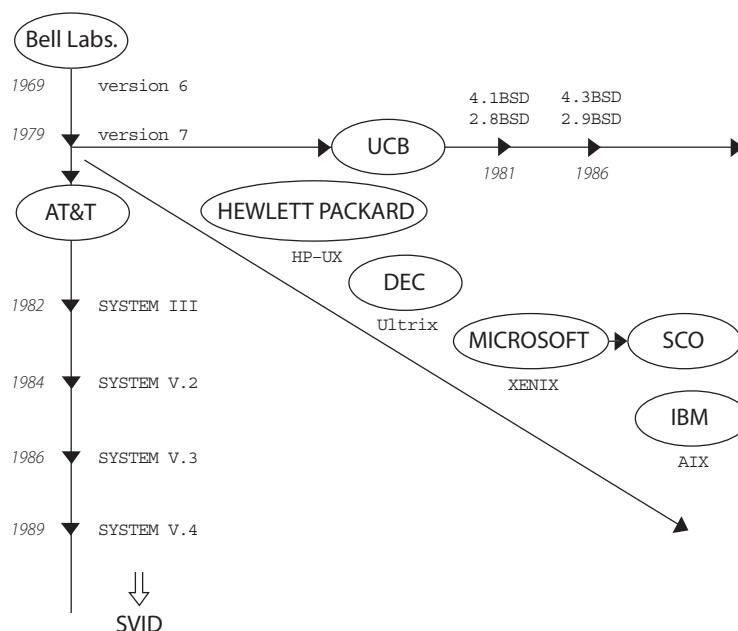
## 1.2 HISTORIE

Vznik operačního systému UNIX je datován rokem 1969 a situován do střediska výzkumu firmy AT&T Bell Laboratories. Ken Thompson, Denis Ritchie a Brian Kernighan, pracující na projektu nového univerzálního operačního systému MULTICS, se snažili vytvořit příjemnější programovací prostředí než nabízel právě MULTICS. Skupina odpadlíků poměrně rychle a z vlastní iniciativy vytvořila zárodek dnešního UNIXu na PDP-7. Vedení Bell Labs. ale odmítlo dále tento projekt podporovat a teprve po návrhu vývoje prostředí pro práci s dokumentací v UNIXu pro patentové oddělení firma vývoj financovala a podpořila počítačem typu PDP-11.

V průběhu několika let Ritchie vyvinul beztypový programovací jazyk BPCL, ze kterého po zavedení datových struktur vznikl jazyk C. Vývoj jazyka C byl veden výhradně záměrem pro přepis UNIXu do vyššího programovacího jazyka a motivován snahou o přenositelnost. V této době se UNIX začíná skutečně používat nejen v patentovém oddělení, ale i dalších odděleních Bell Laboratories. Koncem 70. let se zdá být základní verze operačního systému podle představ autorů hotova a Bell Labs. po dohodě se státní správou a školstvím uvolňují UNIX version 7 pro použití i mimo výzkumné středisko. UNIX version 7 (označovaná zkráceně v7) má z pohledu uživatele všechny charakteristické rysy dnešních verzí. Je to zejména:

- hierarchická struktura adresářů
- práce s procesy jako nositeli změn v datové základně
- podpora volání jádra pro odstínění uživatele od konkrétního technického zařízení
- textově orientovaná interaktivní práce více uživatelů u terminálů

Základním testem pro UNIX byl přenos v7 na počítač jiného typu, a to INTERDATA 8/32, který potvrdil přenositelnost podle proklamovaných tezí. Následující vývoj probíhá pod vedením jednak AT&T a jednak University of California v Berkeley (UCB), jak ukazuje obrázek 1.1.



Obr. 1.1 Verze operačního systému UNIX

AT&T vyvíjí snahu dopracovat UNIX do podoby komerčně dodávaných systémů pro profesionální použití, UCB se věnuje vývoji další systémové nadstavby, především v oblasti počítačových sítí.

UNIX SYSTEM V je dominující verze, která je odrazovým můstkem současným implementacím na různých počítačích od různých výrobců. V průběhu 80. let se objevuje velké množství operačních systémů, které se vzhledem k drahé licenci na zdrojové texty chovají jako UNIX, ale není zaručena přenositelnost v nich odladěných programů do jiných systémů typu UNIX. AT&T proto vydává doporučení SVID (System V Interface Definition), které stanovuje základní přenositelnost programů pro UNIX na úrovni zdrojových textů vzhledem k voláním jádra, které normalizuje. Dnes známá SVID3 je obecně uznávaným základním dokumentem pro UNIX a zabývá se specifikací i na úrovni příkazových řádků nástrojů programátora, konvencí jazyka C atd. Tato kniha SVID3 plně akceptuje; odchylky, které v některých systémech stále existují, budou v textu zvláště komentovány.

Distribuce systémů s označením BSD (Berkeley System Distribution) z UCB byla zahájena na počítačích VAX s označením 4.xBSD, kde x je označení verze a 2.xBSD pro počítače PDP všech modelů. Tým odborníků s označením BSD je dodnes světově respektovanou vývojovou skupinou ovlivňující nové směry UNIXu.

Zvětoví výrobci počítačů čerpající z obou větví jsou na obr. 1.1 uvedeni v elipsách, pod nimiž je označení jejich komerčně dodávaného operačního systému UNIX, vždy vyhovující doporučení SVID. Přestože obyčejný uživatel rozpozná jinou implementaci pouze podle jména, jsou systémy pojmenovány různě, a to i přes úsilí sjednotit všechny tyto systémy pod jménem UNIX.

Ve snahách o normalizaci hraje důležitou roli evropská skupina výrobců X/OPEN (založená v roce 1984) vydávající doporučení (X/OPEN Portability Guide), které se v podstatných partiích zcela shoduje s SVID.

Normu POSIX vlastní IEEE (Institute of Electrical Electronic Engineers v U.S.A.) pod evidencí Standard 1003.1 – 1988 a definuje standardní rozhraní operačního systému a prostředí na bázi UNIX. POSIX je ale obecnější normou, snažící se definovat přenositelnost programů na úrovni zdrojových textů tak, aby byly přenositelné mezi různými typy operačních systémů.

Dnes je možné UNIX provozovat na počítačích různé třídy. Některé verze jsou určeny výhradně pro danou třídu, např. XENIX a SCO UNIX pro oblast osobních počítačů. Modely počítačů firmy HEWLETT PACKARD řady 9000, která celá pracuje v HP-UX, zase pokrývají škálu počínající typem pracovní stanice (modely 345, 375...) až po modely střední třídy podporující práci více než 50 uživatelů současně (např. model 832). Totéž platí pro počítače firmy IBM. Jejich AIX je možné provozovat na sálových počítačích, ale také např. na osobním počítači PS/2 model 80. Důležitá je otevřenost a uživatelská jednodušnost všech těchto systémů, vyplývající především z dodržování SVID, POSIX a X/OPEN.

### 1.3 DOKUMENTACE

Standardně dodávaná dokumentace dodnes vychází ze struktury a referencí dokumentace pro UNIX v7. Ta rozlišovala dvě základní části označované jako reference (manuals, zkráceně *man* nebo *mans*) a články (documents, zkráceně *doc* nebo *docs*).

Články ve v7 se věnovaly především popisu charakteristik operačního systému, nebo byly stručným (ale kvalifikovaným) úvodem pro začátečníka. Jejich autory byli zakladatelé UNIXu D. Ritchie, B. W. Kernighan, K. Thompson, A. S. Bourne a další, kteří pracovali na zvláštních nástrojích programátora (např. `awk(1)`), což je specializovaný jazyk pro práci s texty autorů A. Aho, P. Weinbergera a B. Kernighana, nebo `yacc(1)` a `lex(1)` pro vývoj překladačů od S. Johnsona, M. Leska a E. Schmidta). Celkový počet přibližně třiceti článků rozšířily verze systémů BSD, které pokračovaly v koncepci popisného i výukového článku pro nově vzniklé partie.

Reference je podrobným popisem jednotlivého příkazu, nebo funkce z pohledu uživatele. Původně byly rozděleny do 8 dílů takto:

- 1 příkazy obyčejného uživatele (Commands)
- 2 volání jádra (System Calls)
- 3 podprogramy (Subroutines)
- 4 formáty souborů (File Formats)
- 5 různé (Miscellaneous)
- 6 hry (Games)
- 7 speciální soubory (rozhraní pro technické vybavení – Special Files)
- 8 příkazy privilegovaného uživatele (Superuser Commands)

Každý díl má reference abecedně seřazené. Reference je zvláštním způsobem formátovaný dokument výrazně členěný na části jméno (NAME) pro označení reference, formát použití (SYNOPSIS), popis (DESCRIPTION) a viz také (SEE ALSO) pro označení dalších s problematikou souvisejících referencí téhož nebo ostatních dílů. Každý díl obsahuje referenci nazvanou intro (introduction), úvod, který popisuje obsah dílu.

Jak články, tak reference byly ve v7 i v BSD systémech dodávány na magnetickém médiu, články jako textové soubory obecně dostupné prohlížecími nebo tiskovými programy a reference navíc podporované programem označovaným jako `man`, takže při běžné komunikaci uživatele u terminálu bylo možné např. psát

```
$ man 1 ls
```

což znamenalo vyhledání a prohlížení reference z dílu 1 se jménem `ls` (příkaz pro výpis obsahu adresáře). Označení dílu (`1`) bylo možné vynechat a program potom prohledával všechny díly a hledal referenci daného jména.

Komerční důvody změnily charakter dodávané dokumentace. Firemní literatura k dodávaným systémům se přidržela členění na reference a články, explicitně je ale takto neuvádí. Reference jsou po odmlce v 80. letech opět dodávány na magnetickém médiu a za chodu dostupné pomocí `man(1)`. Členění do dílů se převážně dodržuje, změny bývají ve 4.–7. dílu, kdy se specializace dílů pro různé výrobce liší. Obvykle chybí díl 6, protože rozvoj her v UNIXu byl nepatrný. Bohužel existují ale systémy, kde struktura i označení dílů je zcela jiná. Např. SCO UNIX (nebo SCO XENIX) má značení dílů písmenové, např.

C	příkazy
CP	příkazy programátora
S	podprogramy a volání jádra
ADM	příkazy privilegovaného uživatele
HW	rozhraní technického vybavení atd.

a např. spojení podprogramů a volání jádra jejich promísením je pro začátečníka nepříjemné, protože každé volání jádra je pro UNIX principiální jasně charakterizující vlastností, které má při používání tvar volání obvyčejného podprogramu. Uživatel se proto ze svazku `S` nedozví nic o principech obsažených v jádru systému (volání jádra je přibližně 60, podprogramů 200–300), protože z reference není zřejmé, zda se jedná o funkci nebo volání jádra.

Firemní literatura většinou nenabízí články, namísto nich výrobci poskytují uživatelům několik knih a každá se věnuje jedné oblasti, např. jedna kniha je věnována údržbě a instalaci operačního systému, další příkazovým interpretům nebo nástrojům programátora. Jedna z nich je ale také vždy určena začátečníkům (Getting Started) a tou by měl nový uživatel systému začít.

Kromě dokumentace, kterou kupující dostává se sadou distribučních magnetických médií, je v západním světě neustále vydávána záplava knih o operačním systému UNIX, a to jak pro začátečníky, tak i pro systémové programátory. Některé z nich jsou uvedeny v části Literatura na konci knihy.

### 1.4 POZNÁMKY

UNIX jako každý víceuživatelský operační systém vyžaduje fyzickou osobu s funkcí správce systému. V dnešní době širokého používání osobních počítačů s primitivním monitorem uživatelského sezení MS-DOS je tento fakt pro většinu uživatelů (a kupodivu i programátorů) překvapením. Naopak představitelé firem vlastnících sálůvých počítačů nebo počítačů střední třídy s terminálovou sítí a diskovou pamětí sdílenou řadou uživatelů to považují naštěstí za samozřejmost. UNIX je operační systém, který vznikl a byl vyvíjen pro provoz aplikací na počítači z více míst současně. Samozřejmostí je dnes i napojení na počítačovou síť a s tím opět spojená provozní údržba. Základní činnosti správce systému popisuje kap. 9. Je to zejména pravidelná kontrola dat uložených na discích, jejich úschova a možnost obnovy v případě havárie disku. Dále musí být správce systému schopen připojovat a odpojovat terminály (viz kap. 7) nebo novou periférii, což znamená připojit k systému nový ovladač (driver, kap. 9). Zodpovídá rovněž za správnou instalaci nových aplikací a jejich vhodné umístění v operačním systému.

UNIX je obvykle prodáván jako sada distribučních médií s dokumentací. Součástí dokumentace je příručka věnovaná instalaci. Některé firmy (např. HEWLETT PACKARD) při prodeji počítače kromě dodání kompletní distribuční sady instalaci pro objednanou konfiguraci také provedou. Ovšem seriózní prodejce instalaci se zárukou provádí (např. WYSE UNIX).

Kupující se při výběru z katalogu rozhoduje, zda objednává kompletní operační systém, nebo pouze jeho část.

Základní UNIX je totiž sám rozdělen na 2 části, a to:

UNIX Operating System	– je základní funkční operační systém
UNIX Development System	– je vývojové prostředí, jehož součástí je především jazyk C a jeho prostředí a převážná většina nástrojů programátora

Samotný Operating System je vhodné používat tam, kde nebude žádná aplikace programována, tedy tam, kde je instalována provozní verze aplikace. Naopak Development System je důležitý pro každého programátora aplikací. Pro jeho instalaci a provoz je ale nutné mít už instalovanou část Operating System.

V následujícím textu knihy se u překladu anglických názvů přidržuji konvencí stanovených knihou [4]. Nové termíny, použité v této knize, byly konzultovány s překladateli připravovaných publikací o UNIXu (např. [5]), s normou pro názvosloví výpočetní techniky a s lingvisty. Jsem zastáncem kulturního a jednoznačného vyjadřování, nikoliv nepřesného a zavádějícího žargonu.

V příkladech programování a definic datových struktur systému se vyjadřuji jazykem C, jehož znalost při čtení knihy považuji za nutnou. Začátečníkům doporučuji ke studiu jazyka C [6] nebo [4].

V příkladech komunikace uživatele u terminálu jsou tučně vyznačeny texty, které píše uživatel na klávesnici, naopak výpis textů operačního systému je uváděn v běžné tloušťce písma. V odkazech na jednotlivé části operačního systému používám konvence odkazů v referenční části dokumentace (viz čl. 1. 3). Znamená to, že budu-li např. hovořit o příkazu výpisu adresáře se jménem `ls`, uvedu pouze `ls` ( `1` ) nebo budu-li hovořit o formátu proveditelného souboru, uvedu pouze `a.out` ( `4` ). Čísla v závorkách určují díl reference, název pak jméno reference.

## 2 Základní schéma

### 2.1 VSTUP UŽIVATELE DO SYSTÉMU

```
login:
```

je text, který nově přichází uživatel čte na obrazovce terminálu. Aby se přesvědčil, že je terminál stále připojen k počítači, kde běží UNIX, stisknutím klávesy Enter se výpis textu login: zopakuje. Uživatel může nyní psát na klávesnici svoje jméno, které je počítači známo a které uživateli přidělil správce systému. Po odeslání jména klávesou Enter se na obrazovce objeví text

```
password:
```

a UNIX opět čeká na reakci uživatele. Uživatel má obvykle svá data v počítači chráněna před neoprávněným přístupem heslem. Heslo nyní napíše na klávesnici (přitom se psaný text na obrazovce nezobrazuje). Pokud je zápis jména i hesla správný, na obrazovce se před uživatelem objeví zpráva o vstupu do systému, uvítací text, atp. Výpis se zastaví a UNIX čeká na vstup od uživatele. Na obrazovce se objeví:

```
$
```

Od tohoto okamžiku je uživatel regulérně přihlášen v systému a může interaktivně využívat všech zdrojů výpočetního systému běžného uživatele.

Po vstupu do systému je uživatel evidován jako obyčejný (znak výzvy pro komunikaci je \$), anebo privilegovaný (znak výzvy je #). Obvykle má systém jednoho privilegovaného uživatele se jménem `root` (tzv. superuživatel). Po přihlášení pod tímto jménem má uživatel neomezená přístupová práva ke všem datům a zdrojům výpočetního systému. Obyčejní uživatelé jsou hlídáni systémem a mohou přepisovat nebo rušit jen vlastní data na základě požadavků pro operační systém. Jiná než privilegovaná a neprivilegovaná úroveň není v UNIXu zavedena.

Uživatel se může ze systému odhlásit uzavřením vstupu pro komunikaci v rámci svého sezení, což je pro UNIX znak Ctrl-d (současné stisknutí klávesy Ctrl a d, kódování v ASCII \004). Sezení je ukončeno a UNIX na terminál opět vypíše text končící

```
login:
```

### 2.2 ZÁKLADNÍ KOMUNIKACE

Uživatelé komunikují s operačním systémem pomocí příkazového řádku. Chceme-li např. znát seznam všech souborů v našem adresáři po přihlášení, můžeme psát příkazový řádek

```
$ ls -a
```



## 2.2 Základní schéma

kde znak `$` je znak výzvy pro zápis příkazového řádku (vypisuje UNIX) a příkazový řádek je text `ls -a`. Jméno příkazu je `ls` a část `-a`, nazývaná volby příkazu, určuje změnu chování příkazu oproti implicitní formě. Odpovědí systému je seznam všech souborů, které jsou evidovány v adresáři. Jsme-li přihlášení vůbec poprvé, pravděpodobně nám UNIX odpoví:

```
.  
..  
.profile
```

přitom každý řádek odpovídá jednomu jménu souboru.

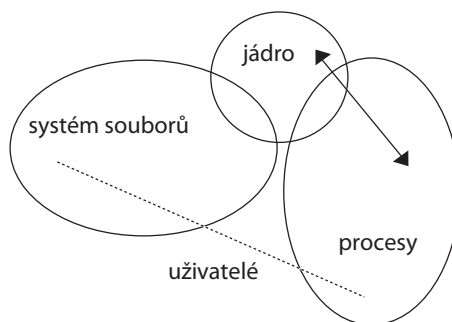
Budeme-li si v následujícím kroku chtít zobrazit obsah souboru se jménem např. `.profile`, napíšeme:

```
$ cat .profile
```

a na obrazovce terminálu je zobrazen obsah souboru `.profile` (obsahuje nastavení výchozího stavu uživatelské komunikace se systémem).

V uvedeném příkladu příkazového řádku pro zajištění našeho požadavku vytváří UNIX proces, který provádí příkaz `cat` a manipuluje se souborem `.profile`. Oba dva fenomény, soubor i proces, jsou podporovány ve své existenci fenoménem třetím, kterým je jádro (kernel).

Jádro je program, který je zaveden po zapnutí počítače do operační paměti a tam je spuštěn. Je programem, který z počítače na úrovni technického vybavení vytváří virtuální počítač, odstíní uživatele od přímého styku s technickým vybavením, umožňuje komunikaci se strojem ve formě vyššího programovacího jazyka a dokáže zajistit sdílení technických zdrojů několika uživatelům najednou. Pro uchování dat na vnějších paměťových médiích (zejména magnetických discích) zajišťuje přístup ke struktuře dat nazývané systém souborů (file system). Pro uživatele to znamená, že může svá data ukládat do souborů a opět je ze souborů vybírat pro další zpracování. Proces je realizace akce uživatele nebo systému. Je nositelem změn v datové základně konkrétní instalace operačního systému. Procesem je např. provedení kopie souboru na tiskárnu, překlad z úrovně zdrojového textu programu do úrovně strojového kódu, spuštění databáze atd. Pomocí volání jádra (System Calls) probíhá interakce procesů s jádrem. Volání jádra má z hlediska psaní programu tvar volání knihovni funkce; je to např. žádost o zpřístupnění dat z disku, žádost o vytvoření nového procesu atd. Schematicky lze základní tři komponenty systému zobrazit podle obr. 2.1:



Obr. 2.1 Jádro, procesy, systém souborů



## 2.3 SOUBORY

Soubor je v UNIXu posloupnost slabik bez jakékoliv další struktury. Rozlišujeme tři hlavní skupiny souborů:

- obyčejné soubory
- adresáře
- speciální soubory

Každý soubor je ve vnitřní struktuře systému souborů reprezentován i-uzlem (i-node, index node). I-uzel obsahuje všechny atributy souboru a odkazy na datovou část souboru. Atributem souboru je např. informace, zda je soubor adresářem nebo obyčejným či speciálním souborem, kdo je jeho vlastníkem, jaká k němu mají uživatelé přístupová práva pro čtení nebo zápis. Výpis hlavních atributů souboru `.profile` v pracovním adresáři nám zajistí příkaz

```
$ ls -l .profile
```

kde odezvou v našem případě bude

```
-rw-r----- 1 petr group 506 May 12 10:37 .profile
```

Každý řádek uvedeného výpisu se vztahuje vždy k jednomu souboru. První znak výpisu na řádku je typ souboru a může být

- obyčejný soubor
- d adresář
- l nepřímý odkaz
- c znakový speciální soubor
- b blokový speciální soubor

Ve výpisu atributů souboru následuje dále devět znaků, které určují přístupová práva k souboru, přitom jednotlivé trojice postupně pro vlastníka souboru, skupinu, do které vlastník patří, a pro ostatní uživatele. V každé trojici mohou přístupová práva znamenat

- r čtení souboru je dovoleno
- w zápis do souboru je dovolen
- x obsah souboru může být spuštěn jako proveditelný program
- přístup k souboru je zakázán

Uživatelé jsou rozděleni do skupin. Příslušnost ke skupině určí správce systému obvykle při registraci uživatele v systému. Uživatelé stejné skupiny jsou si potom bližší z hlediska přístupu k souborům než uživatelé jiných skupin. Soubor `.profile` v našem příkladu může vlastník číst i modifikovat a členové téže skupiny číst. Ostatní práva jsou odebrána.

## 2.3 Základní schéma

Po atributu přístupových práv následuje počet odkazů na soubor z různých míst systému souborů a v našem případě je jeden.

Atribut `petr` je jméno vlastníka souboru `.profile` a `group` je skupina souboru `.profile`.

Následuje velikost souboru ve slabikách (506 znaků) a datum a čas poslední modifikace souboru. Poslední vypsanou položkou na řádce je jméno souboru.

Obyčejný soubor obsahuje data uživatele. Obyčejným souborem je např. soubor `.profile`. Obsah obyčejného souboru můžeme zpřístupnit příkazem `cat(1)` (podle uvedeného příkladu čl. 2.2) nebo příkazem `more(1)`:

```
$ more .profile
```

kdy je obsah souboru `.profile` zobrazován po stránkách. Další příkaz podobný `more(1)` je `pg(1)`.

Uvedený způsob výpisu obsahu souboru je korektní, je-li obyčejný soubor souborem textovým, tj. obsahuje-li znaky z dolní části tabulky ASCII, které jsou běžně zobrazitelné na obrazovce terminálu nebo na tiskárně a jsou používány k záznamu běžně čitelného textu (zdrojové texty programů, dokumenty, telefonní seznamy atd.). Jejich společným znakem je řazení do záznamů ukončených znakem nového řádku. Oproti tomu soubory binární obsahují znaky z celé tabulky ASCII a jsou obvykle čitelné zvláštními aplikacemi nebo jádrem. Binární soubor je např. program v proveditelné podobě, systémové tabulky atd. Obsah binárního souboru si můžeme prohlížet pomocí příkazu `od(1)`:

```
$ od -c a.out
```

Příkaz `od(1)` bez použití volby `-c` vypisuje obsah souboru `a.out` v osmičkové soustavě. Volba `-x` mění výpis z osmičkové soustavy na šestnáctkovou, `-d` na desítkovou a `-c` je výpis znakový, což znamená, že všechny běžně zobrazitelné znaky jsou vypisovány svým obsahem a ostatní ve svém ekvivalentu v osmičkové soustavě. Výpis je uskutečňován po 16 znacích na řádku, řádky jsou počítány (v osmičkové soustavě) v levém sloupci. Je-li ve výpisu samostatně na řádku uveden znak `*`, znamená to opakování právě uvedeného řádku. Kolik takových znaků je opakováno, zjistíme podle počítadla znaků v levém sloupci výpisu.

Adresář je binární soubor s atributem příznaku adresáře a obsahuje seznam souborů. Po přihlášení je uživateli nastaven určitý adresář, který je označován jako domovský (home directory). Uživatel v průběhu sezení může měnit nastavení na různé adresáře. Adresář, který má právě nastaven, je označován jako pracovní (current directory). Obsah adresáře je zobrazitelný příkazem `ls(1)`, ale ve skutečnosti jde o rozpis obsahu i-uzlů. Pracovní adresář je binární soubor a jeho obsah lze vypsat příkazem

```
$ od -c .
0000000 355 004 . \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000020 # \0 . . \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000040 0 \0 . p r o f i l e \0 \0 \0 \0 \0 \0
```

Každý řádek výpisu odpovídá jednomu souboru. První dva znaky (za počítadlem) jsou identifikací čísla i-uzlu a dále následuje 14 znaků jména souboru. Pro operační systém je rozhodující číslo i-uzlu, se kterým je spojeno v adresáři jméno souboru.

Výpis adresáře způsobem `ls -ai od -c` nás upozorňuje na přítomnost dvou souborů se jménem `.` a `..`. Z výpisu příkazu

```
$ ls -la
total 6
drwxr-xr-x  2 petr group   48 May 12 10:37 .
drwxr-xr-x 16 bin  bin   256 May 12 10:37 ..
-rw-r----- 1 petr group  506 May 12 10:37 .profile
```

vidíme, že se jedná o adresáře. Soubor `..` je nadřazeným adresářem a `.` je aktuální adresář (ukazuje na i-uzel vypsaneho adresáře). Systém souborů tvoří z hlediska adresářů hierarchickou strukturu, která je zajištěna položkami `.` a `..` v každém adresáři. První řádek uvedeného výpisu (`total 6`) nás informuje o celkovém obsazení souborů adresáře v blocích.

Hierarchie je završena adresářem označovaným `/`, nazývaným také kořenový adresář (root directory). Kořenový adresář obsahuje podadresáře, které dále mohou obsahovat podadresáře do teoreticky libovolné hloubky. Celý systém souborů je spojen ve strom adresářů. V rámci tohoto stromu má uživatel přidělen adresář, který je mu nastaven po přihlášení do systému a kterým začíná jeho uživatelská oblast dat (jeho uživatelský podstrom). Je to jeho domovský adresář (home directory).

Výpis jména pracovního adresáře získáme příkazem `pwd (1)` (print working directory):

```
$ pwd
/usr/petr
$
```

Výpis `/usr/petr` nás informuje o nastavení na adresář `petr`, jehož nadřazeným adresářem je adresář `usr` a tomuto je nadřazen adresář `/` (kořenový adresář). Výpis je označován jako cesta (path) k adresáři (v tomto případě k pracovnímu adresáři).

Změnit svůj pracovní adresář můžeme příkazem `cd (c`hange directory):

```
$ cd /usr
$ pwd
/usr
$
```

Použije-li uživatel příkaz `cd` bez argumentu, nastavuje si tak domovský adresář.

```
$ cd
$ pwd
/usr/petr
$
```

## 2.3 Základní schéma

Změna pracovního adresáře podléhá samozřejmě kontrole oprávnění vstupu do adresáře podle přístupových práv, vstup do adresáře je dovolen příznakem `x` ve výpisu atributů adresáře.

Adresář vytvoříme příkazem `mkdir (1)` (make directory) a zrušíme příkazem `rmdir (1)` (remove directory):

```
$ mkdir testy                (vytvoření adresáře testy)
$ ls -l
total 2
drwxr-xr-x 2   petr   group   32   May 13 11:27 testy
$ cd testy                  (nastavení na adresář testy)
$ mkdir soubory
$ cd                        (nastavení na domovský adresář)
$ rmdir testy
rmdir: testy: Directory not empty
```

Adresář totiž můžeme zrušit, jen je-li prázdný, takže píšeme:

```
$ rmdir testy/soubory
$ rmdir testy
$
```

Cesta k adresáři může být absolutní (`/usr/petr/testy/soubory`) nebo relativní (`testy/soubory`) vzhledem k pracovnímu adresáři.

Používat můžeme také jména adresářů `.` a `..`. Např.:

```
$ pwd
/usr/petr
$ cd ..
$ pwd
/usr
$
```

Adresář obsahuje obyčejné soubory. Základní manipulace se soubory jsou kopie, přejmenování a zrušení souboru. Kopii souboru vytvoříme příkazem `cp (1)` (copy):

```
$ cp .profile text
```

kde jako první argument příkazu je jméno existujícího souboru (`.profile`) a druhý argument je jméno nově vytvářeného souboru (`text`).

Přejmenovat soubor můžeme příkazem `mv (1)` (move):

```
$ mv text iniscript
```

kde jako první argument je původní jméno existujícího souboru (*text*) a druhý argument jeho nové jméno (*iniscrypt*). Vzhledem k tomu, že je pro systém rozhodující číslo i-uzlu, je zřejmé, že je v tomto případě provedena změna položky jména v pracovním adresáři.

Soubor rušíme příkazem `rm(1)` (*re*move):

```
$ rm iniscrypt
```

Použijeme-li volbu `-i`

```
$ rm -i iniscrypt
iniscrypt: ? _
```

`rm(1)` očekává potvrzení o rušení (*yes*=rušíme soubor, cokoliv jiného znamená, že soubor není zrušen). Dále je možné použít volbu `-r`, jejíž pomocí můžeme rušit rekurzivně celý podstrom:

```
$ pwd
/usr/petr
$ mkdir testy
$ cd testy
$ cp /usr/petr/.profile iniscrypt
$ cd
$ rm -r testy
$ ls -l testy
testy not found
$
```

Při použití příkazu `cp /usr/petr/.profile iniscrypt` používáme absolutní cestu k souboru `.profile` a relativní cestu k souboru `iniscrypt`. Vzhledem k možnosti využít adresáře se jménem `..` můžeme příkaz přepsat na

```
$ cp ../.profile iniscrypt
```

Adresář `.` můžeme využít pro referenci pracovního adresáře také v příkazu `cp(1)`, kde na místě druhého argumentu může být uvedeno jméno adresáře a koncové jméno souboru je při kopii převedeno na nově vznikající soubor.

```
$ cp ../profile .
```

okopíruje z nadřazeného adresáře soubor `.profile`. Nově vzniklý soubor má pak také jméno `.profile`.

Obyčejný soubor vzniká obvykle jako produkt editace (editorem `ed(1)`, `ex(1)` nebo `vi(1)`, viz Příloha B) nebo některé jiné aplikace (překladače, databáze atd.). Pomocí programu `cat(1)` lze soubor s textovým obsahem vytvořit také, a to pomocí mechanismu

## 2.3 Základní schéma

přesměrování. Každý program spuštěný z příkazového řádku obvykle pracuje interaktivním způsobem, standardně čte z klávesnice terminálu a výsledky své činnosti zobrazuje na obrazovce terminálu. Vstupu z klávesnice říkáme standardní vstup (standard input) a výstupu na obrazovku standardní výstup (standard output). Programy pracují jako filtry, data ze standardního vstupu předávají na standardní výstup. Napíšeme-li příkaz `cat (1)` bez argumentů, `cat (1)` očekává vstup dat nikoliv z určeného souboru, ale z klávesnice:

```
$ cat
zkusebni text
zkusebni text
dalsi radek zkusebniho textu
dalsi radek zkusebniho textu
^d$
```

A po řádcích jej předává na standardní výstup. Standardní výstup můžeme přepnout na soubor pomocí znaku `>` takto:

```
$ cat > ztext
zkusebni text
dalsi radek zkusebniho textu
^d$ ls -l ztext
-rw-r--r-- 1 petr group 43 May 13 13:46 ztext
$ cat ztext
zkusebni text
dalsi radek zkusebniho textu
$
```

Soubor `ztext` je nově vytvořen a pokud již dříve existoval, je zkrácen na nulovou délku a naplněn výstupem programu `cat`. Pokud soubor existoval a chceme-li zachovat jeho obsah, můžeme nový text k obsahu souboru jen připojit tak, že standardní výstup přesměrujeme využitím dvojznaku `>>`:

```
$ cat >> ztext
jeste jeden radek zkusebniho textu
^d$ cat ztext
zkusebni text
dalsi radek zkusebniho textu
jeste jeden radek zkusebniho textu
$
```

a naopak, pokud by soubor se jménem `ztext` doposud neexistoval, je vytvořen s nulovou délkou a obsah z klávesnice je k němu připojen.

Příkazem `find(1)` vyhledáme soubor podle některého jeho atributu, a to v dané části stromu adresářů. Např.

```
$ pwd
/usr/petr
$ find . -name ztext -print
/usr/petr/ztext
...
```

prohledává podstrom začínající pracovním adresářem `.` a na standardní výstup vypisuje (na základě volby `-name`) cesty všech souborů, jejichž jméno je `ztext`. Výpis je zajištěn pomocí volby `-print`. Lze totiž požadovat nejen výpis cesty k souboru, ale i provedení určité akce. Např.

```
$ find /usr/petr -name core -exec rm {} \;
```

hledá v podstromu adresáře `/usr/petr` všechny soubory se jménem `core` a ruší je. Určit jiný atribut, podle kterého `find` (1) vybírá soubory, můžeme použitím jiné volby, např.

```
$ find /usr -user petr -print
```

vyhledá všechny soubory od adresáře `/usr`, které jsou ve vlastnictví uživatele `petr`. Konečně

```
$ find /usr -user petr -type d -print
```

vypíše všechny adresáře v podstromu `/usr`, které vlastní uživatel `petr` (lze použít více voleb současně), a

```
$ find /usr/petr /usr/jan -mtime -7 -print
```

vypíše cestu k souborům podstromů `/usr/petr` a `/usr/jan` (lze stanovit více výchozích adresářů současně), které byly změněny v posledních 7 dnech.

Speciální soubor je přístup k periférii výpočetního systému. Speciální soubory jsou standardně uloženy v podstromu začínajícím adresářem `/dev` a uživateli jsou dostupné podle přístupových práv. Přístup uživatele např. k tiskárně může být pomocí speciálního souboru `lp`:

```
$ ls -l /dev/lp
c-w--w--w- 1      bin      bin      6, 0    May 14 09:14 /dev/lp
$
```

kde význam vypsáných atributů speciálního souboru se shoduje s významem atributů obyčejného souboru kromě místa, kde bývá uvedena velikost v počtu slabik. Zde jsou zobrazena dvě čísla oddělená čárkou (6, 0), která nazýváme (po řadě) hlavní číslo speciálního souboru (major number) a vedlejší číslo speciálního souboru (minor number). Hlavní číslo

## 2.3 Základní schéma

je identifikace typu zařízení (tiskárna, disk, ...) a vedlejší určuje specifika konkrétní periferie (např. o kterou periferii v pořadí se jedná a jakým způsobem bude ovládána).

Vzhledem k uvedenému příkladu může uživatel psát data na tiskárnu ze souboru např. `ztext` takto:

```
$ cat ztext > /dev/lp
```

což mu umožní jednak přesměrování a jednak atribut přístupu pro zápis k souboru `/dev/lp`. Přestože je v mnohých instancích zápis na tiskárnu takto povolen, je lépe právo pro zápis ponechat pouze superuživateli, a ostatním uživatelům umožnit tisk na tiskárnu pomocí mechanismu spooling. To znamená, že požadavky na tisk jsou řazeny do fronty typu FIFO (First In, First Out) a postupně odebírány a posílány do speciálního souboru operačním systémem. Tisk přesměrováním dvou uživatelů současně totiž způsobí na tiskárně smíchání obsahů obou souborů. Pomocí mechanismu spooling uživatel tiskne příkazem

```
$ lp ztext
printer-268
```

obsah souboru `ztext`. Výpis textu `printer-268` je označení ve frontě požadavků na tisk. V době tisku (nebo jen existence ve frontě) může uživatel použít

```
$ cancel printer-268
```

a tím požadavek z fronty zruší.

Jiným příkladem je úschova dat na přenosné médium, jako je např. disketa nebo magnetická páska. Magnetickou pásku ve většině systémů využívá pouze superuživatel pro úschovu dat a v rámci celého systému souborů, takže speciální soubor magnetické pásky může např. vypadat:

```
$ ls -l /dev/rmt
crw-rw---- 1      root    root      10, 0   May 15 10:27 /dev/rmt
$
```

kde jsou práva pro zápis a čtení obyčejnému uživateli odebrána.

U diskety se předpokládá lokální úschova dat. Uživatel proto soubor

```
$ ls -l /dev/rfd
crw-rw-rw- 3      bin      bin       2, 0    May 15 09:24 /dev/rfd
$
```

využívá pro zápis i čtení např. archivačním programem `tar(1)`. Příkaz

```
$ tar cf /dev/rfd ztext
```



provede archivaci souboru `ztext` z pracovního adresáře na disketu (volba `c=create` je vytvoření archivace, `f=file` znamená, že následující argument příkazu je určení speciálního souboru). `tar(1)` je definován při použití voleb bez znaku `-`.

```
$ tar tf /dev/rfd
```

vypíše obsah archivace (table) a

```
$ tar xf /dev/rfd ztext
```

soubor se jménem `ztext` z diskety přesouvá do pracovního adresáře (`extract`). Není-li uvedeno jméno souboru, je přesouván celý obsah archivace.

Jména speciálních souborů se řídí konvencemi danými výrobcem, nikoliv obecnou normou. Přesto existují zvyklosti pro jména speciálních souborů, jako např.:

<code>console</code>	operátorská konzola
<code>clock</code>	hodiny
<code>fd</code>	disketa
<code>dsk</code>	disk
<code>kmem</code>	operační paměť jádra
<code>lp</code>	tiskárna
<code>mem</code>	operační paměť
<code>mt</code>	klasická magnetická páska
<code>tty</code>	terminál

kde uvedené texty zastupují pouze používaný základ jména, protože např. disketa pro sekvenční přístup (po znacích) mívá předponu `r`, tj. `rfd` (stejně tak magnetická páska) a dále u více připojených periférií stejného typu je příponou jména pořadí periferie, např. `tty00`, `tty01`, ...

Přístup k periférii je buď znakový nebo blokový. Typicky blokové zařízení je disk, znakové terminál. Ale i s diskem můžeme pracovat sekvenčně po znacích, proto je i pro něj vytvořen znakový speciální soubor. Znakový speciální soubor má při výpisu `ls -l` atributů v prvním sloupci znak `c`, blokový znak `b`.

Speciální soubor vytváří privilegovaný uživatel pomocí příkazu `mknod(8)`, kdy respektuje všechny vazby na operační systém a připojené periferie (viz kap. 9).

Změna atributů souboru (ať už obyčejného, adresáře, speciálního) znamená změnu dat v i-uzlu. Uvedme si pro tyto změny několik příkladů.

Pomocí příkazu `chown(1)` můžeme měnit vlastníka souboru. Příkazem

```
$ chown root ztext
```

### 2.3 Základní schéma

předáme vlastnictví souboru `ztext` uživateli `root`. Opětné navrácení souboru uživateli `petr` ale může uskutečnit pouze uživatel `root`. Obdobně lze pomocí příkazu `chgrp(1)` změnit příslušnost souboru ke skupině.

Příkaz `chmod(1)` mění přístupová práva souboru. Změnu můžeme zapsat buď číselně, např.

```
$ chmod 775 davka
```

což je specifikace přístupových práv v osmičkové soustavě. Souboru se jménem `davka` budou přístupová práva nastavena způsobem

```
rwX rwx r-x
111 111 101
 7   7   5
```

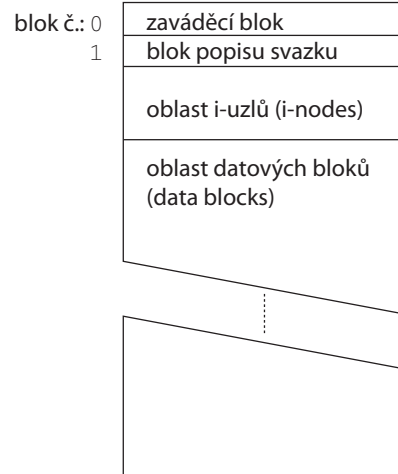
což odpovídá hodnotě jednotlivých cifer v osmičkové soustavě vždy pro trojici bitů oprávnění. Mnemonický zápis vychází ze zadání vlastníka souboru (`u=user`), skupiny (`g=group`), ostatních (`o=others`) nebo všech (`a=all`) uživatelů a označení přístupových práv `r` (`r=read`) pro čtení, `w` (`w=write`) pro zápis a `x` (`x=execute`) pro provádění.

Dále je možné vlastníkovi, skupině nebo ostatním právo nastavit (=), připojit (+) nebo odebrat(-). Např.

```
$ chmod a=rx,g+w,u+w ztext
```

je nastavení, které odpovídá předchozímu příkladu s numerickým nastavením (nejprve je všem nastaveno pouze právo čtení a provádění a dále je přístup pro zápis přiznán nejprve skupině a pak vlastníkovi souboru).

Systém souborů je realizován na magnetickém médiu. Logická struktura magnetického média je svazek (angličtina používá výraz `filesystem`, v překladu systém souborů). Logiku vnitřní struktury svazku ukazuje obrázek 2.2.



Obr. 2.2 Struktura svazku

Z obrázku je nám známý výraz `i-uzel`. Víme, že `i-uzel` je jednoznačná identifikace souboru, obsahuje jeho atributy, a že s číslem `i-uzlu` je v adresáři spojeno jméno souboru. Číslo `i-uzlu` je jeho pořadí v rámci oblasti `i-uzlů`. Velikost `i-uzlu` je 64 slabik a `i-uzel` obsahuje tyto informace:

- vlastníka souboru
- typ souboru (obyčejný soubor, adresář, ...)
- přístupová práva

- datum a čas poslední manipulace se souborem
- počet odkazů na soubor z různých míst stromu adresářů
- tabulka datových bloků
- velikost souboru

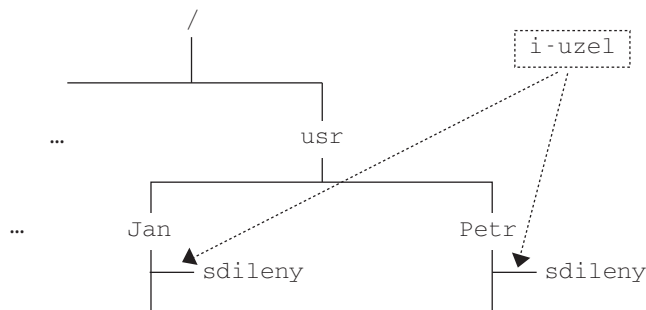
Položka počet odkazů (number of links to the file) je možnost přístupu k témuž souboru z několika různých míst systému souborů, např.:

```
$ pwd
/usr/petr
$ cat > sdileny
^d$
```

(vytvoření souboru s nulovou délkou). Za předpokladu, že soubor `sdileny` dosud v pracovním adresáři (`/usr/petr`) neexistoval, je z oblasti i-uzlů vybrán první neobsazený, souboru je v adresáři přiděleno odpovídající číslo i-uzlu a současně s alokací položek i-uzlu je nastaven počet odkazů na 1. Příkazem

```
$ ln sdileny ../jan/sdileny
```

vytváříme nový odkaz na tentýž i-uzel (za předpokladu, že existuje adresář `/usr/jan` a jeho přístupová práva jsou pro nás příznivá). Situaci ukazuje obrázek 2.3.



Obr. 2.3 Odkaz na tentýž soubor

Z obou adresářů je možný přístup k datům téhož souboru. Je-li `/usr/jan` domovský adresář uživatele `jan` a v systému je současně přihlášen uživatel `petr` i `jan`, současně ze dvou různých terminálů je možné soubor `/usr/petr/sdileny` alias `/usr/jan/sdileny` číst a současně do něj zapisovat (podle nastavených přístupových práv). Můžeme to prověřit příkazem

```
$ ls -i sdileny
829 sdileny
$ ls -i ../jan/sdileny
829 ../jan/sdileny
$
```

## 2.3 Základní schéma

když pomocí volby `-i` požadujeme kromě výpisu jména souboru i číslo odpovídajícího i-uzlu.

Rušíme-li libovolný soubor (příkazem `rm(1)`), systém souborů (pomocí rutin jádra) prohlíží i-uzel a snižuje hodnotu počtu odkazů o 1. Je-li po této dekrementaci hodnota větší než 0, systém rozpojí pouze vazbu jméno souboru – i-uzel. Je-li ale hodnota i-uzlu nulová, i-uzel je uvolněn ze seznamu alokovaných a převeden do seznamu volných, datový obsah souboru je zrušen a soubor i se svou vnitřní strukturou zaniká. Např.:

```
$ rm sdileny
```

zruší odkaz z adresáře `/usr/petr`. I-uzel zůstává alokován, s ním všechna data souboru.

```
$ rm ../jan/sdileny
```

zruší poslední odkaz na i-uzel a tím i soubor.

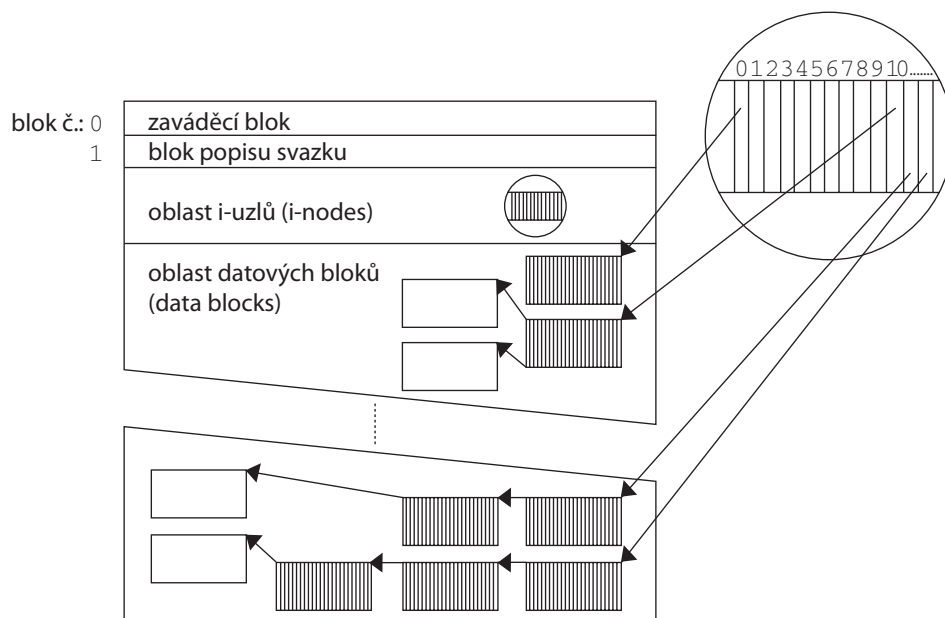
Uvedený způsob práce s více odkazy na jedna data se omezuje na práci s jedním svazkem. Protože by takové zúžení bylo nepraktické, je možné použít odkaz na soubor, který existuje na jiném svazku, k tomu má příkaz `ln(1)` volbu `-s`. Odkaz na soubor jiného svazku je nepřímý odkaz (symbolic link) a v případě výpisu adresáře pomocí `ls(1)` jej rozpoznáme podle znaku `l` v prvním sloupci. Nepřímý odkaz je realizován příznakem v i-uzlu a tak, že v datové části je uložena cesta k souboru na jiném svazku. Nepřímé odkazy přinášejí komplikace správci systému, spojí-li svazky do stromu adresářů jinak než obvykle, protože v případě, že svazek není připojen k odpovídajícímu adresáři, nepřímý odkaz nevede nikam (připojování svazků je popsáno v následujícím textu).

Odkaz (přímý i nepřímý) může být vytvořen i na adresář, je to ale dovoleno pouze privilegovanému uživateli.

Přístup k souboru v UNIXu je obecně vícenásobný. Za předpokladu vhodných přístupových práv mohou různí uživatelé současně číst nebo i zapisovat do jednoho souboru. Jádro pouze zajišťuje výlučný přístup k obsahu souboru v daném čase. V případě, že data sdílí několik uživatelů (např. při práci na společném projektu), musí volit vedoucí projektu (v součinnosti se správcem systému) jiné mechanismy pro zajištění konzistence dat. Jednou z možností je zamykání určité části dat souboru proti zápisu (nebo i čtení) po stanovenou dobu. Zamykání souboru je podporováno voláním jádra `fcntl(2)` a bude diskutováno v kap. 5.

Tabulka datových bloků v i-uzlu má následující organizaci. Prvních 10 hodnot tabulky (0–9) odkazuje přímo na datové bloky. Jedenáctá hodnota je odkaz na blok nepřímé reference, což znamená, že teprve odkazovaný blok obsahuje vlastní odkazy na datové bloky. Je-li vyčerpán, dvanáctá hodnota je odkaz na blok druhé nepřímé reference (odkaz na blok obsahující odkazy na bloky teprve s referencí na data) a je-li i tato možnost vyčerpána, třináctá hodnota je odkaz na blok třetí nepřímé reference. Situaci ukazuje obr. 2.4:

Počet i-uzlů i velikost datové oblasti jsou dány ve chvíli stavby svazku, kterou superuživatel uskuteční pomocí příkazu `mkfs(8)`. Argumentem příkazu `mkfs(8)` musí být speciální soubor média (čili disku), na kterém svazek vytváříme. Dalším povinným argumentem je velikost svazku v blocích, tj. celkový počet datových bloků na médiu (`mkfs(8)` sám odebere jejich část pro režii svazku).



Obr. 2.4 Část i-uzlu pro alokaci datových bloků

Např.:

```
# mkfs /dev/dsk/0s2 40000
```

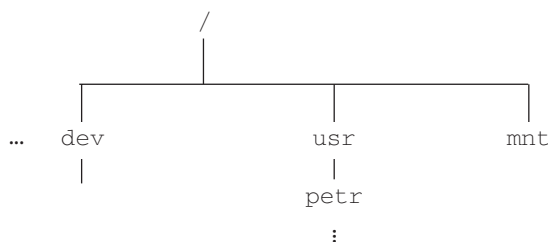
vytvoří na disku prázdný svazek; disk je předpokládané kapacity 40 MB (v případě velikosti diskového bloku 1 KB, velikost diskového bloku je dána typem implementace). Oblast i-uzlů je v tomto případě vypočtena odhadem z velikosti svazku (tento parametr můžeme explicitně v příkazu `mkfs (8)` ovlivnit – viz dokumentace `mkfs (8)`).

Blok popisu svazku (super block) obsahuje globální informace o svazku. Jsou to:

- velikost svazku
- počet volných datových bloků svazku
- seznam neobsazených datových bloků svazku
- ukazatel na první neobsazený blok v seznamu volných bloků
- velikost oblasti i-uzlů
- počet neobsazených i-uzlů svazku
- seznam volných i-uzlů svazku
- ukazatel na první neobsazený i-uzel
- pole zámků pro neobsazené datové bloky a neobsazené i-uzly
- příznak modifikace bloku popisu svazku

Uživatelé operujícímu v rámci systému souborů na svazku ukazuje jádro hierarchickou strukturu adresářů. Od kořenového adresáře systém souborů pokračuje přes podadresáře atd. (viz např. obr. 2.5).

## 2.3 Základní schéma



Obr. 2.5 Fragment svazku z pohledu uživatele

systému souborů, ostatní svazky (na dalších discích, disketách...) jsou pak připojovány k tomuto kořenovému svazku na některý z prázdných adresářů. Mějme např. nově vytvořený svazek na médiu se speciálním souborem `/dev/dsk/0s2`. Příkazem

```
# mount /dev/dsk/0s2 /mnt
```

jej připojíme k adresáři `/mnt` kořenového svazku. Systém souborů je takto rozšířen o další oblast, o další podstrom adresářů. Odkazy na disková média tedy nezadává uživatel, ale definuje je správce systému při startu výpočetního systému. Superuživatel odpojí svazek příkazem `umount` (8), např.:

```
# umount /dev/dsk/02s
```

Uvedený způsob uložení dat na svazku je způsob původní, který současné typy svazků nevyužívají, a to z důvodů zpomalení přístupu k datům při větších svazcích a delším používání. Typ svazku souvisí s vnitřní organizací datové části svazku. V současné době jsou známy např. typy FFS (Fast File System) nebo AFS (Acer File System), které obvykle využívají způsobu organizace svazku navrženého ve 4.3BSD systému (viz [7]) z r. 1984, na rozdíl od označení FS5 (FS5 pracuje s daty uvedeným způsobem). U nové organizace dat jde především o efektivní organizaci svazku za libovolně dlouhého používání a o urychlení přístupu k datům. Svazek má několik míst, kde jsou uloženy informace bloku popisu svazku. Každý blok popisu svazku je dále následován tzv. mapou skupiny cylindrů, jejíž součástí je mapa volných bloků skupiny cylindrů. Oblast i-uzlů následuje vždy za mapou skupiny cylindrů, jejíž součástí je mapa volných bloků skupiny cylindrů. Oblast i-uzlů následuje vždy za mapou skupiny cylindrů (oblasti i-uzlů je tedy také více na svazku). I-uzel má délku 128 slabik a místo deseti přímých odkazů na data jich má dvanáct.

V rámci kořenového svazku dostává smysl zaváděcí blok (boot block). Na svazku s kořenovým adresářem je naplněn strojově závislým programem, který je zavlečen do paměti na pokyn mikroprogramů (firmware, pevné programové vybavení holého stroje) při zapnutí počítače, a po spuštění zavádí a startuje UNIX.

Mezi jádro a systém souborů je vložena systémová vyrovnávací paměť (buffer cache). Její princip za chodu systému je následující: mezi data na připojeném svazku a datovou oblast uživatele je v operační paměti vložena vyrovnávací paměť, do které jádro načítá data po blocích a teprve poté je přenáší do prostoru uživatele. UNIX obecně předpokládá sekvenční

přístup k datům, a proto je následující blok dat souboru již připraven ve vyrovnávací paměti. Velikost systémové vyrovnávací paměti je silně závislá na typu instalace, počtu uživatelů pracujících současně v systému a na typu aplikací, které využívají. Není-li vyrovnávací paměť dostatečně dimenzována vzhledem k uvedeným okolnostem, je pro chod systému spíše zátěží a brzdou než výhodou. Problém totiž nastává, je-li vyrovnávací paměť plně po blocích alokována pro uživatele, kteří se všemi daty aktivně pracují (modifikují je), a algoritmy pro uvolnění některého z bloků vyrovnávací paměti neuspějí. Tehdy je uživatel odstaven do chvíle, kdy je pro něj jinými uživateli uvolněn dostatečný počet bloků. Celková průchodnost operačního systému se tím sníží, ačkoliv stačí systémovou vyrovnávací paměť zvětšit odpovídajícím parametrem jádra (viz kap. 9). Na druhé straně, je-li vyrovnávací paměť příliš velká, může tak zbytečně omezit operační paměť pro práci uživatelů v systému.

Data v systémové vyrovnávací paměti, která uživatel změnil a doposud nebyla přepsána na disk (což uživatel neví), jsou v nebezpečí ztráty v případě havárie počítače (výpadek el. proudu, nekvalitní technické vybavení). Nebezpečí nesouvisí jen se ztrátou datových bloků, ale také s nekonzistencí mezi daty a strukturou systému souborů. Všechny připojené svazky jsou totiž ve vyrovnávací paměti virtualizovány, takže jsou v ní uloženy i všechny bloky popisu svazku atd. Kolize ztráty dat nebo porušení konzistence svazku částečně řeší UNIX příkazem

§ sync

(je dostupný libovolnému uživateli), na jehož základě jádro provede aktualizaci dat na připojených svazcích vzhledem k obsahu vyrovnávací paměti. Za normálního chodu systému je tato akce obvykle automaticky prováděna každých 30 vteřin.

Výhoda přímého přístupu k datovým blokům na disku se výrazně projeví v manipulaci přesunu (seek). Uživatel může programově za pomoci volání jádra `lseek ( 2 )` přesouvat pozici čtení nebo zápisu v otevřeném souboru na pozici jinou, byť mnoho slabik vzdálenou. Jádro vypočítá pořadí bloku dat, který má být uživateli takto zpřístupněn, a podle tabulky alokovaných bloků v i-uzlu přesune do vyrovnávací paměti odpovídající blok dat, aniž by musela prohledávat předchozí datové bloky. Přesunem lze v mnoha případech zefektivnit práci aplikace s daty s pevnou délkou formátu.

## 2.4 PROCESY

Obdobně jako systém souborů, tvoří i procesy hierarchickou strukturu. Je-li nastartován operační systém, tj. je-li spuštěno jádro, vzniká za jeho podpory proces nazývaný **swapper** (někdy **sched**). Je to proces, který je identifikován jako proces č. 0. Jeho hlavním smyslem je pracovat pro údržbu správy paměti, ale také, ihned na začátku své činnosti, sám vytváří další proces pomocí odkazu na jádro. Tento další proces je nazýván **init** a má identifikační číslo 1. Proces, který vznikne odkazem na jádro z jiného procesu, nazýváme procesem synovským (child) vzhledem k procesu, který jej takto vytváří a který je označován jako otec nebo rodič (parent). Žádosti o vznik dalšího procesu říkáme **fork** (rozvětvení) a proces jej uplatní pomocí volání jádra `fork ( 2 )`. Proces **init** je otcem dalších procesů. **init** udržuje několik úrovní stavu systému, přitom je každá úroveň reprezentována množinou procesů, které **init** vytváří a dohlíží na jejich stav. Hlavní dvě úrovně jsou označovány jako úroveň jednouživatelská (single user) a úroveň

## 2.4 Základní schéma

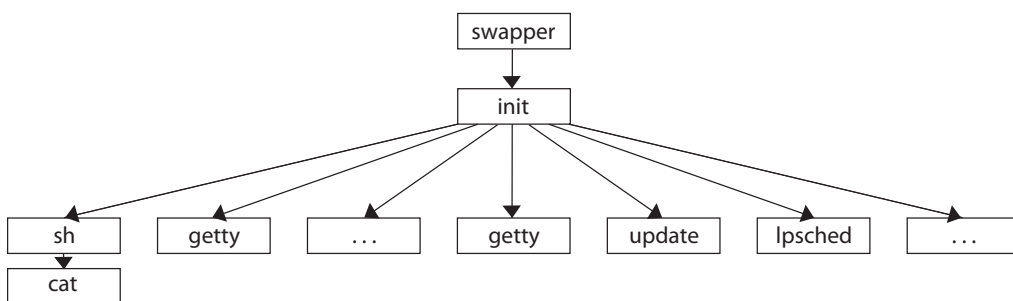
víceuživatelská (multi user). Na úrovni jednouživatelské vytváří **init** jen několik dalších procesů, které umožňují vstup do systému pouze jednomu uživateli, a to privilegovanému. Je to úroveň pro údržbu a celkové změny v systému. Úroveň víceuživatelská je standardní úroveň pro běh systému, kdy se interaktivně z terminálů uživatelé přihlašují do systému a pracují v něm.

Aby proces **init** umožnil uživateli vstup do systému, vytváří proces, jehož standardním vstupem i výstupem je terminál. Takový proces je nazýván **getty** a komunikuje s uživatelem v době přihlašování. Procesům **getty** (jejich počet je dán počtem terminálů) jádro přiděluje pro jednoznačnou identifikaci poslední přidělené číslo procesu zvýšené o 1. Proces **getty** se v případě správného přihlášení mění voláním jádra `exec(2)` na proces **sh**, který realizuje běžnou komunikaci uživatele se systémem a je nazýván příkazovým interpretem (např. Bourne shell), **sh** na obrazovce vypisuje znak `$` (nebo `#`) a interpretuje příkazový řádek uživatele.

Příkazový řádek uživatele, např.

```
$ cat ztext
```

je interpretován procesem **sh** vytvořením nového procesu (voláním jádra `fork(2)`) se jménem **cat**. **sh** je tedy rodičem **cat**. Hierarchickou strukturu stromu procesů ukazuje obr. 2.6.



Obr. 2.6 Hierarchická struktura procesů

Na obrázku jsou navíc procesy **update** a **lpsched**, které jsou příkladem dalších synovských procesů **initu** a přitom se nevztahují k žádnému terminálu. Takové procesy slouží k zajištění určitých akcí iniciovaných systémem na základě určité události (např. při přechodu systému na jinou úroveň). Část takových procesů běží v nekonečné smyčce, probouzí se přitom k akci na základě časového limitu nebo pokynu jádra. Takovému procesu říkáme démon (daemon) a příkladem může být právě proces **update**, který periodicky zajišťuje spooling tiskárny.

Identifikační číslo procesu má zkratku PID (process identification) a je to atribut každého procesu pro jeho jednoznačné odlišení od ostatních procesů jádrem, uživatelem nebo kterýmkoli procesem. PID je celé kladné číslo a je přidělováno nově vznikajícímu procesu podle naposledy použité hodnoty zvýšené o 1. Hodnota PID v instalacích, kde pracuje několik desítek uživatelů současně za nepřetržitého provozu systému několika dnů i týdnů, jistě dosáhne nejvyšší možné hodnoty. Tehdy jádro začíná opět od hodnoty 0. Vynechává přitom ta PID, která jsou přidělena aktivním procesům a měla by tak stejnou hodnotu.



Procesy můžeme rozdělit na systémové (systémem zvýhodňované) a ostatní. Systémový proces je např. **init**, **swapper**, do skupiny ostatních patří **sh**, **cat** atd. Rozdíl mezi systémovým a jiným procesem je zejména ve stanovení jeho dynamické priority, což je číselná hodnota, podle které systém jednomu z procesů přidělí čas procesoru. Víceuživatelský operační systém UNIX zajišťuje sdílení zdrojů výpočetního systému více uživatelům pomocí více procesů, které běží zdánlivě současně. Ve skutečnosti jádro uděluje čas procesoru vždy jen jednomu procesu a jen na omezenou dobu. Vyhoví tak v průběhu velmi krátkého časového intervalu všem procesům, takže se zdá, že procesy běží (uživatelé pracují) současně. Měřítkem přidělení času procesoru jednomu z procesů je dynamická priorita procesu. Je vypočtena pro každý proces z jeho výchozí uživatelské priority, což je celé kladné číslo v intervalu obvykle 0–39 (menší hodnota znamená vyšší prioritu), a z času systému. Dynamická priorita je vypočítávána všem procesům vždy po uplynutí 1 vteřiny, anebo na základě interakce procesu s jádrem. Protože interakce z terminálu znamená zpracování vstupu jádrem, zvýhodňovány jsou procesy podporující práci u terminálu. Z vlastnosti zvýhodňování interaktivních procesů jádrem plyne pro UNIX označení interaktivní operační systém. Hranicí v hodnotě výchozí priority je hodnota 20, která je dělicím bodem mezi systémovými a jinými procesy. Žádný obyčejný proces nemůže mít po dobu svého běhu nastavenou dynamickou prioritu na hodnotu nižší než je 20. Tím je co nejvíce zajištěna průchodnost systému a ochrana proti zahlcení.

Výchozí priorita při spuštění procesu je právě 20. Proces se ale může části své priority vzdát ve prospěch ostatních. Při spouštění procesu např. lze psát

```
$ nice -5 find / -name core -print > fcore
```

kdy před vlastní příkaz `find ( 1 )` přepíšeme příkaz `nice ( 1 )`. Volba `-5` je hodnota, která udává, o kolik bude výchozí priorita spouštěného procesu pro `find ( 1 )` nižší. V našem případě bude výchozí priorita procesu **find** 25. Napíšeme-li

```
$ nice find ...
```

výchozí priorita je implicitně snížena o 10 (je tedy 30). Privilegovaný uživatel může prioritu spouštěného procesu i zvyšovat, a to způsobem

```
# nice -10 ...
```

takže výchozí priorita je o 10 zvýšena ( $20 - 10 = 10$ , výchozí priorita je 10).

Dříve než může být procesu přidělen procesor, musí být proces zaveden do operační paměti. Ve chvíli, kdy je proces vytvořen, je snahou jádra přidělit mu požadovanou paměť. Není-li dostatečně velká požadovaná paměť volná, jádro uplatňuje na procesy v operační paměti algoritmy se snahou odsunout některé procesy mimo paměť a zajistit tak dostatek volné paměti pro nový proces. Orientace jádra zde vychází z posouzení doby procesu již strávené v paměti. Proces nejdéle sídlící v paměti je odsunut. Místo, kam jsou procesy



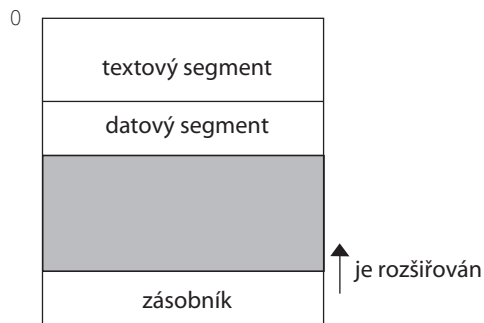
kde `soubor` je jméno souboru s programem. Jádro po skončení procesu, který interpretuje program `soubor`, okopíruje obsah souboru s programem do odkládací oblasti a toto umístění si poznamená. Při opětovném spuštění programu jádro využívá přímého kontaktu s textem programu v odkládací paměti, takže doba spuštění se tím zkrátí.

Uživatel přihlášený do systému je vlastníkem všech procesů, které interaktivně spustil. V případě, že se nejedná o privilegovaného uživatele, jádro při spouštění zjišťuje přístupová práva k souboru, v němž je program uložen. Nemá-li uživatel přístupová práva pro provádění (x-bit), program nemůže být spuštěn. Dále je možné, že může program spustit, ale využívá některé volání jádra, na které má oprávnění pouze superuživatel. V tomto případě proces nebude pracovat správně. Je ale běžné, že systém umožní v rámci spuštěného programu privilegovanou akci, a sice tak, že v místě nastaveného bitu proveditelnosti je namísto označení `x` označení `s` (tzv. nastavení s-bitu), např.:

```
# chmod g,o+s soubor
```

Princip je v tom, že je uživatel v akci regulován programem, který vlastní superuživatel. Nastavením s-bitu může také obyčejný uživatel dát k dispozici kontrolované manipulace se svými daty ostatním uživatelům.

Proces je v době běhu v paměti rozdělen na tři části – segmenty (viz obr. 2.8).



Obr. 2.8 Adresový prostor procesu

Textový segment obsahuje instrukce programu a v datovém segmentu jsou datové struktury programu. Pokud není řečeno jinak, UNIX považuje textový segment za část, která není v době běhu procesu měnitelná a je-li vytvořeno několik procesů s odkazem na tentýž program (např. editor `vi` (1), program `getty` (8), příkazový interpret `sh` (1) atd.), běžící procesy sdílí tentýž textový segment. Každý takový nově vzniklý proces proto požaduje přidělení operační paměti pouze pro datový segment a zásobník.

Procesy jsou sdružovány do skupin, které nazýváme rodiny procesů. Každý proces má právo prohlásit se (pomocí volání jádra `setpgid` (2)) za vedoucího skupiny (otce rodiny), a tím se osamostatnit od rodiny, do které dosud patřil. Každý shell se po přihlášení uživatele prohlašuje za otce rodiny, osamostatňuje se tak a všichni jeho synové (nezaloží-li si také svoji rodinu) jsou s ním existenčně spjati. Ukončí-li totiž otec rodiny svoji činnost, všichni jeho synové umírají také (pokud nejsou v supervizorovém režimu). V řádkové komunikaci u terminálu může uživatel příkazovým řádkem končícím na znak `&` vytvořit proces, který běží, a uživatel může zadávat další příkazy, tj. vytvářet další procesy, např.

```
$ cc velkyprog.c &
159
$
```

## 2.4 Základní schéma

(překlad programu v jazyce C), kde vypsané číslo je PID vzniklého procesu. Pokud ale ještě před dokončením takového procesu dá pokyn k odhlášení, ukončí tak činnost nejen procesu **sh**, ale také procesu **cc**, syn je násilně přerušen. Syna prohlásíme za samostatného příkazem

```
$ nohup cc velkyprog.c &
```

a po odhlášení běží proces **cc** až do regulérního ukončení.

Komunikace procesů je na základní úrovni zajištěna zasíláním signálů (signals). Signály jsou významově číslovány od 1, přitom např. přerušení z klávesnice odpovídá signálu č. 2. Např. proces **sh** může poslat svému synovi, jehož PID zná (v příkladu 159), signál č. 2 takto

```
$ kill -2 159
```

Jméno příkazu přitom odpovídá dřívějšímu významu použití signálu. Zaslat procesu signál totiž znamenalo proces ukončit. Výchozí reakce procesu na příchod různých signálů jsou dnes stanoveny a nemusí to být vždy ukončení. Proces také může reakci na příchod signálu měnit, může např. pouze provést určitou akci a pokračovat dál v činnosti, nebo signál ignorovat, anebo provést určitou akci a svoji činnost ukončit. Závisí to na způsobu použití volání jádra `signal(2)`. Proces posílá jinému procesu signál voláním jádra `kill(2)`. Musí ale znát PID procesu, kterému signál posílá.

Komunikace dvou procesů na úrovni přenosu dat je možná pomocí roury (pipe), kdy procesy pracují podle schématu

PRODUCENT | KONZUMENT

tak, že proces PRODUCENT předává svá výstupní data na vstup procesu KONZUMENT.

Schéma odpovídá zápisu spojení dvou procesů v příkazovém řádku. Spojení je provedeno tak, že standardní výstup procesu PRODUCENT je přepnut do části paměti pro rouru a standardní vstup procesu KONZUMENT je na tutéž paměť napojen jako proces, který z roury data čte. Roura pracuje sekvenčně způsobem FIFO, KONZUMENT dostává nejprve data, která PRODUCENT do roury zapsal jako první. Po příkazovém řádku

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

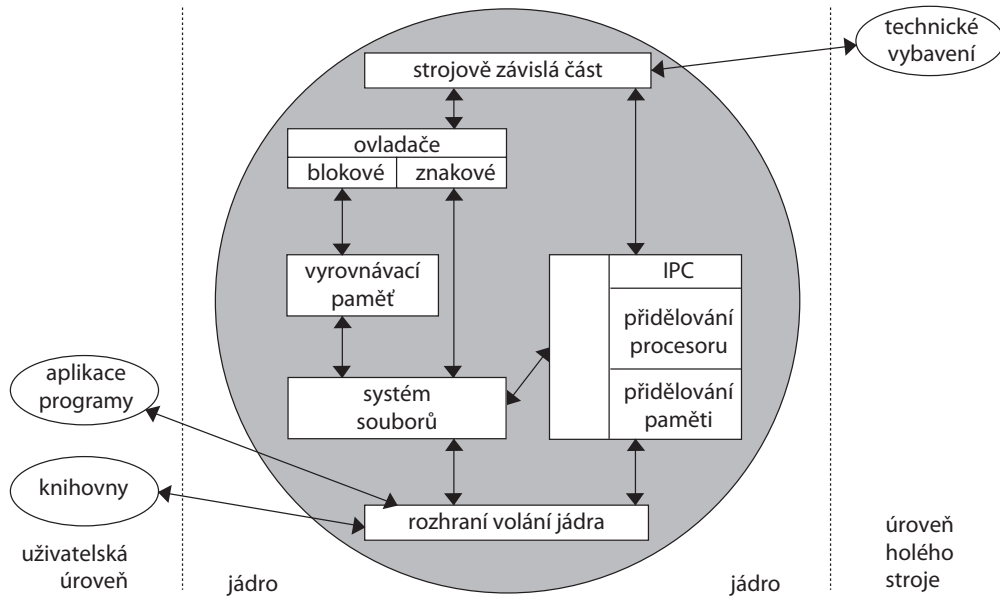
nebude výstup procesu **ls** (seznam souborů) vypisován na obrazovku terminálu, ale bude zapisován do vytvořené roury a proces **wc** (příkaz s volbou `-l` počítá řádky standardního vstupu) bude zpracovávat data čtená z roury. Jádro zajistí pozastavení procesu PRODUCENT, je-li roura plná, anebo procesu KONZUMENT, je-li roura prázdná. PRODUCENT uzavírá rouru znakem konce souboru (CTRL-d) a KONZUMENT po přečtení konce souboru také, čímž roura zaniká.

Komunikace procesů pomocí roury je omezena na procesy, které jsou v přímém příbuzenském vztahu. Takto nemohou např. spolupracovat procesy vytvořené různými uživateli. Tento a další problémy při komunikaci procesů řeší část systému nazývaná jako IPC (Interprocess Communication).

V rámci IPC je možné vytvářet a používat pojmenovanou rouru (named pipe), kdy je roura vytvořena trvale a je spojena se jménem souboru (zvláštního typu). Dále je v IPC možné, aby dva libovolné procesy spolupracovaly na úrovni předávání zpráv (messages), je možné, aby sdílely data (shared memory) svých datových oblastí, a konečně je využitelný mechanismus Dijkstrových semaforů (semaphores). Tato část je podrobně diskutována ve čl. 5.4. IPC ale řeší situaci nejen v rámci jednoho výpočetního systému, ale pomocí schráněk (sockets) také v počítačových sítích.

## 2.5 JÁDRO

Jádro (kernel) je programové vybavení, které pracuje na počítači bez jakékoliv další programové podpory. Je velkým rozhraním mezi uživatelem a technickým vybavením výpočetního systému. Zajišťuje realizaci odkazů uživatelů nebo procesů na periferie a vůbec všechny technické zdroje počítače. Dále udržuje a podporuje v činnosti systém souborů a běžící procesy. Celkovou strukturu jádra ukazuje obr. 2.9.



Obr. 2.9 Struktura jádra

Uživatelská úroveň je jádrem podporována přes rozhraní volání jádra. Stručně řečeno, program používá volání funkce, která je voláním jádra, a program tak vstupuje do režimu obsluhy jádrem (proces vstupuje do supervizorové úrovně). Činnost jádra je v okamžiku, kdy zajišťuje určitou akci vyžádanou procesem, jiným procesem nepřerušitelná. Vzhledem k časové prodlevě, po kterou proces v supervizorové úrovni zůstává, není vhodný pro řízení procesů v reálném čase. Snaha vyřešit problém reálného času je dlouholetá, její výsledky se zatím odrážejí v operačním systému HP-UX od firmy HEWLETT PACKARD, kde je jádro

## 2.6 Základní schéma

upraveno tak, aby doba strávená v supervizorové úrovni procesu byla minimalizována. Jiným příkladem úpravy operačního systému UNIX pro reálný čas je LynxOS rovněž americké firmy Lynx Real-Time Systems. I verze UNIX SYSTEM V.4 má podle dokumentace možnost práce procesů v reálném čase. Praktické zkušenosti ale ukazují, že jde prozatím o úpravu jádra, která nemá reálné využití (interakce ostatních uživatelů se výrazně zpomalí). SVID3 práci v reálném čase už ale zahrnuje (viz Příloha D – `priocntl(2)` a ostatní dokumentace UNIX SYSTEM V.4).

### 2.6 UŽIVATELÉ

Uživatel je poslední entita, která souvisí se základy operačního systému UNIX. Viděli jsme v čl. 2.4, že v době, kdy je uživatel v systému přihlášen, je dynamicky reprezentován akcemi, které vykonává pomocí skupiny procesů. Vzhledem k tomu, že je mu v době přihlášení přidělen nový proces shellu, který sám sebe prohlásí za otce rodiny, odpovídá uživateli rodina procesů tohoto shellu.

Staticky je uživatel reprezentován všemi soubory, u kterých je označen jako vlastník (zejména jsou to data začínající jeho domovským adresářem). Dále je to uživatelova registrace v tabulkách souborů `/etc/passwd` a `/etc/group`. V `/etc/passwd` odpovídá registraci jednoho uživatele 1 řádek souboru. Na řádku jsou uvedeny jednotlivé položky uživatele oddělené znakem `:`. Např.

```
...
petr:*:236:50:Petr Nevecny, tel 537973:/usr/petr:/bin/sh
...
```

První položka na řádku je jméno uživatele. Druhá je jeho heslo, které je zde v zakódované podobě. V současných systémech je na místě položky hesla, pokud je přístup heslem chráněn, uveden znak `*` (nebo `x`), a šifrované heslo je uloženo na jiném, běžně nesdělovaném místě (např. v `/etc/.secure`). Následující položka je číselná identifikace uživatele, která je v rámci instalace jedinečná (zde 236). Tato hodnota je např. uložena v `i`-uzlu souboru, který uživatel vlastní. Dále následuje identifikace skupiny, do které uživatel patří (50).

Položka identifikace skupiny koresponduje se souborem `/etc/group`, kde je uveden seznam všech skupin, na každém řádku jedna, např.

```
...
group:50:jan,petr
...
```

a položky mají po řadě význam: jméno skupiny, heslo (uvedená skupina je bez hesla), číselná identifikace a seznam uživatelů patřících do skupiny a oddělených znakem `,`. Uživatel 236 tedy patří do skupiny se jménem `group`.

Další položka `/etc/passwd` po čísle skupiny je komentář. Následuje cesta k domovskému adresáři, podlení shell nastavuje uživateli pracovní adresář v době přihlášení. Konečně poslední položkou je cesta k programu, který bude spuštěn jako příkazový interpret pro přihlášení uživatele. Tato položka bývá naplněna buď `/bin/sh` (Bourne shell) nebo

`/bin/csh` (C-shell), anebo kteroukoliv jinou aplikací, která je schopna vést dialog s uživatelem u terminálu. Výhodou je, že při ukončení činnosti (ať už regulérním nebo z důvodů havárie programu) je uživatel odhlášen, takže nezůstává po ukončení aplikace v systému a nemůže neodborným zásahem poškodit uložená data. Je-li v souboru `/etc/passwd` poslední položka na řádku vynechána (řádek končí znakem `:`), uživateli je standardně spouštěn program souboru `/bin/sh`.

## 3. Hierarchie adresářů

### 3.1 JMÉNO SOUBORU

Jméno souboru je 14 znakový textový řetězec, který nesmí obsahovat znak binární nuly `\0` a znak lomítko `/`. Znak `\0` je pro UNIX znakem konce textového řetězce (a tedy i jména souboru) a znak `/` je používán pro určení místa souboru ve struktuře adresářů. Některé systémy (např. 4.3BSD) dovolují řetězec jména souboru prodloužit na 255 znaků, vyžadují ale omezení osmého paritního bitu znaku (nelze pojmenovat soubor v národním prostředí).

Pro konvence jména souboru není obecně žádné další omezení. V kap. 4 se seznámíme s metajazykem komunikace uživatele se systémem (příkazových interpretů shell) a poznáme, že ve jménu souboru se nedoporučuje používat řídicí znaky metajazyka (např. znak `*` atd.). Stejně tak se (dle kap. 2) pochopitelně nepoužívá např. jako jméno souboru řetězec `..` nebo `..`.

Konvence pro jména souborů nejsou striktně zavedeny, určité dále popsané konvence se vztahují pouze na aplikaci, která se souborem pracuje, nikoliv obecně na UNIX. Znak `.` nemá také obecně ve jménu souboru zvláštní význam, ten mu přisuzuje zase jen konkrétní nástroj programátora nebo aplikace. Znak `.` může být použit na libovolném místě jména souboru, např. v souboru `.profile` je první znak (tečka) rozhodující pro příkazový interpret Bourne shell, který soubory takového jména v rámci sezení uživatele považuje za soubory řídicí; jejich obsah souvisí s definicí prostředí uživatele sezení. Proto soubory začínající znakem `.` nezařazuje do seznamu datových souborů. Základní úzus pro jména souborů pak dále opět souvisí se znakem `.`, který určuje příponu jména souboru. Následující tabulka ukazuje seznam přípon používaných různými nástroji programátora:

přípona      význam pro obsah souboru

<code>.c</code>	zdrojový text v jazyce C
<code>.h</code>	definice pro program v jazyce C (vsouvaný, tzv. hlavičkový soubor)
<code>.s</code>	zdrojový text v jazyce assembler
<code>.i</code>	zdrojový text v jazyce C po zpracování textovým makroprocesorem
<code>.o</code>	modul programu v přeloženém tvaru
<code>.y</code>	zdrojový text pro nástroj programátora <code>yacc(1)</code> (pro popis gramatiky navrhovaného umělého jazyka)
<code>.l</code>	zdrojový text pro nástroj programátora <code>lex(1)</code> (provádí lexikální analýzu)
<code>.a</code>	soubor s knihovnou přeložených modulů (vytváří nástroj programátora <code>ar(1)</code> )

Soubor s proveditelným obsahem nemá žádnou příponu. Proveditelný soubor je např. `/bin/ls` (soubor se jménem `ls`, který je uložen v adresáři `bin`), který je pro systém program příkazu `ls(1)` pro výpis adresáře. Stejně tak všechny spustitelné uživatelské programy jsou bez přípon a rozlišují tak množinu použitelných příkazů. Přeložit program uložený v souboru se jménem `prog.c` můžeme za pomoci překladače jazyka C (příkaz `cc(1)`) takto:

```
$ cc -o prog prog.c
```



kde volba `-o` stanovuje v následujícím argumentu jméno souboru, ve kterém bude uložena proveditelná podoba programu (přeložená a sestavená). Uživatel potom spouští program zápisem

```
$ prog
```

přítom, umí-li `prog` pracovat s argumenty příkazového řádku, mohou tyto v zápisu následovat.

Příkazem

```
$ cc prog.c
```

vytváříme proveditelnou podobu programu, která bude uložena v souboru se jménem `a.out`, a příkaz

```
$ a.out
```

tento program spouští.

Příkazem

```
$ mv a.out prog
```

můžeme v případě ukončení ladění přejmenovat `a.out` na dále používané jméno souboru. Vzhledem k tomu, že ze jména souboru jednoznačně nevyplývá jeho obsah, je uživateli k dispozici příkaz `file(1)`, který má formát

```
file jméno_souboru ...
```

a který z analýzy prvního diskového bloku souboru rozhodne o tom, zda je soubor binární nebo textový (tj. zda jej budeme prohlížet příkazem `more(1)` nebo `od(1)`), případně `file(1)` pozná, zda se jedná např. o zdrojový text jazyka C, příkazový soubor některého z shellů, o jaký paměťový model proveditelného souboru jde atd.

Vzhledem ke zvyku používat alespoň v určitých případech ve jménu souboru znak `.` pro oddělení přípony charakterizující obsah souboru, část jména souboru před tečkou je označována jako základní část jména souboru (base name).

Příkaz

```
basename jméno_souboru [přípona]
```

vypíše na standardní výstup základní část řetězce parametru `jméno_souboru`. Volitelná přípona je explicitní zadání přípony, např.

```
$ basename muj.c .c
```

```
muj
```

nebo

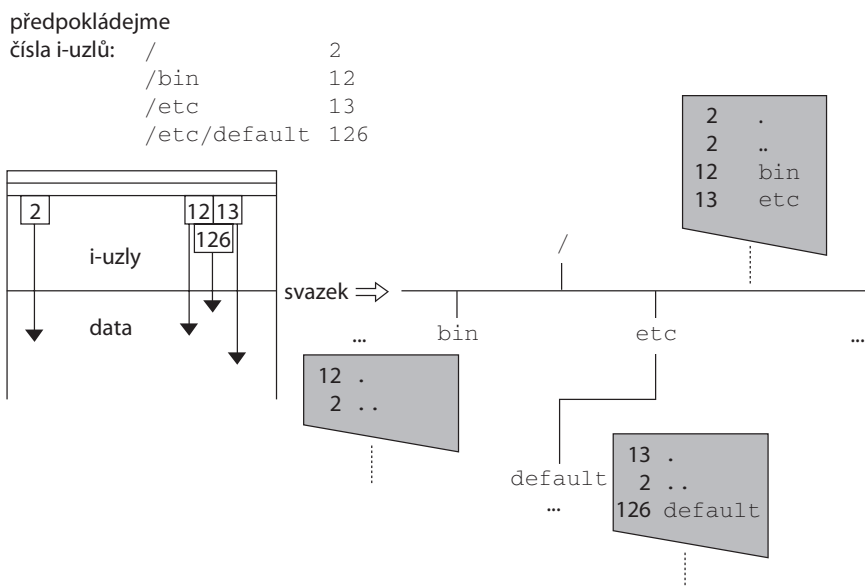
## 3.2 Hierarchie adresářů

```
$ basename ../text.old
text
```

Příkaz `basename (1)` se totiž vztahuje i na předponu jména souboru.

### 3.2 STROM ADRESÁŘŮ

Adresář je soubor, který obsahuje jména souborů s odkazy na jejich atributy a obsah. Součástí obsahu adresáře je soubor se jménem `.`, který je odkazem na sebe sama (i adresář má atributy a obsah) a `..`, který je odkazem na nadřazený adresář. Adresář má takto ve svém obsahu uvedeno, kde je evidován jako jeden ze souborů. Tím je zajištěna provázanost adresářů mezi sebou a z toho také vyplývá stromová struktura všech adresářů systému souborů. Např.



Obr. 3.1 Fragment stromu adresářů

Na obr. 3.1 je u každého adresáře v otevřeném obdélníku schématicky zobrazen jeho obsah. Každému jménu je přiřazeno číslo i-uzlu. Reference výchozího adresáře (se jménem `/`) je pro `.` i `..` tentýž i-uzel. Číslo i-uzlu je ukazatel do oblasti i-uzlů snížený o 1. Je to výjimka oproti obvyklému začátku od 0. 0 zde má totiž význam neobsazeno.

Uživatel používá soubor prostřednictvím jména souboru. Je-li uživatel přihlášen, je mu po celou dobu sezení vždy přiřazen některý adresář. Pracuje-li se soubory tohoto pracovního adresáře (current directory), odkazy mohou být relativní neboli vztažené k tomuto adresáři. Např. je-li naším pracovním adresářem soubor se jménem `etc`, který je podadresářem `/`, můžeme psát

```
$ more passwd
```

a tím si prohlížíte soubor se jménem `passwd`, který je evidován v pracovním adresáři. Oproti relativním odkazům může uživatel používat také odkazy absolutní, kdy nevyužívá nastavení pracovního adresáře. Absolutní odkaz je např.

```
$ more /etc/passwd
```

kdy bude obsah souboru, který je uložen v adresáři `etc`, vypisován stejně jako v předchozím případě, ať už je náš pracovní adresář jakýkoliv.

Ke každému souboru vede cesta (path), v systému souborů někdy označovaná jako předpona. Cesta začíná vždy kořenovým adresářem (root directory), který je označován `/`. V cestě k souboru ho dále následuje seznam adresářů, kterými procházíme, chceme-li získat odkaz na vlastní soubor. Např. `/usr/jan/projekt/proj.c` je odkaz na soubor se jménem `proj.c`, odkaz na něj je přitom uložen v adresáři `projekt`, adresář `projekt` je odkazován z adresáře `jan` a ten je podadresářem adresáře `usr`. Adresář `usr` je evidován jako soubor kořenového adresáře. Vzhledem k možnosti nastavení pracovního adresáře, např.

```
$ cd /usr/jan
```

můžeme používat relativních odkazů, např.

```
$ cat project/proj.c
```

využívá předchozího nastavení pracovního adresáře. Relativní odkaz nezačíná `/` a UNIX jej uvažuje jako pokračování v cestě od pracovního adresáře. V rámci relativního odkazu můžeme používat odkaz na bezprostředně nadřazený adresář pomocí jména `..`. Za předpokladu pracovního adresáře `/usr/jan` např.

```
$ cp ../petr/text text
```

je totéž, co

```
$ cp /usr/petr/text /usr/jan/text
```

při použití absolutních zápisů. Referenci `..` lze používat v každém adresáři, např.

```
$ cat ../../etc/fstab
```

a odkaz `.` používáme pro zjednodušení místo jména pracovního adresáře, např.

```
$ cp /usr/include/*.h .
```

### 3.3 Hierarchie adresářů

kopíruje všechny soubory adresáře `/usr/include`, které končí na `.h`, do pracovního adresáře (znak `*` nahrazuje ve jméně souboru libovolný textový řetězec). Příkaz `cp(1)` má totiž také tvar

```
cp soubor soubor ... adresář
```

Např.

```
$ cp /usr/include/stdio.h /usr/include/file.h .
```

Uživateli přihlášenému do systému je nastaven výchozí adresář. Je nazýván domovským adresářem (home directory) a uživatel jej také nastavuje jako pracovní adresář v průběhu sezení příkazem

```
$ cd
```

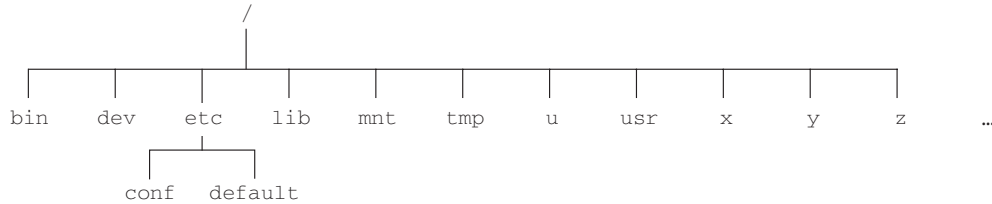
(bez argumentů). Domovským adresářem začíná uživatelova datová oblast. Domovský adresář je uživateli přidělen správcem systému a je evidován v souboru `/etc/passwd` v 6. položce (viz čl. 2.6). Uživatel je jeho vlastníkem a pomocí příkazů `mkdir(1)` a `rmdir(1)` si vytváří a mění svůj datový podstrom. V této části má nejvyšší práva pro přístup k souborům (to je dáno jeho vlastnictvím všech dalších souborů).

Pro čtení je i obyčejnému uživateli dostupný obsah kteréhokoliv souboru (až na některé výjimky, které přinesly poslední úpravy systému vzhledem k budování ochrany dat) nebo průchod kterýmkoliv adresářem. Jak už jsme uvedli, systémový strom adresářů začíná pevně danou strukturou adresářů. Její další rozšiřování provádějí buď uživatelé zvětšováním svých podstromů nebo správce systému. Ten také zvětšuje možnou kapacitu datové oblasti, protože může např. k adresáři `/u` připojit další, dříve inicializovaný disk. Uživatel se na nový disk neodkazuje identifikací disku (jako např. v RSX-11M DK:1 nebo v MS-DOS D:), ale pracuje s ním tak, že ho rozšiřuje vytvářením podadresářů a ukládáním dat do části stromu začínající adresářem `/u`. Obyčejný uživatel obvykle topologii spojení disků a jejich částí ve strom adresářů ani nezná, přestože je mu aktuální stav dostupný příkazem

```
$ /etc/mount
```

### 3.3 HIERARCHIE SYSTÉMOVÝCH ADRESÁŘŮ

Při studiu a používání operačního systému UNIX je velmi důležité dobře se orientovat v systémovém stromu souborů a adresářů. Základní struktura adresářů, která vychází z adresáře `/`, je standardní (viz obr. 3.2). Běh operačního systému je od ní neoddelitelný, proto ji označujeme jako systémovou. Které adresáře obsahuje, popíšeme v tomto článku. Jednotlivé verze systému mohou mít některá další rozšíření, která souvisejí s provozem systému, nikdy se ale nemůže stát, aby výrobce zásadně změnil následující jména a strukturu. Riskoval by tím nepřenositelnost mnoha programů, které s ní pracují.

Obr. 3.2 Obsah adresáře `root`

V průběhu uživatelského sezení příkazový interpret shell realizuje vlastní příkazy (tzv. vnitřní příkazy) a příkazy, které zajišťuje spuštěním programu podle jména příkazu (říkáme jim vnější příkazy). Programy pro realizaci všech hlavních vnějších příkazů jsou uloženy v adresáři `/bin`. Jména souborů zde odpovídají jménům příkazů. Při výpisu obsahu adresáře

```
$ ls -l /bin
```

nalezneme v seznamu např. soubory se jmény `/bin/ls`, `/bin/cp`, `/bin/rm` atd. Příkazy `ls(1)`, `cp(1)`, `rm(1)` atd. jsou vnější příkazy uživatelského sezení. Příkladem vnitřního příkazu uživatele je `cd`, jehož provedení realizuje shell, a proto soubor odpovídajícího jména v `/bin` nenalezneme.

Každá periférie výpočetního systému je uživatelským dostupná pouze prostřednictvím jádra operačního systému. Součástí jádra je knihovna vstupně/výstupních modulů, v rámci níž pro každou periférii nalezneme odpovídající sadu funkcí, která je ovladačem (driver) periférie. Práce s touto sadou funkcí a tím i periférií je možná pouze pomocí speciálního souboru (viz čl. 2.3), se kterým pak pracujeme pomocí volání jádra (viz kap. 5). Všechny speciální soubory jsou standardně umístěny v adresáři `/dev`. Přestože obyčejný uživatel tiskne na tiskárnu pomocí mechanismu spooling příkazem `lp(1)`, privilegovaný uživatel může na tiskárnu vypsat text mimo frontu požadavků, a sice zápisem do speciálního souboru tiskárny `/dev/lp`:

```
# cat ztext > /dev/lp
```

(volání jádra pro zápis zde realizuje shell ve spolupráci s vnějším příkazem `cat(1)`). Přímo se speciálními soubory obvykle pracuje pouze privilegovaný uživatel nebo některý z démonů. Obecně pro kohokoli je ale dostupný speciální soubor `/dev/null` (prázdná periférie). Je využíván buď jako místo, kam lze přesměrovat nežádoucí výpis, anebo jako zdroj pro vznik souboru s nulovou délkou (prázdný soubor). Např.:

```
$ cat /dev/null > prazdny
$ cp /dev/null takyprazdny
$ ls -l prazdny takyprazdny
-rw-r--r-- 1 jan group 0 Dec 30 15:18 prazdny
-rw-r--r-- 1 jan group 0 Dec 30 15:19 takyprazdny
```

### 3.3 Hierarchie adresářů

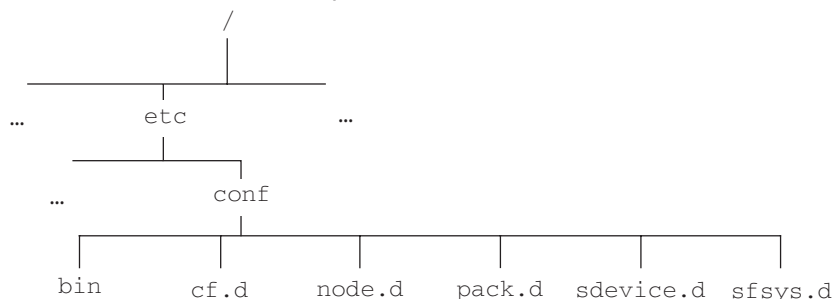
vytváří dva prázdné soubory `prazdny` a `takyprazdny` a

```
$ rm -r /tmp/muj 2> /dev/null
```

potlačí výpis standardního chybového výstupu příkazu `rm(1)` na obrazovku (kanál č. 2 je přepnut na prázdnou periferii, jádro data přijímá, ale nikam dál nepředává – data mizí v odpadkovém koši).

Typická jména speciálních souborů a jejich vazba na operační systém byla popsána v čl. 2.3.

Pro bezchybný chod operačního systému je význačný obsah adresáře `/etc`. Jsou v něm uloženy důležité tabulky, jako např. dynamicky se měnící tabulka právě přihlášených uživatelů `utmp`, právě připojených svazků `mnttab`, tabulky registrace uživatelů pro vstup do systému `passwd`, `group`, textový soubor, jehož obsah se každému uživateli zobrazí na obrazovce terminálu po přihlášení `motd` atd. Rovněž jsou zde uloženy programy pro příkazy, které jsou navíc dostupné privilegovanému uživateli, jako je např. `mkfs` pro vytváření struktury svazku systému souborů na magnetickém disku, `fsck` pro kontrolu svazku, `mount` pro jeho připojení atd. Je zde také uložen program procesu č. 1 `init`, strojově závislé programy používané v zaváděcím bloku svazků pro různé typy disků atd. Studium obsahu tohoto adresáře patří k základním činnostem správce systému. Podadresář `/etc/conf` je dnes již standardně výchozím adresářem podstromu pro generaci jádra. Struktura jeho podadresářů není prozatím přesně stanovena, má ale základní tvar podle obr. 3.3.



Obr. 3.3 Podstrom generace jádra

Adresář `/etc/conf/bin` obsahuje programy, které podporují vytvoření nového jádra. Jádro je vytvořeno (generováno) v adresáři `/etc/conf/cf.d`, kde jsou uloženy i důležité hlavičkové soubory (`.h`) pro konfiguraci jádra nebo další, obsahující struktury periférií jádra. Standardní moduly jádra a moduly vstupně/výstupních rozhraní v binárním tvaru jsou obsahem adresáře `/etc/conf/pack.d`, částem jádra a konkrétním perifériím jsou věnovány zde vytvořené další podadresáře. Do těchto míst bývá také kopírován ovladač nově připojované periferie. Adresář `/etc/conf/sdevice.d` obsahuje popis připojení periférií a úzce souvisí se souborem `/etc/conf/cf.d/sdevice`.

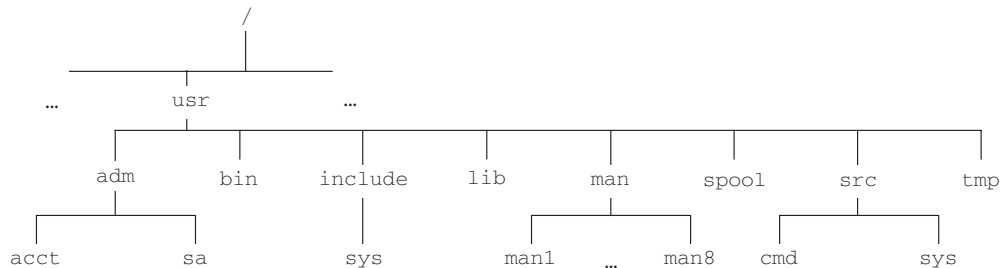
Poměrně nedávno standardizovaný adresář `/etc/default` je využíván pro implicitní nastavení práce některých nástrojů programátora (např. `/etc/default/tar`) nebo pro práci démonů (`/etc/default/lpd`) atd.

Hlavní knihovny funkcí jazyka C a volání jádra jsou umístěny v adresáři `/lib`. Charakteristické je označení souborů s knihovnami písmeny S, M, L, H ve jménu souboru, které rozlišují použití různých paměťových modelů sestavovaného programu (Small, Middle, Large, Huge – viz kap. 6).

Adresář `/mnt` je pomocným adresářem, který je používán pro připojování svazků na omezenou dobu. Je využíván především správcem systému. Adresáře `/u`, `/x`, `/y`, `/z` jsou adresáře, které správce systému používá pro pokračování stromu adresářů na dalších svazcích. Konvenci `/u` zavedl SCO XENIX, `/x`, `/y`, `/z` UNIX SYSTEM V, není ale nutné ji přísně dodržovat. Některé systémy mají k dispozici adresář `/users`, kterým začíná uživatelská oblast dat. Vytvoření adresáře libovolného jména pro připojení disku s daty aplikace nebo uživatelů záleží obvykle na libovůli správce systému.

`/tmp` je veřejně dostupná oblast pro přechodnou úschovu dat. Je využívána aplikacemi nebo nástroji programátora např. k úschově výstupních dat, která jsou dále vstupní pro následně spuštěný proces tam, kde není možné provozovat oba procesy současně propojením rourou nebo jinou přímou pamětí. Je např. využívána překladači jazyků pro úschovu výsledků jednotlivých průchodů. Oblast začínající adresářem `/tmp` je i obyčejnému uživateli dostupná pro zápis i čtení (tzn. i vytváření podadresářů). Oblast je ale obvykle úplně vyprazdňována rutinami při spouštění víceuživatelského režimu, nelze proto data uschovávat v době mimo práci aplikace.

Systémový strom adresářů pokračuje adresářem `/usr`. Jeho vznik byl dán myšlenkou vytvořit základní systémový svazek (s dosud popsanými adresáři) odděleně od zbývajících dat operačního systému (začínajících adresářem `/usr`). Svazkem, který byl připojován na adresář `/usr`, totiž zpočátku začínala uživatelská oblast (zde byly umístovány domovské adresáře všech uživatelů). S rostoucím objemem dat zvláště vývojového prostředí se stala tato oblast ještě před rokem 1979 místem pro umístování nových komponent operačního systému. Časem s rostoucí kapacitou diskových médií se část systému souborů začínající adresářem `/usr` přestala instalovat na oddělený svazek a začala být součástí systémového stromu adresářů na výchozím (kořenovém) svazku. Z těchto historických důvodů jsou některé adresáře systémového stromu zdánlivě duplicitní. Systémový strom adresářů pokračuje adresářem `/usr` touto strukturou podle obr. 3.4.



Obr. 3.4 Oblast zbylé části systémového stromu adresářů `/usr`

UNIX od doby svého vzniku podporuje evidenci spotřeby času stráveného uživatelem v systému. Při zapnutém mechanismu této evidence používá systém adresář `/usr/adm/acct/`,

### 3.3 Hierarchie adresářů

zejména pro archivaci takto získaných dat. Adresář `/usr/adm/sa` je adresářem úschovy dat sledování průchodnosti systému v určitých časových periodách. V souboru `/usr/adm/pub/ascii` je textově uložena tabulka ASCII. Adresářem `/usr/adm` také začíná podstrom modulů pro jazykové kontroly, např. v adresáři `/usr/adm/spell` jsou uloženy soubory pro slovní kontrolu dokumentace prostředkem `spell(1)`.

Adresář `/usr/bin` obsahuje další programy pro externí příkazy v rámci uživatelského sezení, `/usr/lib` je obdobou adresáře `/lib`. Jsou zde ukládány všechny další knihovní moduly jazyků nebo jiných aplikací.

`/usr/include` slouží pro evidenci standardních vsouvajících souborů pro jazyk C. Je to adresář pro textový makroprocesor jazyka C, na základě např. textu ve zdrojovém kódu

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

je uvažován jako adresář, kde bude obsažen soubor se jménem `stdio.h`. Podadresář `/usr/include/sys` obsahuje systémově závislé definice ve vsouvajících souborech (např. `<sys/files.h>`).

Osmdělů dokumentace `man` je uloženo v adresářích `/usr/man/man1`, `/usr/man/man2` atd. V jednotlivých souborech jsou zde standardně uloženy jednotlivé reference dané příručky ve formátu pro prostředek přípravy dokumentů `nroff(1)`. Mnohé verze operačního systému mají vytvořeny také adresář `/usr/catman`, kde bývají ukládány již jednou formátované soubory referenční příručky, takže při opětovném požadavku na zobrazení je již dříve zpracovaný textový soubor pouze zobrazován programem `pg(1)` na obrazovku terminálu. Šetří tím strojový čas (protože `nroff(1)` je časově náročný), ale nikoliv místo na disku. Proto program pro správu databáze dokumentace `man` po uplynutí určitého času (obvykle 1 měsíc) formátované a nepoužívané reference vypouští.

Adresář `/usr/spool` podporuje v činnosti funkci formátování požadavků způsobem FIFO např. na tiskárnu, do počítačové sítě atd. `/usr/tmp` je obdobou adresáře `/tmp`.

Adresářem `/usr/src`, který ve většině komerčních instalací není přítomen, začínala oblast zdrojových textů systému, a to jednak jádra (`/usr/src/sys`) a jednak všech nástrojů programátora (`/usr/src/cmd`). V oblasti začínající adresářem `/usr/src/sys` bylo také dříve generováno nové jádro. Na úrovni pouze binárního kódu jej dnes nahradil adresář `/etc/conf` s podobnou strukturou.

Na obr. 3.2 jsou v adresáři `/` také uvedena jména obyčejných souborů `/boot` a `/unix`. `/unix` je binární obraz jádra, je zaveden při zapnutí do paměti počítače pomocí rovněž samostatného programu `/boot`. Soubor se jménem `/unix.old` slouží pro možné zavedení předchozí verze jádra po případně neúspěšné generaci.



## 4 Bourne shell

Shell je prostředek komunikace uživatele s operačním systémem. Historicky prvním a dodnes nejdůležitějším je Bourne shell, pojmenovaný po svém autorovi A. S. Bourneovi, který jej navrhl a naprogramoval na přelomu 60. a 70. let. Principy Bourne shell přejala všechny následující příkazové interprety, obvykle jde pouze o jiný uživatelský přístup.

UNIX je interaktivní operační systém, podporuje zejména sezení uživatele u terminálu. Je to od chodu systému neoddělitelný víceuživatelský stav systému, kdy všechny výpočetnímu systému známé terminály jsou dostupné pro komunikaci s uživatelem. Při vstupu systému do víceuživatelského režimu podle tabulky připojených terminálů je pro každý terminál procesem **init** (proces s PID=1) startován proces nazývaný **getty**, který na obrazovku terminálu vypisuje oznámení o připojení k systému, výzvu k přihlášení se. Je to řetězec `login:`, kterému obvykle předchází jméno konkrétní instalace UNIXu pro odlišení v rámci sítě. Např.:

```
sysV.001!login:
```

kde `sysV.001` je označení uzlu v síti.

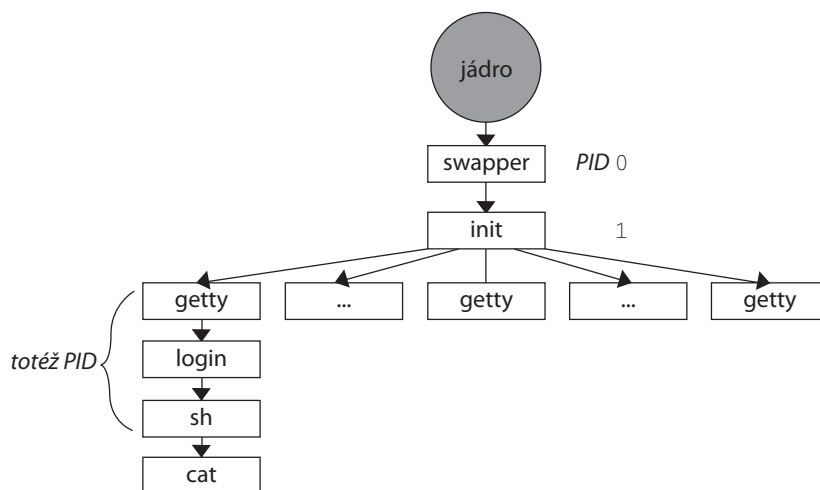
Úvodní text je na obrazovku terminálu vypisován procesem **getty**, který také dále čeká na vstup z klávesnice. Co očekává, je jméno uživatele, který do systému vstupuje (přihlašuje se). **getty** však dále vstupní řetězec od uživatele pouze přijme a předává procesu **login** (někdy nazývaný **logger**), který jej analyzuje a vždy požaduje odpovídající heslo uživatele. Vždy, to znamená i v případě, že uživatel vstup do systému chráněný heslem nemá. Tehdy stačí na výzvu

```
password:
```

pouze stisknout klávesu Enter. V opačném případě uživatel vypisuje svoje heslo, které se na obrazovce neopisuje.

Pokud uživatel napsal správně svoje jméno i heslo, operační systém jej registruje jako přihlášeného, proces **login** startuje komunikaci pomocí příkazového interpretu Bourne shell.

Proces **init** jako otec všech procesů v systému vytváří procesy **getty**, jejichž počet odpovídá logicky existujícím periferiím, přes které uživatel vstupuje do systému; každý **getty** přepisuje sám sebe (mechanismem `exec(2)`) na proces **login**, a ten následovně opět pomocí `exec(2)` na proces **sh** (shell). Ukončí-li uživatel komunikaci (klávesou Ctrl-d), končí činnost procesu **sh**, proces **init** tuto skutečnost registruje a vytváří nový proces **getty** (pomocí `fork(2)` a pak teprve `exec(2)`), který vypíše na obrazovku řetězec končící na `login:`. Schéma přihlašování a odhlašování ukazuje obr. 4.1.



Obr. 4.1 Vstup uživatele do systému

Výsledným procesem v úvodní sekvenci je **sh**, shell, příkazový interpret. Může jím být i jiný libovolný program např. sběru dat. Jaký program je uživateli po přihlášení pro komunikaci spuštěn, je jednoznačně spojeno se jménem uživatele. V daném okamžiku je v tabulce evidovaných uživatelů oprávněných přihlásit se do systému v souboru `/etc/passwd` s jedním jménem spojen pouze jeden příkazový interpret. Záměnou jména programu v této tabulce lze dosáhnout jiné komunikace uživatele se systémem.

V tabulce `/etc/passwd` nalezneme standardně také uživatele se jménem `uucp`, který má na místě příkazového interpretu program `uucico`. Uživatel `uucp` je využíván při propojení dvou systémů typu UNIX sériovou linkou (na úrovni přenosu souborů) a program `uucico` je program zajišťující přenosový protokol dat. `uucico` využívá podpory ovladače sériové linky. Programové rozhraní sériové linky je také využíváno při propojení systému se specializovanou periferií, která dodržuje přenos dat na této úrovni. Podrobněji se terminálovým rozhraním budeme zabývat v kap. 7, podrobný popis sériového rozhraní nalezne uživatel v dokumentaci `termio(5)`.

Program příkazového interpretu Bourne shell je uložen v souboru `/bin/sh`. Bourne shell bývá také označován jako **sh**, protože příkazem `sh(1)` může být uživatelsky spuštěn. Bourne shell je schopen plnit několik funkcí. Může pracovat v interakci s uživatelem jako interpret jeho příkazů pro komunikaci se systémem, může být používán obecně jako komunikační program a může se používat také jako programovací jazyk. V následujícím textu si uvedeme použití **sh** zejména jako interpretu příkazů, v druhé části kapitoly jeho programovací schopnosti (čl. 4.10 a dále).

## 4.1 PŘÍKAZOVÝ ŘÁDEK

Po přihlášení se **sh** na terminálu ohlásí výpisem znaku `$` pro obyčejného uživatele, znakem `#` pro privilegovaného. Znak `$` nebo `#` je znakem výzvy pro zadání příkazového řádku.

Příkazový řádek má formát

```
arg0 arg1 arg2 ...
```

kde `arg0`, `arg1` atd. jsou textové řetězce oddělené navzájem mezerou nebo tabulátorem.

Řetězec `arg0` je vždy jméno příkazu. Shell zahrnuje skupinu vnitřních příkazů. Jsou to akce, kterými uživatel požaduje určitou změnu v chování procesu **sh** (např. změna pracovního adresáře pomocí `cd`). Neodpovídá-li jméno žádnému z vnitřních příkazů, shell považuje příkaz za vnější a hledá v adresářích `/bin` a `/usr/bin` soubor se jménem shodným se jménem příkazu (např. `ls(1)`). Pokud jej nalezne, vytvoří synovský proces, který bude řízený programem souboru. Nenajde-li soubor jména příkazu ani zde, hledá jej konečně v pracovním adresáři; říkáme, že jde o příkaz vnější lokální, a shell s ním pracuje jako s vnějším (např. `a.out`).

Další argumenty příkazového řádku mohou mít různý význam daný funkcí příkazu. Pokud jsou volby využívány, je jako první znak řetězce `arg1` uveden znak `-`. Další textové řetězce od `arg2` jsou obvykle jména souborů nebo textové řetězce, se kterými spuštěný program pracuje. Chybí-li volby, je za `arg2` považován `arg1` atd., počet argumentů (textových řetězců) je proměnlivý. Zbytek příkazového řádku počínaje `arg1` budeme označovat jako parametry příkazu `arg0`. Např.

```
$ who
```

je příkaz s použitím pouze `arg0`. Příkaz `who(1)` nemá použity žádné volby ani jiné parametry, jeho funkcí je zde zobrazit na terminálu statistiku všech přihlášených uživatelů.

Příkaz

```
$ ls
```

opíše na obrazovku seznam jmen souborů pracovního adresáře. V uvedeném zápisu nemá parametry.

```
$ ls -l
```

má použitou volbu `-l`, která znamená, že seznam jmen souborů má být obohacen o výpis hlavních atributů souborů (`long`).

```
$ ls -l soubor
```

využívá dalšího parametru, příkaz vypíše z pracovního adresáře pouze atributy souboru zadaného parametrem příkazu `soubor`, pokud soubor tohoto jména existuje (je-li soubor daného jména adresářem, příkaz vypíše jeho obsah).

Obvykle pokračuje příkazový řádek dalšími jmény souborů, jichž může být proměnný počet, a akce stanovená příkazem je provedena s každým z nich. Lze psát

## 4.2 Bourne shell

```
$ ls -l soubor1 soubor2 soubor3
```

kdy budou vypsaný atributy souborů uvedených v parametrech `soubor1`, `soubor2`, `soubor3`.

Příkazový řádek může také obsahovat znaky, které mají speciální význam pro práci příkazového interpretu a které mění formát příkazového řádku. Nazýváme je význačné znaky (nebo speciální znaky, příp. řídící) a v dalším textu uvedeme jejich seznam.

První skupina význačných znaků souvisí s principem přepínání standardního vstupu a výstupu.

### 4.2 STANDARDNÍ VSTUP A VÝSTUP

Většina programů, která kdy v UNIXu byla a bude napsána, má interaktivní charakter. Rozumíme tím, že program čte vstup z klávesnice uživatele terminálu a výsledky své činnosti vypisuje na jeho obrazovku.

Každý proces v UNIXu má k dispozici parametrem jádra stanovený počet komunikačních kanálů (50–60) pro manipulaci se soubory. Otevření souboru znamená alokaci kanálu, obsazení další položky v tabulce. Nově vytvořený proces přebírá (dědí) po svém otci mimo jiné také tuto tabulku otevřených kanálů. Tabulka kanálů obsahuje 3 záznamy i v případě, že otec žádný ze souborů explicitně neotevřel. Jsou to obsazená místa kanálů č. 0, 1 a 2.

Kanál č. 0 je označován jako standardní vstup a je otevřen pro čtení z klávesnice terminálu, kanál č. 1 je standardní výstup, je otevřen pro zápis na obrazovku terminálu a stejně tak je pro zápis na obrazovku terminálu otevřen kanál č. 2, označovaný jako standardní chybový výstup.

Takže např. program `ls(1)` využívá standardní výstup (kanál č. 1) pro výpis atributů souboru. Příkaz `rm(1)` (rušení souboru) v příkazovém řádku

```
$ rm -r text
text: ? _
```

žádá potvrzení nalezeného souboru `text` ke zrušení ze standardního vstupu (kanál č. 0), když předtím na standardní výstup vypíše jméno rušeného souboru.

V rámci shellu je možné manipulovat se standardním vstupem i výstupem pomocí speciálních znaků. Prvnímu způsobu manipulace říkáme přesměrování standardního vstupu a výstupu a druhému roura (pipe).

Příkaz

```
$ ls -l > seznam
```

nevypisuje atributy souborů pracovního adresáře na obrazovku terminálu, výpis je ukládán do souboru se jménem `seznam`. Je toho dosaženo zápisem speciálního znaku `>`, který shell rozpozná a interpretuje jako přepnutí standardního výstupu procesu `ls` do souboru se jménem `seznam`. Soubor `seznam` je nově vytvořen, pokud již existoval, jeho předchozí obsah je ztracen. Standardní výstup lze přesměrovat i způsobem, kdy původní obsah souboru

bude zachován, přepnutý obsah standardního výstupu k němu bude pouze připojen. Toho dosáhneme zápisem `>>` :

```
$ ls -l >> seznam
```

Znakem `<` přepínáme standardní vstup.

```
$ rm -i < soubor < potvrzeni
```

čte program `rm(1)` ze souboru se jménem `potvrzeni` namísto z klávesnice terminálu. Ukončí-li `rm(1)` svoji činnost před vyčerpáním obsahu souboru `potvrzeni`, soubor je korektně uzavřen.

Ostatní kanály zmíněné tabulky otevřených souborů v Bourne shell lze přepínat uvedením čísla kanálu před znakem přepnutí. Prostředek `time(1)` je program, který měří čas spotřebovaný procesem za jeho existence v systému. Např.

```
$ time ls
```

provede řádně výpis adresáře na obrazovku, ale po ukončení procesu `ls(1)` bude ještě následovat informace o času procesu v systému. Protože je tato informace vypisována do kanálu č. 2, zápis

```
$ time ls > seznam 2> cas
```

způsobí výpis programu `ls(1)` do souboru `seznam` a informace od procesu `time(1)` do souboru `cas`. Jiným příkladem je přepnutí diagnostiky překladače jazyka C:

```
$ cc prog.c 2>chyby
```

Chybové zprávy překladače zdrojového textu v jazyce C uložené v souboru `prog.c` jsou přepnuty do souboru se jménem `chyby`. Lze také obecně spojovat tok dat určitého kanálu s kanálem jiným. Např. standardní chybový výstup lze napojit na kanál č. 1 zápisem

```
... 2>&1
```

takže například

```
$ time ls > mereni 2>&1
```

spojí kanál č. 2 a č. 1 a data obou kanálů ukládá do souboru `mereni`.

Obecně lze psát

```
m< &n nebo m> &n
```

kde *m* a *n* jsou čísla kanálů.

### 4.3 Bourne shell

Z uvedeného je patrné, jak lze pomocí přesměrování z interaktivního programu získat program upravující daným způsobem data, program, který data filtruje – v dalším označován jako filtr.

Program `cb(1)` (`C-beautifier`) umí zdrojový text v jazyce C zkrášlit tak, že tam, kde je to zapotřebí, vsouvá na začátek řádku tabulátory, aby byly řídicí struktury zřetelně čitelné. Program `cb(1)` pracuje nad standardním vstupem a výstupem, lze např. psát

```
$ cb < prog.c > progcb.c
```

kde `cb(1)` filtruje data ze souboru `prog.c` do souboru `progcb.c`.

#### 4.3 ROURA

Roura je prostředek pro spojení dat dvou procesů. Proces `PRODUCENT` zapisuje data na standardní výstup a proces `KONZUMENT` data čte ze standardního vstupu. Znakem `|` vytváří shell rouru:

```
PRODUCENT | KONZUMENT
```

Stejného efektu dosáhneme přesměrováním vstupu a výstupu na pomocný soubor. Sekvence příkazů

```
$ PRODUCENT > tmp
$ KONZUMENT < tmp
$ rm tmp
```

má týž efekt, ale realizace je zcela jiná. Roura je totiž vytvořena v operační paměti, kdy procesy `PRODUCENT` i `KONZUMENT` běží současně. Roura má omezenou velikost; je-li procesem `PRODUCENT` zaplněna, proces je jádrem pozastaven do chvíle, než proces `KONZUMENT` odečte data z roury. Naopak, je-li roura prázdná, proces `KONZUMENT` je pozastaven a čeká se, kdy proces `PRODUCENT` opět naplní rouru daty. Data jsou z roury čtena v pořadí, v jakém byla do roury zapisována (FIFO).

Tímto mechanismem, který jádro realizuje na požádání shellu, lze propojovat sekvenčně současně několik procesů. Příkazový řádek pak nazýváme kolona.

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

Tato kolona na standardní výstup vypisuje počet souborů v daném adresáři, program `wc(1)` s volbou `-l` počítá řádky standardního vstupu. Program `grep(1)` umí vyhledat v textu řetězec znaků daný prvním parametrem. Kolona

```
$ who | grep jan | wc -l
```

vypíše na standardní výstup, kolikrát je uživatel `jan` přihlášen.

Užitečný je program `tee(1)`. Jeho název je odvozen z vodoinstalační terminologie. Jde o odbočku z roury. Odbočka svede standardní vstup do souboru daného parametrem

a současně jej předává na standardní výstup. Příkazový řádek

```
$ who | tee kdo | grep jan | wc -l
```

má tentýž význam jako předchozí kolona, navíc ale bude do souboru `kdo` uložen výpis programu `who(1)`, což je seznam přihlášených uživatelů. Posledním příkladem

```
$ cc prog.c 2>&1 | more
```

je prohlížení standardního chybového výstupu překladače jazyka C.

#### 4.4 EXPANZNÍ ZNAKY JMEN SOUBORŮ

Je to druhá skupina speciálních znaků, vztahuje se ke jménům souborů a umožňuje vymezit podmnožinu souborů. Napíšeme-li na klávesnici příkaz

```
$ ls -l *.c
```

znak `*` je uvažován jako expanzní, shell prohledává pracovní adresář a jména všech souborů, která končí řetězcem `.c`, nahradí v příkazovém řádku za text `*.c`. Znak `*` znamená libovolný řetězec znaků jakékoliv (i nulové délky), např.

```
$ ls
ab
com.sh
muj.h
muj.c
prog.c
$ ls *.c
muj.c
prog.c
$
```

Přitom program `ls(1)` jména souborů zjistí a opíše na standardní výstup. Expanze jmen souborů provede shell (nakonec spouští `ls muj.c prog.c`). Lze to ověřit např. nejjednodušším výpisem obsahu adresáře

```
$ echo *
```

kde `echo(1)` je program opisující na standardní výstup zbytek příkazového řádku.

Chceme-li náhradu ve jménu souboru pouze v rámci jednoho znaku, používáme expanzní znak `?`. Příkaz

```
$ ls a?
```

## 4.5 Bourne shell

vypisuje jména všech souborů, která jsou dvouznaková a která končí znakem `a`.

Konečně můžeme použít expanzní znaky `[ a ]`. Používají se ve dvojici, přitom první zahajuje a druhý uzavírá výčet znaků, které mohou být vybrány. Např.

```
$ ls [abcd] z
```

vypíše jména všech souborů pracovního adresáře, která končí znakem `z`, jsou dvouznaková a začínají jedním ze znaků `a`, `b`, `c` nebo `d`.

Pro snadnější zápis můžeme použít znak `-` jako určení rozsahu.

```
$ ls [a-d] z
```

má tentýž význam. Shell expanduje znaky určené pomocí `-` podle pořadí v tabulce ASCII, viz. Příloha A.

### 4.5 DALŠÍ ZNAKY ZVLÁŠTNÍHO VÝZNAMU

Sekvenci příkazů lze psát v rámci jednoho vstupního řádku. Znakem `;` oddělujeme příkazy:

```
$ pwd; ls -l
```

Nejprve je na obrazovce vypsána cesta k pracovnímu adresáři a po ukončení práce programem `pwd` (`1`) jsou vypsány atributy souborů pracovního adresáře programem `ls` (`1`).

Sekvence příkazů můžeme sdružovat do skupin. Závorky `{ a }` vymezují skupinu příkazů, je na ní ale implicitně startován nový proces `sh`.

```
$ { cat /usr/jan/seznam; ls;} | grep *.c
```

vypisuje na obrazovku terminálu řádky souboru `/usr/jan/seznam` a ta jména souborů pracovního adresáře, která obsahují řetězce končící na `.c` (jména souborů obsahující zdrojové texty v jazyce C).

Prozatím jsme uváděli na příkazovém řádku pouze jména vnějších příkazů. Tento program byl pomocí shellu nalezen v odpovídajícím adresáři ve stejnojmenném souboru a byl spuštěn proces, který jej realizoval. Příkazy, které můžeme zadávat na klávesnici terminálu, mohou být také vnitřní příkazy, mezi něž patří např. `cd` (`change directory`), který mění pracovní adresář. V případě `cd` to znamená, že nastavení pracovního adresáře se může u nově spuštěných procesů `sh` změnit, jejich otec si ale své nastavení ponechá. Např.

```
$ cd bin      ;; zmeni pracovni adresar na /bin
$ sh          ;; spousti novy Bourne shell jako syna
$ cd /usr/jan ;; nastavuje novy pracovni adresar
$ ls          ;; vypis adresare /usr/jan
```



```
...
$ ^d$ pwd      ;; ukončení práce nové spuštěného shellu
/bin
$ : na předchozím příkazovém řádku jste vypis jména prac. adresare
$
```

Uvedenou sekvenci lze zapsat pomocí závorek ( a ) :

```
$ cd /bin; (cd /usr/jan; ls); pwd
```

V příkladu bez kulatých závorek jsme použili další speciální znak, znak komentáře. Je jím : , který musí být uveden na místě příkazu, a zbytek příkazového řádku musí od něj být oddělen mezerou ( : je vnitřní příkaz). Stejnou funkci má speciální znak # , který lze ale použít pouze pro komentáře v příkazových souborech (viz čl. 4.9).

#### 4.6 POPŘEDÍ A POZADÍ, PROCESY

Všechny příkazy, které jsme prozatím v této kapitole uváděli, byly spouštěny na popředí (foreground). Znamená to, že příkaz byl interpretován a uživatel očekával výpis výzvy \$ , kterou mu bylo sděleno, že interpretace byla ukončena. Bez toho, abychom čekali na ukončení příkazu a bylo nám umožněno zadávat další příkazy, je možné spustit příkaz na pozadí (background). Dosáhne se toho speciálním znakem & , který bude uveden jako poslední na příkazovém řádku.

```
$ ls -l > seznam &
231
$
```

Na pokyn \$ můžeme ihned zadávat další příkaz. Shell vypsal na standardní výstup identifikační číslo procesu (PID) spuštěného na pozadí, zde 231. Slouží uživateli k manipulaci s procesem programu na pozadí. Pokud by se např. proces spuštěný na pozadí zacyklil, může mu uživatel zaslat signál, na který proces reaguje ukončením činnosti. PID má každý proces v době svého vzniku přiděleno, u procesů běžících na popředí se implicitně nezobrazuje. PID všech procesů spuštěných v rámci sezení u terminálu lze získat příkazem ps ( 1 ) :

```
$ ps
PID      TTY      TIME    COMMAND
31        01        0:03      sh
44        01        0:00      ls
45        01        0:00      ps
$
```

Vypsaná tabulka uvádí pro každý proces postupně identifikační číslo procesu (PID), označení terminálu, odkud byl proces spuštěn (TTY), celkový čas existence procesu v systému v hodinách a minutách (TIME) a příkaz, kterým byl proces vytvořen (COMMAND).

## 4.6 Bourne shell

Uživatel může procesu, jehož PID zná, zaslat signál vnějším příkazem `kill(1)`. Jak mnemonický název napovídá, ve většině zasílaných signálů přijímající proces ukončí svoji činnost. Pro různé implementace bývá definován různý počet signálů (SVID3 dnes už definuje 28 signálů), jejich seznam je uveden v čl. 5.2, kde je také podrobně diskutován princip zasílání, přijímání a maskování signálů. Proces může signál zachytit a zpracovat (maskovat), ignorovat, anebo nechat implicitně k vyřízení jádru (to je obvykle ukončení procesu). Signál, který nelze ignorovat ani maskovat, je signál č. 9.

```
$ kill -9 44
```

je pokyn k ukončení činnosti procesu s `PID=44`, který byl aktivován uživatelem, a to za jakékoliv situace. Zápis

```
$ kill 44
```

znamená zaslání signálu č. 15. Signál č. 2 je ekvivalentní přerušení procesu běžícího na popředí z klávesnice (obvykle klávesa Del nebo Ctrl-c).

Uživatel může snadno zjistit PID všech procesů běžícího systému vnějším příkazem

```
$ ps -e
```

Z takto získané tabulky může zjistit čísla procesů a případně se pokusit jim zasílat signály. Samozřejmě si uživatelé navzájem nemohou procesy rušit. To je umožněno pouze superuživateli.

Samotný proces **sh** dovoluje uživateli nastavovat reakci na příchozí signál vnitřním příkazem `trap`.

```
$ trap 'date' 2 15
```

maskuje přijetí signálu 2 a 15. Po příchodu některého z nich je interpretována sekvence příkazů uvedená v apostrofech (zde jen příkaz `date`, výpis data a času) a dále shell pokračuje od místa, kde byl přerušen příchodem signálu. Ignorovat signál č. 3 lze

```
$ trap '' 3
```

Příkaz

```
$ trap
```

vypíše nastavení maskování signálů a příkazem

```
$ trap 2 3 15
```

obnovujeme původní nastavení reakce na příchod signálu.

Na dokončení procesu běžícího na pozadí můžeme počkat vnitřním příkazem `wait`.

```
$ wait 44
```

čeká shell na dokončení procesu s `PID=44`. Teprve pak vypíše na obrazovku `$` a čeká na další vstup. Příkazem

```
$ wait
```

čeká shell na všechny procesy spuštěné na pozadí. Není-li žádný takový, `$` je vypsán okamžitě.

Uživatel tedy může procesy spouštět na pozadí nebo na popředí, může synchronizovaně čekat na jejich dokončení, případně je předčasně ukončit zasláním signálu. Nemůže ale proces běžící na pozadí přesunout na popředí nebo naopak. Tento problém řešil příkazový interpret C-shell pomocí mechanismu `jobs` (práce, viz čl. 4.14). Dnes je podle SVID3 `sh(1)` rozšířen na `jsh(1)`, který mechanismus `jobs` obsahuje také.

Příkazem `jsh(1)` je spouštěn Job Control shell. Obsahuje všechny možnosti interpretu Bourne shell, které jsou uvedeny v této kapitole. Navíc obsahuje část řízení prací, která je dostupná pouze v interaktivním režimu, eviduje příkazové řádky spuštěné uživatelem jako práce (`jobs`) a eviduje je celočíselným označením (které nijak nesouvisí s `PID` procesů). Uživatel může spustit několik procesů na pozadí (samostatně nebo v koloně) a má-li možnost zadávat příkazový řádek, vnitřní příkaz `jobs` mu vypíše seznam prací spuštěných z jeho terminálu. Součástí výpisu je celočíselná identifikace prací, kterou dále využívá v příkazech `bg` (background) a `fg` (foreground) pro převod práce na pozadí nebo popředí. Číslo práce při použití v příkazovém řádku musí předcházet řídicí znak `%`, např.

```
$ fg %2
```

převádí práci číslo 2 na pozadí. Práci na popředí můžeme pozastavit (a získat tak možnost komunikace s **jsh**) definovaným znakem klávesnice (bývá to `Ctrl-z`, nastavuje se a mění pomocí `stty(1)`, viz kap. 7), který je systémem zpracován jako zaslání signálu `SIGTSTP` (viz čl. 5.2) procesu **jsh**. V `jsh(1)` můžeme používat vnitřní příkazy `wait` a `kill` v rozšířeném formátu, v parametru příkazu používáme také označení práce, např.

```
$ kill -9 %2
```

je zaslání signálu č. 9 všem procesům, které jsou součástí práce číslo 2. Ukázka řízení prací je také uvedena u poznámek k C-shell ve čl. 4.14.

## 4.7 ZNAKY VÝLUKY

Každému význačnému znaku lze vypnout jeho význam, a to znakem výluky `\`. Znak obrácené lomítko (backslash) zapsaný před znakem snižuje význam znaku na úroveň ostatních znaků, shell tedy neuvažuje jeho speciální význam. Znamená to např., že lze použít znak `*` ve jménu souboru:

## 4.7 Bourne shell

```
$ cat /dev/null > \*
```

znamená vytvoření souboru nulové délky se jménem `*` v pracovním adresáři. Pro zrušení takového souboru musíme psát

```
$ rm \*
```

a ne

```
$ rm *
```

protože ve druhém případě by shell využil speciálního významu znaku `*` a zrušil tak všechny soubory adresáře.

Znak `\` respektuje ovladač terminálu, což nám umožňuje vypínat význam např. i znaku mezery, tabulátoru, znaku nového řádku nebo znaku pro zrušení předchozího znaku (backspace). Příkazem

```
$ cp /dev/null a\ b
```

můžeme vytvořit soubor se jménem `a b`, nebo příkazem

```
$ cp /dev/null a←b
```

soubor jména `a←b` (symbol `←`=backspace), označuje zrušení předchozího znaku), který bude ale při výpisu

```
$ ls
```

čitelný jako soubor se jménem `b` (`a` je překryto znakem `b`). Pokud soubor takového jména nebyl vytvořen úmyslně, je obtížné jej z adresáře odstranit. My ale umíme podle kap. 2 zobrazovat data adresáře jako obsah binárního souboru způsobem

```
$ od -c .
```

odkud si celé jméno včetně nezobrazených znaků zjistíme a použitím znaku výluky s ním můžeme pracovat.

Vymezit řetězec znaků, v němž nemá být u žádného uvažován jeho speciální význam, můžeme ohraničením mezi apostrofy

```
$ echo '\&'
\&
```

Jako řetězec znaků můžeme také označit text ohraničený znaky uvozovky `"`. I v takto vymezeném textovém řetězci je potlačen speciální význam význačných znaků. Speciální znaky,

kteřé uvedeme ve čl. 4.8, týkající se substituce příkazů (opačný apostrof `) a proměnných (\$) jsou však interpretovány ve svém zvláštním významu.

#### 4.8 PROMĚNNÉ A PROSTŘEDÍ UŽIVATELE U TERMINÁLU

V úvodu kapitoly jsme řekli, že shell je také programovací jazyk. Jde o jazyk interpretační, který má datové struktury jediného typu – textové proměnné. Proměnnou uživatel vytváří a současně definuje její obsah zadáním

```
$ jmeno=obsah
```

kde `jmeno` je jméno proměnné a `obsah` je textový řetězec, kterým bude proměnná naplněna. Před a za operátorem = nesmějí být mezery.

```
$ database=informix
```

Proměnná se jménem `database` (je příp. vytvořena) je naplněna řetězcem `informix`. Obsah proměnné je textově substituován na místě, kde jménu proměnné předchází znak \$:

```
$ echo $database
informix
```

A může-li nastat kolize, je jméno proměnné uzavřeno mezi znaky { a } :

```
$ data=/usr/data
$ echo $database ${data}base
informix /usr/database
```

Obsah proměnné lze naplnit také standardním výstupem některého programu. V příkladu

```
$ dir=`pwd`
```

je proměnná `dir` naplněna textovým řetězcem, který za normálních okolností program `pwd` (1) vypisuje na standardní výstup. Příkaz je uzavřen mezi znaky obráceného apostrofu `.

Obsah proměnné můžeme dále naplnit vstupem z klávesnice, tedy ze standardního vstupu, příkaz

```
$ read vstup
```

očekává na následujícím řádku vstupu textový řetězec ukončený znakem nového řádku, kterým naplní obsah proměnné `vstup`. Proměnná `vstup` nemusela být dříve inicializována.

Vnitřním příkazem `set` můžeme získat seznam všech proměnných včetně výpisu jejich obsahu:

```
$ set
```

## 4.8 Bourne shell

```
HOME=/usr/jan
IFS=

MAIL=/usr/spool/mail/jan
MAILCHECK=600
PATH=/bin:/usr/bin:.
PS1=$
PS2=>
SHELL=/bin/sh
TERM=ansi
TZ=MET-1
jmeno=obsah
database=informix
data=/usr/data
dir=/usr/jan
$
```

Z příkladu vidíme námi definované proměnné `jmeno`, `database`, `data` a `dir` na konci výpisu. Předchází jim seznam proměnných (jejichž jména jsou označena velkými písmeny), které jsme jako uživatelé nedefinovali, ale které byly definovány v okamžiku startu příkazového interpretu. Jejich význam je důležitý pro práci shellu a je následující

proměnná	její obsah
HOME	domovský adresář
IFS	znak pro ukončení vstupu z klávesnice (je nastaven na znak nového řádku)
MAIL	poštovní schránka elektronické pošty uživatele (viz příkaz <code>mail(1)</code> )
MAILCHECK	časový interval, po jehož uplynutí shell hledá změnu v poštovní schránce
MAILPATH	seznam jmen souborů, dalších poštovních schránek uživatele
PATH	seznam cest k programům vnějších příkazů; standardně <code>/bin</code> , <code>/usr/bin</code> a pracovní adresář
PS1	výzva pro zápis příkazového řádku
PS2	druhá, pokračovací výzva; uživateli je nabídnuta, pokračuje-li příkaz na více řádcích; uživatel může explicitně stanovit následující řádek jako pokračovací znakem výluky <code>\</code> na místě před znakem nového řádku; <code>PS2</code> bývá nastavena na <code>&gt;</code>
SHELL	cesta k programu běžícího interpretu
TERM	typ terminálu (viz kap. 7)
TZ	časová zóna, označení a relativita vzhledem k Greenwich poledníku

Proměnné, které už dále nebudeme potřebovat, lze zrušit vnitřním příkazem `unset`:

```
$ unset jmeno database data dir
```

Běžící proces řízený programem `/bin/sh` vytváří synovské procesy pro uspokojení požadavků uživatele. Synovské procesy pracují v rámci prostředí, kde jsou prováděny (prostředí uživatele), tj. jsou jim pro čtení dostupné obsahy proměnných shellu. Implicitně ale není dostupná žádná z nich, dostupnost je nastavována vnitřním příkazem `export`, např.

```
$ database=ingres
$ export database
```

stanovujeme viditelnost obsahu proměnné `database` v synovských procesech. Příkaz `export` bez argumentů vypíše seznam všech exportovaných proměnných.

Podle konvencí práce v jazyku C víme, že příkazový řádek je programu dostupný z argumentů funkce `main()`, tj. proměnných `argc`, `argv`:

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

Seznam proměnných hlavní funkce programu můžeme ale rozšířit o proměnnou `envp` takto:

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

Přitom pole řetězců `envp` obsahuje všechny proměnné a jejich obsahy, které byly označeny příkazem `export`.

Prostředím uživatele u terminálu rozumíme všechny právě definované proměnné. Jejich seznam lze obdržet pomocí `set`. Seznam všech proměnných, které byly importovány z procesů otce, obdržíme příkazem `env(1)`.

## 4.9 PŘÍKAZOVÉ SOUBORY – SCÉNÁŘE

Sekvencí příkazů interpretu shell můžeme naplnit textový soubor:

```
$ cat > cmdsoubor
pwd
ls -l
^d$
```

Takový soubor můžeme interpretovat novým (synovským) procesem shellu:

```
$ sh cmdsoubor
```

kde `cmdsoubor` je soubor s příkazy pro shell. Přiznáme-li souboru se jménem `cmdsoubor` proveditelnost příkazem `chmod(1)`, lze příkazový soubor interpretovat bez uvedení příkazu `sh(1)`:

## 4.10 Bourne shell

```
$ chmod 775 cmdsoubor
$ cmdsoubor
```

I tento zápis znamená vytvoření synovského shellu. Chceme-li příkazový soubor interpretovat v rámci aktuálního shellu, můžeme použít vnitřní příkaz `.` (tečka):

```
$ . cmdsoubor
```

Podle určitých konvencí pro jména souborů interpretují příkazové interprety při přihlašování (nebo i odhlašování) uživatele soubory smluvených jmen. Pro Bourne shell je směrodatný obsah souboru `.profile`, který má uživatel uložen v domovském adresáři a jehož obsah interpretuje ihned po přihlášení uživatele ještě před výpisem znaku odezvy `$` (některé proměnné jsou definovány na tomto místě). V případě interpretu C-shell jsou důležité soubory `.login` (je interpretován po přihlášení se), `.logout` (před odhlášením), `.cshrc` (interpretován vždy po spuštění z příkazového řádku).

### 4.10 PROGRAMOVÁNÍ

Každý proces při ukončení své činnosti oznamuje způsob ukončení operačnímu systému návratovým kódem. Návratový kód je při psaní programu zadáván celočíselným parametrem volání jádra `exit(2)` nebo řídicí struktury `return` pro ukončení funkce `main()`:

```
main()
{
...
exit(0);
...
}
```

Je-li návratový kód programu nulový, je v konvencích UNIXu uvažován jako logická pravda, je-li nenulový, je uvažován jako logická nepravda. Tento návratový kód je možné v shellu analyzovat a měnit podle něj tok řízení. Vnitřní příkaz, řídicí struktura **if**, má následující formát

```
if cmdlist
then cmdlist2
else cmdlist3
fi
```

kde `cmdlist` označuje sekvenci příkazů. Na jejím místě za klíčovým slovem vnitřního příkazu **if** je v obvyklém případě uveden pouze jeden příkaz. Větvení je potom dáno návratovým kódem tohoto příkazu nebo návratovým kódem příkazu, který byl v uvedené sekvenci proveden jako poslední. Vyzkoušejme si příklad. Do souboru `prog.c` editujeme



```
main()
{
    exit(35);
}
```

Po překladu

```
$ cc prog.c
```

vzniká přeložený a sestavený program v souboru `a.out`. Píšeme

```
$ if a.out
> then echo ano
> else echo ne
> fi
ne
$
```

A opravíme-li parametr 35 ve funkci `exit` na 0, větvení `if` projde příkazem `echo ano`. Znaky `>` jsou označením pro pokračování příkazu na následujícím řádku (obsah proměnné `PS2`).

Pro testování stavu obsahu proměnných a souborů je v UNIXu k dispozici program `test(1)`. Jeho návratová hodnota je závislá na pravdivosti zapsaného výroku, např.

```
$ test $database = "informix"
```

vrací návratový status pravdy, pokud proměnná `database` obsahuje textový řetězec `informix`.

Formát příkazu `test(1)` je

```
test arg
```

kde `arg` může být

<code>-r file</code>	soubor se jménem <i>file</i> existuje a je dostupný pro čtení
<code>-w file</code>	totéž pro zápis
<code>-f file</code>	soubor se jménem <i>file</i> existuje a není adresářem
<code>-d file</code>	soubor se jménem <i>file</i> existuje a je adresářem
<code>-z s</code>	délka textového řetězce <i>s</i> je nulová
<code>-n s</code>	délka textového řetězce <i>s</i> je nenulová
<code>s1=s2</code>	řetězce <i>s1</i> a <i>s2</i> jsou si rovny
<code>s1!=s2</code>	řetězce <i>s1</i> a <i>s2</i> si nejsou rovny
<code>n1 op n2</code>	výhodnocení vztahu dvou čísel algebraicky; <i>op</i> může nabývat
<code>-eq</code>	rovnost
<code>-ne</code>	nerovnost

## 4.10 Bourne shell

```
-gt n1 je větší než n2
-ge n1 je větší nebo rovno n2
-lt n1 je menší než n2
-le n1 je menší nebo rovno n2
n1 a n2 jsou zapsány jako textové řetězce obsahující pouze cifry,
test ( 1 ) provádí konverzi do numerické podoby a vyhodnocuje.
```

Příklad scénáře:

```
read vstup           ;# naplnění proměnné vstup z klávesnice
if test -n "$vstup"   ;# testujeme proměnnou na nenulovou délku
then
    echo $vstup        ;# opisujeme vstupní text na standardní výstup
fi
```

Ve většině v současné době běžících instalací UNIXu je program `test ( 1 )` spojen (je vytvořen nový odkaz příkazem `ln ( 1 )`) ještě se jménem `[`. Lze tedy např. psát

```
...
if [ -n "$vstup" ]
...
```

a použít tak hranaté závorky pro vymezení podmínky. Nebo je `test` zahrnut do množiny vnitřních příkazů shellu (viz Příloha C).

Dalším vhodným příkazem, tentokrát pro algebraické vyhodnocení textových řetězců, je `expr ( 1 )`.

```
$ expr 1 + 1
2
$
```

který na standardní výstup vypisuje výsledek aritmetické operace dané argumenty příkazu. Uvědomme si opět, že jak operátor, operandy výrazu, tak i výsledek jsou textové řetězce. Inkrementaci proměnné `i` můžeme provádět pomocí

```
i=`expr $i + 1`
```

Příkaz `expr ( 1 )` uvažuje 2 operandy. Na místě operátoru může být uvedeno

+	pro součet,
-	rozdíl,
*	násobení,
/	celočíslné dělení
%	zbytek po dělení,

anebo některý z relačních operátorů `<`, `<=`, `=`, `!=`, `>=`, `>`, kdy jsou operandy porovnány numericky; pokud jsou nečíselné, pak lexikograficky podle ASCII.

Cykly můžeme programovat pomocí vnitřních příkazů `while`, `until` a `for`:

```
while cmdlist1
do
  cmdlist2
done
```

kde sekvence příkazů `cmdlist2` se provádí, pokud je výsledek `cmdlist1` pravdivý, a v případě

```
until cmdlist1
do
  cmdlist2
done
```

bude sekvence `cmdlist2` opakována, pokud bude výsledek `cmdlist1` nepravdivý.

Uvedme si rozšíření scénáře pro výpis načteného vstupu (předchozí scénář):

```
while read vstup
do
  if [ -n "$vstup" ]
  then echo $vstup
  fi
done
```

Příklad je scénářem, který přenáší standardní vstup na standardní výstup s výjimkou prázdné řádky.

Cyklus `for` má formát:

```
for n in word1 word2 ...
do
  cmdlist
done
```

Seznam příkazů `cmdlist` bude proveden pro všechny instance textových řetězců `word1`, `word2` atd. proměnné `n`. Např. cyklus

```
$ for i in *
> do
> if [ -f $i ]
> then
> echo $i
```

## 4.10 Bourne shell

```
> fi
> done
```

vypisuje vždy na nový řádek každé jméno obyčejného souboru pracovního adresáře (vynechá podadresáře).

Ve čl. 4.8 jsme uvedli seznam proměnných, které jsou definovány bez iniciativy uživatele. V době běhu **sh** existují další proměnné, jejich obsah se ale dynamicky mění podle současného stavu spuštěných procesů. Jsou to:

proměnná	její obsah
?	návratový status posledního provedeného příkazu
\$	PID procesu provádějícího tento shell
!	PID procesu jako posledního spuštěného na pozadí
#	počet pozičních parametrů scénáře
0	první řetězec příkazového řádku scénáře (jméno scénáře)
1	první argument scénáře
2	druhý argument scénáře
3	...
...	...
9	devátý argument scénáře
*	příkazový řádek scénáře vyjma jména scénáře (\$1 \$2 ... \$9)

Vyzkoušejte

```
$ echo $? ;: vypisuje navratovy status posledniho provedeneho prikazu
```

nebo

```
$ echo $! ;: PID posledniho procesu na pozadi
...
$ echo $$ ;: PID shellu
```

Proměnné se jmény 0, 1, 2, ..., \*, # souvisejí s příkazovým řádkem při spuštění nového interpretu pro scénář. Editujeme-li např. v souboru se jménem `testarg`

```
echo $*, celkem=$#
```

spuštění znamená

```
$ sh testarg a b c
a b c, celkem=3
```

Proměnná `$` je např. využívána k jednoznačné identifikaci jména pomocného souboru. Přesměrováním

```
$ cat /dev/null > /tmp/sh$$
```

vytvoříme v pracovní oblasti /tmp (kap. 3) soubor končící PID (v textové podobě) shellu, který příkaz interpretuje. Takto vzniklý soubor je pro tentýž shell dostupný opět pomocí proměnné \$ :

```
trap 'rm /tmp/sh$$; exit' 2
```

kde soubor rušíme a vnitřní příkaz `exit` končí činnost shellu.

Přerušení cyklů nebo omezení jejich činnosti lze provést vnitřními příkazy

<code>break</code>	přeruší tělo cyklu a pokračuje příkazovým řádkem za vnitřním příkazem <code>done</code>
<code>continue</code>	přeruší tělo cyklu a pokračuje následující iterací cyklu
<code>exit</code>	končí práci shellu, scénář je takto ukončen na jiném místě než na konci souboru

Např.

```
while read vstup
do
  if [ -n "$vstup" ]
  then echo $vstup
  else break
  fi
done
```

ukončí opis standardního vstupu v případě prázdného řádku na vstupu.

Dynamickou proměnnou `*` nesmíme zaměňovat za expanzní znak jmen souborů.

```
for i in *
...
```

je cyklus pro všechny soubory pracovního adresáře a

```
for i in $*
...
```

je cyklus pro jednotlivé argumenty scénáře. Uložme např. do souboru se jménem `testprogram` tento scénář:

```
for i in $*
do
  if [ -f /bin/$i ] || [ -f /usr/bin/$i ]
  then
    echo $i je takovy vnejsi prikaz
  else
    echo $i neni vnejsim prikazem
```

## 4.10 Bourne shell

```
fi
done
```

V příkazu `if` jsme zde použili operátor `||` pro stanovení disjunkce dvou příkazů (analogicky pro konjunkci `&&`). Scénář jinak hledá v adresářích `/bin` a `/usr/bin` jméno souboru podle jednotlivých argumentů scénáře:

```
$ sh testprogram rm mv abc
rm je takovy vnejsi prikaz
mv je takovy vnejsi prikaz
abc neni vnejsim prikazem
$
```

Zápis cyklu pro všechny argumenty scénáře lze zkráceně zapisovat pouze

```
for i
...
```

Poslední řídicí struktura je přepínač `case`. Má formát

```
case word in
  patt1) cmdlist ;;
  patt2) cmdlist ;;
  ...
  *) cmdlist ;;
esac
```

kde je provedena sekvence příkazů `cmdlist` podle shody textového řetězce `word` s některým textovým řetězcem `patt1`, `patt2`, ...

Každý `cmdlist` je ukončen řetězcem `;;` a shell dále pokračuje následujícím příkazem `esac`. Znak `*` označuje variantu, kdy nevyhovuje žádná z předchozích. Např.

```
case $1 in
  kopie) mkdir zaloha
    cd zaloha
    cp ../ * .
    cd ..
    ;;
  ruseni)
    echo -n "Zrusit vsechny soubory adresare `pwd` [a/n]?:"
    read vstup
    if [ "$vstup" = "a" ] || [ "$vstup" = "A" ]
    then
      rm *
    fi
fi
```

```
;;
*)
  echo Program udrzba, format: udrzba kopie \ | ruseni
;;
esac
```

Bude-li uvedený scénář uložen v souboru se jménem `udrzba`, zápis (v případě proveditelnosti souboru)

```
$ udrzba kopie
```

vytvoří kopii obyčejných souborů pracovního adresáře do podadresáře `zaloha` a

```
$ udrzba ruseni
```

odstraní po potvrzení všechny obyčejné soubory rovněž z pracovního adresáře.

## 4.11 VNOŘENÍ, REKURZE, FUNKCE

Ve čl. 4.9 jsme hovořili o principu provádění scénáře. Každá taková instance `sh (1)` vytváří podle pokynů scénáře další procesy a řídí jejich průběh. Jedním z takových synovských procesů může být také další shell řízený opět nějakým scénářem. Hloubka takového vnoření je teoreticky neomezená, prakticky je však v dané instanci omezená maximálním počtem procesů, které může uživatel vytvořit v rámci sezení.

Vnořený shell může být dále řízen tímž scénářem jako jeho otec – vzniká rekurze. Uvedeme příklad rekurzivního scénáře, pomocí něhož lze získat stromovou strukturu podadresářů pracovního adresáře. Uvažujme jméno souboru se scénářem `strom` a předpokládejme, že je uložen v domovském adresáři právě přihlášeného uživatele:

```
d=`pwd`
echo ""
echo "$d : "
# vnorenym shellem interpretujeme scenar adresar
sh $HOME/adresar
for i in *
do
# testujeme, zda je soubor podadresarem
if [ -d $d/$i ]
then
  cd $d/$i ;# je-li, vstupujeme do nej,
  sh $HOME/strom ;# rekurze scenarem strom
  cd ..
fi
done
```

## 4.11 Bourne shell

a v příkladu použitý scénář `adresar` obsahuje:

```
for i in *
do
    if [ -d $i ] ;# je vypis pouze polozek adresare
    then echo $i ;# ktere jsou podadresarem
    fi
done
```

Použijeme-li scénář `strom` v domovském adresáři `/usr/jan`, který má např. strukturu podle obr. 4.2 a kde uvedená jména odpovídají adresářům, příkaz `strom` má tento efekt:

```
$ strom

/usr/jan :
hry
prace

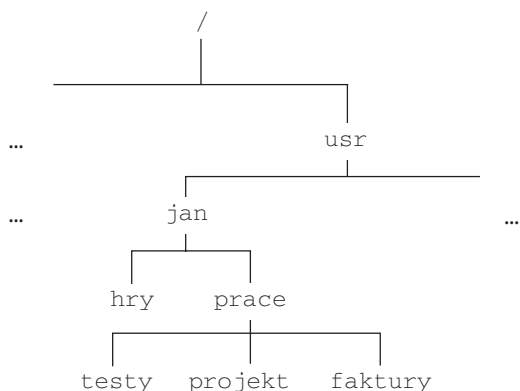
/usr/jan/hry :

/usr/jan/prace:
testy
projekt
faktury

/usr/jan/prace/testy :

/usr/jan/prace/projekt:

/usr/jan/prace/faktury:
```



Obr. 4.2 Příklad uživatelského podstromu

V případě scénáře `adresar` se můžeme vyhnout novému shellu a programovat `adresar` jako funkci scénáře `strom`. Takovou definici zapíšeme na začátku scénáře ve formátu

```
jmeno() { cmdlist ;}
```

kde `jmeno` je název funkce a `cmdlist` její tělo, např.

```
adresar() { for i in *
do
    if
    ...
done ;}
```



a ve scénáři příkaz

```
sh $HOME/adresar
```

nahradíme příkazem

```
adresar
```

## 4.12 LADĚNÍ SCÉNÁŘŮ

Po programování následuje etapa ladění. Činnost scénářů můžeme lépe sledovat pomocí volby `-x` nebo `-v` při spuštění shellu:

```
$ sh -x strom
...
```

Volbou `-x` získáme výpis všech příkazových řádků, které byly právě interpretovány (např. cykly jsou rozvinuty atd.). Volba `-v` je naopak výpis příkazových řádků před interpretací. Přitom jsou vypisované řádky pro `-x` označeny v prvním sloupci znakem `+`.

Volby pro ladění se automaticky nepřenášejí do vnoření dalších shellů.

Volby `-x` a `-v` můžeme nahradit pro určitou část scénáře příkazem `set` (v rámci scénáře):

```
set -x (nebo set -v)
```

a příkazem

```
set -
```

ladění vypínáme.

## 4.13 VSTUP Z TEXTU SCÉNÁŘE

Dvojnázev speciálního významu `<<` přesměruje standardní vstup na text scénáře. Znamená to, že standardní vstup příkazu ve scénáři nebude ani klávesnice, ani soubor s daty, ale text, který je součástí scénáře. Tento text začíná za příkazovým řádkem, ve kterém je použit `<<` a ukončen opakováním textu použitého za `<<` samostatně na řádku. Např.

```
cat >$1 << /*EOF
Následující text je
presmerovan do souboru se jmenem
podle prvnioho argumentu tohoto scenare.
A nasledujicim radkem je toto presmerovani ukonceno:
/*EOF
```

### 4.14 NĚKOLIK POZNÁMEK K C-SHELL

Vzhledem k nedostatkům Bourne shell vzniklo v průběhu vývoje operačního systému UNIX několik dalších příkazových interpretů. Na začátku 80. let vytvořil student William Joy na University of California v Berkeley interpret, který nazval C-shell. Hlavní myšlenkou byla snaha vytvořit způsob komunikace s operačním systémem velmi podobný jazyku C (odtud i název). Myšlenka usnadnit uživateli komunikaci byla jistě dobrá, ale se změnou syntaxe způsobila nepřenositelnost scénářů z Bourne shell.

Avšak C-shell nejen že připodobnil zápis řídicích struktur jazyku C, zavedl numerické proměnné a proměnné typu pole textových řetězců atd., ale pokusil se odstranit i jiné problémy, které Bourne shell neřešil. Je to zejména:

#### – zamezení přepisu existujících souborů:

Přesměrování `>` v Bourne shell na obyčejný soubor znamená vytvoření tohoto souboru, a pokud soubor existuje, je zkrácen na nulovou délku a poté naplněn standardním výstupem. Každému takovému nebo jakémukoliv jinému přepisu existujícího souboru přesměrováním můžeme v C-shell zabránit příkazem

```
% set noclobber ;; znak vyzvy v C-shell je %
```

který vytvoří proměnnou se jménem `noclobber`. Takže

```
% who > kdo
kdo : file exists.
% who >> kdo ;; soubor je chráněn i proti rozsíření
kdo : file exists.
% who >>! kdo ;; znak ! za přesměrováním ruší ochranu
% unset noclobber ;; ruší zamezení přepisu
```

#### – přesměrování:

C-shell je rozšířen o přesměrování standardního chybového výstupu přidáním znaku `&`.

```
% cc program.c >& chyby
```

je zápis přesměrování standardního chybového výstupu překladače jazyka C do souboru `chyby`. Přitom je zápis ekvivalentní obdobě ... `2>&1` (čl. 4.2), protože i zde je kanál 1 a 2 spojen. A analogicky platí pro `>>&` připojení ke standardnímu chybovému výpisu.

#### – řízení prací:

Spustíte-li v Bourne shell proces nebo kolonu na příkazovém řádku v popředí, neovlivníte běh procesů jinak, než že je přerušíte z klávesnice. V C-shell, podobně jako v Job Control shell, je možné proces pozastavit a po chvíli obnovit chod procesu od místa pozastavení, a to buď na popředí nebo na pozadí. Pozastavení procesu běžícího na popředí dosáhne uživatel klávesou Ctrl-z. Pro pokračování v běhu slouží vnitřní příkazy `bg` (background, na pozadí) a `fg` (foreground, na popředí), přitom může být příkaz následován číslem práce. Práce (job) je příkaz nebo kolona. Ke zjištění čísla práce slouží příkaz `jobs`, např.:

```
% jobs
[1] Running cc program.c
[2] Stopped ls | tee dir | grep *.c
```

Co řádek výpisu, to práce, číslo práce je první položka řádku uzavřená do hranatých závorek. A např.

```
% bg 2
```

přesouvá práci č. 2 na pozadí, ze stavu `Stopped` je převedena do stavu `Running`.

#### –mechanismus historie:

Je uchování dříve použitých příkazových řádků. Je realizován vnitřním příkazem `history`. Proměnná stejného jména musí být nastavena na hodnotu odpovídající počtu posledních zapamatovaných řádků. Příkaz vypisuje seznam posledních (v našem případě dvaceti) dosud použitých příkazových řádků:

```
% set history = 20
...
% history
10 who
...
18 ls -l
19 pwd
%
```

kde v uvedeném výpisu je každý dříve použitý příkaz očíslován. Opakovat některý z příkazů lze pomocí znaku `!`:

```
% !28
ls -l
...
```

(zápis `!!` je žádost o zopakování naposledy provedeného příkazového řádku z historie), přitom je možné v takto opětovně zadávaném příkazovém řádku požadovat změny. Editace je prováděna v řádkovém režimu obdobném jako v editoru `ed(1)`.

#### – mechanismus záměn:

Umožňuje definovat synonyma pro příkazový řádek. Vnitřní příkaz `alias` má formát

```
alias [ jmeno [definice] ]
```

a např.

```
% alias l ls -l
```

#### 4.14 Bourne shell

---

definuje nový příkaz se jménem `ll`, přitom jeho obsah je dán definiční částí `ls -l`, kterou C-shell jméno nahradí a provede. Dále

```
% alias ll pwd; ls -l
```

uvažuje stejnou sekvenci, a vzhledem k tomu, že jde o textovou substituci, můžeme např. psát

```
% ll; date
```

definované jméno jako součást příkazového řádku.

C-shell se dočkal brzké obliby a rychle se stal od UNIXu těžko oddělitelným. Jeho nenahraditelnost je dána také tím, že je pomocí něj naprogramováno několik důležitých nástrojů programátora (tools – viz kap. 6). V této souvislosti je důležité znát způsob interpretace programů. V souboru, který má nastavenou proveditelnost, může být uložen obraz programu ve strojovém kódu pro přímé provádění procesorem. Pokud jde ale o soubor textový, UNIX jeho obsah interpretuje pomocí `/bin/sh`. Je-li dále první znak textového souboru `#`, je interpretován pomocí `/bin/csh` (C-shell).

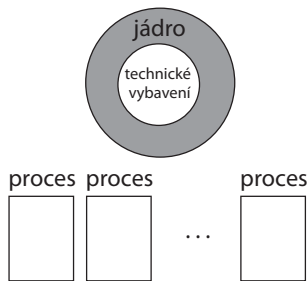
## 5 Volání jádra

Jádro (kernel), obr. 5.1,



Obr. 5.1 Jádro

obaluje technické vybavení (hardware) a umožňuje více uživatelům současně využívat v daném čase technické zdroje výpočetního systému. Kromě režie jádra (správa paměti, správa procesů, systémová vyrovnávací paměť atd. – viz kap. 2) poskytuje jádro procesům služby pro práci s technickými zdroji počítače (obr. 5.2).



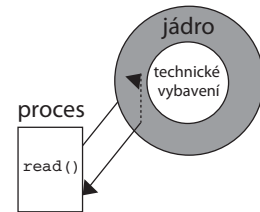
Obr. 5.2 Jádro a procesy

Tyto služby odpovídají konstrukci a funkci operačního systému UNIX a každý proces je využívá prostřednictvím volání jádra (System Calls).

Je to např. žádost procesu o přenos dat z určité periferie do datové oblasti procesu, v UNIXu nazvaná `read(2)`. Proces volá jádro v době svého běhu a vstupuje tak do supervizorového režimu, do režimu jádra. Jádro od této chvíle provádí požadovanou službu tak, že interpretuje odpovídající část svého textového segmentu (viz obr. 5.3).

V době realizace volání jádra je jádro nepřerušitelné jiným procesem. Proces volá jádro stejným způsobem jako obvyklou (ať už uživatelem definovanou nebo knihovní) funkci. Při sestavování programu do běhuschopné podoby jsou odkazy na funkce volání jádra získány z knihovny modulů

```
/lib/?libc.a
```



Obr. 5.3 Proces a jádro

kde `?` je znak určující paměťový model (`L`, `M`, `S` – `Long`, `Middle`, `Small` – viz kap. 6). Funkce z této knihovny (např. funkce `read()`) obsahují především instrukci vstupu do jádra (`trap`). Tím přechází uživatelský program do privilegované fáze svého běhu, která je realizována provedením odpovídající části textového segmentu jádra.

### 5.1 UŽIVATEL

Návratová hodnota volání jádra je celočíselná hodnota. Jádro signalizuje kolizi (odmítne vykonat akci) návratovou hodnotou `-1`.

Pro analýzu kolize má proces k dispozici globální celočíselnou proměnnou `errno`, a ta je jádrem naplněna označením důvodu kolize. Symbolické konstanty takových stavů má

## 5.2 Volání jádra

uživatel k dispozici v díle ( 2 ) dokumentace man (viz čl. 1.2) konkrétního UNIXu. V tomto díle má uživatel také uveden přesný formát a popis volání jádra.

Volání jádra, kterými operační systém uspokojuje požadavky uživatele, můžeme rozdělit do částí zabývajících se

- procesy,
- systémem souborů,
- komunikací mezi procesy,
- sítěmi,
- ostatní.

### 5.2 PROCESY

Proces je základním elementem chodu systému.

Uživateli je k dispozici volání jádra `fork( 2 )`, které má formát:

```
int fork(void);
```

Jádro vytvoří nový proces, který je

- identický s procesem, který jej vytváří, tzn. je vytvořen nový proces, jehož textový segment, datový segment i zásobník jsou tytéž jako segmenty volajícího procesu, proces je pouze odlišen novým PID
- provádění nově vytvořeného procesu pokračuje od místa volání `fork( 2 )`
- nově vzniklý proces je označován jako syn volajícího procesu, který je otcem

Od této chvíle otec i syn pracují nad týmž textovým kódem. Z návratové hodnoty `fork( 2 )` může proces identifikovat svoji totožnost. Je-li návratová hodnota 0, proces je syn, tedy nově vzniklý proces, je-li návratová hodnota nenulové kladné číslo, proces je otec a návratová hodnota odpovídá PID syna. Např.:

```
extern int errno;          /* v případě kolize je možné identifikovat
                             duvod */

perror(retez)              /* funkce panic error */
char *retez;
{
    fprintf(stderr, "%s\n", retez);
    exit(-1);
}

main()
{
```

```

switch(fork()){
    case -1: perror("Chyba!"); break;
    case 0:  printf("Syn\n"); break;
    default: printf("Otec\n"); break;
}
exit(0);
}

```

kde od volání jádra `fork(2)` v případě bezchybného provedení běží dva procesy, z nichž každý vypíše svoji totožnost. V případě kolize je na standardní chybový výstup vypsán pouze text `Chyba!` a žádný proces – syn nevzniká.

Synchronizovat lze procesy pomocí volání jádra `wait(2)`. V uvedeném příkladu nelze dopředu jednoznačně říct, který proces svoji identifikaci vypíše jako první (záleží to na modulech jádra pro správu přidělování paměti). Otec ale může na svého syna čekat. V tomto případě můžeme v uvedeném příkladu ve větvi `default` psát

```

...
main()
{
    int status;
    switch(fork()){
    ...
        default: wait(&status); printf("Otec\n"); break;
    ...
}

```

Volání jádra `wait(2)` má formát

```
int wait(int *stat_loc);
```

a tímto voláním je volající proces jádrem pozastaven do chvíle, kdy jeden ze synů ukončí svoji činnost. Proměnná `stat_loc` je naplněna příčinou ukončení, ve spodních osmi bitech je kód příčiny, v horních osmi je koncový stav syna (hodnota parametru `parm` příkazu `exit(parm)`). Návrátová hodnota `wait(2)` je PID ukončeného syna. Nebyl-li dříve žádný proces voláním `fork(2)` vytvořen, `wait(2)` je okamžitě ukončen s návratovou hodnotou `-1` a proměnná `errno` je ve stavu `ECHILD`.

Zmíněný příkaz `exit(2)` je voláním jádra. Má formát

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

a je žádostí o ukončení běhu procesu. Hodnotou proměnné `status` může být otec končícího procesu informován o způsobu ukončení, pokud to vyžaduje pomocí `wait(2)`.

## 5.2 Volání jádra

Tato hodnota také odpovídá návratové hodnotě příkazu v Bourne shell, která je v tomto případě určující pro tok řízení (viz čl. 4.10).

Volání jádra `exec(2)` umí přepsat textový a datový segment, který odpovídá programu v argumentech volání. Např.

```
...
main()
{
    int chpid, status;

    switch((chpid=fork())){
        case -1: perror("Chyba!");
        case 0: execl("/bin/ls", "ls", "-l", NULL);
        default: while(wait(&status)!=chpid); break;
    }

    ...
} /* konec main() */
```

otec čeká na ukončení svého syna, který provede výpis obsahu pracovního adresáře.

Formát volání `exec(2)` má několik tvarů. Uvedme nejpoužívanější z nich:

```
int execl(const char *path, const char *arg0, const char *arg1,
        ..., const char *argn, (char *)0);
```

kde argument `path` je cesta souboru s programem, jehož textový segment je nahrazen stávajícím, argumenty `arg0`, `arg1`, ..., `argn` jsou parametry odpovídající řetězcům pole `argv` funkce `main()` ve spuštěném procesu.

```
main(int argc, char *argv[])
{
    ...
}
```

Výčet argumentů je ukončen `NULL` specifikací.

Formát

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

dává možnost zápisu příkazového řádku v jednom textovém řetězci, takže lze namísto

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



psát

```
execv("/bin/ls", args);
```

kde `args` je definováno jako

```
char *args[]={ "ls", "-l" };
```

konečně je možné použít způsob `execve(2)` (analogicky `execle(2)`) ve formátu

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

kde `envp` obsahuje prostředí procesu v definovaných proměnných (např. exportovaných proměnných příkazového interpretu Bourne shell) způsobem

```
PROM=obsah
```

kde `PROM` je jméno proměnné a `obsah` je její hodnota (textový řetězec). Místo přístupu k prostředí procesu je pak právě pole textových řetězců `envp` v argumentech `main()` spuštěného procesu

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

Upozorňovat se navzájem na vznik určité situace mohou procesy pomocí signálů, a to pomocí volání jádra `kill(2)`. Má formát:

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

přítom signál `sig` je zaslán procesu s identifikačním číslem `pid` (PID). Např.

```
main()
{
    int pid;

    switch(pid=fork()){
        case -1: exit(-1); /* chyba pri vytvareni syna */
        case 0: for(;;); /* syn aktivne ceka ve smyce */
        default: sleep(10); /* otec po 10 vterinach */
            switch(kill(pid, 2)){
                /* posila synu signal c. 2 */
                case -1: perror("Chyba kill()!");
                case 0: printf("O.K.\n");
                break;
            }
    }
```

## 5.2 Volání jádra

```
    }  
    exit(0);  
}
```

V příkladu PID syna využívá otec k zaslání signálu č. 2 po 10 vteřinách čekání. Funkce `sleep(3)` je standardní, ve svém argumentu obsahuje počet vteřin, po který je proces pozastaven. Signál č. 2 odpovídá pokynu přerušení z klávesnice, proces syn je tehdy zrušen.

Signál může být také zaslán synem otci. Syn zjišťuje PID svého otce voláním jádra `getppid(2)`. Proces může své vlastní PID zjistit voláním jádra `getpid(2)`:

```
int getpid(void);  
int getppid(void);
```

Předchozí příklad můžeme obměnit tak, aby signál posílal syn otci takto:

```
main()  
{  
    switch(fork()){  
        case -1: perror("Chyba fork() !");  
        case 0: sleep(10);  
                switch(kill(getppid(), 9)){  
                    case -1: perror("Chyba kill()!");  
                            /* je nutno zastavit otce  
                               jinými prostředky */  
                    case 0: printf("O.K.\n");  
                            break;  
                }  
                break;  
        default: for(;;);  
    }  
    exit(0);  
}
```

kde otci syn posílá signál č. 9 (nejsilnější výzva k ukončení).

Uvedme si tabulku signálů, které mohou být procesy vysílány a přijímány. U každého čísla je uvedena jeho symbolická definice a význam, jehož je signál nositelem:

číslo	označení	implicitně	význam
1	SIGHUP	ukončení	odpojení terminálu
2	SIGINT	ukončení	přerušení z klávesnice
3	SIGQUIT	*	konec s uložením obrazu paměti
4	SIGILL	*	neznámá instrukce

5	SIGTRAP	*	ladící přerušení
6	SIGABRT	*	ukončení z důvodu v/v
7	SIGEMT	*	instrukce EMT
8	SIGFPE	*	kolize reálného čísla
9	SIGKILL	ukončení	okamžité ukončení procesu
10	SIGBUS	*	kolize sběrnice
11	SIGSEGV	*	selhání segmentace
12	SIGSYS	*	chybný tvar volání jádra
13	SIGPIPE	ukončení	zapisovanou rouru nikdo nečte
14	SIGALRM	ukončení	konec časového intervalu
15	SIGTERM	ukončení	ukončení
16	SIGUSR1	ukončení	první signál definovaný uživatelem
17	SIGUSR2	ukončení	druhý signál definovaný uživatelem
18	SIGCHLD	ignorování	změna stavu synovského procesu
19	SIGPWR	ignorování	kolize zdroje
20	SIGWINCH	ignorování	změna velikosti okna
21	SIGPOLL	ukončení	příznak při práci s PROUDY
22	SIGSTOP	pozastavení	signál pozastavení procesu
23	SIGTSTP	pozastavení	pozastavení procesu uživatelem
24	SIGCONT	ignorování	pokračování v činnosti
25	SIGTTIN	pozastavení	čekání na vstup
26	SIGTTOU	pozastavení	čekání na výstup
27	SIGXCPU	*	konec časového kvanta procesoru
28	SIGXFSZ	*	překročení stanovené délky souboru

Na příchod signálu může proces reagovat několika způsoby. Může signál nechat k vyřízení jádru, což ve většině případů znamená ukončení práce procesu (odtud mnemonický zápis kill). Dále může signál ignorovat, anebo ho zachytit a reagovat na něj určitou akcí. Signál č. 9 (SIGKILL) nelze zachytit ani ignorovat. V případě ukončení procesu jádrem u signálů v tabulce označených znakem \* jádro proces ukončí a v pracovním adresáři procesu vytváří obraz paměti procesu v době příchodu signálu a ukládá jej do souboru se jménem `core`. Pro možnost zpracování signálu slouží volání jádra `signal(2)`:

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

Znamená to, že pomocí funkce `func` je ošetřen příchod signálu `sig`. Po příchodu signálu je předáno řízení funkci `func` a pokud ona sama neukončí činnost práce procesu, po návratu z ní proces pokračuje od místa příchodu signálu. Např.

```
handler(s)
int s;
{
    signal(s, handler);
    printf("Nelze mne lehce prerusit\n");
}
```

## 5.2 Volání jádra

```
main()
{
    signal(SIGINT, handler);
    ...
}
```

je pomocí funkce `handler()` ignorováno přerušení z klávesnice. Po příchodu signálu je ve funkci `handler()` nastaveno opětovně maskování signálu (po příchodu signálu je totiž nastavena zpětně implicitní reakce na signál), a protože nenásleduje žádná akce, proces pokračuje dál.

Funkce `sleep(3)`, použitá v předchozích příkladech, je programována pomocí zasílání a přijímání signálů. Jádro totiž umožňuje použít volání jádra `pause(2)`, po němž je proces zablokován v další činnosti až do příchodu signálu. Poté je odblokován a případně pokračuje obsluhou funkcí. Proces může jádro požádat o zaslání signálu `SIGALRM` po uplynutí určitého počtu vteřin voláním jádra `alarm(2)`. Formáty jsou

```
int pause(void);

unsigned int alarm(unsigned int sec);
```

Potom lze funkci `sleep(3)` programovat:

```
handler_al()
{
}

sleep(sec)
int sec;
{
    signal(SIGALRM, handler_al);
    alarm(sec);
    pause();
}
```

### Definované funkce

```
#define SIG_DFL (int(*)())0
#define SIG_IGN (int(*)())1
```

při použití volání jádra `signal(2)` jsou k dispozici pro stanovení ignorování signálu (`SIG_IGN`) a pro ponechání jádra k vyřízení (`SIG_DFL`).

Procesy jsou sdružovány do skupin. PID vedoucího skupiny označujeme jako identifikaci skupiny (process group identification – PGID). Za znalosti PGID lze pomocí `kill(2)` poslat signál skupině procesů. Voláním jádra `getpgrp(2)` (v jeho návratové hodnotě) získáme PGID procesu:

```
int getpgrp(void);
```

Proces ale může sám sebe prohlásit za vedoucího takto vzniklé nové skupiny voláním jádra `setpgrp(2)`:

```
int setpgrp(void);
```

kde v návratové hodnotě je nové PGID (PID volajícího procesu). Od této chvíle všechny procesy jím vytvořené patří do jeho skupiny, pokud si samy nevytvoří skupinu novou.

Vedoucím skupiny procesů spuštěných z terminálu je proces příkazového interpretu (shell). Zašleme-li mu signál č. 9, on sám i všechny jeho synovské procesy, které si nevytvořily novou skupinu, končí svoji činnost. Situace se ovšem komplikuje, jsou-li některé procesy v supervizorovém režimu a očekávají např. příchod dat z periferie. Takový proces nezaniká, ale je mu přiřazen nový otec, kterým je **init**, proces s PID=1.

Uživatel, který vstupuje do systému přihlášením, je identifikován svým číslem uživatele (UID – user identification) a číslem své skupiny (GID – group identification), a to podle tabulky uživatelů `/etc/passwd` a `/etc/group`. Jak bylo popsáno v kap. 2, uživatel má tímto stanovena přístupová práva k souborům. Obsahuje-li soubor binární obraz spuštěného procesu, spustitelnost je dána nastavením odpovídajícího x-bitu pro skupinu uživatelů. V době běhu je však UID i GID procesu dáno jednak uživatelem, který proces spouští (tzv. reálné UID a GID), a jednak vlastníkem souboru s binárním obrazem (efektivní UID = EUID a efektivní GID = EGID). Je-li namísto x-bitu nastaven s-bit, jsou v době běhu procesu uživateli propůjčena práva vlastníka souboru. Tím je možné uživateli regulovaně dát k dispozici privilegované akce systému. Efektivní i reálnou identifikaci procesu zjistíme pomocí volání jádra

```
unsigned short getuid(void);    reálná identifikace vlastníka procesu
```

```
unsigned short getgid(void);    reálná identifikace pro skupinu vlastníka
                                procesu
```

```
unsigned short geteuid(void);   efektivní identifikace vlastníka procesu
```

```
unsigned short getegid(void);   efektivní identifikace pro skupinu vlastníka
                                procesu
```

Jádro umožňuje procesu přepínat identifikaci procesu mezi vlastníkem souboru a vlastníkem procesu pomocí

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

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

Přitom je-li efektivním vlastníkem procesu superuživatel, nastaví se UID nebo GID jako reálný i efektivní vlastník procesu. Není-li efektivním vlastníkem procesu superuživatel, je

## 5.3 Volání jádra

v UID nebo GID stanovena identifikace pro přepnutí na reálného nebo efektivního vlastníka procesu.

Z kap. 2 víme, že při startu je procesu stanovena uživatelská priorita, která je společně s doposud spotřebovaným časem procesu měřítkem k výpočtu dynamické priority. Podle dynamické priority je vybranému procesu přidělen čas procesoru. Výchozí uživatelská priorita je stanovena na 20, může se ale pohybovat v rozmezí 0–39 (0 je nejvyšší a 39 nejnižší priorita). Uživatel může pomocí volání jádra `nice(2)` uživatelskou prioritu ovlivnit.

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

Pokud proces stanovuje `incr` jako celou kladnou hodnotu, o tuto je mu priorita snížena. Zápornou hodnotou prioritu zvyšuje, ale pouze v případě privilegovaného procesu.

Zamknutí procesu v paměti dosáhne proces patřící privilegovanému uživateli voláním jádra

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

Proces je dále obcházen při určování, který z procesů má být z operační paměti přesunut do odkládací oblasti na disku (swap area) až do chvíle buď ukončení procesu, anebo zrušení zámku opět pomocí `plock(2)`. Hodnota `op` může být

PROCLOCK	zamknut je textový i datový segment
TXTLOCK	zamknut je pouze textový segment
DATLOCK	zamknut je pouze datový segment
UNLOCK	je zrušení zámků

S pamětí dále souvisí volání jádra

```
char *sbrk(int incr);
```

```
char brk(char *endds);
```

a obě rozšiřují datovou oblast procesu. Volání jádra `sbrk(2)` rozšiřuje datovou oblast o počet slabik daný argumentem `incr` a `brk(2)` posouvá hranici datové oblasti na novou hodnotu `endds`.

## 5.3 Systém souborů

Vzhledem k návaznosti na předchozí čl. 5.2, uveďme si volání jádra pro zjištění přístupových práv souboru:

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

kde `path` je cesta k souboru a hodnotou `amode` testujeme daná přístupová práva podle efektivní identifikace vlastníka procesu. `amode` může být

<code>R_OK</code>	přístup ke čtení
<code>W_OK</code>	přístup pro zápis
<code>X_OK</code>	provedení souboru
<code>F_OK</code>	existence souboru

Je-li přístup k souboru možný, návratová hodnota volání jádra je 0. Např.:

```
...
if(!access("/usr/include/stdio.h", R_OK | !W_OK){
    printf("O.K.\n");
}
...
```

fragment programu vypisuje text `O.K.`, je-li přístup k danému souboru pro čtení, ale je zakázán pro zápis.

Je-li vlastník procesu také vlastníkem souboru, může proces voláním jádra

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

změnit pro soubor daný cestou `path` přístupová práva bitovými příznaky v argumentu `protect`. Kolize nastává, není-li uživatel superuživitelem a požaduje-li změnu přístupových práv pro zapůjčení identifikace, na kterou nemá právo. Tehdy jsou tyto příznaky nulovány.

Vlastník souboru nebo superuživatel také může voláním jádra

```
int chown(const char *path, int uid, int gid);
```

změnit vlastníka souboru `path`, tj. předat jej uživateli danému identifikací `uid`, což je jednoznačná identifikace uživatele v rámci instalace, a `gid`, která určuje skupinu, do které nový vlastník patří. Číselné hodnoty `uid` a `gid` jsou evidovány pro všechny uživatele oprávněné vstupovat do systému (přihlašovat se) v tabulkách `/etc/passwd` a `/etc/group`.

Důležitá volání jádra pro práci se systémem souborů souvisejí se zpřístupněním obsahu souboru. Soubor otevíráme voláním jádra

```
int open(const char *path, int oflag [, int protect]);
```

kde v řetězci `path` zadáváme cestu souboru, `oflag` je způsob otevření souboru, který může být např.

### 5.3 Volání jádra

<code>O_RDONLY</code>	pouze pro čtení
<code>O_WRONLY</code>	pouze pro zápis
<code>O_RDWR</code>	pro čtení i zápis
<code>O_APPEND</code>	při každém zápisu bude soubor rozšiřován na konci souboru
<code>O_CREAT</code>	pokud soubor neexistuje, je nově vytvořen
<code>O_SYNC</code>	při zápisu dat do souboru jsou data přenesena ihned ze systémové vyrovnávací paměti na médium, proces je zablokovan do ukončení přenosu dat

Volitelná část `protect` je zadávána v případě použití `O_CREAT` a stanovuje přístupová práva vytvářeného souboru. Návratová hodnota `open(2)`, pokud jde o nezáporné celé číslo, je ukazatelem do tabulky otevřených souborů, který nazýváme deskriptorem souboru. Deskriptor souboru pak používáme v další manipulaci se souborem. Velikost tabulky otevřených souborů je dána hodnotou odpovídajícího parametru jádra při generaci systému a bývá stanovena nad 50 deskriptorů. Přitom je běžně již při startu procesu obsazen deskriptor č. 0 (standardní vstup – `stdin`), č. 1 (standardní výstup – `stdout`) a č. 2 (standardní chybový výstup – `stderr`). Při použití volání jádra `open(2)` je v tabulce nalezen první volný deskriptor v pořadí od 0, a ten je přidělen.

Uzavření souboru (uvolnění deskriptoru z tabulky) provedeme voláním jádra

```
int close(int fd);
```

kde `fd` je deskriptor dříve otevřeného souboru. Přitom není nutné před ukončením práce procesu explicitně uzavírat všechny právě otevřené soubory; po zániku procesu jádro ruší styk se soubory zrušením celé tabulky deskriptorů.

Voláním jádra

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

vytváříme soubor. Můžeme vytvářet i soubor již existující, tehdy je původní obsah souboru vyprázdněn a velikost souboru je zkrácena na nulu. Přístupová práva u již existujícího souboru zůstávají zachována, u nově vznikajícího jsou vytvořena podle odpovídajícího parametru `protect`.

Hodnota `protect` u volání jádra `open(2)` i `creat(2)` je určující pro přístupová práva vznikajícího souboru. Rozhodující je ale také dříve nastavená maska vytváření souborů, jejíž bitový komplement s `protect` přístupová práva stanovuje. Tuto masku nastavíme voláním jádra

```
int umask(int mask);
```

Maska je děděna z otce na syna a její běžná hodnota je 0133. Obvyklou hodnotou `protect` přitom bývá 0644,



Pro čtení z otevřeného souboru používáme volání jádra.

```
int read(int fd, char *buf, unsigned count);
```

a pro zápis

```
int write(int fd, const char *buf, unsigned count);
```

Jedním voláním jádra přenášíme počet slabik daných hodnotou `count`. `fd` je deskriptor otevřeného souboru a `buf` je pole znaků v datové oblasti procesu, kam nebo z které je posloupnost slabik přenesena. `read(2)` i `write(2)` jako návratovou hodnotu vracejí počet skutečně přenesených slabik. U `read(2)` je návratová hodnota menší než `count` signalizací konce souboru.

Příklad kopie souboru je:

```
#include      <stdio.h>
#include      <fcntl.h>
#define       BSIZE      30000
#define       PROTECT    0644
char buf[BSIZE];

main(argc, argv)
int argc;
char *argv[];
{
    int sid, tif;
    int n;
    if(argc!=3){
        perror("kopie: použijte: kopie odkud kam");
    }
    if((sid=open(argv[1], O_RDONLY))== -1){
        perror("kopie: nemohu otevrit zdrojovy soubor");
    }
    if((tid=creat(argv[2], PROTECT))== -1){
        perror("kopie: nemohu vytvorit cilovy soubor");
    }
    while((n=read(sid, buf, BSIZE)) > 0){
        if(write(tid, buf, n) != n){
            perror("kopie: chyba zapisu");
        }
    }
    exit(0);
}
```

### 5.3 Volání jádra

Proveditelná podoba programu je uložena v souboru se jménem `kopie` a je striktně vyžadováno, aby zadání v příkazovém řádku obsahovalo po řadě zdrojový a cílový soubor.

Velikost přenášeného bloku dat je zde stanovena na 30 000 slabik. Často ale z důvodů ať paměťových či ryze praktických nebude program potřebovat přenos tak velkého objemu dat najednou. Optimální hranici pro kompromis mezi opakovanou žádostí o přenos dat jádrem a požadavkem procesu na daná data zajišťuje v jazyce C konstrukce zvaná streams. Známa konstrukce nabízí sadu funkcí `fopen(3)`, `fclose(3)`, `getc(3)`, `putc(3)`, ..., přitom režii vyrovnávací paměti pro práci se souborem zajišťují streams. Jde o běžně využívané funkce, a proto je doporučuji pozornosti čtenáře.

Pro práci s deskriptory souborů je k dispozici volání jádra `dup(2)`

```
int dup(int fd);
```

`fd` je již dříve alokovaný deskriptor. Návrátová hodnota volání je nový deskriptor souboru, který je ale spojen se stejným souborem jako deskriptor `fd`. Jde o odkaz na soubor z více míst v procesu.

V i-uzlu souboru je uložena informace o počtu odkazů na soubor z různých adresářů (viz čl. 2.3). Další odkaz na soubor získáme voláním jádra

```
int link(const char *path, const char *newpath);
```

Z uvedeného formátu je zřejmé, že pro volání stanovujeme cestu k existujícímu souboru `path` a cestu k novému souboru `newpath`. Nový soubor se objevuje v systému souborů, ale není pro něj vytvořen nový i-uzel. Je to jen nový odkaz na tatáž data na disku.

Nepřímý odkaz (Symbolic Link) vytváří volání jádra

```
int symlink(const char *path, const char *newpath);
```

Zjistit obsah datové části nepřímého odkazu, tj. cestu k souboru nepřímého odkazu, může volání jádra

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

kde `path` je jméno souboru a `buf` je místo, kam jádro vloží obsah datové části souboru `path`. `bufsize` je velikost pole `buf`, data v `buf` po úspěšném návratu z jádra nejsou ukončena znakem binární nuly.

Naopak, snížit počet odkazů o 1 v i-uzlu lze voláním jádra

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

kde řetězcem znaků `path` uvádíme cestu k souboru. Je zrušena vazba adresář – i-uzel a počet odkazů v i-uzlu je snížen o 1. Je-li po odpočtu výsledný počet odkazů 0, je to znamení ke

zrušení souboru, i-uzel je uvolněn ze seznamu alokovaných, stejně jako datové bloky s ním spojené.

Efektivní přesun aktuální pozice pro čtení nebo zápis v otevřeném souboru (ukazovátka) dosáhneme voláním jádra

```
long lseek(int fd, long int offset, int from);
```

Po otevření souboru je ukazovátka nastaveno na jeho začátek. Při použití `lseek(2)` argumentem `offset` určujeme počet slabik, o které má být ukazovátka přesunuto od pozice `from`. Jako `from` může být použito

```
SEEK_SET    od začátku souboru
SEEK_CUR    od současné pozice
SEEK_END    od konce souboru
```

Např. nastavit ukazovátka na začátek souboru umíme zápisem

```
lseek(fd, 0L, SEEK_SET);
```

Návratová hodnota `lseek(2)`, pokud všechno proběhlo bez kolize, je nově nastavená pozice. Pouhé zjištění pozice ukazovátka umíme způsobem

```
long ukazovatko;
...
ukazovatko=lseek(fd, 0L, SEEK_CUR);
```

Jádro udržuje ukazovátka pro každý otevřený soubor. S odkazem na kap. 2, kde jsme uvedli, že soubor může být vícenásobně přístupný, je zřejmé, že pro každé nové otevření souboru vzniká ukazovátka nové, a to ať už v rámci více procesů nebo pouze jednoho.

Získat všechny důležité informace o souboru uložené v jeho i-uzlu můžeme voláním jádra

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

`path` je cesta k souboru a obsah struktury `buf` je naplněn informacemi o souboru. Struktura `buf` má složky:

```
mode_t st_mode;    /* přístupova práva */
ino_t st_ino;      /* číslo i-uzlu */
dev_t st_dev;      /* identifikace svazku */
dev_t st_rdev;     /* identifikace zařízení, pouze jedna-li se
                    o speciální blokový nebo znakový soubor */
nlink_t st_nlink;  /* počet odkazu */
uid_t st_uid;      /* vlastník souboru */
gid_t st_gid;      /* skupina */
off_t st_size;     /* velikost souboru ve slabikách */
```

### 5.3 Volání jádra

```
time_t st_atime;    /* datum a cas posledniho pristupu */
time_t st_mtime;    /* datum a cas posledni zmeny */
time_t st_ctime;    /* datum a cas posledni zmeny atributu */
long st_blksize;    /* optimalni navrhovana prenosova velikost */
long st_blocks;     /* pocet alokovanych diskovych bloku */
```

Proces může jádro požádat o výhradní přístup k souboru. Proces zamyká soubor voláním jádra `fcntl(2)` a je možné zamykat pouze jeho část (segment). Segment je dán pozicí svého začátku a počtem slabik. Nelze zamykat segmenty vzájemně se překrývající. `fcntl(2)` je volání jádra pro manipulaci s otevřeným souborem. Má formát:

```
int fcntl(int fd, int cmd [, arg]);
```

`cmd` je operace se souborem `fd` a `arg` je argumentem operace. Jedna z možných operací je vytvoření zámku

```
fcntl(fd, F_GETLK, *lock)
```

a voláním

```
fcntl(fd, F_SETLKW, *lock)
```

pracujeme se zámkem. `lock` je ukazatel na strukturu

```
struct flock {
    short l_type;    /* typ zamku */
    short l_whence;  /* zacatek segmentu */
    long l_start;    /* vztah zacatku segmentu k souboru */
    long l_len;      /* delka segmentu */
    short l_pid;     /* identifikace procesu */
    short l_sysid;   /* identifikace systemu */
};
```

kde pomocí proměnné `l_type` stanovíme prováděnou akci jako

<code>F_RDLCK</code>	zamknutí pro čtení
<code>F_WRLCK</code>	zamknutí pro zápis
<code>F_UNLCK</code>	zrušení zámků

`l_whence` určuje začátek segmentu, a to od pozice dané `l_start`, která může stanovit začátek souboru (hodnotou 0), současnou pozici (1) a konec souboru (2). Délka segmentu `l_len` nesmí být záporná při stanovení začátku souboru. Je-li hodnota `l_len` nulová, stanovujeme zámek od začátku segmentu do konce souboru, včetně případného rozšíření souboru. Takže nastavením

```

struct flock lock;
...
lock.l_type=(F_RDLCK | F_WRLCK);
lock.l_whence=0;
lock.l_start=0;
lock.l_len=0;
...
fcntl(fd, F_SETLKW, *lock);
...

```

zamykáme celý soubor daný identifikací `fd` pro čtení i zápis. Proměnné `l_pid` a `l_sysid` jsou naopak při volání s operací `F_GETLK` naplněny informací o zámku, pokud tento již existuje.

Není-li možné segment uzamknout, je volající proces pozastaven do chvíle, kdy jiný proces svůj zámek zruší. Namísto `F_SETLKW` můžeme použít operaci `F_SETLK`, což má za následek nikoliv čekání na zrušení zámku, ale ukončení volání s návratovou hodnotou `-1`.

Z důvodu jednodušší manipulace může proces zamknout data v souboru také voláním jádra

```
int lockf(int fd, int function, long size);
```

kde `fd` je deskriptor souboru, `function` způsob zamykání:

<code>F_LOCK</code>	zamknutí s výhradním přístupem (zda pro čtení, zápis atd. určuje způsob otevření souboru z <code>open(2)</code> )
<code>F_TEST</code>	pouze zjistí, zda data nejsou zamknuta jinými procesy
<code>F_TLOCK</code>	testuje, zda data nejsou zamknuta, a nejsou-li, zamyká je
<code>F_ULOCK</code>	zrušení zámku

a `size` je velikost zamknutého segmentu souboru ve slabikách. Začátek zamykaného segmentu je dán ukazovátkem otevřeného souboru.

Adresář je binární soubor se zvláštní strukturou. Lze na něj uplatnit uvedené volání jádra, stejně jako na obyčejný soubor. Volání jádra, které se vztahuje výhradně k adresářům, je

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

a určuje změnu pracovního adresáře procesu.

Pro práci se speciálními soubory uveďme dvě volání jádra. První z nich je `mknod(2)` pro vytvoření speciálního souboru, druhé je `ioctl(2)`, pomocí něhož můžeme s odpovídající periferií manipulovat. Formáty jsou

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

### 5.3 Volání jádra

a

```
int ioctl(int fd, int cmd [, arg]);
```

kde pro `ioctl(2)` je `fd` deskriptor otevřeného speciálního souboru (voláním jádra `open(2)` a `cmd` je operace, která se společně se svými argumenty `arg` předává jádrem periférii. `cmd` i `arg` jsou silně závislé na typu periférie a jejího ovladače. Typ `arg` je závislý na `cmd`.

Zajímavým použitím `ioctl(2)` je vsouvání modulů pro úpravu dat mezi proces a ovladač periférie. Tento princip navrhl v r. 1984 D. M. Ritchie [8] a znamená to, že pomocí operace `PUSH` může běžící proces do oblasti přenosu dat mezi jeho datovou oblastí a periférií vsunout frontu dvojic – `PROUD`, které zajišťují konverzi dat. Modulů s frontou dvojic může být přitom teoreticky neomezené množství. Další operací `PUSH` vsouváme další modul a operací `POP` naopak naposledy vložený modul rušíme. Na principu `PROUD`ů pracuje např. virtualizace více terminálů na jednom fyzickém terminálu. `PROUD`y jsou ale především používány pro implementaci síťových protokolů (viz kap. 8).

Volání jádra `mknod(2)` vytváří speciální soubor, jehož jméno je dáno cestou k souboru `path`. V argumentu `mode` je zadávána žádost o vytvoření buď znakového nebo blokového speciálního souboru společně s přístupovými právy. V argumentu `dev` stanovujeme hlavní (major) a vedlejší (minor) číslo souboru (viz kap. 2).

Voláním jádra `mknod(2)` vytváříme i-uzel. Seznam možných hodnot argumentu `mode` je totiž následující:

<code>S_FIFO</code>	soubor pojmenované roury
<code>S_IFCHR</code>	znakový speciální soubor
<code>S_IFDIR</code>	adresář
<code>S_IFBLK</code>	blokový speciální soubor
<code>S_IFREG</code>	obyčejný soubor

Pojem pojmenovaná roura souvisí s komunikací mezi procesy a popíšeme jej v následujícím článku (5.4). Adresář vytvořený pomocí `mknod(2)` má přiřazen i-uzel s odpovídajícím obsahem, ale obsah adresáře zatím není úplně v pořádku, nejsou v něm totiž vytvořeny dvě základní položky `.` a `..` (reference na sebe sama a nadřazený adresář), a tím ani vazba do struktury systému souborů. O to se musí postarat proces v dalším běhu sám (např. `mkdir(1)`).

Další hodnoty argumentu `mode` jsou

<code>S_ISUID</code>	s-bit pro vlastníka
<code>S_ISGID</code>	s-bit pro skupinu
<code>S_ENFNT</code>	t-bit

a přístupová práva, mohou být

<code>S_IRUSR</code>	čtení pro vlastníka
<code>S_IWUSR</code>	zápis pro vlastníka

S_IXUSR	proveditelnost pro vlastníka
S_IRGRP	
S_IWGRP	
S_IXGRP	čtení, zápis, proveditelnost pro skupinu
S_IROTH	
S_IWOTH	
S_IXOTH	čtení, zápis, proveditelnost pro ostatní

**Svazek** je uspořádání dat na magnetickém médiu s náhodným přístupem tak, že je akceptovatelné jádrem jako systém souborů a adresářů. Struktura svazku byla popsána v kap. 2 a dále se jí budeme věnovat v kap. 9. Zde si uvedme volání jádra, která umožňují připojit a odpojit svazek k (od) systému souborů a zjistit základní informace o připojeném svazku.

Volání jádra

```
int mount(const char *spec, const char *dir, int rw,
          const char *fstype, const char *dataptr, int datalen);
```

umí připojit svazek. `spec` je jméno speciálního souboru, `dir` je adresář (obvykle prázdný), který bude spojen s kořenovým adresářem připojovaného svazku, a `rw` zadává čtení a zápis (hodnotou 0) nebo možnost pouze čtení (1) ze svazku; `fstype` je jméno typu svazku (viz kap. 2). Jádro pochopitelně připojí pouze svazky, jejichž typ zná a podporuje. Poslední dva argumenty `dataptr` a `datalen` jsou vztaženy ke konkrétnímu typu svazku a není-li vyžadována speciální manipulace, mohou oba být nulové.

Svazek odpojujeme voláním jádra

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

kde `spec` je speciální soubor svazku.

Konečně voláním jádra

```
int ustat(int dev, struct ustat *buf);
```

můžeme zjistit počet volných bloků a volných i-uzlů na připojeném svazku. `dev` obsahuje identifikaci zařízení (hlavní a vedlejší číslo) a struktura `buf` po volání obsahuje informace o svazku (viz obsah souboru `/usr/include/ustat.h`).

## 5.4 KOMUNIKACE MEZI PROCESY

Procesy mohou komunikovat na základní úrovni pomocí signálů (čl. 5.2). Tento způsob je bohužel omezen pouze na upozornění na určitou událost a reakci na ni. Dále se u procesů předpokládá znalost svých PID.

Dva procesy mohou spolu komunikovat na úrovni předávání dat pomocí roury (pipe). Roura je paměťová oblast sdílená dvěma procesy tak, že jeden proces do roury pouze zapisuje a druhý proces z ní pouze čte. Jedná se o frontu FIFO (First In First Out), to znamená, že čtoucí

## 5.4 Volání jádra

proces obdrží nejprve data, která zapisující proces poslal do roury jako první. V mnemonice Bourne shell můžeme schématicky psát

```
PRODUCENT | KONZUMENT
```

kde znak `|` je symbol roury a proces PRODUCENT do roury zapisuje, kdežto KONZUMENT z roury data čte.

Proces může vytvořit rouru voláním jádra

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

a získává tím deskriptor čtení z roury `pd[0]` a deskriptor zápisu `pd[1]`. Ukažme si mechanismus `pipe(2)` na realizaci příkladu příkazového řádku Bourne shell

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

jehož výsledným efektem je výpis počtu položek pracovního adresáře na standardní výstup. Napíšeme text programu, kde vzniknou dva synovské procesy a každý z nich bude realizovat jeden z programů `ls` a `wc`:

```
#include <stdio.h>
int pd[2];
int status;
main()
{
    switch(fork()) {
        case -1: exit(1); /* Chyba */
        case 0:  /* první syn - KONZUMENT, realizuje program wc */
            pipe(pd);
            switch(fork()){ /* první syn vytváří svého
                               partnera pro komunikaci */
                case -1: exit(-1); /* Chyba 1. syna */
                case 0:  /* druhý syn - PRODUCENT, realizuje
                           program ls */
                    close(1);
                    dup(pd[1]);
                    close(pd[0]);
                    execl("/bin/ls", "ls", NULL);
                default : /* první syn */
                    close(0);
                    dup(pd[0]);
                    close(pd[1]);
                    execl("/bin/wc", "wc", "-l", NULL);
            } /* konec switch() při vytvoření druhého syna */
        default : /* Otec čeká na ukončení komunikace synu */
    }
```



```

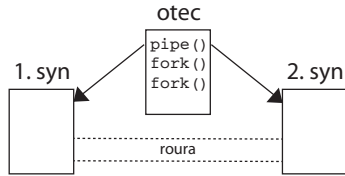
wait(&status) ;
exit(status & 0377);
} /* konec switch() */
} /* konec main() */

```

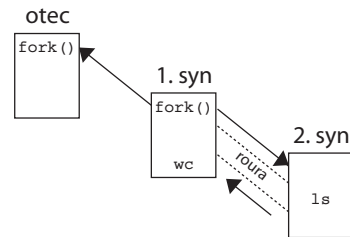
Uvedený příklad je schématicky zobrazen na obrázku 5.4.

Spuštěný proces vytváří syna (je označen jako 1. syn) a čeká na jeho dokončení. Po dokončení vrací tentýž návratový status. Syn vytvoří rouru voláním jádra `pipe(2)` a voláním `fork(2)` svého syna (2. syn). Chová se dále jako KONZUMENT, uzavírá svůj standardní vstup a voláním `dup(2)` si jako svůj standardní vstup připojí deskriptor roury pro čtení, uzavře rouru pro zápis a čtením svého standardního vstupu bude nyní připraven odebírat data z roury. Nakonec sám sebe přepíše programem `wc(1)`. Proces pokračuje

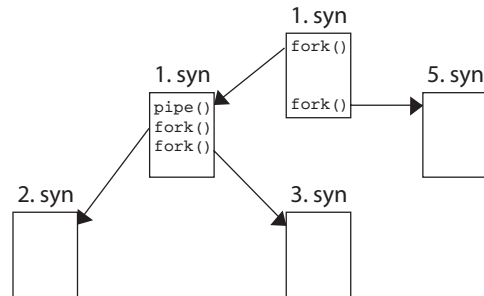
dál podle textu `wc(1)`, zdědil přitom atributy svého otce; vstup programu `wc(1)` bude tedy přesměrován na vstup roury. Jeho syn (označován jako 2. syn) se naopak chová jako PRODUCENT, zdědil kromě všech otevřených kanálů také deskriptory vytvořené rourou, uzavírá svůj standardní výstup, voláním jádra `dup(2)` si jako svůj standardní výstup připojí deskriptor roury pro zápis, uzavře rouru pro čtení, a pokud nyní bude zapisovat na svůj standardní výstup, zapisuje do roury. Přepisuje se poté programem `ls(1)`, který dědí všechny jeho atributy. Tím jsou oba komunikující procesy vytvořeny a současně je mezi nimi vytvořen jednosměrný komunikační datový kanál zvaný roura. Oba komunikující procesy v příkladu uzavírají nepoužitý konec roury, aby se rozpoznalo, kdy PRODUCENT končí. Jádro rozpozná konec činnosti PRODUCENTA teprve tehdy, až zápisový konec roury není otevřen pro žádný proces. PRODUCENT je vytvářen jako druhý syn, protože ukončení jeho činnosti, které bude následovat za uzavřením roury, znamená ukončení činnosti všech jeho synů; v opačném případě by to znamenalo i ukončení činnosti `wc(1)`.



Obr. 5.5 Roura dvou synů stejné úrovně



Obr. 5.4 Roura `ls -l | wc`



Obr. 5.6 Rodina procesů

Není příliš obtížné modifikovat uvedený příklad na situaci podle obrázku 5.5 a čtenář si ji může z pilnosti naprogramovat jako domácí úkol.

Těžko bychom ale dokázali zabezpečit situaci pro sdílení téže roury procesem otec a 3. syn ve schématu obr. 5.6.

Nebo dokonce mezi 5. a 3. synem. Roura je totiž vytvořena 1. synem a tím je otci nedostupná. Uvedený problém pro UNIX řeší pojmenovaná roura (named pipe).

## 5.4 Volání jádra

Pojmenovaná roura vznikne pomocí volání jádra `mknod(2)` (viz čl. 5.3), je tak vytvořena vazba mezi jménem souboru (jménem roury) a rourou. Pojmenování roury se tedy vztahuje k systému souborů. Taková roura je od svého vytvoření dostupná kterémukoli procesu, který má oprávnění přístupu pro zápis nebo čtení. Pojmenovaná roura v systému souborů je rozpoznatelná svým i-uzlem a při výpisu pomocí `ls -l` má v prvním sloupci výpisu znak `p`. Např.

```
$ ls -l /usr/spool/lp/fifos/FIFO
prw----- 1 lp bin 0 Jul 31 14:23 /usr/spool/lp/fifos/FIFO
$
```

je využití pojmenované roury pro řízení synchronizace výstupu dat od více procesů na jednu tiskárnu. Jinak má pojmenovaná roura všechny potřebné vlastnosti roury dříve popsané (princip FIFO).

Pojmenovaná roura nemá sama o sobě k dispozici mechanismus pro organizovaný přístup pro čtení nebo zápis všech oprávněných procesů. Takový mechanismus musí být zajištěn jiným způsobem. Procesy s pojmenovanou rourou pracují jako s obyčejným souborem pomocí volání jádra `open(2)`, `close(2)` atd.

Další způsob komunikace procesů pro UNIX pokračuje fenoménem IPC (Interprocess Communication). V IPC mohou libovolné procesy komunikovat pomocí předávání zpráv (messages), sdílené paměti (shared memory) a semaforů (semaphores).

Oblast zpráv procesů je rozdělena podle klíčů. Každý klíč identifikuje frontu zpráv. Proces může voláním jádra

```
int msgget(key_t key, int flag);
```

požádat o připojení na frontu zpráv s klíčem `key` (obvykle je typu `long`). Rozhodující je také příznak napojení `flag`, protože např. v případě, že fronta zpráv s takovým klíčem doposud neexistuje, můžeme kombinací

```
flag & IPC_CREAT
```

požadovat vytvoření nové fronty. Návrátová hodnota volání je identifikace pro další práci s frontou zpráv (jinak v případě vytváření zprávy určuje `flag` přístupová práva k frontě zpráv).

Zprávu zašleme voláním jádra

```
int msgsnd(int id, void *msg, size_t count, int flag);
```

a získáme voláním jádra

```
int msgrcv(int id, void *msg, int count, long type, int flag);
```

kde `id` je identifikace fronty zpráv získaná návratovou hodnotou `msgget(2)`, `count` je počet slabik přenášené zprávy a `flag` je specifikace přenosu zprávy podle případných okolností (např. je možné hodnotou `MSG_NOERROR` požadovat přenos pouze daného počtu slabik, přestože nabízená zpráva ve frontě je delší). Struktura `msg` obsahuje položky

```
long mtype;
char mtext[];
```

kde `mtype` je typ zprávy a `mtext` obsah zprávy. Proměnná `type` může nabývat těchto hodnot:

0	je přijata první zpráva z fronty
větší než 0	je požadována první zpráva daného typu
menší než 0	první zpráva je nižšího typ než uvedené zpráva <code>type</code>

`msgsnd(2)` je úspěšné, je-li proces oprávněn zprávu k frontě připojit, podle přístupových zpráv, a jsou-li systémové zdroje (paměť, počet zpráv atd.) dostatečné. V případě úspěchu jádro probouzí proces, který na zprávu tohoto typu čeká. Proces pomocí `msgrcv(2)` může totiž být zablokovan do chvíle příchodu zprávy.

Pomocí volání jádra

```
int msgctl(int id, int cmd, struct msqid_ds *mstatbuf);
```

kde `id` je identifikace fronty zpráv, můžeme získat informace o frontě anebo je měnit. Je-li `cmd=IPC_STAT`, čteme charakteristiky fronty zpráv do struktury `mstatbuf` (přístupová práva, aktuální počet zpráv ve frontě, PID procesu, který jako poslední zapisoval do fronty atd.) a pomocí `IPC_SET` naopak charakteristiky frontě zpráv nastavujeme. Je-li `cmd=IPC_RMID`, fronta zpráv je zrušena (tehdy je na místě `*mstatbuf 0`).

Následující příklad je dvojice procesů, z nichž první pracuje v režimu poskytování služeb (server) a druhý je uživatel těchto služeb (client). Proces poskytování služeb běží v nekonečné smyčce a na žádost vrací zprávu obsahující jeho PID. (Pro vytvoření jedinečného klíče, odvozeného z řetězce znaků, slouží funkce `getkey(3)`, v příkladu je použita hodnota 15.)

Proces poskytování služeb:

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

#define MSGKEY 15

struct msgform {
    long mtype;
    char mtext[256]
} msg;
```

## 5.4 Volání jádra

---

```
int id;
main()
{
int pid, *pint;

/* vytvarime novou frontu zprav */
id=msgget(MSGKEY, 0777 | IPC_CREAT);
for(;;) {
    msgrcv(id, &msg, 256, 1L, 0);
    pint=(int *) msg.mtext;
    pid=*pint;
    msg.mtype=pid;
    *pint=getpid();
    /* do fronty posilame sve pid */
    msgsnd(id, &msg, sizeof(int), 0);
}
}
```

a proces využívající služeb:

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

#define       MSGKEY 15

struct msgform {
    long mtype;
    char mtext[256];
};

main()
{
    struct msgform msg;
    int id, pid, *pint;

    id=msgget(MSGKEY, 0777);

    pid=getpid();
    pint=(int *)msg.mtext;
    *pint=pid;
    msg.mtype=1;
    msgsnd(id, &msg, sizeof(msg), 0);
    msgrcv(id, &msg, 256, pid, 0);
}
```

Proces se obrací na démon (který běží v nekonečné smyčce), ve zprávě mu posílá své PID (které může řídicí proces později využít pro jiný typ komunikace) a dostává od něj v přijaté zprávě PID řídicího procesu. Řídicí proces může být v tomto případě zrušen pouze signálem `SIGKILL`, který nelze maskovat. Protože fronta zpráv nebyla před tímto ukončením zrušena pomocí

```
msgctl(id, IPC_RMID, 0);
```

zůstává v paměti dokud všechny procesy, které ji využívají, neskončí svoji činnost. Čtenář si může doplnit program řídicího procesu o funkci `handler()`, která např. na základě příchodu signálu `SIGINTR` zrušení zpráv provede.

Další formou komunikace mezi procesy je sdílená paměť. Syntax komunikace je podobná jako u předávání zpráv. K dispozici je datová paměť, jejíž část (oblast) lze alokovat podle daného klíče pro několik procesů. Přístup k oblasti je uskutečňován podle odpovídajícího klíče.

Volání jádra

```
int shmget(key_t key, int size, int flag);
```

alokuje oblast sdílených dat s klíčem `key` o velikosti `size` slabik. Příznakem `flag` můžeme obdobně jako u zpráv požadovat pouhé připojení k oblasti (`0777`), nebo vytvoření nové datové oblasti (`IPC_CREAT`, `IPC_PRIVAT`, `IPC_EXCL` – viz díl (2) uživatelské dokumentace). Návrátová hodnota `shmget(2)` je identifikace datové oblasti v rámci volajícího procesu. Potom volání jádra

```
void *shmat(int id, void *addr, int flag);
```

podle dané identifikace `id` zpřístupňuje sdílenou datovou oblast podle virtuální adresy `addr` (je-li hodnota `addr` nulová, jádro zpřístupňuje celou oblast). Je-li oblast právě zpřístupněna jiným procesem, `shmat(2)` čeká na přístup (hodnotou příznaku `flag=SHM_RND`, což neplatí, pokud je oblast pro proces stanovena jako dostupná pouze pro čtení (nastavitelné a zjistitelné pomocí `shmctl(2)`). Opouštíme-li oblast sdílené paměti a uvolňujeme-li ji pro přístup jiným procesům, využíváme volání jádra

```
int shmdt(void *addr);
```

kde `addr` je návratová hodnota úspěšné `shmat(2)` (virtuální adresa v rámci zpřístupňované oblasti).

Voláním jádra

```
int shmctl(int id, int cmd, struct shmid_ds *buf);
```

můžeme nastavit charakteristiky datové oblasti s identifikací `id` specifikovanou ve struktuře `buf` s hodnotou operace `cmd=IPC_SET`. Hodnotou `cmd=IPC_STAT` naopak do struktury

## 5.4 Volání jádra

buf charakteristiky datové oblasti načítáme a pomocí `IPC_RMID` rušíme datovou oblast vytvořenou pomocí `shmget(2)`. Je-li oblast využívána ještě jinými procesy, jádro čeká, dokud oblast nebude uvolněna všemi zúčastněnými procesy (pomocí `semctl(2)` a `IPC_RMID`), a teprve potom datovou oblast ruší.

Semafor, implementované v operačním systému UNIX, vychází s Dekkerova algoritmu semaforů, publikovaného Dijkstrou v r. 1968 [9]. Původně pro UNIX používaný způsob výlučného přístupu ke zdroji výpočetního systému pomocí vytváření a testu existence souboru (např. se jménem `LOCK`) byl nahrazen mechanismem zajišťujícím výlučný přístup za pomoci známých **P** a **V** operací nad celočíselnou datovou strukturou nazývanou semafor. **P** operace sníží hodnotu semaforu o 1, je-li tato hodnota větší než 0, a **V** operace hodnotu semaforu o 1 zvýší. UNIX využívá principu semaforů, dovoluje ale, aby hodnota semaforu byla větší než 1. Jádro zajišťuje výlučnost operace nad semaforem. UNIX uvažuje každý semafor jako

- hodnotu semaforu
- PID procesu, který naposledy se semaforem pracoval
- počet procesů, které čekají na zvýšení hodnoty semaforu
- počet procesů, které čekají na nulovou hodnotu semaforu

Volání jádra pro práci se semaforey jsou `semget(2)`, `semop(2)`, a `semctl(2)`. Použití je podobné mechanismu sdílené paměti nebo předávání zpráv. Odkaz na klíč `key` zde má význam odkazu na pole semaforů.

```
int semget (key_t key, int count, int flag);
```

zpřístupňuje pole semaforů odpovídající klíči `key`, které bude mít `count` položek (semaforů). `flag` má ekvivalentní význam jako např. u předávání zpráv. Návratovou hodnotou identifikujeme přidělenou skupinu semaforů a používáme ji ve volání jádra

```
int semop(int id, struct sembuf *oplist, unsigned count);
```

což je vykonání operace nad semaforem. `oplist` je ukazatel na pole operací se semaforey, `count` je jeho velikost. Každý prvek `oplist` obsahuje

```
short sem_num; /* identifikace (poradí) semaforu ve skupine */
short sem_op; /* operace nad semaforem */
short sem_flag; /* příznaky vztahující se k operaci */
```

Operaci `sem_op` můžeme vyjádřit

- zápornou hodnotou, kdy v případě, že součet `sem_op` a hodnoty semaforu je  $\geq 0$ , přičte hodnotu `sem_op` k hodnotě semaforu; jinak je proces blokován (podle příznaku `sem_flag`)
- kladnou hodnotou, kdy je k hodnotě semaforu připočtena hodnota `sem_op`
- nulou, proces pokračuje, je-li hodnota semaforu 0; jinak je proces zablokován (podle příznaku `sem_flag`)

## A třetí volání jádra

```
int semctl(int id, int number, int cmd,
           union semun { int val; struct semid_ds *buf;
                        ushort *array;
                        } arg);
```

je určeno pro zvláštní manipulace `cmd` se semaforey ve frontě `id`. `cmd` např. stanovuje nastavení nebo zjištění přístupových práv k semaforům, zjišťování informací o procesech čekajících na určitou hodnotu semaforů atd. `number` i zde identifikuje semafor.

Protože komunikace dvou procesů v počítačové síti bude předmětem diskuse kap. 8, na tomto místě uvedeme pouze stručnou ukázkou. Klasická metoda komunikace dvou procesů v síti se opírá o volání jádra `ioctl(2)` a mechanismus PROUDŮ (STREAMS), které jsou pomocí `ioctl(2)` vsouvány jako převodní moduly mezi ovladač a data procesu. Vzhledem k dosavadní nejednotnosti v sítích byla používána málo. Z několika dnes už standardizovaných způsobů komunikace v síti vyberme zde na ukázkou způsob vyvinutý pro systémy BSD (University of California v Berkeley) v první polovině 80. let, označovaný jako schránky (sockets, Berkeley sockets), viz [10],

Za voláním jádra musí být v implementaci (jádra) přítomna možnost propojení dvou operačních systémů.

Z hlediska schránek lze rozlišit 3 úrovně:

- úroveň volání jádra
- úroveň komunikačního protokolu (např. TCP/IP)
- úroveň přenosového zařízení (ovladač, např. zařízení Ethernet)

Schránky z pohledu uživatele budou reprezentovány úrovní volání jádra, jsou to volání jádra `socket(2)`, `bind(2)`, `connect(2)`, `listen(2)`, `accept(2)`, `recv(2)`, `send(2)`, `getsockname(2)`, `getsockopt(2)` a `setsockopt(2)`. Schránky pamatují nejen na možnost komunikace procesů v síti, ale stejného způsobu komunikace mohou docílit i procesy lokálního systému. Schránky jsou sdružovány do skupin (domains). Pro lokální komunikaci jsou určeny typy skupin UNIX system domain a pro práci v sítích Internet domain. Typ komunikačního protokolu může být stálé spojení (virtual circuit), např. TCP – Transmission Control Protocol a typu datagram (Internet Domain) např. UDP – User Datagram Protocol.

Uživatel používá volání jádra

```
int socket(int format, int type, int protocol);
```

pro zpřístupnění komunikace. `format` určuje způsob komunikace (např. `AF_INET` je síťový styk typu DARPA, `AF_UNIX` je lokální komunikace). `type` určuje typ komunikace (stálé spojení konstantou `SOCK_STREAM` nebo datagram konstantou `SOCK_DGRAM`) a `protocol` stanovuje způsob komunikačního protokolu.

Spojit návratovou hodnotu volání jádra `socket(2)` s konkrétním jménem (souboru) můžeme voláním jádra

## 5.4 Volání jádra

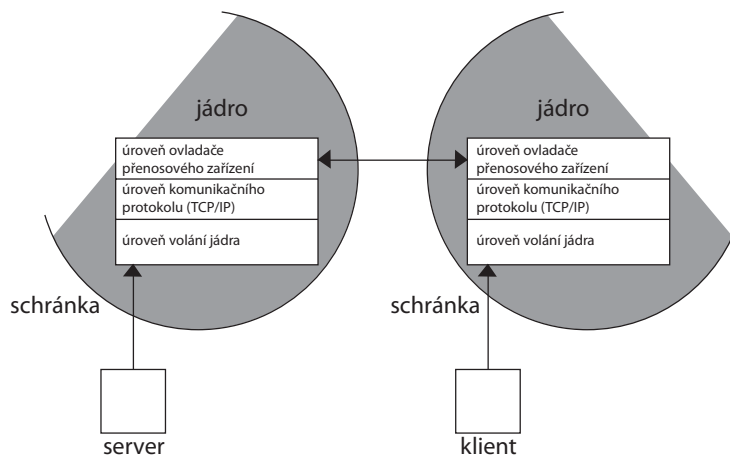
```
int bind(int sd, struct sockaddr *address, int length);
```

kde `sd` je návratová hodnota `socket(2)` (deskriptor schránky), `address` určuje strukturu pro spojení, formát `address` je proměnlivý podle způsobu komunikace a `length` je délka `address`. Např. pro lokální komunikaci je `address` jméno souboru.

Proces vlastní komunikaci navazuje voláním jádra

```
int connect(int sd, struct sockaddr *address, int length);
```

a parametry mají též význam jako u `bind(2)`.



Obr. 5.7 Schránky, jádro, síť

Schránky podporují komunikaci způsobu klient – server (client – server), proces je klient na základě poskytované služby svého protějšku (serveru) ze stejné schránky (obrázek 5.7). Server stanovuje maximální délku fronty příchozích požadavků pro napojení na schránku od klientů voláním jádra

```
int listen(int sd, int qlength);
```

v parametru `qlength`. Server přijímá požadavek napojení z fronty pomocí

```
int accept(int sd, struct socaddr *address, int *addrlen);
```

kde `sd` je deskriptor schránky, `address` je ukazatel na oblast dat procesu, kterou jádro naplní adresou spojovaného procesu klient.

Vlastní přenos dat probíhá pomocí

```
int recv(int sd, char *buf, int length, int flags);
```



pro příjem dat od schránky s deskriptorem `sd` do pole `buf`. `length` je očekávaný počet přijímaných znaků, přitom v návratové hodnotě volání jádra se proces dovídá skutečný počet přenesených slabik. Pro vyslání dat do schránky je k dispozici volání jádra

```
int send(int sd, char *msg, int length, int flags);
```

s analogickým významem parametrů.

Uvedme ještě, že je k dispozici také volání jádra `getsockname(2)`, které umí procesu sdělit jméno schránky stanovené dřívějším voláním jádra `bind(2)`. `getsockopt(2)` a `setsockopt(2)` slouží k získání nebo nastavení charakteristik schránky. Uzavření schránky (uvolnění deskriptoru) provedeme voláním jádra

```
int shutdown(int sd, int mode);
```

kde pomocí `mode` stanovíme, o kterou stranu schránky se jedná.

## 5.5 OSTATNÍ VOLÁNÍ JÁDRA

Se systémem souborů souvisí ještě volání jádra `sync(2)`, které pracuje s daty v systémové vyrovnávací paměti. Má formát

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

a provádí aktualizaci dat na všech připojených svazcích vzhledem k virtualizaci systémového stromu souborů a adresářů ve vyrovnávací paměti. Modifikovaná data jsou fyzicky přenesena na média.

Pro práci se systémovým časem jsou k dispozici volání jádra `time(2)` a `stime(2)`.

```
time_t time(time_t * tloc);
```

v návratové hodnotě získá proces počet uplynulých vteřin od 0:00 1. ledna 1970 greenwichského času. Pokud před voláním `tlock! = 0`, je i `tlock` naplněna touto hodnotou. Údaj ve vteřinách umí zpracovat do podoby jakou má běžné datum např. standardní funkce `ctime(3)`, která z obsahu proměnné typu `long` vytvoří textový řetězec běžného zápisu data a času v anglosaském světě, nebo funkce `localtime(3)` jejíž návratová hodnota ukazuje na strukturu s položkami minuty, hodiny, dne, měsíce atd.

Nastavit datum a čas může privilegovaný uživatel pomocí

```
int stime(const long *tptr);
```

kde proměnná `tptr` ukazuje na počet vteřin od 0:00 1. ledna 1970 greenwichského času.

Se sítěmi souvisí volání jádra

## 5.5 Volání jádra

---

```
int uname(struct utsname *name);
```

kde struktura `name`, která je po volání naplněna, obsahuje položky

```
char sysname[SYS_NMLN]; /* jmeno instalace systemu */
char nodename[SYS_NMLN]; /* jmeno uzlu v siti */
char release[SYS_NMLN]; /* verze systemu */
char version[SYS_NMLN]; /* pokračování verze systemu */
char machine[SYS_NMLN]; /* typ pocitace */
```

A na závěr ještě uvedme tři volání jádra, která podporují ladění programů.

Pomocí

```
long int times(struct tms *tbuf);
```

získá proces informace o spotřebovaném čase procesoru pro uživatelskou fázi a fázi supervizorovou (viz čl. 2.4), a to nejen svého vlastního času, ale i součet těchto časů pro všechny jeho synovské procesy.

Pomocí

```
int ptrace(int cmd, int pid, int addr, int data);
```

může proces trasovat jiný proces daný identifikací `pid` (PID). `cmd` je způsob trasování a pomocí

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

můžeme získat statistiku frekvence doby trvání funkcí běžícího procesu.

## 6. Programátor

### 6.1 METODIKA PROGRAMOVÁNÍ

Píšeme-li program a má-li k něčemu být užitečný, pracuje se vstupními daty, která svou činností mění na data výstupní. Vstup a výstup programu zajišťuje jádro. Programátor využívá jádro pomocí volání jádra. Volání jádra je proto nutno znát a správně využívat, protože tato volání charakterizují operační systém UNIX.

Programátor ale není omezován pouze na tuto úroveň styku se systémem. Pomineme-li rozsáhlé knihovny funkcí, UNIX obsahuje velké množství (200–300) již hotových programů pro práci s daty, kterým říkáme nástroje programátora (tools), zkráceně nástroje. Uživatel je spouští zadáním (vnějšího) příkazu pro svůj shell. V odpovídajícím dílu referenční příručky (1) nalezne vyčerpávající popis. Znalostí všech nástrojů programátor šetří čas, protože jen málo textových manipulací s daty doposud nebylo naprogramováno. Nástroj je naprogramován tak, aby plnil jednu funkci, ale bezchybně a bez výjimek. K různým modifikacím základní funkce lze použít volby.

Shell není jen prostředkem komunikace uživatele se systémem. Bourne shell umožňuje pomocí vnitřních příkazů seřadit nástroje do sekvencí a podle výsledků jejich práce řídit zpracování. Znamená to, že nejde jen o komfort práce u terminálu, ale také o programovací možnosti na úrovni práce procesů. Shell testuje ukončení práce svého synovského procesu podle jeho návratového kódu, syn svůj návratový kód stanovuje v parametru volání jádra `exit(2)` a vyžaduje-li to programátor, tento návratový kód může ovlivnit zpracování sekvence nástrojů. Píšeme-li např.

```
find /usr/tmp -name $$tmp
```

při interaktivní komunikaci `find(1)` nevypisuje žádnou zprávu o nalezení souboru se jménem `$$tmp` (`$$` je substituováno za PID shellu). `find(1)` ale úspěch nebo neúspěch ohlásí v návratovém kódu a my můžeme např. psát

```
if find /usr/tmp -name $$tmp
then
  cd /usr/tmp
  ...
fi
```

Řazení nástrojů do sekvencí zpracovaných shellem je důvod, proč nástroj v případě správné práce standardně nevypisuje žádnou zprávu (pracuje mlčky). Pro komentování neočekávaného vývoje práce používá nástroj standardní chybový výstup (kanál č. 2). Standardní výstup je totiž v případě programování nejčastěji přepínán na soubor (... > `jméno_souboru`), na vstup roury (`PRODUCENT | KONZUMENT`), nebo je vkládán do obsahu proměnné (`PROM=`nástroj``).

## 6.1 Programátor

Pomocí přesměrování můžeme řešit i problém zařazení obrazovkově orientovaného nástroje do scénáře. Např. můžeme pomocí editoru `vi (1)` měnit první výskyt textu `Petr` v souboru `clanek` na text `Petr Nevecny`, z klávesnice postupně zadáváme (viz Příloha B):

```
vi clanek
/Petr
wiNevecny<Esc>ZZ
```

Neinteraktivně lze `vi (1)` za tímto účelem využít v řádkové komunikaci jeho části `ex (1)` takto (viz Příloha B):

```
ex clanek < PetrN 2&>1 >/dev/null
```

kde soubor `PetrN` bude obsahovat sekvenci znaků pro odpovídající textovou záměnu, oba výstupní kanály přepínáme do prázdného zařízení. Takový příkazový řádek lze používat ve scénáři. Uvedená změna chování nástroje je jedním z důvodů, proč jsou obrazovkově orientované nástroje ovládány klávesami s písmeny. Nejčastější je případ alternace řídicích kláves obrazovky (šipky, funkční klávesy atd.) alfabetskými klávesami. Řízení obrazovky se totiž pro různé typy terminálu liší (viz čl. 7.5). Ošidné místo v uvedeném příkladu (ale i obecně) je v použití klávesy `<Esc>`. V obrazovkových aplikacích se používá pro návrat zpět. Mnohé terminály ale používají `<Esc>` jako lokální řídicí klávesu (např. terminál VT100 má klávesu šipka vpravo nahrazenou sekvencí znaků `<Esc>[C`. Aplikace odlišují časovou prodlevu použití `<Esc>` samostatné od použití v řídicí sekvenci. Stiskneme-li `<Esc>`, následující klávesu stiskneme o něco později oproti terminálu, který celou sekvenci vyšle najednou přenosovou rychlostí. Soubor `PetrN` z příkladu by proto musel obsahovat prodlevu mezi znakem `<Esc>` a `Z`, což lze obtížně zajistit, proto nástroje alternují i klávesu `<Esc>`; pro `vi (1)` musíme použít jeho řádkový režim, editor `ex (1)`, takto

```
/Petr/s/Petr/Petr Nevecny/
x
```

Rozhodne-li se programátor pro vytvoření nového nástroje, může jej naprogramovat jako jeden proces (např. v jazyce C) nebo jako scénář pro shell. V obou případech by ale měl pamatovat, že:

- příkazový řádek pro spuštění nástroje musí být ošetřen tak, aby mohl nástroj pracovat na vstupu buď s libovolným počtem souborů, které jsou dány argumenty nástroje nebo (bez uvedení jmen těchto souborů) pracovat se standardním vstupem; nástroj lze potom používat v koloně jako filtr dat
- jedná-li se o obrazovkově orientovanou aplikaci, ovládání by mělo být dvojí, pro pohodlnou interakci uživatele a pro neinteraktivní práci ve scénáři, plněním funkce modulu

- nástroj musí pracovat s návratovým kódem (ve scénářích lze použít vnitřní příkaz shellu `exit`)
- důležité je prostředí provádění nástroje; z množiny proměnných shellu lze stanovit ty, jejichž jméno i obsah bude viditelný v synovských procesech; množina takto exportovaných proměnných do jiného procesu tvoří jeho prostředí provádění; v shellu exportujeme proměnnou vnitřním příkazem `export`, na úrovni volání jádra používáme variantu `execve(2)` (viz Příloha C), kde v parametrech uvedeme seznam textových řetězců ve tvaru `PROM=obsah`; v synovském procesu (nástroji) je prostředí provádění čitelné z obsahu proměnné `envp` funkce `main()` (viz čl. 4.8)

Při programování je nutné mít neustále na paměti hierarchickou strukturu procesů od místa spuštění řídicího procesu (shellu). Synovské procesy, ať už v přímé interakci nebo při provádění scénáře, spolupracují pomocí souborů, roury, proměnných atd. Mohou využívat společnou paměť nebo frontu zpráv (IPC). Na úrovni shellu mohou být synchronizovány posíláním signálů příkazy `kill(1)`, `trap` nebo vzájemným čekáním na syny příkazem `wait`. Realizace aplikace pomocí několika spolupracujících procesů je pro UNIX pravidlem, nikoliv výjimkou.

Bourne shell nepředstavuje jedinou možnost, jak lze interpretovat scénář. Programování blížící se notaci jazyka C zahrnuje C-shell. Bohužel neumí interpretovat také scénáře pro Bourne shell, což se mu v současné době stává osudným. Naopak úspěch některých nedávno vzniklých shellů spočívá právě v programové kompatibilitě na původní Bourne shell. Takovým je např. KornShell.

## 6.2 VÝVOJ PROGRAMU

Programovací jazyk C je mateřským mlékem operačního systému UNIX. Přestože mu v této knize nevěnujeme přímou pozornost, jeho ovládání je pro práci v UNIXu nezbytné. Překlad zdrojového textu v jazyce C uloženého v souboru `prog.c` spouští příkazový řádek

```
$ cc prog.c
```

`cc(1)` je řídicí program, který postupně startuje jednotlivé části překladače jako samostatné (synovské) procesy. Těmito částmi jsou textový makroprocesor, jednotlivé průchody překladače a sestavující program. Textový makroprocesor, který je odděleně přístupný příkazem `cpp(1)`, zpracuje příkazy začínající znakem `#` v prvním sloupci řádku (`#include`, `#define`, `#pragma`, ...) a vyřeší tak vsunutí hlavičkových souborů s příponou `.h` (příkaz `#include`), rozvinutí maker a náhradu textů (`#define`), oddělí části zdrojového textu, které nejsou určeny k tomuto překladu (`#ifdef`) atd. Vlastní překladač pracuje v několika průchodech. Po vytvoření přeloženého modulu následuje etapa sestavení. Řídicí program `cc(1)` fázi sestavení realizuje příkazem `ld(1)`. Sestavující program `ld(1)` je možné spouštět nezávisle na `cc(1)`. Výsledkem překladu v podobě uvedeného příkazového řádku je proveditelný program uložený v souboru `a.out` v pracovním adresáři. Jiné jméno souboru s proveditelným programem získáme volbou `-o`:

## 6.2 Programátor

```
$ cc -o prog prog.c
```

Proveditelný program je uložen v souboru `prog` pracovního adresáře. V případě, že budeme sestavovat několik modulů, překlad jednoho z modulů bez spuštění `ld(1)` provedeme pomocí `-c`:

```
$ cc -c modul.c
```

kdy v pracovním adresáři vznikne soubor `modul.o` (`object`). Překlad ale můžeme zastavit také i v některých jiných fázích, lze také:

- získat zdrojový text programu za fází textového makroprocesoru (v souboru s příponou `.i`):

```
$ cc -P prog.c
```

- získat odpovídající zdrojový text programu v assembleru (přípona `.s`)

```
$ cc -S prog.c
```

Text v assembleru (lze jej překládat pomocí `as(1)`) je důležitý, nemáme-li v systému prostředky pro ladění programů na úrovni jazyka C (např. `sdb(1)`). Musíme se pak spokojit se standardním ladícím programem `adb(1)`, který pracuje na úrovni assembleru.

Další užitečné volby se vztahují k textovému makroprocesoru. Použitím

```
$ cc -DSYMBOL prog.c
```

definujeme ve fázi překladu textovou konstantu `SYMBOL` stejně jako by byl použit příkaz

```
#define SYMBOL
```

ve zdrojovém textu souboru `prog.c` a

```
$ cc -I/usr/local/include prog.c
```

definuje další adresář, ve kterém má makroprocesor vyhledávat hlavičkové soubory při zpracování příkazu

```
#include <hlavicky.h>
```

v textu `prog.c`. Volba dává přednost uvedenému adresáři před standardním (`/usr/include`). Vícekrát použitá volba `-I` znamená postupné hledání v adresářích definovaných zleva doprava v příkazovém řádku.

Sestavující program `ld(1)`

```
ld [volby] soubor ...
```

má definovanu množinu voleb disjunktů s množinou voleb `cc(1)`, protože v příkazovém řádku `cc(1)` musí být možné zadávat volby týkající se sestavení. Např. uvedená volba `-o` u `cc(1)` je volba `ld(1)`. Důležitá je volba `-l`, pomocí níž stanovujeme knihovnu modulů, ve které budou hledány externí symboly nepokryté samotným sestavovaným programem. Textový řetězec následující za `-l` je určení knihovny, např.

```
$ ld *.o -lm
```

je sestavení všech modulů pracovního adresáře v souborech končících na `.o`; program může používat odkazy na funkce z knihovny v souboru `/lib/Llibm.a`; proveditelný program bude v souboru `a.out` pracovního adresáře, `m` je část jména `/lib/Llibm.a`, která se používá v příkazovém řádku. Definice adresáře `/lib` a `/usr/lib` a část jména `Llib` je dána implementací `ld(1)` a může se v různých systémech lišit. Nahlédneme-li do adresáře `/lib`, vidíme, že knihoven s odpovídajícím jménem je v adresáři více, liší se v prvním znaku jména souboru (`Llib`, `Slib`, `Mlib`, ...). Obsah knihovny je závislý na použitém paměťovém modelu programu, které bývají v UNIXu definovány nejméně čtyři a jsou označovány jako Small, Middle, Large a Huge. Jeden z nich je při použití `ld(1)` implicitní, programátor může volbou (obvykle `M`) model programu explicitně stanovit. Vzhledem k tomu, že je tato část úzce závislá na typu procesoru, není definována v SVID a čtenář by se měl orientovat v dokumentaci konkrétní instalace.

Knihovnu modulů, kterou chce používat ve svých programech, může programátor vytvořit a udržovat příkazem `ar(1)`. Knihovnu vytváříme např.:

```
$ ar -q libloc funcf.o funcg.o
```

kde předpokládáme existenci modulů `funcf.o` a `funcg.o` v pracovním adresáři (vznikly pomocí `cc(1)` s volbou `-c`). Pokud soubor se jménem `libloc` v pracovním adresáři neexistuje, je nově vytvořen a jeho obsahem je knihovna s uvedenými moduly. Pokud existuje a má formát knihovny, jsou moduly do knihovny připojeny. Obsah knihovny můžeme prověřit volbou `-t` (`table`) a `-v` (`verbose`):

```
$ ar -tv libloc
rw-r--r--    55/50      264      Jan      20 10:19 1993 funcf.o
rw-r--r--    55/50      264      Jan      20 09:50 1993 funcg.o
```

Výpis je podobný jako u příkazu `tar(1)` s volbami `tv` (viz čl. 2.3). Na místě vlastníka a skupiny modulu je uveden jejich číselný ekvivalent (podle `/etc/passwd`). Připojit další modul ke knihovně lze opět volbou `-q` uvedeným způsobem. Vyjmout modul z knihovny (zrušit jeho existenci v knihovně) můžeme volbou `-d`

```
$ ar -d libloc funcf.o
```

## 6.2 Programátor

zatímco modul z knihovny okopírovat do souboru a v knihovně jej v ponechat můžeme příkazem

```
$ ar -x libloc
```

kdy bez uvedení jména modulu bude kopírován celý obsah knihovny do jednotlivých souborů se jmény shodnými s moduly. Vyměnit modul za jiný lze volbou `-r`. I pomocí něj lze vytvořit knihovnu modulů, jestliže dosud neexistovala. Celý formát `ar(1)` je

```
ar určující_volby [doplňující_volby] knihovna [modul ...]
```

a *určující\_volby* je typ manipulace s knihovnou (`qtX`) a volitelné *doplňující\_volby* se vztahují ke způsobu provedené manipulace (`v`).

Pro vhodnou organizaci vytvářené knihovny slouží `lorder(1)`. Jeho vstupem je seznam přeložených modulů a výstupem dvojice charakterizující umístění jmen funkcí v těchto modulech. Výstup `lorder(1)` lze setřídít nástrojem `tsort(1)` a předložit ke zpracování `ar(1)`. Vytvoření knihovny modulů, která je optimální pro prohledávání `ld(1)`, je např.:

```
$ ar -cr libloc `lorder *.c | tsort`
```

Přeložený modul můžeme z hlediska použitých jmen symbolů prohlížet příkazem `nm(1)`, např.

```
$ nm funcf.o
```

ale také

```
$ nm a.out
```

`nm(1)` vypisuje jména symbolů podle jejich tabulky na konci souboru s modulem. Protože je v okamžiku odladěného programu zbytečná, je možné ji odstranit příkazem `strip(1)`

```
$ strip funcf.o a.out
```

což lze zajistit i volbou `-s` sestavujícího programu `ld(1)`. Dalším informativním nástrojem při vývoji programu je `size(1)`, pomocí něhož můžeme zjistit velikost jednotlivých segmentů modulu (textového segmentu, datového segmentu a zásobníku). Použití je jednoduché:

```
size modul ...
```

a na standardní výstup dostáváme velikost segmentů.

Sestavený proveditelný program bývá obecně uložen v souboru se jménem `a.out`. `a.out(4)` je popis struktury tohoto souboru. Může být odlišný vzhledem k danému typu



procesoru a zpracování v centrální jednotce počítače. Vždy ale obsahuje základní hlavičku, v jejímž magickém čísle (magic number) je dán paměťový model a způsob provádění následujícího kódu. V poslední době se často hovoří o binární přenositelnosti programů. Mnozí světoví výrobci UNIXu garantují přenositelnost binárního kódu programů a knihoven vzájemně na úrovni různých typů UNIXu, např. je dána možnost přenosu mezi instalacemi na PC, mezi SCO UNIX a Interactive UNIX, což je ovšem příklad stejné centrální jednotky. Definice binární přenositelnosti prozatím SVID nezahrnuje.

Při překladu překladačem `cc(1)` programátor brzy zjistí, že nemá dostatek informací o syntaktických chybách v programech. Otázka sémantického rozboru je při překladu vynechána úplně. Analýzu programu psaného v jazyce C provádí program `lint(1)`. Na standardní výstup vypisuje diagnostiku syntaktické a zčásti i sémantické kontroly, důležitá je také kontrola na přenositelnost. Formát je jednoduchý:

```
lint [volby] soubor ...
```

Program `cxref(1)` produkuje seznam symbolů a modulů s odkazem na soubory se zdrojovými texty, ve kterých byly použity včetně uvedení jejich řádků. Formát je

```
cxref [volby] soubory
```

kde *soubory* je kolekce jmen souborů, ve kterých jsou uloženy zdrojové texty jednotlivých modulů kontrolované aplikace.

Analýzu programu na úrovni grafu toku řízení provádí prostředek `cflow(1)`. Vychází ze zdrojových textů v jazyce C, lexikálního analyzátoru `lex(1)`, konstrukturu gramatik umělých jazyků `yacc(1)`, assembleru a z přeložených modulů (z hlediska tabulky symbolů). Výsledkem je výpis grafu toku řízení a odpovídající analýza.

Fáze dynamického ladění programů je podporována programy `sdb(1)` a `adb(1)`. Ladění programů psaných ve vyšším programovacím jazyce (jako je např. C) je pomocí `sdb(1)` (symbolic debugger). Pracuje s proveditelnou podobou programu a případně se souborem obsahujícím obraz paměti v době havárie programu se jménem `core`. `core` vzniká v pracovním adresáři procesu, kterému byl jádrem zaslán signál určitého typu. Zpracování tohoto signálu jádrem znamená ukončení procesu a v souboru `core` vytvoření jeho obrazu paměti (viz tabulka signálů v čl. 5.2). Chceme-li plnohodnotně využívat možnosti `sdb(1)`, musíme překládat laděný program s volbou `-g`:

```
$ cc -g prog.c
```

což znamená vytvoření proveditelného kódu v souboru `a.out` s připojením dalších informací o kódu (viz [11]). `sdb(1)` má příkazový řádek tvaru

```
sdb [a.out [core [adresáře]]]
```

Ladíme program v souboru `a.out`, k němuž přísluší obraz paměti v souboru `core`. *adresáře* je seznam adresářů, ve kterých jsou uloženy zdrojové texty programu. Píšeme-li

\$ `sdb`

ladíme `a.out` `s core` (je-li v pracovním adresáři) a zdrojové texty jsou hledány v pracovním adresáři procesu `sdb`. V době práce `sdb(1)` můžeme program `a.out` spustit pod naším řízením. Znamená to, že lze nastavit body zastavení běhu programu a v těchto bodech prohlížet obsahy proměnných, zásobníku, obsahy strojových registrů, měnit obsahy datových struktur a pokračovat k dalšímu nastavenému bodu zastavení. Ukončení práce `sdb(1)` dosáhneme vnitřním příkazem `q`.

Adekvátní ladící prostředek na úrovni assembleru je `adb(1)`, který vznikl s první verzí systému UNIX (`sdb(1)` byl vytvořen v rámci skupiny BSD). Přestože SVID3 definuje `sdb(1)`, `adb(1)` v ní už není zahrnut. `adb(1)` je ale přesto obsažen prakticky v každé instalaci systému, jeho příkazový řádek má tvar

```
adb [a.out [core]
```

`adb(1)` přitom nepoužívá zdrojový text, prohlížení programu je dáno požadavky uživatele, který stanoví, jakým způsobem požaduje strojový kód zobrazit, zda jako instrukci assembleru, nebo jako data v určeném formátu. `adb(1)` se velmi často používal jako editor binárních souborů. Vnitřní příkaz `l` (nebo `L`) totiž umí vyhledat určenou slabiku (nebo slovo) a vnitřní příkaz `w` (nebo `W` pro slovo) přepíše slabiku novým obsahem. `adb(1)` končí svoji činnost příkazem `$q`.

Ve čl. 5.6 jsme uvedli volání jádra `ptrace(2)`, které je využíváno při dynamickém ladění programů prostředky `adb(1)` a `sdb(1)`. Jde vlastně o možnost řízení synovského procesu pro statické ladění programů. Další tamtéž uvedené volání jádra `profil(2)` je využíváno funkcí `monitor(3)`:

```
#include <mon.h>
```

```
void monitor(int(*lowpc)(), int (*highpc)(), WORD *buffer,  
            int bufsize, int nfunc);
```

a umožňuje snadnější sledování chodu programu. Výsledkem programu `a.out`, který `monitor(3)` využívá, je také soubor `mon.out` se statistickými údaji o chodu `a.out`. `mon.out` je ve formě tabulky zobrazitelný pomocí `prof(1)`.

Příkaz `time(1)` s formátem

```
time příkazový_řádek
```

provede *příkazový\_řádek* jako nový proces a po jeho ukončení vypíše ve vteřinách jeho čas strávený v systému, a to jednak v uživatelské a jednak v supervizorové fázi.

Sledovat průběh volání jádra programu umožňuje program `truss(1)`. Na rozdíl od předchozích ladících prostředků sleduje vztah proces – jádro a vydává o tom statistiku.

Několikaletý vývoj grafického rozhraní X-WINDOW SYSTEM pro UNIX (viz. čl. 7.6) přináší v poslední době konečně své plody v podobě integrovaného prostředí překladačů jazyka C, např. pro operační systém HP-UX. Jednotný přístup uživatele k takovým aplikacím zatím ale není jednoznačně stanoven, zvláště pro možnost implementace prakticky libovolného grafického systému do X-WINDOW. Výrobci UNIXu, navíc poučení z neúspěchu pokusů o standard vizuálního obrazovkového shellu, se standardem nespěchají. Také z tohoto důvodu SVID3 o grafických prostředcích programátora prozatím mlčí a nedoporučuje žádnou orientaci do budoucna.

## 6.3 NÁSTROJE

UNIX nezachovává verze obsahu souborů. Přesměrování, editace nebo jiná manipulace vedoucí ke změně obsahu souboru znamená ztrátu původního obsahu. Proto pro úschovu a údržbu různých verzí zdrojových textů programů vznikl podsystém SCCS (Source Code Control System). Vytváří jej několik samostatných programů, které dále uvádíme. Základní myšlenkou SCCS je vytvoření archivu zdrojových textů, které pak rozlišujeme číslem jejich verze. Databázi zdrojových textů říkáme soubor SCCS. Tato databáze je uložena v souboru začínajícím povinně na `s.`. Vytvořit soubor SCCS lze příkazem `admin(1)`, např.

```
$ admin -n -i prog.c s.prog.c
```

vytváří databanku v souboru `s.prog.c`, výchozí je zdrojový text `prog.c` souboru a výchozí verze má implicitní označení 1.1. `s.prog.c` se v další práci vztahuje k verzím programu `prog.c`. Takovým položkám v souboru SCCS říkáme delta a pro vytváření dalších verzí zdrojového textu slouží příkaz `delta(1)`. Např.

```
$ delta -r1.2 s.prog.c
```

vytvoří novou verzi (1.2) v souboru SCCS programu `prog.c`. Kopírovat verzi zdrojového textu z databanky do souboru lze pomocí `get(1)`,

```
$ get -r1.1 s.prog.c
```

Program `prc(1)` slouží k získání informací o dané verzi z souboru SCCS, `val(1)` je kontrola správné struktury souboru SCCS a `what(1)` slouží k výpisu identifikace souborů končící na `.c`, `.a` a souboru `a.out` v souvislosti s verzí z databanky souboru SCCS. Zbývají programy `rmcl(1)` pro zrušení verze z databanky, `unget(1)` pro návrat od posledního `get(1)` a `sact(1)` pro výpis stavu editovaných verzí odpovídajícího souboru SCCS. Jiné programy podsystému SCCS SVID3 nezahrnuje. Dříve používané příkazy se staly součástí některého z uvedených příkazů, příkaz `help(1)` byl pro svůj obecný název vynechán a popis SCCS příkazů je dostupný pomocí standardní práce s referenční příručkou `man(1)` (viz kap. 1).

Dalším velmi důležitým nástrojem programátora je `make(1)`. Princip vychází z myšlenky, která napadla každého programátora, který napsal program o několika modulech. Programátor definuje programovaný projekt v souboru `makefile` (nebo `Makefile`) v adresáři, kterým

### 6.3 Programátor

začíná podstrom zdrojových textů projektu. Struktura popisu určuje, v jakém pořadí jsou zdrojové texty jeden na druhém závislé. Např. při změně hlavičkového souboru (`.h`) je nutné znovu přeložit všechny moduly zdrojových textů, ve kterých jsou definice v hlavičkovém souboru použity, ale žádné jiné. Část souboru `makefile`, která popisuje závislost, může např. být

```
prog.o : def.h okna.h prog.c
        cc -c prog.c
```

kde text `prog.o` následován znakem `:` je návěští. Na řádku pak pokračuje seznam závislostí, v tomto případě jmen souborů, na kterých závisí provádění následujících řádků. Následující řádky obsahují příkazy pro shell, které bude `make(1)` interpretovat, bude-li závislost návěští splněna. Každý řádek s příkazem musí začínat znakem tabulátoru. Uvedený příklad ukazuje popis překladač modulu `prog.o`, a to tehdy, jestliže soubor `prog.o` neexistuje, nebo byl-li alespoň jeden ze souborů `def.h`, `okna.h`, `prog.c` modifikován dříve než `prog.o`. V sezení můžeme psát

```
$ make prog.o
```

a `make(1)` prověří podle uvedeného popisu závislosti, existence a čas změny obsahu souborů a podle toho buď následující překlad provede, nebo považuje situaci za odpovídající popisu.

V seznamu závislostí nemusí být uvedena pouze jména souborů, ale může zde být uvedeno některé další návěští v souboru `makefile`. Např.

```
prog.o : def.h okna.h prog.c
        cc -c prog.c
projekt: projekt.c prog.o
        cc -c projekt.c
        cc -o projekt projekt.o prog.o
```

je text souboru `makefile` rozšířen o návěští `projekt`. Na základě

```
$ make projekt
```

pak proběhne překlad a sestavení do souboru `projekt` za předpokladu, že soubor `projekt` zatím neexistuje, nebo je starší než `projekt.c`, a nebo bylo zapotřebí nově vytvořit soubor `prog.o` po prozkoumání závislostí odpovídající návěští `prog.o`.

Při popisu, vlastně programování skladby modulů projektu, můžeme také využívat textové proměnné, které definujeme způsobem

```
PROM=obsah
```

v souboru `makefile`, např.

```
CCOPT="-O"
```

a obsah substituujeme v příkazu souboru `makefile` např.

```
cc $ (CCOPT) -o prog.c
```

Pomocí `make(1)` můžeme pracovat i na velmi rozsáhlých programových systémech, které se skládají z několika různých programů. V části příkazů se nemusí popis omezovat pouze na využívání překladačů nebo sestavujícího programu, ale může zde být využit jakýkoliv nástroj programátora; např. pro kopii nebo rušení (instalace vytvářeného programového podsystemu) apod. lze `make(1)` využívat i při údržbě datové základny (viz kap. 9).

Nástrojů, které s programováním bezprostředně souvisejí, je v UNIXu ještě celá řada. Nástroj `m4(1)` je textový makroprocesor vybavený lépe než makroprocesor používaný překladačem jazyka C. Při vytváření popisu umělého jazyka je velmi vhodný `yacc(1)`, který generuje syntaktický analyzátor vytvářeného jazyka ve tvaru zdrojového textu jazyka C a `lex(1)`, program, který dokáže provádět lexikální analýzu zdrojového textu.

Další nástroje používá programátor jako pracovní moduly své aplikace. S textovými soubory pracují `diff(1)`, `cut(1)`, `paste(1)`, `grep(1)` atd. Složitější operace při práci s texty můžeme naprogramovat v `awk(1)`. Úpravu dokumentace zahrnuje textový procesor `nroff(1)`. `nroff(1)` je nejen vlastní formátovací program, ale také označení celé sady nástrojů pro práci s dokumentací, např. `tbl(1)` pro práci s tabulkami, `eqn(1)` a `neqn(1)` při vytváření matematických vzorců atd. Seznam nástrojů podle SVID3 a jejich formáty použití uvádím v Příloze E.

# 7. Terminál

## 7.1 ZMĚNA ZPŮSOBU KOMUNIKACE

Uživatel v rámci svého sezení

\$

nebo

#

komunikuje s operačním systémem prostřednictvím příkazového řádku. Znamená to, že zapsaný text je předán komunikujícímu procesu teprve po stisknutí klávesy Enter. Uživatel může změnit způsob komunikace pomocí příkazu `stty(1)`, např.:

```
$ stty cbreak
```

což znamená, že systém bude přenášet vstupní data z terminálu nikoli po řádcích, ale po znacích, jakmile budou na klávesnici zapsány. Tato změna je výhodná pro ovládání vstupu procesů stiskem jedné klávesy, znemožňuje ale právě zapsaný znak zrušit a nahradit jej jiným, což v řádkové komunikaci při omylech v textu potřebujeme. Návrat z přímého režimu zpět do řádkového provedeme

```
$ stty -cbreak
```

Pomocí `stty(1)`, jehož formát je

```
stty [-a] [-g] [volby]
```

můžeme v rámci parametru *volby* nastavovat i jiné charakteristiky komunikace terminálu se systémem. Např. protokol XON-XOFF, což znamená, že na základě definované klávesy Ctrl-s (\023 podle ASCII viz Příloha A) pozastavíme příliš dlouhý výpis na obrazovce terminálu a klávesou Ctrl-q (\021 podle ASCII tamtéž) pokračování výpisu obnovíme. Komunikační protokol XON-XOFF je pro uživatele implicitně nastaven, ale uživatel jej může pomocí

```
$ stty -ixon
```

vypnout, a případně pomocí

```
$ stty ixon
```

opět nastavit.



## 7.2 Terminál

(echo jej znovu zapíná), což můžeme využít při vstupu textu zpracovaného scénářem shellu, který nemá být viditelný:

```
stty -echo
read vstup
stty echo
```

protože ovladač zajišťuje opis znaků z klávesnice na obrazovku. Můžeme také stanovit změnu přenosové rychlosti sériového rozhraní

```
$ stty 2400
```

na 2400 bps (bitů/vteřinu), což asi stěží použijeme v běžné uživatelské komunikaci, a podobně také změnu parity přenosu na sudou, lichou nebo změnu počtu stopbitů. Tyto atributy jsou ale nastavovány v době přihlašování uživatele a jsou předmětem následujícího článku.

### 7.2 NASTAVENÍ VÝCHOZÍCH CHARAKTERISTIK

Uvážíme-li nejpoužívanější způsob připojení terminálu k počítači, jedná se o sériový plně duplexní přenos. Správce operačního systému předpokládá správnost připojení na technické úrovni (bývá standardu RS 232 C), za kterou odpovídá správce technického zařízení. Operační systém nabízí procesům komunikaci s terminálovou linkou pomocí odpovídajících speciálních souborů. Jak bylo uvedeno v kap. 2 a 3, tyto soubory jsou umístěny v adresáři `/dev` a jejich jména začínají textem `tty`, případně mohou být v nových verzích UNIX SYSTEM V umístěny v podadresáři `/dev/term`. Např.

```
$ ls -l /dev/term/ttyAA /dev/term/ttyaa
crw--w--w- 1 root root 15,128 Dec 18 13:37 /dev/term/ttyAA
crw--w--w- 1 root jan 15,0 Jan 7 16:15 /dev/term/ttyaa
```

Speciální soubory terminálů jsou pouze znakové (c). Je zvykem používat pro jednu periferii dva speciální soubory s odlišným vedlejším číslem, hodnota nad 128 včetně přitom charakterizuje způsob připojení pomocí modemu. Přístupová práva a vlastnictví těchto speciálních souborů odpovídají dynamicky se měnícímu uživateli, který je právě na terminál přihlášen. Každý uživatel má pro snazší práci se svým terminálem k dispozici také virtuální speciální soubor

```
crw--w--w- 1 root jan 15,0 Jan 7 16:15 /dev/term/tty
```

který na logické úrovni odpovídá souboru konkrétní linky. Uživatel pak může ve svých programech nebo v rámci svého sezení používat odkaz na soubor `/dev/term/tty`. Ve skutečnosti je odkaz jádrem v konkrétní situaci převeden na speciální soubor odpovídající aktuálnímu sezení.



Proces **init** je otcem všech procesů **getty**, které čekají na terminálech na přihlášení uživatele. Každý proces **getty** je jednoznačně spojen s konkrétním speciálním souborem. Spojení vytvoří proces **init** podle tabulky `/etc/inittab`. Obsah tabulky `inittab` a práci procesu **init** s ní popíšeme podrobně v kap. 9. Nyní je pro nás důležitá část tabulky obsahující příkazové řádky pro spuštění procesu **getty**, např.

```
ttyaa:2:respawn:/etc/getty /dev/term/ttyaa 9600
```

Příkazovému řádku pro **getty** předchází několik textů, které oddělují znaky : . Jedná se po řadě o označení řádku tabulky ( `ttyaa` ), označení, ve kterém režimu má být **getty** spuštěn (2 je víceuživatelský režim), a způsob práce procesu **init** s procesem **getty** (pomocí `respawn` vyžadujeme, aby při uvolnění linky byl **getty** nově startován). Příkazový řádek **getty** má 2 parametry, kde první z nich je speciální soubor terminálu, se kterým bude **getty** spojen. Z toho je patrné, že počet řádků tabulky se startem **getty** odpovídá počtu připojovaných terminálů. Řádek ale nemusí být z tabulky `/etc/inittab` vymazán, je-li právě odpovídající terminál např. v poruše, stačí pouze přepsat text `respawn` na `off`, **init** pak řádek při práci vynechá. V případě textové náhrady editorem zůstane pro chod systému stav a počet připojených terminálů nezměněn. Pro aktualizaci skutečnosti podle tabulky `/etc/inittab` je nutno psát

```
# init Q
```

(**init** tabulku sám opakovaně neprohlíží).

Druhým parametrem příkazového řádku **getty** je určení základních charakteristik terminálu při vstupu uživatele do systému. Náš příklad uvádí text `9600`, který mnemonicky napovídá, že se jedná o připojení související s přenosovou rychlostí `9600 bps`. `9600` ale znamená i další nastavení. Text `9600` je reference do tabulky `/etc/gettydefs`. Tabulka obsahuje vždy několik různých způsobů připojení terminálu. Jsou navzájem odděleny prázdným řádkem, každý má přitom tvar

```
návěští#úvodní_nastavení#konečné_nastavení#  
přihlašovací_řetězec#následující_návěští
```

např.

```
9600 # B9600 HUPCL # B9600 CS8 SANE HUPCL TAB3 ECHOE IXANY #  
\r\n@!login : # 9600
```

kde znak `#` je oddělovač jednotlivých položek. Bez zvláštního upozornění může popis způsobu pokračovat na několika řádcích. Proces **getty** pracuje podle způsobu daného položkou `návěští`, která je právě druhým parametrem při spouštění (`9600`). **getty** nastaví při přidělení odpovídajícího speciálního souboru charakteristiky sériové linky tak, jak je popsáno v položce `úvodní_nastavení`, a tyto charakteristiky jsou platné po dobu uživatelova přihlašování (zápis jména a hesla uživatele). Položka `konečné_nastavení` určuje charakteristiky sériové linky poté, co přihlášení úspěšně proběhlo a proces **getty** se proměnil (pomocí

`exec(2)`) na proces příkazového interpretu. *přihlašovací\_řetězec* je text, který **getty** vypisuje jako výzvu k přihlášení a tu opakuje vždy, stiskne-li příchozí uživatel Enter (konvenčně text končí na `login :`). Způsob nastavení výchozích charakteristik v položce *úvodní\_nastavení* nemusí odpovídat technickému stavu rozhraní RS-232 (např. je-li na periférii nastavena jiná přenosová rychlost). Pokud **getty** tyto nesrovnalosti rozpozná, pokusí se nastavit sériovou linku jiným způsobem, který je dán položkou *následující\_návěští*. Určuje jeden ze způsobů připojení téže tabulky. Je ale možné, aby položka *návěští* a *následující\_návěští* byly shodné, v případě neúspěchu není sériová linka nastavena a není umožněna komunikace se systémem.

V uvedeném příkladu je *návěští* i *následující\_návěští* označeno jako 9600 a *přihlašovací\_řetězec* bude vždy začínat vynecháním jednoho řádku (v textu je možné používat znakových konstant jazyka C) a textu `login :` bude vždy předcházet označení instalace systému pro komunikaci více počítačů v síti (@ substituuje první řádek souboru `/etc/systemid`). V uvedeném příkladu na místě položek *úvodní\_nastavení* a *konečné\_nastavení* je sekvence textových konstant, které **getty** používá pro nastavení charakteristik terminálu. Rozhraní terminálové linky je z hlediska schopností ovladače podrobně popsáno v `termio(5)`, kde můžeme najít také význam konstant používaných v tabulce `/etc/gettydefs`. Význam konstant uvedených v příkladu je následující:

B9600	stanovení přenosové rychlosti 9600 bps (analogicky např. B2400 odpovídá nastavení na 2400 bps)
CS8	je vyžadován přenos znaků využitím všech 8 bitů (analogicky CS7 ...)
HUPCL	i po uzavření speciálního souboru zůstává sériová linka evidována operačním systémem v odpovídajícím nastavení
ECHOE	nastavuje zobrazování stisku klávesy s významem zrušení znaku jako znak Backspace (návrat o znak zpět), znak mezeru a opět znak návratu o 1 znak
TAB3	tabulátory jsou nahrazeny mezerami
IXANY	určuje, zda v případě pozastavení komunikace v protokolu XON-XOFF je libovolný znak pokynem pro pokračování v přenosu dat
SANE	nastavuje všechny ostatní charakteristiky na implicitní hodnoty

Editací tabulky `/etc/gettydefs` může správce systému měnit výchozí charakteristiky uživatele terminálu při vstupu do systému a vytvářet nové způsoby připojení. Po editaci `/etc/gettydefs` se doporučuje použít příkaz

```
# getty -c /etc/gettydefs
```

kde volba `-c` (check) vyžaduje kontrolu formátu obsahu tabulky. Charakteristiky terminálu jsou v tomto případě vypisovány v konvencích výpisu příkazu `stty(1)`.

Program `stty(1)` nebo `getty(8)` využívá pro komunikaci s ovladačem terminálového rozhraní volání jádra `ioctl(2)`, které je k dispozici libovolnému uživateli v jeho programech. Podle formátu `ioctl(2)` (viz čl. 5.4) lze po otevření speciálního terminálového souboru voláním jádra `open(2)` např. použít `ioctl(2)` pro nastavení přenosové rychlosti 9600 bps s respektováním všech 8 bitů takto:

```

#include <termio.h>

int fd;
struct termio nastaveni;
...
fd=open("/dev/ttyaa", O_RDWR | O_EXCL);
/* ziskame informace o dosavadnim nastaveni */
ioctl(fd, TCGETA, &nastaveni);
/* zmena charakteristik */
nastaveni.c_cflag |= B9600 | CS8;
/* provedeni zmeny */
ioctl(fd, TSETA, &nastaveni);
...
close(fd);
...

```

Po otevření speciálního souboru `/dev/ttyaa` pro čtení i zápis (`O_RDWR`) s výhradním přístupem (`O_EXCL`) prvním `ioctl(2)` získáme informace o původním nastavení (`TCGETA`). Jádro naplní obsah struktury `nastaveni`. Požadovanou změnu provedeme změnou obsahu položky `c_cflag` za použití konstant, jejichž identifikace odpovídá popisu v souboru `/etc/gettydefs`, a použitím `ioctl(2)` tentokrát pro nastavení charakteristik (`TSETA`).

Studiem položek struktury `termio`, která je definována v souboru `/usr/include/sys/termio.h` a podrobně komentována v `termio(5)` získáme přehled o možnostech práce se sériovým rozhraním. SVID3 definuje strukturu `termio` (nebo `termios`) takto

```

tcflag_t c_iflag; /* input modes */
tcflag_t c_oflag; /* output modes */
tcflag_t c_cflag; /* control modes */
tcflag_t c_lflag; /* local modes */
cc_t c_cc[NCCS]; /* control chars */

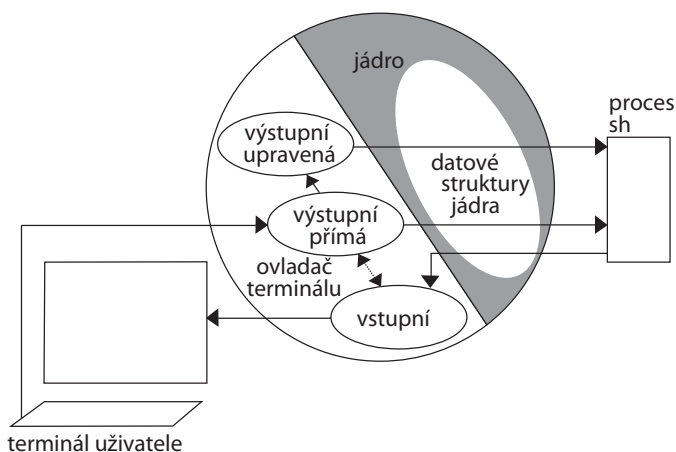
```

Vstupní charakteristiky terminálu jsou stanoveny v položce `c_iflag` (`tcflag_t` je obvykle `unsigned short`). Můžeme zde stanovit režim XON-XOFF a také můžeme stanovit způsob zpracování konce odesílaného řádku z klávesnice (použitím znaku `\n` nebo `\r` a `\n` apod.). `c_oflag` se vztahuje ke způsobu komunikace pro výpis na obrazovku. Nastavujeme např. časovou prodlevu po výpisu řádku na obrazovce nebo způsob náhrady znaku tabulátoru (`\t`) za daný počet mezer, i zde můžeme stanovit znak konce řádku jako pouhé `\n` nebo `\r` a `\n`. V příkladu použitá položka `c_cflag` stanovuje způsob komunikace, tj. např. přenosovou rychlost, počet přenášených bitů ve znaku, nastavení sudé nebo liché parity přenosu, počet stopbitů. Konečně `c_lflag` je používán pro práci místního charakteru terminálu. Je zde např. nastavován příznak opisu znaku klávesy stisknuté uživatelem na obrazovku terminálu. Definice řídicích znaků např. protokolu XON-XOFF nebo znaku přerušení z klávesnice je ve znakovém poli `c_cc`.

## 7.3 OVLADAČ

Na periférii terminál musíme nahlížet jako na znakovou periférii vstupu (klávesnice) a výstupu (obrazovka). UNIX podporuje takové striktní oddělení, připojení terminálu proto musí být plně duplexní. Z toho vyplývající možnost současného výpisu na obrazovku i čtení z klávesnice vyžaduje správu fronty znaků pro vstup i výstup. Standardní nastavení terminálu je např. opisování znaků z klávesnice na obrazovku při zápisu příkazového řádku. Ovladač proto zajišťuje přenos znaků mezi vstupní a výstupní frontou. Přenos znaků mezi datovým prostorem procesu a terminálem je obvykle po řádcích. Ukončení vstupu z klávesnice (odeslání stiskem klávesy Enter) je proto obvykle pokynem pro odeslání výstupní fronty znaků ovladače do jádra. Protože je možné (a při programování obrazovkově orientovaných programů nutné) nastavit přenos znaku okamžitě po jeho zapsání na klávesnici, ovladač udržuje jednu vstupní a dvě výstupní fronty každého terminálu podle obr. 7.2 :

Přímá výstupní fronta (raw) je vázána na výstupní frontu na úrovni popisu znaků na obrazovce. Výstupní fronta upravená (cooked) slouží k odeslání opsané přímé fronty směrem k procesu. Přímá fronta obsahuje všechny znaky zapsané uživatelem na klávesnici, zatímco v upravené už nenalezneme např. znaky, které jsou pokynem pro vymazání předchozího zapsaného znaku atd. Ve chvíli přepnutí na přímý režim pracuje proces s okamžitým stavem přímé výstupní fronty. Způsob komunikace (nastavení charakteristik přenosu dat) může proces kdykoliv změnit (pomocí `ioctl(2)`).



Obr. 7.2 Vstupní a výstupní fronty ovladače terminálu

## 7.4 NÁRODNÍ PROSTŘEDÍ

UNIX původně pracoval výhradně s anglosaskými texty. Vzhledem k tomu, že anglická abeceda a všechny znaky běžně zobrazitelné na obrazovce a generované klávesnicí terminálu jsou v tabulce ASCII umístěny v dolní části (kódované 0–128 v desítkové soustavě), obecně veškerá práce s operačním systémem UNIX předpokládala textovou informaci ve slabice v prvních sedmi bitech. Osmý bit byl velmi často používán jako poznámkový k různým účelům.

Celý problém podpory uživatele operačním systémem pro práci v národním prostředí nespočívá pouze v možnosti editace textu s diakritickými znaménky (např. úpravou editoru `vi(1)`), případně v možnosti jeho zpracování textovým procesorem (úpravou `nroff(1)`).

Ve smyslu obsahu kap. 6 musí mít uživatel zajištěn veškerý komfort práce s texty i pomocí nástrojů programátora, bez obavy ze zničení textu ignorováním významu osmého bitu. Celkově můžeme nároky na práci operačního systému v národním prostředí rozdělit do několika úrovní:

1. Fyzický terminál a ovladač terminálu pracují plně se znaky v 8 bitech.
2. Všechny nástroje programátora (editory, textové procesory, `diff(1)`, `awk(1)`, `cc(1)`, ...) pracují bez výjimky se znaky v 8 bitech.
3. Jádro všechny texty uvažuje v 8 bitech reprezentace (např. jméno souboru může obsahovat znaky národního prostředí).

Snaha přizpůsobit UNIX národnímu prostředí má už svoji historii, ale přesto dodnes málokterý výrobce garantuje podporu ve smyslu všech tří uvedených bodů. Obvyklý postup výrobců je zajistit obsluhu terminálu pro zobrazování a generaci množiny znaků příslušného prostředí. Uživatel ale většinou musí k editaci textu používat jen některé prostředky (např. editor `lyrix`), a pro práci s textovým procesorem používá před vstupem textu filtr, který převede každý znak s kódováním nad 128 na textový řetězec odpovídající makrodefinici pro `nroff(1)` (v SCO UNIX má takový filtr název `trchan`). Bohužel ani v definici umístění znaků národního prostředí v tabulce ASCII nevládne shoda. Existenci normy ISO 8859 (definující češtině odpovídající část LATIN2) ignorovala firma IBM a zavedla vlastní kódování, v jehož důsledku vznikl kód s označením PC LATIN-2 zahrnující češtinu. Z PC LATIN-2 vychází i norma ČSN 369103. Obecně lze ale současný stav v českých zemích považovat za krajně nejasný.

Úpravu operačního systému ve smyslu všech tří uvedených úrovní splňuje v současné době málokterý výrobce. Vážným kandidátem je už několik let HP-UX firmy HEWLETT PACKARD. Doufejme, že nové vydání SVID přinese v dohledné době jednotný pohled na řešení problému.

V současné době se SVID3 vyjadřuje pro úpravu pro národní prostředí ve třech termínech. Jednak zavádí výměnu textových zpráv (message handling), kdy hovoří o nástrojích programátora, pomocí nichž lze vyhledat, prohlížet a vyměňovat zprávy v programech, které jsou běžně vypisovatelné při komunikaci s uživatelem. Dále je to termín sekvence pro třídění (collating sequence), který se vztahuje k definici pořadí znaků tabulky ASCII vzhledem k jejich lexikografickému uspořádání. Konečně třetí termín je podpora rozšířené množiny znaků (wide character support). Týká se práce programátora při zpracování znaků národního prostředí, s odvoláním na zavedený datový typ `wchar_t` v definici ANSI C. UNIX SYSTEM V přitom nabízí podprogramy převodu na datový typ `char` a zpět. Všechny tři termíny SVID3 uvádí vzhledem k připravované podpoře práce v národním prostředí operačního systému SYSTEM V, která bude doplněna programy a podprogramy pro úpravu do konkrétní jazykové mutace.

## 7.5 ŘÍZENÍ OBRAZOVKY

Při vytváření prvních obrazovkově orientovaných programů pro UNIX vyvstal problém jejich provozování na různých typech terminálů. Vzhledem k tomu, že UNIX dokáže podporovat práci několika uživatelů pracujících současně na terminálech různých výrobců, musí být zaručen provoz téže aplikace v různém lokálním prostředí. Je nevýhodné a zbytečné udržovat několik různých verzí téhož programu pro jednotlivé typy terminálů.

## 7.5 Terminál

Aplikace pracující v textovém režimu a s obrazovkovými operacemi (menu, práce v několika oknech současně atd.) využívají pro styk s terminálem knihovnu funkcí CURSES. Množinu definovaných funkcí používá programátor nezávisle na řídicích znacích konkrétního terminálu. Např. funkcí `move(3)`

```
int move(int y, int x);
```

přesune program kurzor z aktuálního místa na obrazovce na místo dané souřadnicemi `y` (řádky) a `x` (sloupce). Vlastní funkce `move(3)` provádí přesun kurzoru pomocí řídicích znaků konkrétního terminálu. Knihovna CURSES má totiž k dispozici databanku s obsahem popisu řídicích funkcí různých typů terminálů. Po přihlášení uživatele je pro aplikaci podstatný obsah proměnné `TERM` příkazového interpretu. Je ukazatelem do databáze na popis terminálu, se kterým uživatel pracuje.

Základní princip práce uživatele s CURSES je v definici několika různých oken v rámci obrazovky. Okno definujeme pomocí funkce `newwin(3)`

```
WINDOW *newwin(int nlines, int ncols, int begin_y, int begin_x);
```

kde parametrem `nlines` určujeme počet řádků okna, `ncols` počet sloupců a pomocí `begin_y` a `begin_x` umístění levého horního rohu okna (`begin_y` pro řádek a `begin_x` pro sloupec). Návrátová hodnota `newwin(3)` je ukazatel na strukturu `WINDOW`, která je definována v souboru `/usr/include/curses.h` a obsahuje základní informace okna po dobu jeho existence, např. současnou pozici kurzoru v okně, rozměry okna a další. Okno rušíme funkcí

```
int delwin(WINDOW *win);
```

kde `*win` je hodnota získaná při definici pomocí `newwin(3)`. Např.

```
idwin=newwin(10, 15, 0, 0);
```

definuje okno o rozměrech 10 řádků a 15 sloupců, okno je umístěno do levého horního rohu obrazovky.

Pomocí `newwin(3)` můžeme definovat různá okna na obrazovce navzájem se různě překrývající. Při využívání CURSES musí programátor v úvodu práce použít funkci

```
WINDOW *initscr(void);
```

pro načtení informací o terminálu a inicializaci všech potřebných struktur pro práci s okny. Funkcí

```
int endwin(void);
```

naopak uzavírá práci se sadou funkcí a obnovuje předchozí režim práce s terminálem. V rámci práce s definovanými okny totiž můžeme (a potřebujeme) měnit např. režim řádkového

vstupu na přímý, nebo jinak stanovit charakteristiky terminálu. Např. pomocí sekvence funkcí

```
cbreak(); noecho();
```

vytváříme typický režim práce s okny, vstup z klávesnice je akceptován po znacích a vstupní znak přitom není zobrazován. Použitím `endwin(3)` CURSES restaurují stav charakteristik terminálu před spuštěním aplikace.

Při práci s oknem má programátor k dispozici funkce vstupu a výstupu. Použitím `initscr(3)` má bez uvedení dalšího definováno základní okno, které se velikostí shoduje s obrazovkou. Použití např. `wmove(3)`

```
wmove(idwin, y, x);
```

pro přesun pozice kurzoru na  $(y, x)$  v rámci okna `idwin` je vzhledem k definici základního okna možné psát jako

```
wmove(stdscr, y, x);
```

což je ekvivalentní standardně definovanému makru

```
move(y, x);
```

a takové vynechání určení okna je zavedeno pro všechny funkce vstupu a výstupu pracující s celou obrazovkou jako s jedním oknem (viz Příloha G).

Použijeme-li výstup textu do okna, využíváme funkce např. (obvykle je `typedef unsigned short chtype;`)

```
int waddch(WINDOW *win, chtype ch);
```

kteřá vloží na aktuální pozici okna `win` znak `ch` a funkce

```
int waddstr(WINDOW *win, chtype *chstr);
```

do okna `win` od aktuální pozice vloží řetězec znaků `chstr`. Pro formátovaný zápis do okna můžeme použít funkci

```
int wprintw(WINDOW *win, char *fmt [, arg] ...);
```

kteřá vloží text do okna `win` podle formátu `fmt` a seznamu argumentů `arg ...` (jde o ekvivalenci s funkcí `fprintf(3)` pro formátovaný výstup do určeného komunikačního kanálu). Funkce

```
int wdelch(WINDOW *win);
```

## 7.5 Terminál

vyjme z okna `win` znak z pozice kurzoru; zbytek řádku v okně je posunut o znak doleva, poslední znak řádku je nahrazen mezerou a

```
int wborder(WINDOW *win,
            chtype ls, chtype rs, chtype ts, chtype bs,
            chtype tl, chtype tr, chtype bl, chtype br);
```

vykreslí po obvodu okna `win` rámeček, ostatní parametry funkce definují znaky, které rámeček vytvářejí (left side, right side, ... top left, top right, ...), je možné použít definice `ACS_VLINE`, `ACS_HLINE`, `ACS_ULCORNER`, `ACS_URCORNER`, `ACS_BLCORNER`, `ACS_BRCORNER`, které odpovídají znakům daného typu terminálu.

Naopak pro vstup z definovaného okna funkce

```
int wgetch(WINDOW *win);
```

čte z klávesnice (standardního vstupu) znak v prostředí okna `win`. Funkce

```
int wgetstr(WINDOW *win, char *str);
```

čte v prostředí okna `win` ze standardního vstupu řetězec znaků a ukládá jej do `str` a

```
int wscanw(WINDOW *win, xchar *fmt [, arg] ...);
```

čte ze standardního vstupu text a konvertuje jej podle formátu `fmt` do obsahu proměnných `arg ...`, použití je ekvivalentní funkci `fscanf(3)`.

Manipulace s obsahem každého okna probíhá v datovém prostoru procesu. Přepis obsahu okna `win` na obrazovku provede teprve funkce

```
int wrefresh(WINDOW *win);
```

CURSES v tomto případě přepisují pouze ty části, které neodpovídají obsahu okna (promítají se pouze změny od posledního `wrefresh(3)`).

Následující příklad je jednoduchou ukázkou vytvoření dvou oken a jejich střídání:

```
#include <curses.h>

main()
{
    WINDOW win1, win2, wins, *pwin1, *pwin2, *pwin, *pwins;

    pwin1=&win1;
    pwin2=&win2;
```



```

pwins=&wins; initscr();
noecho(); cbreak();
curs_set(0); /* vypiname zobrazovani kurzoru */

pwin1=newwin(20,20,0,0);
/*vytvorene okno vybavime rameckem podle standardnich definic znaku */
wborder(pwin1, ACS_VLINE, ACS_VLINE, ACS_HLINE, ACS_HLINE,
        ACS_ULCORNER, ACS_URCORNER, ACS_BLCORNER,
        ACS_BRCORNER);
wmove(pwin1, 0, 1);
wprintw(pwin1,"Hlavni okno");

pwin2=newwin(15,30,10,10);
pwins=newwin(15,30,10,10);
/* druhe vytvorene okno vybavime rameckem; 0 jsou analogicke
predchozimu pouziti funkce */
wborder(pwin2, 0, 0, 0, 0, 0, 0, 0, 0); wmove(pwin2, 0, 1);
wprintw(pwin2,"napoveda :");

overwrite(pwin2, pwins); /* zaloha obsahu druheho okna */

/* na obrazovku vypiseme obsah prvnio okna */
overwrite(pwin1, stdscr);
refresh();
pwin=pwin1;
for (;;) {
    switch(wgetch(pwin)) {
        case KEY_ENTER : /* Stisk klavesy Enter, v pripade
                           aktualniho prvnio okna
                           vypisujeme take druhe okno */
            overwrite(pwins, stdscr);
            refresh() ;
            pwin=pwin2;
            break;
        case KEY_EXIT: /* Stisk klavesy Esc - navrat
                        k prvnimu oknu nebo ukonceni prace
                        programu */
            if(pwin==pwin2) {
                wclear(pwin2);
                overwrite(pwin2, stdscr);
                overwrite(pwin1, stdscr);
                refresh();
                overwrite(pwins, pwin2);
                pwin=pwin1;
            }
    }
}

```

## 7.5 Terminál

```
        else {
            clear();
            refresh();
            endwin();
            exit(0);
        }
        break;
default:
    beep();
    break;
    }
}
} /* konec main() */
```

V příkladu je kromě dříve uvedených funkcí použita také důležitá funkce

```
int overwrite(WINDOW *srcwin, WINDOW *dstwin);
```

která provede kopii okna `srcwin` do `dstwin` (pouze v datové oblasti procesu). Je použita pro zachování obsahu okna `win2`. V nekonečném cyklu `for(;;)` čteme znak z právě aktuálního okna a v případě stisku klávesy Enter zobrazujeme druhé okno (obsahuje text `napoveda :`), v případě aktivního druhého okna vrátíme na obrazovku obsah prvního okna (druhé okno z obrazovky smažeme) a nastavujeme aktivitu na první okno. V příkladu je také použita funkce `curs_set(3)`, kterou zde pomocí parametru `0` vypneme zobrazování kurzoru.

CURSES umí pracovat také s barevnými alfanumerickými terminály, a to pomocí funkcí např.

```
int start_color(void);
```

která inicializuje používání barev,

```
int has_colors(void);
```

testuje, zda má terminál možnost barevného prostředí,

```
int init_pair(short pair, short f, short b);
```

definuje dvojici barev pro text (foreground) a pozadí (background).

CURSES v dnešní podobě pamatují i na práci se sadou menu, panely a formuláři. Funkce a jejich použití jsou definovány SVID3 a jejich popis nalezne čtenář v dokumentaci konkrétního systému. Seznam všech základních funkcí CURSES je uveden v Příloze G.

Při sestavování programu využívajícího knihovnu CURSES musí programátor standardně použít volbu `-lcurses` pro zpřístupnění použitých funkcí v programu:

```
cc ... *.o -lcurses [-lx]
```

Knihovna funkcí CURSES se při startu programu orientuje podle proměnné `TERM` prostředí, ve kterém byl proces spuštěn (obvykle proměnná některého příkazového interpretu). `TERM` definuje typ terminálu, protože její obsah je jméno položky v databázi terminálů. V adresáři `/usr/lib/terminfo` je řada podadresářů s jednoznačným jménem. Jméno adresáře je prvním znakem textu obsahu proměnné `TERM`. Např. je-li

```
TERM=ansi
```

CURSES hledají soubor se jménem `/usr/lib/terminfo/a/ansi`. Obsah binárního souboru `ansi` je popis terminálu. Vznikl překladem zdrojového textu popisu, který je běžně uložen v souboru `/usr/lib/terminfo/terminfo.src`, pomocí programu `tic(8)`

```
# cd /usr/lib/terminfo
# tic terminfo.src
```

`tic(8)` rozpozná identifikaci popisu jednotlivých terminálů v argumentu textového souboru a ukládá efektivní popis pro CURSES do uvedených adresářů (které v případě potřeby vytváří). V souboru `terminfo.src` můžeme také najít popis pro terminál označený jako `ansi`:

```
ansi|ansic|ansi80x25|Ansi standard console,
    am, eo, xon, bce,
    cols#80, lines#25, colors#8, pairs#64,
    op=\E[37;40m,
    setf=\E[3%p1%dm,
    setb=\E[4%p1%dm,
    bel=^G, blink=\E[5m, bold=\E[1m, cbt=\E[Z,
    clear=\E[2J\E[H, cr=\r, cub1=\b, cud1=\E[B, cuf1=\E[C,
    cup=\E[%i%p1%d;%p2%dH, cuu1=\E[A,
    dch1=\E[P, dll1=\E[M, ed=\E[J, el=\E[K, home=\E[H,
    ht=\t, ich1=\E[@, ill1=\E[L, ind=\E[S, kbs=\b,
    kcub1=\E[D, kcud1=\E[B, kcufl1=\E[C, kcuu1=\E[A,
    kf0=\E[V, kf1=\E[M, kf2=\E[N, kf3=\E[O, kf4=\E[P,
    kf5=\E[Q, kf6=\E[R, kf7=\E[S, kf8=\E[T, kf9=\E[U,
    khome=\E[H, kend=\E[F, kpp=\E[I, knp=\E[G,
    ri=\E[T, rmso=\E[m,
    rmul=\E[m, sgr0=\E[10;0m, smso=\E[7m, smul=\E[4m,
    it#8, ht=^I, cbt=\E[Z,
    rmacs=\E[10m, smacs=\E[12m, acsc=k?lZm@jYnEwBvAuCt4qDx3,
    rev=\E[7m, invis=\E[8m,
```

První řádek výpisu je označení (text obsahu proměnné `TERM`), jeho ekvivalenty (odděleny znakem `|`), případně komentář. Následující, tabulátorem začínající řádky, obsahují položky popisu funkcí terminálu (capabilities). Jsou odděleny znakem `,` a jejich význam je uveden v tabulce odkazu `terminfo(4)`. V našem příkladu je klávesa Home (položka `khome`)

## 7.5 Terminál

definována pomocí tří znaků, a sice `Esc`, `[`, `H` (`\E` je textový popis klávesy `ESC`, `^` je textový popis stisku doplňující klávesy `Ctrl`), obrazovka je definována na 25 řádků (`lines#25`), 80 sloupců (`cols#80`) atd.

V některých systémech není zdrojový text popisu terminálů na disku přítomen. Správce systému jej může získat z binární podoby příkazem `untic(8)`, např.:

```
# untic /usr/lib/terminfo/a/ansi
```

kdy je textová podoba popisu zobrazena na standardní výstup. `untic(8)` není definován v SVID3.

Obsah proměnné shellu `TERM` je nastavován uživateli v době přihlašování podle obsahu tabulky `/etc/ttytype`, kde je určena korespondence mezi speciálním souborem terminálu a obsahem proměnné `TERM`, např. řádek

```
ansi ttyaa
```

definuje obsah proměnné `TERM` pro uživatele, který vstupuje do systému na terminálu speciálního souboru `/dev/ttyaa`. Vzhledem k vývoji počítačových sítí a možností emulace několika různých typů terminálů na fyzickém zařízení, má uživatel obvykle v domovském adresáři v souboru `.profile` nastaven příkaz `tset(1)` (SCO UNIX), kterým je testován obsah proměnné `TERM` a typ terminálu, který uživatel obvykle používá. Nejde-li o shodu, `tset(1)` požaduje pro typ terminálu odpověď uživatele. V případě chybně zadaného typu terminálu budou obrazovkové aplikace pracovat v prostředí uživatele nekorektně. Na řádkovou komunikaci nemá tato chyba vliv. `tset(1)` není definován v SVID3.

Dříve používaná (a SVID3 nedoporučovaná) databanka terminálů s ekvivalentním významem pro `CURSES` je `termcap`. Bývá obsahem souboru `/etc/termcap`, který je textový. Popis jednotlivých terminálů je zde velmi podobný jako v `terminfo.src`. V dokumentaci `termcap(4)` a `terminfo(4)` u systémů, které stále ještě obě varianty podporují, je dokonce stanovena relace mezi položkami popisu terminálů. SVID3 definuje program `captoinfo(8)`, který je převodním programem mezi podobou popisu terminálu v `termcap` do zdrojového textu pro `tic(8)` `terminfo`.

Prostředí uživatele může také obsahovat proměnnou `TERMINFO`, kde je explicitně stanoven adresář pro databázi `terminfo`.

Knihovna `CURSES` má nevýhodu zpracování obrazovkových operací v datové oblasti procesu a jejich přenos přes rozhraní na fyzický terminál. Při nižších rychlostech (pod 9600 bps) je komunikace viditelně pomalá. `CURSES` také nepodporují grafické operace obrazovky. Řešení zejména těchto problémů přineslo prostředí práce uživatele v operačním systému UNIX pod názvem X-WINDOW SYSTEM.

## 7.6 X-WINDOW SYSTEM

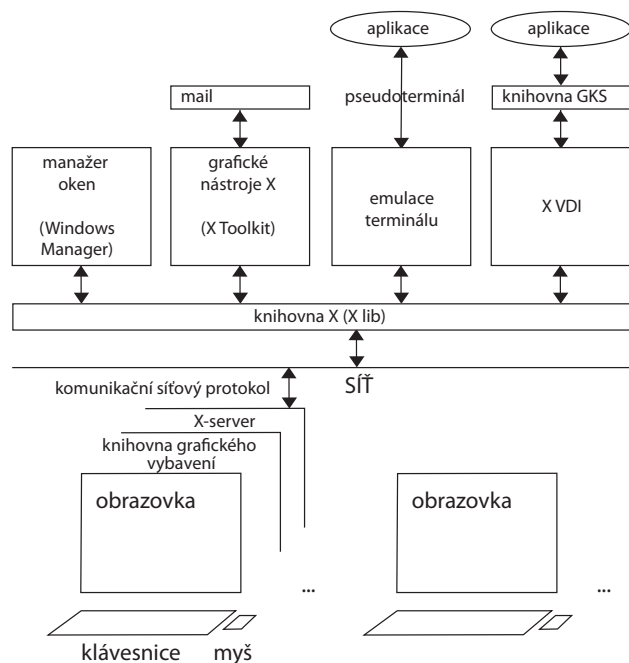
Pro grafické práce uživatele rozhraní na obrazovce terminálu využívá UNIX grafický systém X-WINDOW SYSTEM označovaný nejčastěji pouze jako X, který byl koncipován a vyvíjen v MIT (The Messachusetts Institute of Technology) od r. 1984. Na projektu se také podílela firma DEC (Digital Equipment Corporation) a později projekt začali podporovat i někteří světoví výrobci pracovních stanic, jako např. HEWLETT PACKARD nebo Sun. Rokem 1988 je datován vznik sdružení X-Consortium všech vážných zájemců o účast na vývoji X. X-Consortium dnes zahrnuje přibližně 70 různých institucí a významných světových výrobců v oblasti computer science. SVID3 obsahuje jako jeden ze svých pěti dílů definici pro práci v X. Je shodná s dokumentací X od MIT. Poslední známá podrobně dokumentovaná verze je X version 11, Release 4 z r. 1991.

Grafický systém X je síťově transparentní a není pouze doménou operačního systému UNIX. X svým obecným použitím a rozsahem z hlediska pokrytí různých grafických adaptérů představuje svět sám o sobě, jehož další vývoj je na UNIXu nezávislý. Základní myšlenka spočívá v oddělení zpracování grafických úkonů na lokálním grafickém terminálu a vlastní práce aplikace. Předpokladem pro X-terminál (lokální technické zařízení schopné pracovat s X) je možnost provozu procesu, který provádí grafické operace na základě požadavků, které přicházejí z centrálního počítače. Tomuto procesu říkáme server. Program běžící na centrálním počítači, který služeb serveru využívá, nazýváme klient. Mezi procesy server a klient bývá vložena síť. Přestože síťová podpora je poměrně rychlý přenosový aparát, oddělení dvou procesů je důsledkem snahy o minimalizaci přenosu dat mezi grafickou stanicí a centrálním počítačem. Druhá varianta (využívaná pro grafické vybavení pracovních stanic) představuje použití nikoliv sítě, ale prostředků meziprocesové komunikace (v UNIXu IPC) a provoz serveru i klienta na tomtéž procesoru. Umístění částí X ukazuje schéma na obr. 7.3.

X-server, pokud komunikuje s klientem prostřednictvím sítě, využívá síťového protokolu, který je obvykle TCP/IP (viz kap. 8). Klientem je na obr. 7.3 např. některá z aplikací využívající grafických nástrojů (X-Toolkit). Klient se opírá při své práci o základní knihovnu funkcí Xlib. Programátor má k dispozici jak funkce pro výstup na obrazovku, tak funkce vstupní. Server tedy nejen provádí zobrazování objektů, ale předává zpět klientovi např. polohu umístění kurzoru atd. Xlib obsahuje možnost definovat standardní typy, s nimiž pak programátor může pomocí funkcí Xlib manipulovat. Základní strukturou je např. okno.

Součástí systému X je manažer oken (WM – Window Manager), což je grafické komunikační prostředí uživatele sedícího u terminálu. Uživatel má k dispozici otevírání oken, v nichž pracuje vždy v jednotlivém sezení v operačním systému, nebo může v okně pracovat určitá aplikace. Definovat a měnit polohu nebo velikost okna myší je běžná činnost. Okno je možné také sbalit do ikony a uvolnit si tak prostor pro jinou práci. Velmi často bývá WM implementován jako součást serveru. Z pohledu manažera oken je zřejmě důležité, aby jeden server dokázal komunikovat s několika klienty, případně aby klienty mohly spolu v rámci X spolupracovat. Spolupráci klientů se věnuje část X nazvaná ICCCM (Inter-Client Communication Conventions Manual).

Další komponentou X pracující nad vrstvou Xlib jsou Grafické nástroje X (X-Toolkit). Je to obecné označení pro knihovnu nadstavby Xlib, v jejímž rámci může programátor pohodlněji programovat provoz oken, menu atd. Jednotlivé grafické nástroje X mají charakteristickou tvář i způsob práce a přístupu uživatele k naprogramované aplikaci. Nejznámější je Xt. Pro UNIX je důležitá jednotnost grafického přístupu uživatele a z tohoto pohledu vznikl návrh



Obr. 7.3 Komponenty grafického systému X

grafického systému Motif podle OSF (Open System Foundation). Motif byl implementován a je používán jako jeden z možných grafických nástrojů v X.

Na obr. 7.3 je aplikace viditelná ještě přes X VDI a knihovnu GKS. Je to příklad dalšího grafického systému, který může využívat aplikace a v konečné části je implementován v Xlib. Jde o ukázkou přenosu jiných grafických systémů do prostředí X.

Vrátíme-li se na úroveň komunikace server – klient, z obrázku je patrná základní vrstva jejich grafické komunikace, a sice Xlib. Programátor v X pracuje s objekty. Klient může definovat objekty a pracovat s objekty typu okno (window), rastrový obrázek (pixmap), barevná mapa (colormap), kurzor (cursor), font (font), grafický kontext (graphic context). Objekt je možné vytvářet, rušit nebo definovat a měnit jeho atributy. Tyto akce provádí klient (pracuje po vytvoření s identifikací objektu – id) a server realizuje pokyny klienta v grafické práci s objektem (přesun okna na jinou část obrazovky atp.). Id objektů jiných klientů je klientem zjistitelné, a tak mohou klienti spolupracovat.

Hlavním předmětem zájmu programátora je okno. Okna jednoho klienta vytváří hierarchickou strukturu, přitom výchozí okno (označené jako root) je celá obrazovka. Hierarchie stanoví překrývání oken, protože potomci okna budou zobrazováni pouze v rámci rozměrů svého otce. Okno lze definovat pouze jako vstupní, to znamená, že např. lze jistou část obrazovky definovat pro identifikaci polohy kurzoru.

Grafický systém X-WINDOW SYSTEM je poměrně nákladná investice. Vlastní programové vybavení je volně přístupné včetně zdrojových textů (rozsahu nad 100 MB), ale předpoklady technického zařízení se pohybují na úrovni síťového propojení (technicky i programově) a úrovni stanice, která je schopna pracovat se serverem, což přibližně odpovídá schopnostech IBM PC/AT. Renomované firmy nabízejí specializované terminály (X-terminal), jejichž cena obvykle převyšuje cenu vhodného osobního počítače. Cena kvalitního programu, který na PC realizuje server, však tento schodek přinejmenším vyrovná.

## 8. Sítě

Prostředky komunikace uživatele a procesů s různými výpočetními systémy řeší počítačové sítě. UNIX ve své základní podobě v r. 1978 obsahoval pouze jednoduchý způsob spojení dvou výpočetních systémů pomocí terminálového rozhraní s názvem UUCP. Propojení přitom předpokládalo na všech výpočetních systémech pouze operační systém UNIX a komunikace se omezovala pouze na emulaci terminálu a přenos souborů. Principiálně stejné možnosti poskytoval prostředek KERMIT (Columbia University New York), který se neomezoval pouze na UNIX, ale dokázal přenášet data (a provádět jejich konverzi) terminálovým rozhraním mezi různými typy operačních systémů. KERMIT ale představuje pro UNIX bezvýznamnou epizodu, dnes není standardně dodáván jako součást systému. Se zapojením skupiny BSD na University of California v Berkeley byl zahájen několikaletý výzkum a vývoj síťového rozhraní pro UNIX. Výsledkem je programový produkt dnes v UNIXu označovaný jako Berkeley Services (známý více pod označením Berknet nebo Berkeley Sockets). Do stejné skupiny služeb bývá běžně také zahrnována podpora síťového rozhraní známá jako ARPA Services (uživatelské programy `telnet(1)` a `ftp(1)`).

Dnes nejvíce rozšířená (a SVID3 podporovaná) uživatelská úroveň sítí má název NFS (Network File System) a byla poprvé realizovaná firmou Sun Microsystem. Jedná se o spojení systémů souborů různých instalací operačních systémů tak, že uživatelům se jeví jako jeden strom adresářů.

Desetiletý vývoj počítačových sítí pro UNIX přinesl sjednocení také na úrovni komunikačních protokolů. Ze spojení skupiny BBN (Bolt, Beranek and Newman) a DARPA (Defense Advanced Research Project Agency) v r. 1980 vznikla v první polovině 80. let implementace síťového komunikačního protokolu pro systémy 4.xBSD s názvem TCP/IP (Transmission Control Protocol/Internet Protocol). Protože TCP/IP není vázáno na konkrétního výrobce a ve svém principu ani na určitý typ operačního systému, jeho rozšíření v rozsahu od střediskových až po osobní počítače v sítích LAN (Local Area Network) nebo WAN (Wide Area Network) znamená také řešení při spojování heterogenních sítí. V současné době je dominantní také pro podporu sítí v UNIXu.

Tato kapitola vysvětlí základní možnosti uživatele při vstupu do počítačových sítí. Od komunikace dvou uživatelů v rámci lokálního operačního systému (`write(1)` a `mail(1)`) přes popis UUCP vysvětlíme také rozšíření jádra ve smyslu PROUDů (STREAMS) pro podporu síťových protokolů. Uvedeme strukturu modelu OSI (Open Systems Interconnection) navrženého EIA (Electronics Industry Association) v počítačových sítích a ukážeme si začlenění jednotlivých popisovaných částí do tohoto modelu.

### 8.1 ZÁKLADNÍ PROSTŘEDKY KOMUNIKACE

Interaktivní rozhovor dvou uživatelů přihlášených v systému je realizován příkazem `write(1)`. K tomu, aby jej uživatel mohl dobře použít, je dobré mít seznam právě přihlášených uživatelů. Můžeme ho získat pomocí příkazu `who(1)`, např.

```
$ who
root    tty01    Dec 19 20:24
petr    ttyaa     Dec 19 16:25
```

V příkazovém řádku můžeme použít volby, které určují formát výpisu seznamu přihlášených uživatelů. Každý řádek výpisu odpovídá jednomu přihlášenému uživateli a má obecný tvar

```
jméno [stav] linka čas [spící] [pid] [komentář] [exit]
```

Bez voleb (uvedený příklad) mají položky výpisu po řadě význam jména uživatele (*jméno*), jména speciálního souboru terminálu, na kterém je uživatel přihlášen (*linka*), bez uvedení cesty `/dev`, data a času, kdy k přihlášení uživatele došlo (*čas*). Při použití volby `-u` (`who -u`) získáme navíc informaci, kolik minut je uživatel na terminálu neaktivní (*spící*), kdy pro případ aktivity v poslední minutě je zde uveden znak `.` a PID procesu shell uživatele (*pid*). Volba `-T` obohatí výpis o položku možnosti komunikace s uživatelem (např. příkazem `write(1)`) (*stav*). Je-li zde uveden znak `+`, sezení uživatele je jiným uživatelům dostupné pro rozhovor, je-li zde `-`, sezení je nedostupné. Použijeme-li v příkazovém řádku také volbu `-H`, výpis bude navíc obsahovat záhlaví. Např.

```
$ who -HuT
NAME      LINE      TIME          IDLE      PID      COMMENTS
root      +tty01    Dec 19 20:24  .         6289
petr      -ttyaa    Dec 19 16:25  .         5234
```

Konečně je možné také použít

```
$ who am i
```

pro výpis parametrů vlastního sezení.

Příkaz `write(1)` je pak používán ve formátu

```
write jméno [linka]
```

Rozhovor uživatel otevírá např. příkazem

```
$ write root
```

Uživateli `root` je vypsána na obrazovce zpráva o zahájení rozhovoru:

```
Message from petr (ttyaa) [Sat Dec 20:34:00] ...
```

a tím se dovídá o zprávě, která mu bude vypisována na obrazovku uživatelem `petr`, který je přihlášen na terminálu speciálního souboru `/dev/ttyaa`. Následující čas je časem zahájení rozhovoru. Bude-li oslovený uživatel přihlášen na několika terminálech současně, může uživatel otevírající rozhovor použít parametr *linka*, kde určí jméno speciálního souboru požadovaného terminálu. Následující text, který bude uživatel `petr` psát na klávesnici, bude zobrazován současně na jeho obrazovce i na obrazovce uživatele `root` (po řádcích), např.:



```
$ write root
Chci upozornit, ze Vase pritomnost
mne ponekud znervoznila!
                                Petr X
^d$
```

(posílaný text je ukončen znakem konce souboru). Oslovený uživatel může v průběhu zprávy vstoupit do rozhovoru a ihned reagovat na zasílaný text:

```
# write petr
```

Ale v případě, že je sezení uživatele `petr` pro komunikaci uzavřeno (`stav=-`), rozhovor nemůže být ze strany uživatele `root` navázán, je odbyt diagnostikou

```
Permission denied.
```

Každý uživatel má totiž možnost chránit se pomocí příkazu

```
mesg [y | n]
```

proti zprávám od ostatních uživatelů (parametr `n`). Parametr `y` změní sezení na přístupné ostatním uživatelům. Bez parametru příkaz vypisuje nastavení:

```
$ mesg
is n
```

Privilegovaný uživatel může použít příkaz `wall(8)`. (write all). Posílá jím zprávu všem právě přihlášeným uživatelům navzdory tomu, zda mají pro zprávy zamezený přístup či nikoliv:

```
# wall
Petre X,
Chci s Tebou hovorit,
jinak ocekavej svůj konec!
                                Root
^d#
```

Příkaz `wall(8)` používá superuživatel většinou pro zprávy zásadního charakteru, jako např. při zastavování chodu systému nebo při jeho přechodu na jinou úroveň (viz čl. 9.2).

Jiný způsob komunikace uživatelů je neinteraktivní, použitím elektronické pošty (electronic mail). UNIX ji podporuje příkazem `mail(1)` a `mailx(1)`.

Poslat poštu uživateli registrovanému v systému (v tabulce `/etc/passwd`, `/etc/group` atd. viz čl. 2.6) můžeme pomocí

```
$ mail petr
```

nebo více uživatelům

```
$ mail petr jan
```

atd. Následující text, přijatý ze standardního vstupu, je uvažován jako text pošty. Je obohacen o záhlaví a přenesen do poštovních schránek označených uživateli. Záhlaví obsahuje řádek se jménem uživatele a označením pošty, odkud je dopis posílán, dále řádek s uvedením délky dopisu ve znacích atd. Je-li pošta posílána sítí, dozvíme se jména všech uzlů, kterými musela projít.

Uživatel otevírá a následně čte obsah své poštovní schránky (obsah souboru podle `$MAIL` nebo `$MAILPATH` viz čl. 4.8) příkazem bez parametrů :

```
$ mail
```

kdy je na standardní výstup vypsán nejprve počet dopisů v poštovní schránce. Uživatel může číst jednotlivé dopisy vnitřním příkazem `p`. Při otevření poštovní schránky je čtení nastaveno na naposledy doručený dopis. Uživatel může měnit čtení (nebo jinou manipulaci s dopisy) pouze odesláním znaku nový řádek (=následující dopis) nebo zápisem znaku `-` (předchozí dopis). Dopisy tvoří frontu a jsou čteny v opačném pořadí než přicházely. Při přechodu na jiný dopis je vypsáno jeho záhlaví. Číst dopisy v pořadí jejich přijetí lze zajistit volbou `-r`. Můžeme ale také číst dopisy i z jiné poštovní schránky než `$MAIL`, a sice pomocí volby `f`, např.:

```
$ mail -r -f staraposta
```

otevřeme poštovní schránku ze souboru `staraposta` a budeme její obsah číst v pořadí FIFO. `mail(1)` jako jeden z mála příkazů v UNIXu obsahuje nápovědu. Při čtení poštovní schránky ji můžeme zobrazit vnitřním příkazem `?`. Dostáváme seznam vnitřních příkazů `mail(1)`

<nový řádek>	přesun na další dopis
+	totéž
d	zrušení dopisu z poštovní schránky a přesun na následující dopis
p	výpis obsahu dopisu
-	přesun na předchozí dopis
s [ <i>soubor</i> ]	uschová dopis do souboru se jménem <i>soubor</i>
w [ <i>soubor</i> ]	uschová pouze text dopisu bez záhlaví
m [ <i>jméno</i> ... ]	pošle aktuální poštu, uživateli <i>jméno</i>
q	uloží nezrušené dopisy zpět do souboru poštovní schránky a ukončí činnost
EOF	(Ctrl-d) totéž
x	uloží zpět do poštovní schránky všechny dopisy a ukončí činnost
! <i>příkazový_řádek</i>	na <i>příkazový_řádek</i> je spuštěn shell
*	vypíše seznam příkazů

S poštou souvisí také proměnná shellu `MAILCHECK`. Obsahuje časový interval (v sekundách), po kterém je vždy kontrolován obsah poštovních schránek. Pokud byla v uplynulém časovém intervalu přijata nová pošta, uživatel je na tuto skutečnost na obrazovce upozorněn zprávou:

```
You have mail.
```

Je-li `MAILCHECK=0`, příchod pošty je uživateli oznámen ihned po následující výzvě k interakci s shellem. Standardně bývá `MAILCHECK=600` (10 minut). Není-li poštovní schránka prázdná, uvedená zpráva se na terminál vypisuje také po přihlášení uživatele.

Větší komfort při práci s odesíláním nebo přijímáním pošty poskytuje podsystém `mailx(1)`, který je rozšířením `mail(1)` a je doporučován SVID3.

V rámci označení uživatele v příkazech elektronické pošty může být uživatel určen také vzhledem k uzlu v počítačové síti, např.:

```
$ mail petr@ncvut.cs
```

je označení uživatele se jménem `petr`, za znakem `@` následuje označení uzlu v síti (podrobněji viz čl. 8.5 a 8.6).

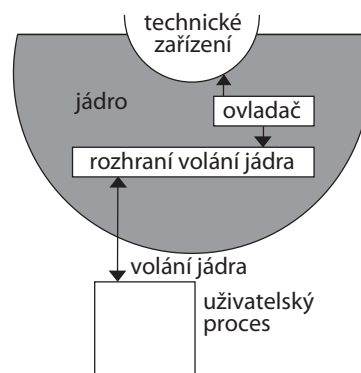
V počítačové síti pracuje elektronická pošta na dvou úrovních. Jednak je to úroveň uživatelského zprostředkovatele (UA – user agent) pro práci s koncovým uživatelem a jednak úroveň zprostředkovatele přenosu (MTA message transfer agent) pro doručení a přijetí dopisu. Úroveň MTA souvisí se způsobem připojení uzlu k síti, což je námětem dalších částí této kapitoly.

## 8.2 PROUDY

UNIX SYSTEM V přinesl pro síťovou komunikaci úpravu jádra označovanou jako PROUDY (STREAMS). Principiálně se jedná o možnost vkládání programových modulů mezi ovladač zařízení a datovou oblast procesu tak, že vkládaný modul přenášená data nějakým způsobem upravuje (obr. 8.1).

Uživatelský proces komunikuje s periferií pomocí speciálního souboru a volání jádra `open(2)`, `close(2)`, `read(2)`, `write(2)`, `ioctl(2)`. V jádru tyto požadavky na periferii přebírá ovladač periferie. Bude-li ale v jádru periferie podporována mechanismem PROUDŮ, bude cesta k periferii v jádru obohacena o záhlaví proudu (podle obrázku 8.2).

PROUD je sekvence programových modulů, které je možné vkládat mezi záhlaví proudu a ovladač (obrázek 8.3).



Obr. 8.1 Styk uživatelského procesu s technickým zařízením

Uživatelský proces si vytváří PROUD sám, protože moduly mezi ovladač a záhlaví proudu vkládá voláním jádra `ioctl(2)`, a to způsobem

```
ioctl (fd, I_PUSH, jmeno);
```

kde `fd` je deskriptor souboru (celočíslná hodnota získaná z návratové hodnoty `open(2)` a `jmeno` je označení vkládaného modulu. `I_PUSH` je příkaz pro vložení modulu `jmeno`, a to za záhlaví proudu před všechny dříve stejným způsobem vložené moduly (vzniká zásobník modulů). Naposledy vložený modul můžeme z PROUDu vyjmout příkazem

```
ioctl(fd, I_POP, 0);
```

Jednotlivé moduly PROUDu jsou duplexní (pracují odděleně pro přenos dat k periférii a od periférie) a komunikují mezi sebou prostřednictvím předávání zpráv. Uživatel komunikuje s periférií na úrovni přenosu dat jako obvykle pomocí `read(2)` a `write(2)`, ale může navíc vložit nebo vyjmout zprávu ze záhlaví PROUDu pomocí volání jádra

```
#include <stropts.h>
```

```
int getmsg(int fd, struct strbuf
    *ctlptr, struct strbuf *dataptr,
    int *flagsp);
```

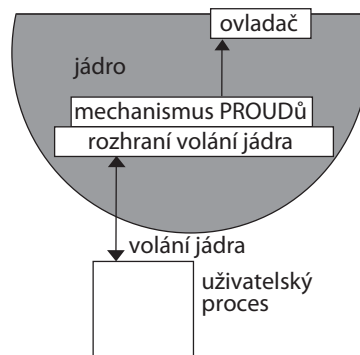
pro vyjmutí zprávy z PROUDu a

```
int putmsg(int fd, const struct strbuf *ctlptr,
    const struct strbuf *dataptr, int flags);
```

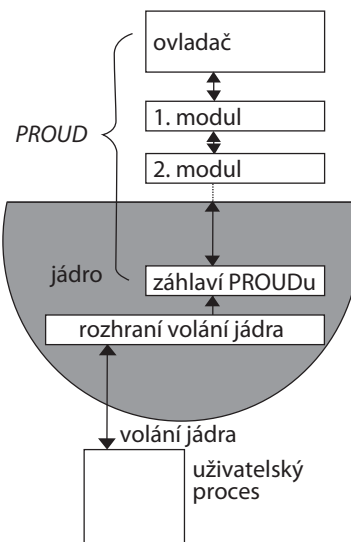
pro vložení zprávy do PROUDu. Přidávaná zpráva je rozdělena na datovou (`dataptr`) a řídicí (`ctlptr`) část. Struktura `strbuf` obsahuje položky:

```
int    len;        /* délka dat */
char   *buf;       /* ukazatel na data */
```

Jakým způsobem je zpráva PROUDem zpracovávána, je dáno funkcí PROUDu a definicí jeho jednotlivých modulů. Předávání zpráv je rozšířeno pomocí `getpmsg(2)` a `putpmsg(2)`,



Obr. 8.2 PROUDově orientované periférie



Obr. 8.3 PROUD

kteřé jsou vybaveny větší flexibilitou v řídicí části zprávy. Konečně volání jádra

```
#include <poll.h>

int poll (struct pollfd fds[],
          unsigned long nfd,
          int, timeout);
```

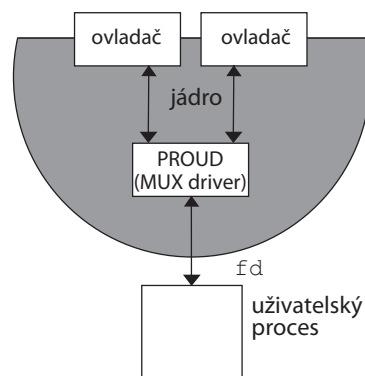
slouží k řízení PROUDu, je-li tento používán současně z několika různých deskriptorů `fds[]` (input/output multiplexing). V poli `fds` jsou definovány příznaky, které jsou pro odpovídající deskriptor důležité a které znamenají získání pozornosti.

Uvedené PROUDY mají lineární charakter, tj. PROUD je sekvence modulů, které pracují se zapisovanými nebo čtenými daty. Jsou proto velmi výhodné pro vytváření jednotlivých vrstev síťových disciplín. PROUDY ale navíc umožňují pracovat sdíleně, jeden proces může definovat PROUD pro dva ovladače (PROUD multiplexuje periferní data, viz obrázek 8.4), což je používáno např. při rozlišení dat několika se scházejících sítí nebo pro aplikace oken terminálů.

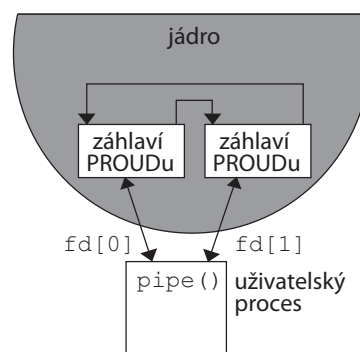
Pomocí PROUDů byl také vyřešen problém obousměrné roury (označované někdy jako roura na principu PROUDů). Volání jádra `pipe(2)` potom znamená vytvoření dvou vzájemně propojených záhlaví PROUDů v jádru (viz také čl. 5.5).

Výhoda nově implementované obousměrné roury je také v možnosti vkládání modulů do jednotlivých PROUDů a v důsledku toho v ovlivňování práce roury (obrázek 8.6).

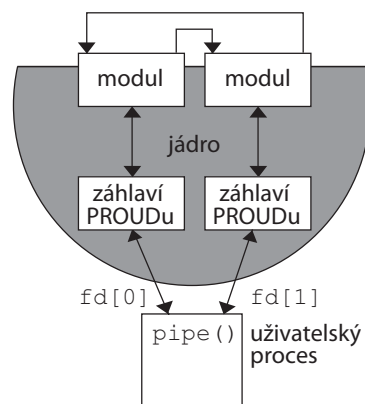
Stranou nezůstává ani problém pojmenované roury, která je řešena pomocí pojmenovaných PROUDů. PROUD může být pojmenován voláním jádra, kde `fdes` je deskriptor vytvořené obousměrné roury a `path` textový řetězec jména souboru v systému souborů.



Obr. 8.4 Sdílený PROUD



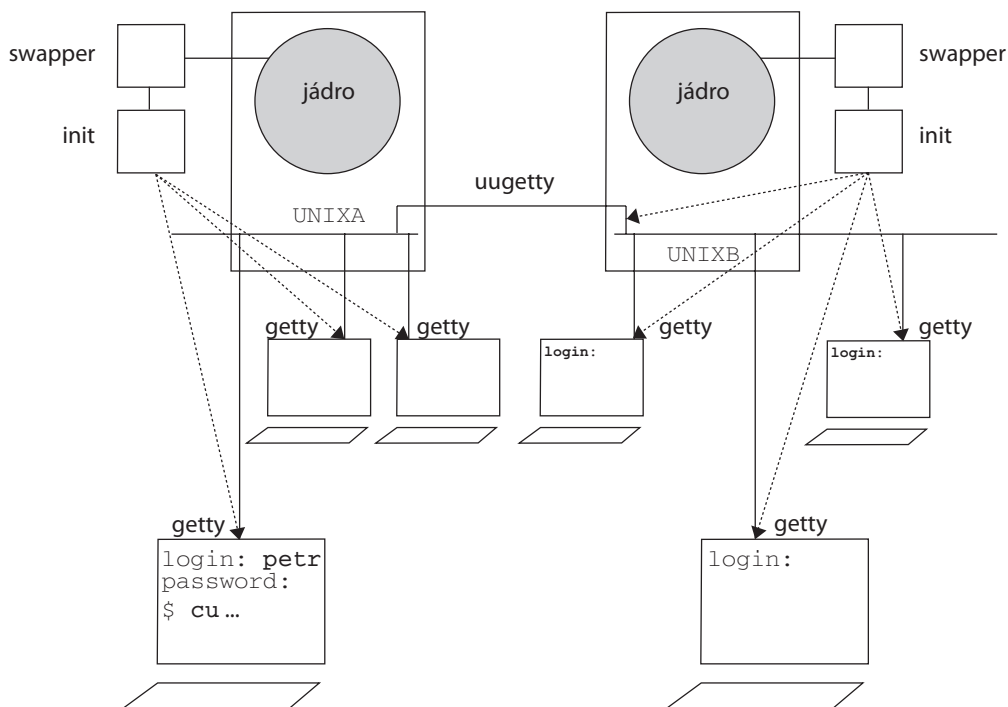
Obr. 8.5 Obousměrná roura na principu PROUDů



Obr. 8.6 Obousměrná roura a její PROUDY

## 8.3 PODSYSTÉM UUCP

Fenomén UUCP je pro UNIX historickou záležitostí. Kdysi vznikl pro spojení dvou nebo více počítačů s operačním systémem UNIX. Přestože mu pozorovatelé (zejména obchodníci) dávali stále menší a menší šanci na přežití, v dnešní neutěšené době různorodých variant rozhraní pro síť získává opět na vážnosti. Myšlenka UUCP je triviální (viz obrázek 8.7). Jádro



Obr. 8.7 UUCP ve dvou systémech UNIXA a UNIXB

v systému UNIXA iniciuje komunikaci uživatelů s operačním systémem pomocí procesů `getty(8)`, které se promění v době přihlašování na uživatelský proces shell nebo jiný komunikační proces. Uvažujeme zjednodušeně propojení výpočetních systémů pomocí sériového rozhraní stejně jako je propojen každý terminál na obrázku. Proces `getty` (nebo `uucp`) pracuje také při spojení dvou systémů. Směrem od systému UNIXB je k UNIXA vyslán procesem `getty` znakový řetězec končící na `@!login :` (`@` je identifikace systému – uzlu v síti). Naproti tomu systém UNIXA neaktivuje ze svého směru tuto linku žádným procesem.

Přihlášeným uživatelům v UNIXA je k dispozici příkaz `cu(1)`, pomocí něhož mohou navázat spojení se systémem UNIXB. Např. píše-li uživatel v UNIXA

```
$ cu -l /dev/term/AA
```

ozve se proces `getty` ze systému `UNIXB`

```
UNIXB!login :
```

a očekává vstup uživatele do systému `UNIXB`. `cu(1)` je program, který emuluje terminálovou komunikaci uživatele vzdáleného systému. `cu(1)` dokáže pracovat i s linkou, která je přístupná přes telefonní ústřednu

```
$ cu 0=923265
```

kde znak `=` je upozornění, kdy má `cu(1)` očekávat další ohlašovací telefonní tón (`0` je např. vstup z firemní do městské telefonní sítě). Uživatel bude ale nejčastěji používat `cu(1)` s označením jména vzdáleného systému, např.

```
$ cu UNIXB
```

V rámci stanoveného propojení operačních systémů jsou naplněny odpovídající tabulky, kde je dána jednoznačná korespondence mezi speciálním souborem (příp. tel. číslem) a jménem vzdáleného systému. Příkazem

```
$ uuname
```

získá uživatel seznam jmen všech evidovaných vzdálených systémů. Pomocí volby `-l` (`local`) získá jméno lokálního systému. Podrobnější informace o označení lokálního systému lze získat příkazem `uname(1)`

```
$ uname
```

což je jméno lokálního systému, ale nikoliv jméno v rámci sítě. Lze ale použít volby s významem:

- `-n` jméno systému v rámci sítě (`nodename`, analogie `uname -l`)
- `-r` označení vydání systému (`release`)
- `-v` verze (`version`)
- `-m` jméno stroje (`machine hardware name`)
- `-a` kumuluje všechny předchozí volby (`all`)

Pomocí příkazu `cu(1)` může uživatel vstoupit do vzdáleného systému běžným způsobem (přihlásit se ve vzdáleném systému známým jménem atd.). Znak `~` je řídicím znakem komunikace uživatele a `cu(1)`. Např. zápisem

```
~.
```

dojde k přerušení spojení a `cu(1)` ukončí činnost. Pomocí `cu(1)` můžeme také přenášet soubory ze vzdáleného systému do lokálního. K tomu slouží

```
~%take from [to]
```

kde *from* je jméno souboru ve vzdáleném systému a pomocí *to* můžeme stanovit jméno cílového souboru v lokálním systému (jinak je dáno podle *from*). Naopak

```
~%put from [to]
```

přenášší *cu(1)* soubor se jménem *from* z lokálního systému do vzdáleného. Pomocí

```
~%cd
```

lze měnit nastavení pracovního adresáře v lokálním systému a

```
~!
```

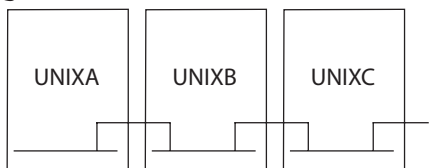
můžeme až do ukončení (Ctrl-d) pracovat s příkazovým interpretem lokálního systému (bude startován proces shell jako syn *cu(1)*).

Za situace zobrazené na obr. 8.7 mohou uživatelé systému **UNIXA** využívat zdrojů uzlu **UNIXB**, použijeme-li pro práci s linkou proces **getty**. Uživatelům systému **UNIXB** můžeme umožnit využívat uzel **UNIXA** buď použitím další sériové linky, nebo změnou směru využití linky určené pro spojení. Tento problém ale řeší také použití procesu příkazu *ugetty(8)*, který je možné spustit na obou stranách spojovaných uzlů a teprve po startu *cu(1)* ustoupí lokální **ugetty** uživatel. Za tímto účelem je zavedena volba *-r* příkazu *ugetty(8)*.

Přenos souborů z jednoho uzlu do jiného umožňuje příkaz *uucp(1)*

```
$ uucp prog.c UNIXB! /usr/jan/prog.c
```

provede kopii souboru *prog.c* pracovního adresáře do souboru */usr/jan/prog.c* uzlu v síti se jménem **UNIXB**. Základní formát příkazu je



Obr. 8.8 Lineární spojení více uzlů

```
uucp [-d] [-j] [-m] [-njméno] [-r] výchozí_soubor cílový_soubor
```

*výchozí\_soubor* je seznam všech jmen souborů určených pro přenos a *cílový\_soubor* je jméno souboru (v případě přenosu jednoho souboru) nebo adresáře, které je cílovým jménem pro přenášený soubor. Parametr *výchozí\_soubor* má formát

```
jméno_uzlu!cesta
```

kde *jméno\_uzlu* je jméno uzlu v síti, odkud je přenos prováděn. Uvážíme-li více uzlů v lineárním propojení (podle obrázku 8.8),



můžeme v určení *jméno\_uzlu* použít průchod několika systémy, např.

```
$ uucp UNIXB!UNIXC! /usr/jan/*.c .
```

kopíruje všechny soubory končící *.c* z adresáře */usr/jan* uzlu *UNIXC* do pracovního adresáře lokálního systému (se jménem *UNIXA* podle obr. 8.8). Příklad a formát *uucp(1)* naznačuje také použití

```
$ uucp UNIXB!UNIXC! /usr/jan/*.c UNIXB! /usr/jan
```

v sezení uživatele uzlu *UNIXA*. V určení *cesta* můžeme také použít odkaz tvaru

```
~jméno
```

kde *jméno* je jméno uživatele, který je registrován v odpovídajícím uzlu, a odkaz pak nahrazuje cestu jeho domovského adresáře. Uvedené volby ve formátu *uucp(1)* mají význam:

- d při přenosu jsou vytvářeny neexistující potřebné adresáře
- j na standardní výstup je zobrazeno označení práce, která realizuje přenos; označení práce můžeme použít v dalších příkazech statistiky přenosu
- m po uskutečnění přenosu je uživateli, který *uucp(1)* použil, poslána pošta
- n *jméno* má tentýž význam jako *~jméno*
- r vlastní přenos není zahájen, je pouze založena evidence přenosu

Program *uucp(1)* pracuje s frontou požadavků na přenos. Do vzdálených uzlů vstupuje jako uživatel *uucp* a proces **uugetty** mu podle tabulky */etc/passwd* startuje proces **uucico**, což je komunikační proces protokolu UUCP. Výpis znaku *\$* po příkazu *uucp(1)* neznamena uskutečnění přenosu, ale pouze registraci ve frontě pro přenos (viz volby *-j*, *-r* a *-m*). Uživatel má pro práci s frontou požadavků na přenos k dispozici příkaz *uustat(1)*,

```
$ uustat -s UNIXB
```

který lze použít pro výpis všech transakcí pro uzel *UNIXB*,

```
$ uustat -u jan
```

pro výpis transakcí aktivovaných uživatelem *jan* a

```
$ uustat -m
```

je výpis všech požadavků na přenos v lokálním uzlu. Příkazem formátu

```
uustat -kidtrans
```

rušíme (`kill`) požadavek přenosu s označením `idtrans` (získaný výpisem nebo volbou `-j` v `uucp(1)`).

Posledním příkazem z kolekce podsystému UUCP je provedení akce v daném uzlu příkazem `uux(1)`

`uux příkazový_řádek`

V parametru `příkazový_řádek` přitom používáme odkaz na uzly v síti, a to jak při zadávání jména příkazu, tak i ve jménech souborů v parametrech příkazu. Např.

```
$ uux "UNIXB!tar -cvf /dev/rmt0 UNIXB!~petr/*.c"
```

spustí v uzlu `UNIXB` program `tar(1)`, který bude na magnetickou pásku lokálního uzlu (danou speciálním souborem `/dev/rmt0`) archivovat všechny soubory z domovského adresáře uživatele `petr` a uzlu `UNIXB` končící na `.c`.

## 8.4 MODEL OSI

Implementace počítačové sítě je vrstvená. Obyčejný uživatel sedící u terminálu většinou rezignuje na znalosti nižších vrstev sítě než je úroveň aplikační. Správce lokálního systému je povinen zajímat se o vrstvu transportní (přenosového protokolu). Osoba zabývající se technickým zařízením sítě naopak pracuje s nejnižší úrovní, tj. s úrovní linkovou. S programátorem sítí se setkává na úrovni vrstvy síťové.

7	Aplikační (Application)
6	Prezentační (Presentation)
5	Relační (Session)
4	Transportní (Transport)
3	Síťová (Network)
2	Linková (Data Link)
1	Fyzická (Physical)

Obr. 8.9 Vrstvy sítě podle OSI

Standard při popisu vrstev dnes znamená model OSI (Open System Interconnection) definovaný institucí ISO (International Standards Organization). Struktura OSI má 7 vrstev (obrázek 8.9).

Fyzická vrstva je definována pro technické zařízení, v rámci ní se hovoří o charakteristikách elektronického přenosu. Je jím např. technické zařízení Ethernet/IEEE 802.3. Vrstva linková je realizována rozhraním odpovídajícím typu technického zařízení. Jedná se o ukládání informací vyšších vrstev do rámců (frames) a technické zajištění přenosu. Vrstva síťová se věnuje způsobu komunikace s jinými uzly v síti. Na této úrovni je nutné akceptovat např. rozhraní X.25. V UNIXu dnes dominuje rozhraní nazývané IP (Internet Protocol). Transportní vrstva je určena definicí přenosového protokolu. Je jím určité TCP (Transmission Control Protocol), přestože jej ISO neuvádí. Nevyhovuje totiž definovaným požadavkům pro ošetření chyb. Naopak

je v modelu OSI podporován protokol UDP (User Datagram Protocol). Čtvrtá vrstva je v UNIXu implementována různým způsobem, jako další příklad můžeme uvést SNA (IBM's System Network Architecture) nebo XNS (Xerox Networking Systems). Vrstva relační (pátá vrstva) je vrstvou vstupu uživatele do sítě. Pro UNIX se jedná především o úroveň využití definovaných volání jádra. Na úrovni vrstvy prezentační dochází ke kódování a komprimaci přenášených textů. Typické je využití jazyka XDR (eXternal Data Representation), který pracuje nad RPC (Remote Procedure Call) relační vrstvy. Konečně nejvyšší vrstva je vrstva aplikační. Je ryze uživatelská a patří sem NFS (Network File System), prostředky virtuálních terminálů `ftp(1)` nebo Berkeley `rlogin(1)` atd.

vrstva	Arpa/Berkeley		NFS
OSI	ARPA	Berkeley	
7	ftp (File Transfer Protocol) telnet (Telnet)	BIND (Berkeley Internet Name Domain Server) rcp (Remote Copy) rlogin (Remote Login) rexec (Remote Execution) remsh (Remote Shell) rwho (Remote Who) ruptime (Remote Uptime) Sendmail	RFS (Remote File Sharing) NFS (Network File System) YP (Yellow Pages) VHE (Virtual Home Environment)
6	SMTP (Simple Mail Transfer Protocol)		XDR (eXternal Data Representation)
5		BSD IPC (Sockets)	RPC (Remote Procedure Call)
4	TCP (Transmission Control Protocol)		UDP (User Datagram Protocol)
3	IP (Internet Protocol)		
2	Ethernet		
1	Ethernet/IEEE 802.3		

Obr. 8.10 UNIX z pohledu OSI

Přestože jsou dnešní používané síťové prostředky v UNIXu v mírném rozporu s OSI (výjimku tvoří SVID3), budeme se ve zbytku kapitoly snažit kontext s OSI udržovat (obr. 8.10 se snaží zobrazit vztah UNIX a OSI).

Z hlediska uživatele a vrstvy aplikační se od provozu sítě zejména vyžaduje:

- výměna pošty
- přenos souborů
- sdílení periferií, např. tiskárny nebo disku
- spouštění procesů ve vzdáleném uzlu
- přihlášení ve vzdáleném uzlu

Na obr. 8.10 jsme vybrali hlavní reprezentanty sítí pro UNIX, které pokrývají uvedenou množinu požadavků na úrovni aplikační vrstvy. Otázka aplikace je věcí programovou, která závisí na možnostech nižších vrstev sítě. Proto si v dalším článku uvedeme stručnou charakteristiku protokolu IP a komunikačních protokolů TCP a UDP. Aplikace, uživatelský a systémový přístup uvedeme ve čl. 8.6, zejména pro NFS ve smyslu doporučení SVID3.

Počítačová síť (computer network) je komunikační systém pro spojení dvou výpočetních systémů. Výpočetní systém v síti označujeme termínem uzel (node, host). Uzel může být osobní počítač, sálový počítač sdílený více uživateli anebo počítač, který zajišťuje pouze veřejnou obsluhu tiskárny nebo sdílení disků (print server, file server).

LAN (Local Area Network) je označení pro místní síť počítačů (obvykle v rozsahu několika kilometrů). Pro LAN je typická vysoká přenosová rychlost (10–16, ale i 100 Mbps). Technické

prostředky bývají obvykle Ethernet a síť s přenosem příznaků (token ring) nebo v poslední době známý FDDI (Fiber Distributed Data Interface) na bázi optických vláken.

WAN (Wide Area Network) spojuje výpočetní systémy v různých městech nebo zemích. Přenosová rychlost je typicky nižší, spojení je pevnou telefonní linkou na rychlosti 9600 bps nebo 1.55 Mbps.

MAN (Metropolitan Area Network) stojí mezi LAN a WAN a je to počítačová síť některého města nebo regionu. Technické principy propojení a přenosové rychlosti odpovídají LAN (spojení bývá koaxiálním kabelem a vzduchem v oblasti VKV).

V následném kroku jde o spojování sítí různé úrovně a různého chování (internet nebo internetwork). Podle vrstvy OSI, na které je spojení dvou sítí provedeno, rozlišujeme způsob spojení

- opakovač (repeater) jde o spojení na úrovni fyzického přenosu, obvykle jde o zdvojení dat dvěma různými směry
- most (bridge) nejčastěji pracují na druhé vrstvě; jde o kopii rámců do sousední sítě
- směrovač (router) je vztažen ke třetí vrstvě vstupu do sítě a podporuje migraci paketů jednotlivých sítí
- brána (gateway) je způsob propojení dvou sítí na úrovni komunikačního protokolu (např. TCP/IP); jde o zabezpečení styku na vysoké úrovni přenosu dat, přičemž každá strana zajistí stejné rozhraní pro síť

Pro uživatele sedícího u terminálu je tato struktura spojování počítačů a sítí skryta. Jeho zájem by měl končit znalostí jmen uzlů v síti a jmen, pod kterými je v těchto uzlech evidován. Ostatní vyplývá z použití aplikační vrstvy.

Adresace systému v síti a procesu v uzlu sítě vyplývá z jednoznačnosti

- sítě
- uzlu v síti
- procesu v uzlu

Z toho plyne asociace dvou procesů v síti. Je dána specifikací 5 položek:

```
{protokol, místní_adresa, místní_proces, vzdálená_adresa,  
vzdálený_proces}
```

např.

```
{tcp, 1.40.254.2, 201, 133.26.35.1, 21}
```

Z myšlenky poloviční asociace, tj. zúžení asociace na {protokol, místní\_adresa, místní\_proces} a {protokol, vzdálená\_adresa, vzdálený\_proces}, principiálně vycházejí schránky typu síť Berknet.

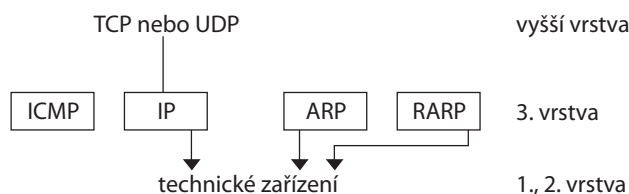
Pro síť jsou také důležité termíny server a klient (server, client), které jsou už uvedeny v kap. 5 a 7. Jde o vzájemnou komunikaci dvou procesů, z nichž server je proces očekávající požadavek jednoho nebo více klientů, který podle svých schopností požadavky realizuje. Klient se obrací na server s požadavky definovanými jejich protokolem. Přenosovým

protokolem obou takových procesů, z nichž každý obvykle pracuje v jiném uzlu sítě, je některý ze standardních protokolů podle čtvrté vrstvy OSI.

## 8.5 PŘENOSOVÝ PROTOKOL TCP/IP

Vrstva vstupu do sítě je reprezentována IP (Internet Protocol). Slovo internet použité v názvu vychází z označení obecného protokolu mezi různými sítěmi. Tento problém řeší projekt DARPA (Defense ARPA), jehož produktem je práce v síti označované jako Internet a odtud je protokol nazýván IP. Je úzce spojen s protokolem vyšší vrstvy TCP, takže je obvykle přenosový protokol označován jako TCP/IP. Na bázi tohoto způsobu propojování počítačů a počítačových sítí dnes pracuje ve světě více než 200 000 síťových uzlů. TCP/IP řeší spojení počítačů různé kategorie, ať už jde o dva osobní počítače nebo o rozsáhlé počítačové sítě.

IP je protokolem síťové vrstvy. Zajišťuje vytvoření paketu s odpovídající výchozí a cílovou adresou. IP je nezávislý na provozu v síti, nezajišťuje doručení paketu, případně jeho opakování – to přenechává vyšší vrstvě. Pro IP je každý paket samostatný, a tak jej posílá do sítě. Vrstva IP má strukturu podle obr. 8.11.



Obr. 8.11 Vrstva IP

Součástí IP může být modul ARP (Address Resolution Protocol) pro převod adres sítě Internet na fyzické (vzhledem k technickému zařízení) a RARP (Reverse Address Resolution Protocol) naopak pro převod fyzických adres na adresy IP.

Nutný je ale modul ICMP (Internet Control Message Protocol), jehož úkolem je zpracovat chyby a řídicí informace mezi bránou a uzlem v síti. Součástí testování stavu sítě je kontrola existence a stavu jednotlivých uzlů sítě programem `ping(1)` (packet internet grouper). Reakce na jeho řídicí znaky zajišťuje právě ICMP.

IP v případě příchozího paketu zjišťuje správnost jeho kontrolních součtů včetně hlavičky IP (20 slabik), která zahrnuje mimo jiné adresu příjemce a adresáta. Neprojde-li test paketu modulem IP, je paket zapomenut. Obohacování paketu o adresy probíhá ve formátu čtyř různých typů, vždy ale s délkou 32 bitů. Tři hlavní typy adres ukazuje obr. 8.12. Z něho vyplývá použití daného typu adresy: bude-li místní síť pracovat s větším množstvím uzlů, je výhodnější si rezervovat pro identifikaci větší rozsah pro označení uzlů a použijeme třídu A, naopak třída C bude vhodná při vstupu do většího počtu sousedních sítí. Adresa 32 bitů je přepočítávána na tzv. **tečkový** formát. Jde o převod z kódování v šestnáctkové soustavě na 4 čísla v desítkové soustavě oddělená tečkou, každé číslo odpovídá



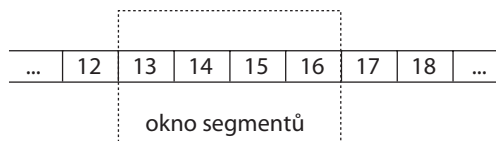
Obr. 8.12 Typy adres IP

jedné slabice. Použijeme-li např. třídu A a budeme kódovat síť č. 1 a uzel 28FE02, pak celkově z 0x0128FE02 dostáváme 1.40.254.2 nebo např. tečkový formát 133.26.35.1 představuje využití kódování adresy třídy B.

Adresa uzlu v síti s přenosovým protokolem TCP/IP musí být unikátní. Jednoznačnost je nutná zvláště v dnešní době, kdy dochází k propojování počítačů a sítí celého světa. Jednoznačné adresy IP v rámci sítě Internet přiděluje NIC (Network Information Center) v rámci SRI International, viz [12]. Správce systému nebo sítě, která je budována, by měl v každém případě žádat o přidělení adresy NIC, a to ať už na úrovni státní, regionální nebo firemní lokality.

Čtvrtá vrstva modelu OSI je v rámci TCP/IP realizována protokoly TCP nebo UDP. Oba využívají nižší (třetí) vrstvu, tedy uvedený protokol IP. TCP je používán aplikační vrstvou pro zajištění duplexního spojení pro proudový přenos dat. UDP je naopak používán pro aplikace bez stálého spojení, datagramově orientované.

TCP zajišťuje přenos paketů v síti na protokolární úrovni. Je odpovědný za ztrátu a žádost o opakování paketu. Vytváří plně duplexní sekvenční kanál mezi dvěma procesy v síti. TCP z dat, která získává z vyšší vrstvy (od procesů), vytváří segmenty. Segmenty čísluje (sequence number), obohatí je o záhlaví a předává IP. TCP je jeden z protokolů posuvného okna (sliding window protocol). Znamená to, že nad připravenými segmenty pro vstup do sítě udržuje okno o počtu



Obr. 8.13 Okno segmentů TCP délky 4

několika segmentů, které právě předal do sítě, ale prozatím nedostal jejich potvrzení. V příkladu na obrázku 8.13 je využíváno okno délky čtyři segmentů. Segmenty č. 13, 14, 15, 16 jsou v síti, prozatím nepřišlo jejich potvrzení od adresáta. Segment č. 12 byl dříve vyslán a potvrzen. Segmenty č. 17 a 18 jsou připraveny pro vstup do sítě, ale prozatím je TCP nepředává. Velikost okna je závislá na propustnosti sítě. Mnohé instalace nastavují délku okna na jeden segment, tím snižují míru zahlcení sítě, ale mnohdy nevyužívají plně její možnosti. Při použití délky okna jeden segment hovoříme o protokolu čekání (stop-and-wait protocol).

Modul TCP je schopen pracovat pro několik uživatelských procesů současně. Pro identifikaci procesu využívá TCP koncové číslo (port number). TCP a UDP mají k dispozici koncová čísla např. v rozsahu 1–255. Nad hodnotou 255 bývají obvykle koncová čísla rezervovaná pro privilegované procesy a dále (v oblasti nad 1023) dočasná koncová čísla (ephemeral port numbers). Při průchodu sítí ke koncovému procesu je používána asociace uvedená v předchozím článku, která zahrnuje pět specifikací. Na místě místní proces a vzdálený proces TCP používá koncové číslo. Koncová čísla TCP ukládá do záhlaví vytvořeného segmentu.

TCP podporuje model komunikace procesů server-klient. V rámci navázání spojení TCP umožňuje navázání aktivního (pro proces klient) a pasivního (pro proces server) spojení (active open, passive open).

Optimální propustnost sítě musí TCP zajistit pomocí řízení toku paketů. Využívá k tomu tzv. okna řízení toku (flow control windows). Potvrzení paketu obsahuje ve svém záhlaví hodnotu délky dat, kterou je v dané chvíli uzel schopen přijmout (může být i 0, což znamená pozastavení přenosu). Velikost segmentu je omezena délkou 65 515 přenosových slabik, ale TCP dělí data obvykle na podstatně menší části.

Záhlaví datového segmentu TCP má délku 20 přenosových slabik a obsahuje tyto hlavní části:

- koncové číslo místního uzlu
- koncové číslo vzdáleného uzlu
- číslo segmentu
- oblast potvrzování
- řídicí část
- okno řízení toku
- kontroly (checksum)
- příznak zvláštní pozornosti (Urgent Pointer)

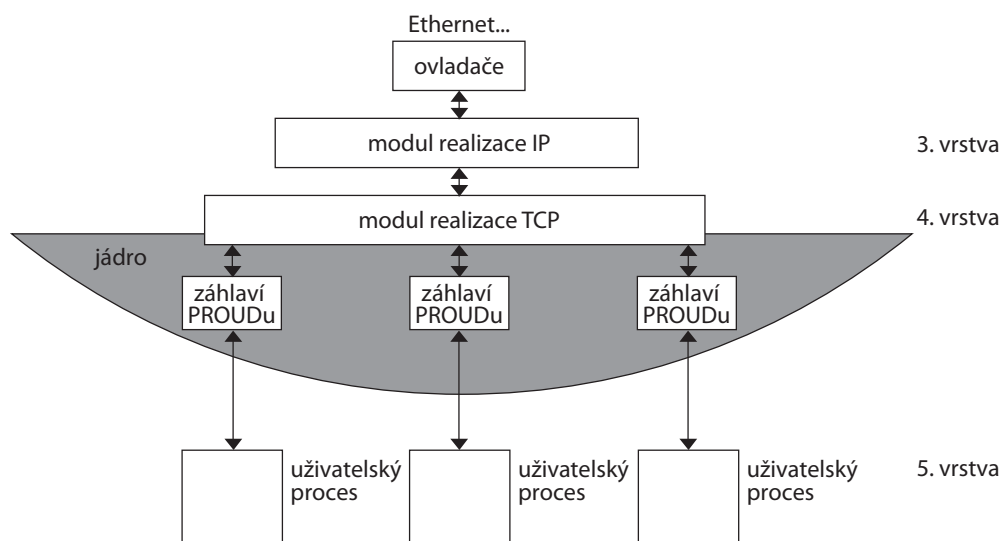
Záhlaví neobsahuje délku segmentu, protože ta je dána nižší přenosovou vrstvou; délku přenosového segmentu stanovuje IP.

Celkový rámec, který prochází sítí, je pak datový segment obohacený o záhlaví TCP, IP a nižších vrstev, např. podle obr. 8.14.

14 slabik	20 slabik	20 slabik (UDP 8)		4 slabiky
záhlaví Ethernet	záhlaví IP	záhlaví TCP	data	ukončení Ethernet

Obr. 8.14 Rámec TCP/IP pro zařízení Ethernet

Implementaci TCP/IP pomocí PROUDů ukazuje obr. 8.15



Obr. 8.15 Implementace TCP/IP pomocí PROUDů

Vrstvy 2–4 jsou součástí jádra. Moduly TCP a IP jsou vkládány v sekvenci PROUDu. Procesy (uživatelské, privilegované, démoni) nejčastěji zajišťují 5.–7. vrstvu.

Protože ne každá aplikace vyžaduje přímé spojení dvou procesů sekvenčním tokem slabik, v rámci protokolu TCP/IP figuruje protokol UDP. Je postaven na stejnou úroveň jako TCP, tj. na čtvrtou vrstvu OSI, a využívá protokolu IP. Je podstatně jednodušší než TCP, ale například nezajišťuje přenos s potvrzením všech segmentů. Znamená to, že veškerou práci pro zajištění přenosu všech dat musí splňovat vyšší vrstva. Záhlaví UDP protokolu má pouze 8 slabik a obsahuje:

- koncové číslo místního uzlu
- koncové číslo vzdáleného uzlu
- délku segmentu dat
- kontrolu

UDP bývá typicky využíván aplikací NFS, která pracuje na sdílení oblastí systému souborů.

### 8.6 SÍŤOVÉ APLIKACE

Aplikační vrstva je určena požadavky uživatele, které jsme uvedli v čl. 8.4. Služby ARPA v rámci aplikací `telnet(1)` a `ftp(1)` nabízejí možnosti přihlášení se ve vzdáleném uzlu, přenos souborů v síti a elektronickou poštu. `telnet(1)` je způsob virtualizace terminálu vzdáleného uzlu. Napíšeme-li

```
$ telnet SASbrno
```

znamená to, že požadujeme vstup do vzdáleného uzlu v síti se jménem `SASbrno`. Napíšeme-li pouze

```
$ telnet
```

vstupujeme do interaktivního režimu, dostáváme výzvu

```
telnet>
```

a můžeme pomocí vnitřních příkazů programu stanovit parametry spojení. Vnitřním příkazem

```
open uzel [ označení ]
```

vstupujeme do sítě a požadujeme uživatelské připojení k uzlu `uzel`. Vnitřní příkaz

```
? [ příkaz ]
```

je nápověda, vnitřní příkaz

```
close
```

uzavírá spojení a



```
quit
```

končí interaktivní komunikaci (vracíme se k lokálnímu sezení). V průběhu trvání vzdáleného sezení můžeme vstoupit do interakce s místním programem `telnet(1)` znakem `Ctrl-]`, který lze předefinovat vnitřním příkazem

```
escape [znak]
```

Program `ftp(1)` (`file transfer program`) zajišťuje přenos souborů mezi dvěma různými uzly. Pomocí `ftp(1)` lze iniciovat v místním uzlu přenos souborů mezi dvěma vzdálenými uzly, `ftp(1)` může pracovat se dvěma kanály TCP při přenosu souborů, tj. lze interaktivně řídit přenos souborů mezi lokálním a vzdáleným uzlem (skrytě pomocí **telnet**). `ftp(1)` má základní příkazový řádek

```
ftp [ volby ] [ uzel ]
```

a rovněž pracuje v režimu server–klient. V okamžiku startu, je-li stanoven vzdálený uzel, dojde ke spojení a na opačné straně pracuje server pro zajištění místních požadavků na přenos souborů (auto-login). `ftp(1)` nabízí po startu výzvu k interakci

```
ftp>
```

kdy můžeme používat vnitřní příkazy k přenosu dat. Stejně jako v případě `telnet(1)`, je i zde vnitřní příkaz

```
? [ příkaz ]
```

který zajišťuje nápovědu (příkaz `?` má synonymum `help`). Vnitřním příkazem `bye` (nebo `quit`) končíme činnost `ftp(1)`, zatímco `close` pouze uzavírá spojení (server je odpojen) a příkaz `open uzel` spojení se vzdáleným uzlem `uzel` navazuje (`ftp(1)` si vyžádá jméno a heslo vzdáleného uživatele). Přenos souborů může např. být příkazem

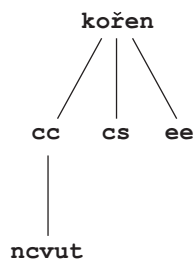
```
ftp> get odtamtud sem
```

což znamená, že požadujeme ze vzdáleného uzlu přenos souboru `odtamtud`, data jsou lokálně uložena do souboru se jménem `sem` v pracovním adresáři. Opačný přenos provedeme příkazem `put`. Oba pracovní adresáře ve vzdáleném i lokálním uzlu můžeme měnit vnitřními příkazy `cd` a `lcd` (`local change directory`).

Pro elektronickou poštu pracuje ve službách ARPA protokol SMTP (Simple Mail Transfer Protocol, šestá vrstva OSI). Jeho využíváním získají aplikace zajištěný přenos pošty, protože SMTP poštu v případě neúspěchu přenosu po jistém čase opakuje (úroveň zprostředkovatele přenosu MTA viz čl. 8.1).

Uživatel při zadávání adresáta v síti používá syntax

```
místní_část@doména
```



Obr. 8.16  
Ukázka DNS

*místní\_část* je označení adresáta v daném uzlu a *doména* je jméno uzlu z hlediska systému DNS (Domain Name System). DNS je kolekce systémových programů, která virtualizuje síť z pohledu uživatele na strom uzlů v síti. Příklad je uveden na obr. 8.16. Cesta od kořene k některému listu grafu je označení některého uzlu v síti. Hierarchie DNS umožňuje její snadnou rozšiřitelnost v případě připojení dalšího uzlu nebo napojení na jinou síť. *doména* je dána cestou od kořene DNS stromu k listu, jednotlivé uzly stromu jsou odděleny znakem `.`, jméno *kořen* se přitom vynechává. Např. z obr. 8.16 lze konstruovat doménu uzlu sítě `ncvut.cs`. Použití domény v adrese je pak např.

`petr@ncvut.cs`

a znamená to, že pošta bude zaslána uživateli `petr` v uzlu `ncvut.cs`. Je zřejmé, že informace DNS stromu a všech odpovídajících tabulek (adres IP atd.) musí být distribuovány v síti. To zajišťují příslušné programy systému DNS v rámci zóny (část DNS stromu) automaticky.

V rámci implementované BSD aplikační vrstvy (na obr. 8.10 Berkeley) může uživatel používat příkaz `rlogin(1)` pro přihlášení se ve vzdáleném uzlu, a to způsobem

```
rlogin uzel [volby]
```

kdy v jednoduchém případě může použít pouze jména vzdáleného uzlu

```
$ rlogin science
```

což znamená, že je uživateli zpřístupněn terminálový vstup do uzlu se jménem `science` a že je přitom přihlášen pod stejným jménem jako v lokálním uzlu. Volbou

```
-l jméno_uživatele  stanovíme jméno uživatele pro přihlášení ve vzdáleném uzlu
-7                  emulace terminálu bude probíhat na přenosu 7 bitů
-8                  emulace terminálu pracuje se všemi 8 bity přenosu
-c znak             je výměna řídicího znaku pro přerušení spojení na znak
```

Řídicí znak nastavitelný volbou `-c` je při spuštění `rlogin(1)` nastaven na `~`. Použijeme-li jej samostatně na řádku, spojení je přerušeno.

Příkazem `rwho(1)` získáme seznam přihlášených uživatelů všech uzlů místní sítě. S `rwho(1)` spolupracuje v každém uzlu démon `rwhod(8)`. Pomocí `ruptime(1)` můžeme pak získat přehled o stavu uzlů v síti při běhu démonů `rwhod(8)`.

Pomocí `rcp(1)`

```
rcp [-r] soubor1 [soubor2 ... ] kam
```

kopírujeme seznam souborů *soubor1*, *soubor2*, ... do cílového adresáře (souboru) *kam*. Volba *-r* je příznak kopie celého podstromu v případě, že jeden ze *soubor1* ... je adresářem. Jméno souboru (adresáře) v příkazovém řádku je v případě odkazu na vzdálený uzel konstruováno způsobem

```
[uživatel@]uzel:cesta
```

např.

```
science:/usr/jan/*.c
```

je odkaz na všechny soubory končící na *\*.c* v adresáři */usr/jan* uzlu *science*. Nebo

```
jan@science:*.c
```

je totéž, jako když má uživatel *jan* v uzlu *science* nastaven domovský adresář na */usr/jan* (expanzní znaky jmen souborů jsou stejně jako v UUCP nahrazeny v cílovém uzlu, lokálně ale v příkazovém řádku musí být vypnut jejich speciální význam).

Příkazem *remsh(1)* můžeme provádět příkaz shellu ve vzdáleném uzlu (analogie *uux(1)* v UUCP, viz čl. 8.3).

Uživatel může také používat funkci *rexec(3)* ve formátu

```
int rexec(char **ahost, int inport, char *user, char *passwd,
          char *cmd, int *fd2p);
```

kterou může použít pro provedení příkazu *cmd* ve vzdáleném uzlu *ahost*.

Síťová aplikace NFS (Network File System) plně podporovaná SVID3 je příklad aplikace v síti pro sdílení zdrojů uzlů sítě. Vychází především z myšlenky spojení stromů adresářů (nebo jejich částí) do jednoho superstromu nebo spíše síťového stromu adresářů.

Doporučení SVID3 stanovuje dva způsoby propojení dvou stromů adresářů různých uzlů. Jednak je to RFS (Remote File Sharing), který nabízí možnost v různých uzlech sdílet celý obsah podstromu uvolněného pro síť, tj. obvyčejné soubory, adresáře, pojmenované roury a speciální soubory. Aplikace NFS umožňuje uvolněný podstrom sdílet dalšími uzly sítě pouze na úrovni obvyčejných souborů a adresářů. Uzel, který uvolňuje pro síť podstrom adresářů, nazýváme server vůči sdílení, a uzly, které podstrom v síti používají, klienty. Server uvolní podstrom příkazem superuživatele *share(8)*. Volbou *-F* stanovujeme typ stromu adresářů, který uvolňujeme, tj. zda se jedná o způsob RFS nebo NFS. Např.

```
# share -F nfs /usr/src
```

uvolňuje uzel podstrom daný adresářem */usr/src*, aplikace je typu NFS. Nebo

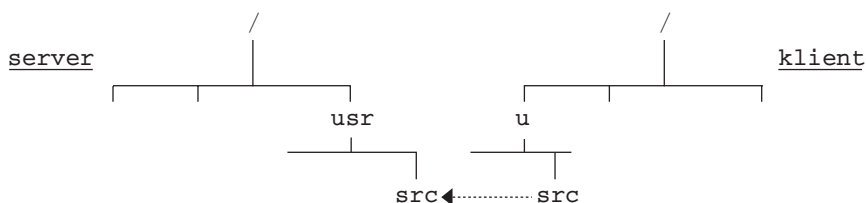
```
# share -F rfs /usr/src SOURCES
```

je uvolňován tentýž podstrom pro aplikaci RFS a je pojmenován jako `SOURCES`. Jméno uvolněného podstromu je využíváno klienty, které si uvolněný podstrom připojují příkazem `mount(8)` (má rozšířené použití v rámci sítě), např.

```
# mount -F rfs SOURCES /u/src
```

což znamená, že síť nabízený strom adresářů typu RFS se jménem `SOURCES` připojujeme k místnímu uzlu do adresáře `/u/src`. Data od této chvíle budou dostupná všem uživatelům uzlu podle přístupových práv (dají se omezit v parametrech příkazu `share(8)`), jako kdyby se jednalo o část systému souborů místního uzlu. Není-li uvolněný strom adresářů pojmenován, připojení probíhá. Např.:

```
# mount -F nfs ncvt.cs:/usr/src /u/src
```



Obr. 8.17 Připojení vzdáleného stromu adresářů

kde `ncvt.cs` je jméno uzlu, který uvolňuje data pro síť. Situaci ukazuje obr. 8.17. Za účelem odpojení sítě poskytovaného systému adresářů je k dispozici rozšířený příkaz `umount(8)` a ke zrušení uvolnění svého místního podstromu pro síť příkaz `unshare(8)`. Další příkazy, které pracují pro údržbu podstromů v síti, jsou např. `adv(8)`, `unadv(8)`, `dfshares(8)`, `nsquery(8)` atd. Správce uzlu musí totiž např. znát seznam a jména veřejných stromů v síti.

Aplikace YP a VHE uvedené na obr. 8.10 jsou další volitelné aplikace v rámci úhrnně označované aplikace NFS (SVID3 je nezahrnuje). YP je aplikace sloužící pro práci s centrální databází sítě v rámci údržby sítě. VHE umožňuje konfigurovat sezení v uzlu tak, že odráží prostředí vzdáleného uzlu.

Od prvních počátků vážného vývoje síťového rozhraní v operačním systému UNIX stála v popředí snaha spojovat nejen počítače jednoho typu operačního systému, ale byla zde také snaha o přijetí standardu Internet a využívání obecně daných přenosových protokolů TCP/IP a UDP/IP při spojování počítačů do heterogenních sítí. V dnešní době jsme svědky implementace nejen uvedených přenosových protokolů do různých operačních systémů (VMS, MS-DOS atd.), ale i aplikací typu NFS nebo FTP atd. Nabídka aplikačního programového vybavení jde obvykle ruku v ruce s podporou X-WINDOW SYSTEM v případě grafické práce, zvláště z důvodu plné síťové transparentnosti systému X, takže uživatel může využívat místních prostředků grafické práce v libovolném operačním systému, jeho vlastní aplikace ale pracuje ve vzdáleném uzlu jako klient sítě (viz kap. 7).

## 9. Údržba

Údržba operačního systému je soubor činností správce výpočetního systému, zajišťující bezchybnou a deterministickou práci konkrétní instalace operačního systému. Takový soubor prací zahrnuje sledování stavu obsahu magnetických médií s daty, sledování funkcí běžícího jádra a korektní zásahy při změně konfigurace výpočetního systému.

Od dob, kdy bylo nutné, aby správce systému prováděl údržbu a zásahy do operačního systému řádkovou komunikací pomocí sady příkazů podle svazku (8) dokumentace, byl učiněn krok kupředu ve smyslu obrazovkově orientovaného shellu správce systému. Jde o interpret příkazů, který je specializován pro akce údržby operačního systému. Uživatelsky je orientován na výběr z menu, což orientaci a práci správce systému podstatně usnadňuje. Bohužel není z hlediska ovládání ve verzích UNIXu jednotný a příkaz pro jeho spuštění je také různý (`sysadmsh` pro SCO UNIX, `sam` pro HP-UX, ...). Také komfort poskytovaných služeb se liší. V lepších případech je při systémové akci uživateli jako komentář vypsán ekvivalent zápisu akce v řádkové komunikaci. UNIX SYSTEM V takový shell nemá a SVID3 jej nedefinuje.

Údržbu operačního systému provádí správce především jako privilegovaný uživatel `root`. Zvláště v řádkové komunikaci je důležité zachovat maximální pozornost, protože privilegovanému uživateli je dovoleno takřka vše, včetně destrukce jádra. Správce by proto měl všechny své zásahy předem promyslet a zůstatvat v privilegované úrovni jen po nejnnutnější dobu.

### 9.1 SPUŠTĚNÍ A ZASTAVENÍ OPERAČNÍHO SYSTÉMU

Po zapnutí počítače jsou provedeny testovací programy uložené v pevných pamětech, které zajišťují provozuschopnost základních komponent počítače (operační paměť, operátorská konzola, základní magnetický disk atd.). Poté je počítač uveden do výchozího stavu, kdy je základní monitor počítače zavlečen do paměti a spuštěn (úroveň firmware), jeho komunikace s člověkem probíhá na operátorské konzole. Je to program umožňující interaktivně provádět např. testování periférií na úrovni technické podpory. Hlavní funkcí monitoru je ale možnost zavlečení do paměti a následné spuštění programu, který je schopen samostatně (standalone) pracovat na holém počítači. Programem tohoto typu je také jádro operačního systému UNIX. Pro firmware je v tomto okamžiku nutné odkazem na některou periférii (magnetický disk, magnetická páska atd.) stanovit místo, kde je samostatný program uložen. Např. odkaz na magnetický disk č. 0 znamená v případě, že je na něm instalován UNIX, zavedení a spuštění jádra operačního systému. V některých případech, jako je např. oblast osobních počítačů, program zde označený jako základní monitor není součástí stroje a počítač po testech technického vybavení očekává samostatný program na standardní periférii (pro PC první disketová mechanika, není-li připravena, aktivní část prvního pevného disku). Z tohoto důvodu, ale také z důvodů větší obecnosti, je UNIX (jeho jádro) zavlékán do operační paměti ve dvou fázích. Ve fázi první je z daného média do paměti zavlečen program, který nahrazuje práci monitoru technického vybavení. Je to samostatný program, který oznámí svůj start na operátorskou konzolu, např.:

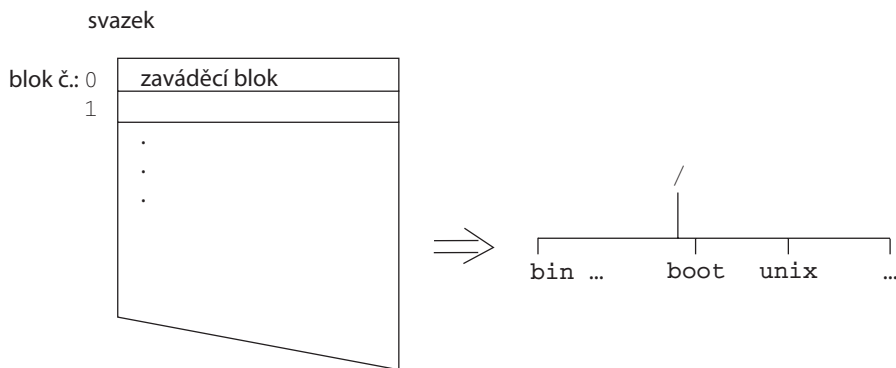
```
Boot
:
```

## 9.1 Údržba

Znak : je znak výzvy pro komunikaci. Tento samostatný program dokáže podle pokynů člověka pracovat se strukturou magnetického média typu UNIX. Napíšeme-li

```
Boot
: fd(0) /unix
```

zadááme pomocí `fd(0)` typ a pořadí periferie před cestou k samostatnému programu (v tomto případě je `fd` první disketová mechanika). `Boot` je samostatný program, který, pokud se jedná o referenci na diskovou periferii, je obvykle uložen v souboru se jménem `/boot`. Rutiny práce s technickým vybavením jej naleznou na disku pomocí obsahu zaváděcího bloku (boot block) svazku (viz čl. 2.3).



Obr. 9.1 Umístění samostatných programů

Zaváděcí blok je naplněn programem, který po zavedení a spuštění hledá na téže svazku v kořenovém adresáři soubor se jménem `boot`. Ten je zaveden do paměti a spuštěn. Obsah zaváděcího bloku je tedy rutina styku operační paměti s konkrétním typem periferie, přitom nesmí přesáhnout velikost jednoho bloku svazku. Rutiny pro zaváděcí blok různých typů disků jsou uloženy v adresáři `/etc` pod jmény souborů, které mnemonicky začínají označením typu periferie (např. v SCO UNIX `fd96ds15boot0` pro disketu 5,25" s kapacitou 1.2MB) a jsou součástí strojově závislé části systému. Při vytváření svazku se zaveditelným operačním systémem je proto nutné mít jako součást svazku komponenty:

- zaváděcí blok naplněný programem, který odpovídá typu periferie
- samostatný program `boot` ve struktuře svazku v kořenovém adresáři
- taktéž v kořenovém adresáři samostatný program jádra `unix`

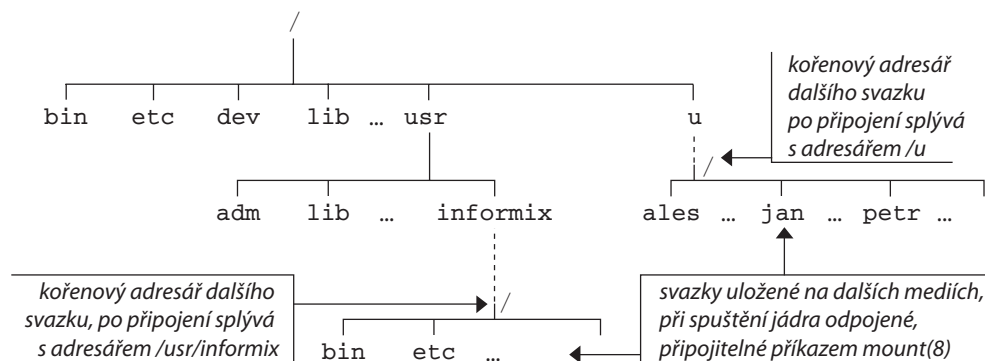
Přitom se výchozí reference technické podpory vždy vztahuje k začátku periferie, což je zaváděcí blok. Ostatní součásti startu systému si postupně předávají řízení, kdy poslední fázi (druhou fází) je zavedení a spuštění samostatného programu `/unix`, kterým je jádro operačního systému.

Je-li jádro zavedeno a spuštěno, obalí technické vybavení, což znamená prověření existence všech periférií, které jsou v tabulkách jádra evidovány. Na konzolu vypisuje identifikaci verze systému a diagnostiku technického vybavení. Jádro testuje velikost zbytku neobsazené

operační paměti a považuje za kořenový svazek ten, ze kterého bylo zavedeno. Vytváří procesy **swapper** a **init** (viz kap. 2) a přechází do úvodního stavu.

Úvodní stav operačního systému je označován jako režim jednoho uživatele (jednouživatelský stav). Je charakterizován tím, že pouze jeden uživatel, a to superuživatel, může na operátorské konzole v systému pracovat. Procesy, které běží, jsou proces **swapper** pro práci s odkládací pamětí, proces **init** jakožto otec terminálových procesů a démonů a terminálový proces operátorské konzoly `sh(1)` privilegovaného uživatele. Pro zajištění ochrany dat je před spuštěním `sh(1)` vyžadován zápis hesla. Teprve při bezchybném zadání hesla uživatele `root` je možné vstoupit do režimu jednoho uživatele.

Režim jednoho uživatele je využíván pro systémovou údržbu. Superuživatel provádí akce, které je obtížné – ne-li nemožné – uskutečnit, pokud v systému pracují další uživatelé. Jedná se např. o kontrolu všech svazků na magnetických discích nebo instalaci nového programového vybavení, regeneraci jádra atd. Pro denní provoz je při startu systému nutná kontrola všech svazků, které budou připojeny k výchozímu stromu adresářů a na kterých budou uživatelé se svými daty pracovat. Jakákoliv nekonzistence dat na některém svazku, způsobená např. výpadkem elektrického proudu (viz čl. 2.3 o systémové vyrovnávací paměti) bez provedení opravy, může totiž při provozu způsobit další poškození vnitřní struktury dat na médiu a vést ke znehodnocení dat na svazku. Privilegovaný uživatel proto v jednouživatelském režimu používá program `fsck(8)` pro kontrolu a případnou opravu svazků. Ve chvíli vstupu do jednouživatelského režimu jádro zná jediný svazek jako připojený. Je jím svazek, odkud bylo zavlečeno jádro, a který tak obsahuje výchozí kořenový strom adresářů. Všechny ostatní svazky, které jej rozšiřují, jsou odpojeny (příklad viz obr. 9.2). Jako první svazek superuživatel prohlíží svazek s kořenovým stromem příkazem

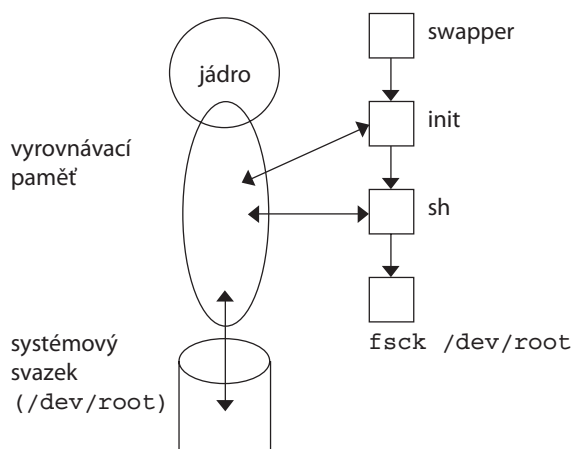


Obr. 9.2 Příklad použití několika svazků

```
# fsck /dev/root
```

protože speciální soubor `/dev/root` je používán pro označení disku se svazkem, na kterém je kořenový strom se zaveditelným jádrem. `/dev/root` je dalším odkazem na speciální soubor systémového svazku. Pokud při této kontrole `fsck(8)` nalezne ve struktuře systému souborů nesrovnalosti (např. nalezne datové bloky, které nejsou zařazeny ani do seznamu volných, ani do seznamu obsazených bloků), nabízí korekci (zařadí takové bloky do seznamu

volných, angl. salvage) a superuživatel s korekcí souhlasí (yes) nebo ne (no). V každém případě je ale po opravě systémového svazku nutné dosáhnout shody mezi obsahem vyrovnávací paměti a obsahem svazku. Situaci lze zjednodušeně zobrazit podle obr. 9.3.



Obr. 9.3 Systémový svazek a vyrovnávací paměť

Za běhu jádra v jednouživatelském režimu je systémový svazek připojen a veškerá data svazku a data, se kterými procesy pracují, procházejí systémovou vyrovnávací pamětí. Je v ní uložen i obsah bloku popisu svazku (super block). V případě opravy pomocí `fck(8)` a následné aktualizace dat na disku pomocí `sync(2)` je nekonzistentní virtualizovaný systémový svazek překopírován opět na médium. Cesta, jak tomu zabránit, je zapomenout obsah systémové vyrovnávací paměti, svazek `/dev/root` odpojit a nově připojit. V mnoha implementacích je to možné pouze vypnutím počítače. Mnohdy ale `fck(8)` na tento problém upozorňuje zprávou

```
***** FILE SYSTEM WAS MODIFIED *****
***** BOOT UNIX!! NO SYNC !!!!! *****
```

kdy superuživatel vyzývá k novému zavedení jádra bez aktualizace vyrovnávací paměti (`NO SYNC!!!!`) na připojeném médiu kořenového svazku. Ještě větší zabezpečení může být implementováno za interakce jádra a `fck(8)`, kdy jádro za pomoci `fck(8)` pozná opravu a samo zapomíná a obnovuje obsah vyrovnávací paměti podle opraveného svazku. V tomto případě na konzolu po zprávě o modifikaci jádro namísto výzvy k novému zavedení vypisuje

```
***** REMOUNTING THE ROOT FILESYSTEM *****
```

a superuživatel může pokračovat v práci bez opětného zavedení jádra.

Jako další akci po kontrole systémového svazku provádí superuživatel kontrolu všech ostatních svazků, které budou uživatelé při běhu systému využívat. Kontrola a opravy se opět provádějí pomocí příkazu

```
fck device
```

kde na místě `device` je uveden speciální soubor kontrolovaného svazku. Kontrolovaný svazek musí být odpojen od systémového svazku, takže při opravě nevzniká nebezpečí rozdílu dat mezi obsahem svazku a vyrovnávací pamětí.

Superuživatel dává pokyn k přechodu z jednouživatelského režimu do víceživatelského



a umožňuje tak ostatním uživatelům přihlašovat se a vstupovat do systému. Tento pokyn je realizován odhlášením superuživatele v jednouživatelském režimu (ukončení práce shellu).

Přechod systému z úrovně jednouživatelské do úrovně víceuživatelské je charakterizován připojením ostatních svazků, vytvořením procesů **getty** pro všechny terminály, ze kterých se budou uživatelé přihlašovat (**getty** vypisuje na každý takový terminál textový řetězec končící na `login :`), vytvořením procesů démonů pro zajištění cyklicky se opakujících akcí (např. spooling pro tiskárnu **lpsched**, cyklická aktualizace obsahu systémové vyrovnávací paměti na připojené svazky **update** atd.), případně démonů zajišťujících síťové propojení nebo specializované akce pro styk se speciální periferií atd. Akce, o které chce správce systému lokální systém rozšířit a které by se měly provádět v tomto přechodovém stadiu, může zapsat do souboru `/etc/rc`, a to způsobem programování v Bourne shell. `/etc/rc` je scénář pro `sh(1)`, který je interpretován při přechodu do víceuživatelského režimu jako poslední akce přechodu. Je důležité si uvědomit, že scénář nemá implicitně stanovený standardní vstup a výstup, všechny procesy spuštěné v `/etc/rc` pracující interaktivně je proto nutné explicitně přesměrovat na operátorskou konzolu. Součástí `/etc/rc` může být např. příkazový řádek

```
/usr/bin/startrobot < /dev/console >/dev/console
```

kde spuštěný proces **startrobot** pracuje s explicitně přepnutým standardním vstupem i výstupem na operátorské konzole (`/dev/console`).

Po interpretaci `/etc/rc` je ukončen přechod do víceuživatelského režimu a uživatelé mohou pracovat v systému pomocí připojených terminálů.

Zastavení operačního systému může uskutečnit pouze privilegovaný uživatel. Používá k tomu příkaz

```
# shutdown
```

který většinou za 15 minut korektně UNIX zastaví. V průběhu této doby vždy po uplynutí několika minut vypisuje na všechny terminály informaci o blížícím se konci práce operačního systému. Uživatelé tak vyzývá k odhlášení se, protože tím mohou nejlépe zabránit ztrátě dat, např. právě editovaných souborů. Po uplynutí čekací doby 15 minut `shutdown(8)` odpojuje uživatele od systému (ruší všechny procesy **getty**), zastavuje činnost všech démonů, aktualizuje vyrovnávací paměť na připojené svazky, svazky odpojuje a zastavuje jádro. Vypisuje zprávu typu

```
System is halted
```

(systém je zastaven) a upozorňuje, že vypnutím a zapnutím počítače můžeme opět UNIX spustit.

## 9.2 Údržba

Za chodu operačního systému ve víceuživatelském režimu je také možná změna režimu opět na jednouživatelský. Příkazem

```
# kill -1 1
```

superuživatel odpojí všechny uživatele (ihned a bez varování) a na operátorské konzole dává superuživateli možnost vstupu do jednouživatelského režimu. Proces **init** se v tomto případě postará o zánik všech procesů kromě sebe sama a procesu **swapper**. Vytváří proces nový, proces pro přihlášení superuživatele. Svazky zůstávají připojeny, systémová vyrovnávací paměť není přepisována na disky. `kill -1 1` tedy znamená pokyn pro zánik všech synů procesu č. 1 – **init**. Dále je na superuživateli, zda svazky odpojí a jaké další akce bude provádět. Systém lze i zde zastavit příkazem `shutdown(8)`. Odhlášením superuživatele přechází systém opět do víceuživatelského režimu.

### 9.2 PROCES INIT

Malá pružnost pouhých dvou režimů operačního systému vedla tvůrce systému UNIX SYSTEM V.2 (AT&T r. 1984) k nové koncepci práce procesu **init**. Jeho funkce je rozšířena o možnost evidence osmi různých režimů (úrovní) a zajištění přechodu systému z dané úrovně na jinou. Úroveň systému je dána skupinou procesů, které vstupem do stanovené úrovně **init** vytváří; všechny existující procesy s výjimkou sebe sama a procesu **swapper** zastavuje. Ve dvou úrovních může být evidován tentýž proces, jeho činnost pak není přerušena, proces existuje a pracuje i přes změnu úrovně. Úrovně jsou označeny 0, 1, 2, ... 6, S a mají význam

- 0 zastavení operačního systému
- 1 úroveň odpovídající jednouživatelskému režimu
- 2 víceuživatelský režim
- 3 připojení systému k síti
- 4 obvykle volná úroveň pro alternaci víceuživatelského režimu
- 5 zastavení systému, vstup do monitoru technického vybavení (firmware)
- 6 zastavení a znovuzavedení operačního systému
- S, s úroveň jednoho uživatele (single-user mode); odpovídá stavu systému po přechodu z víceuživatelského režimu příkazem `kill -1 1`; bývá součástí úrovně 1 jako její poslední etapa; při přechodu do úrovně 1 se dříve ještě provádí akce upozorňující ostatní uživatele na nutnost odhlásit se, odpojí se svazky (`shutdown` bez zastavení počítače s přechodem do jednouživatelského režimu)

Chování procesu **init** vzhledem k úrovním popisuje tabulka `/etc/inittab`. Její struktura odpovídá po řádcích popisu jedné akce jedním řádkem (1 proces). Řádek je konstruován podle schématu

*id:úroveň;způsob\_akce:proces*

Znak : slouží jako oddělovač položek popisu. *id* je identifikace akce, její délka je nejvýše 4 znaky. Např.

```
ttl1a: ...
```

Označení *úrovně* je výčet všech úrovní, při vstupu do kterých je akce provedena. Např.

```
ttl1a:24: ...
```

je evidence pro úroveň 2 nebo 4. *způsob\_akce* zadává, jak má být akce provedena (proces spuštěn). Tato položka může obsahovat např. textové řetězce

respawn	<b>init</b> při vstupu do úrovně vytvoří proces (odpovídající následující položce <i>proces</i> ) a po dobu systému v úrovni jej opět vytvoří v případě zániku procesu
wait	při vstupu do úrovně je definovaný proces spuštěn a <b>init</b> čeká na jeho dokončení; poté pokračuje v realizaci dalších akcí úrovně
once	při vstupu do úrovně je proces spuštěn, <b>init</b> nečeká na dokončení procesu, ani proces nově nevytváří, jestliže zanikne
off	akce je vypnuta, proces není při vstupu do této úrovně vytvářen
initdefault	je označení akce, která se bude provádět při prvním spuštění procesu <b>init</b> (při startu systému)

atd.

Např.

```
ttl1a:24:respawn: ...
```

A konečně poslední položka *proces* je příkazový řádek spuštění procesu, vlastní akce. Např.

```
ttl1a:24:respawn:/etc/getty /dev/tty1a 9600
```

je spuštění procesu **getty** s příslušnými parametry (na terminálu */dev/tty1a* s nastavením charakterizujícím přenosovou rychlost 9600 bps), a to vždy po vstupu do úrovně 2 nebo 4; kdykoliv proces ukončí svoji činnost (uživatel se odhlásí), **init** proces **getty** nově spustí.

Při startu systému je vytvořen proces **init** a poprvé prohlíží tabulku */etc/inittab* (podle akce *initdefault*). Na základě úvodních akcí přechází do výchozího stavu (obvykle jednouživatelského režimu, úroveň S). Není-li žádná akce jako *initdefault* označena, **init** vyžaduje na konzole zadání úrovně, na kterou má operační systém nastavit.

Přechod z jedné úrovně na jinou zadává superuživatel příkazem např.

```
# init 4
```

nebo

```
# telinit 4
```

## 9.2 Údržba

což je pokyn procesu **init** pro přechod na úroveň 4. **Init** uzavírá stávající úroveň (všem procesům, které nejsou označeny pro existenci v nové úrovni, posílá signál `SIGTERM` a po 20 vteřinách `SIGKILL`). Pak otevírá úroveň novou (postupně prochází tabulku `/etc/inittab` a vytváří daným způsobem všechny nové procesy.

Provede-li superuživatel editorem změnu v tabulce `/etc/inittab` (např. změnou `off` na `respawn` u **getty** některého terminálu), může na změnu příkazem

```
# init Q
```

upozornit a **init** aktualizuje stav systému na dané úrovni podle současného textu v `/etc/inittab`.

Konečně se mohou v položce *úroveň* vyskytovat také znaky `a`, `b`, `c`. Neoznačují úroveň, správce systému je může využívat k popisu akcí, které nesouvisejí s přechodem na jinou úroveň, mohou být provedeny v rámci úrovně jako doplněk. Na základě

```
# init a
```

tedy **init** prochází `/etc/inittab` a provádí akce označené v úrovni znakem `a`. Žádný proces implicitně není zastaven.

Uvedme si jednoduchý příklad tabulky `/etc/inittab`:

```
ck::sysinit:/etc/setclock </dev/console >/dev/console 2>&1
is:2:initdefault:
r0:056:wait:/etc/rc0 1> /dev/console 2>&1 </dev/console
r1:1:wait:/etc/rc1 1> /dev/console 2>&1 </dev/console
r2:23:wait:/etc/rc2 1> /dev/console 2>&1 </dev/console
r3:3:wait:/etc/rc3 1> /dev/console 2>&1 </dev/console
co:12345:respawn:/etc/getty console console
wt:123:wait:sleep 1
v1:23:respawn:/etc/getty /dev/vt01 vt01
v2:23:respawn:/etc/getty /dev/vt02 vt02
tt1a:23:respawn:/etc/getty /dev/tty1a 9600
```

V ukázce je na prvním řádku `způsob_akce=sysinit`, což je akce prováděná před zpřístupněním konzoly jako terminálu. Zde je určena pro nastavení data a času. Pomocí `initdefault` stanovujeme úroveň 2 jako úroveň, do níž má **init** systém nastavit po svém startu. Poslední čtyři akce se vztahují k terminálům a akce ve středu příkladu se vztahují na přechod na jednotlivé úrovně 1, 2 a 3, kdy je při přechodu interpretován odpovídající scénář `/etc/rc?` (dříve komentovaný scénář `/etc/rc` bývá vyvoláván na závěr scénáře `/etc/rc2` nebo jiného, týkajícího se přechodu do víceuživatelského režimu).

Příkaz `who (1)` můžeme používat také jako zdroj informací o stavu systému. Příkaz

```
$ who -r
```

vypisuje na standardní výstup označení aktuálního stavu systému podle `/etc/inittab`, zajímá-li nás čas posledního startu systému, používáme volbu `-b`:

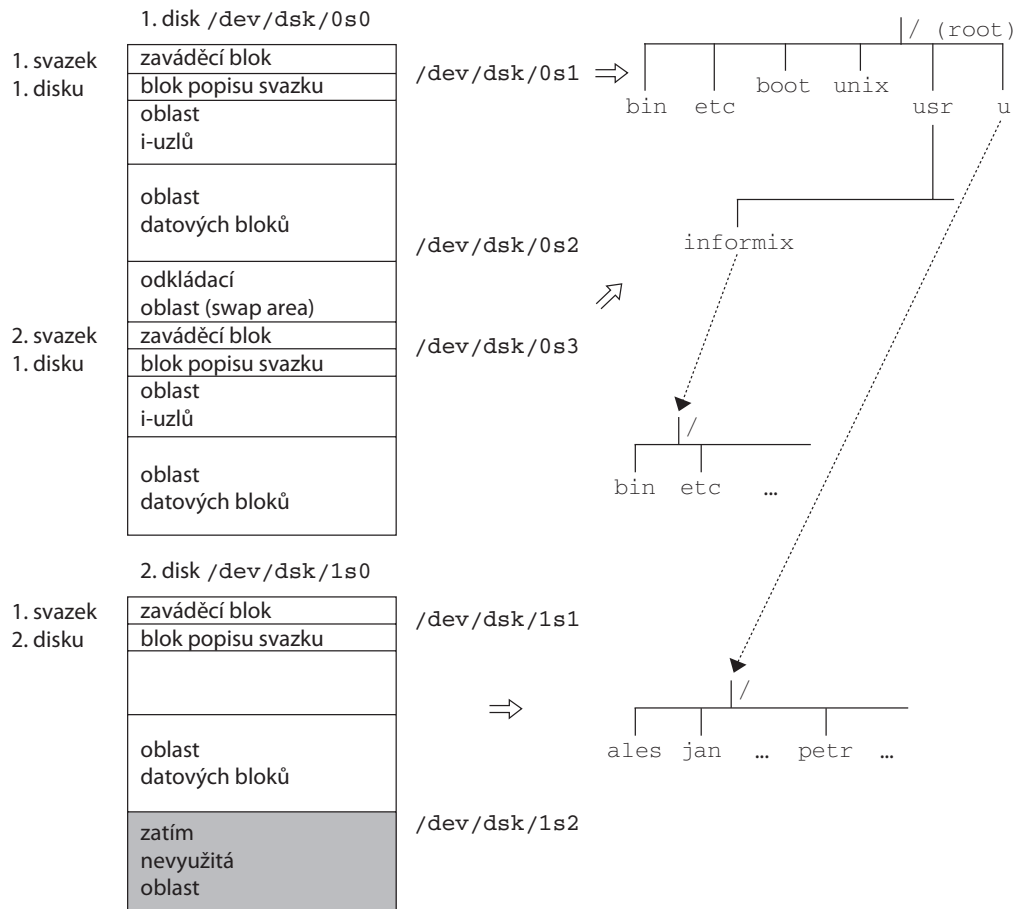
```
$ who -b
```

Důležitý a správcem systému hodně používaný příkaz je `ps (1)`. Vypisuje seznam běžících procesů. Volba `-e`

```
# ps -e
```

zajistí výpis všech procesů, které běží v systému. Zajímavé jsou volby `-f` a `-l`, které zajišťují výpis atributů procesů. Volba `-t` následovaná jménem speciálního souboru terminálu zajistí výpis pouze těch procesů, které se vztahují k danému terminálu a `-u` všech procesů, které odpovídají sezení uživatele, jehož jméno následuje za volbou.

### 9.3 SVAZKY, ÚDRŽBA, OPRAVY



Obr. 9.4 Příklad svazků a jejich obsahů

Podle obr. 9.2 může být systém souborů realizován na několika svazcích, jak ukazuje obr. 9.4, kde na prvním disku (disk č. 0) je vytvořen svazek pro zavedení systému (systémový svazek – svazek root), odkládací oblast a svazek pro databázi informix. Na druhém disku (disk č. 1) je vytvořen svazek uživatelů, přitom pro něj není využit celý disk. Logické spojení svazků v jeden systém souborů je zobrazeno pomocí šipky z přerušované čáry. Vzhledem k tomu, že je svazek root připojen po spuštění systému, propojení dalších svazků uvedeným způsobem zadává superuživatel sekvencí příkazů

```
# mount /dev/dsk/0s3 /usr/informix
# mount /dev/dsk/1s1 /usr/u
```

což může být také doplňkem souboru `/etc/rc`. Uvedené obsazení disků lze popsat v tabulce `/etc/vfstab` (nebo `/etc/fstab`) takto:

```
/dev/dsk/0s3 /usr/informix
/dev/dsk/1s1 /usr/u
```

a umožňuje superuživateli připojit svazky jediným příkazem

```
# mountall
(mount -a ve starších systémech)
```

Tabulka `/etc/vfstab` je podle SVID3 používána zejména pro definici způsobu připojení svazku, jsou-li informace v příkazovém řádku nedostačující. `vfstab` má totiž širší formát a můžeme v ní definovat také např. typ svazku (FS5, FFS, ...). Volbu `-a` ani příkaz `mountall(8)` SVID3 nezahrnuje.

Na svazcích jsou uložena všechna uživatelská i systémová data. Běh operačního systému a práce procesů jsou nositeli změn v obsahu svazků. Konzistentní obsah a struktura svazků je proto nezbytná pro korektní chování operačního systému. Ve čl. 2.4 jsme hovořili o principu systémové vyrovnávací paměti, pomocí níž virtualizujeme připojené svazky, a zmínili jsme se o problémech, které nastanou při ztrátě jejího obsahu způsobenou havárií systému. Z kap. 2 také vyplynulo použití volání jádra `sync(2)` k aktualizaci změněných dat ve vyrovnávací paměti na připojených svazcích. Odpovídající příkaz téhož významu je

```
$ sync
```

a démon, který ve víceuživatelském režimu tuto aktualizaci provádí každých 30 vteřin, nese jméno **update**. V jednouchvatelském režimu **update** není spuštěn, proto si musí superuživatel při důležité změně v datech aktualizaci zajistit sám příkazem `sync(1)`.

K nekonzistenci dat na svazku může dojít při havárii systému, odpojením od el. proudu bez korektního ukončení práce operačního systému nebo nevhodným používáním některých systémových programů, které obcházejí vyrovnávací paměť při práci se svazky.

Na obr. 9.4 jsme disk č. 0 rozdělili na 2 svazky a odkládací oblast. Oddělením dat systému a dat databáze tím, že je uložíme na různé svazky, snižujeme riziko ztráty dat současně

v obou částech disku. Při porušení struktury svazku se totiž může chyba projevit v postupném narůstání problémů při nové alokaci dat. Proto se u velkých disků doporučuje rozdělení do logických celků využitím několika svazků.

Program `fsck(8)` (`filesystem check`) provádí kontrolu a opravu svazků. Při použití v argumentu určujeme speciální soubor svazku. Speciální soubor může být vynechán, `fsck(8)` pak pracuje postupně nad všemi svazky podle tabulky `/etc/checklist` (odkládací oblast neprochází kontrolou `fsck(2)`, protože nemá strukturu svazku). V některých starších systémech je určující obsah tabulky `/etc/fstab` a kontrola všech v ní popsaných svazků se provede pomocí `fsck -a`. Vzhledem k problémům, které mohou nastat při opravě svazku vzhledem k jeho konzistenci s obsahem vyrovnávací paměti, má smysl používat `fsck(8)` pouze na odpojené svazky.

Spustíme-li `fsck(8)` nad svazkem, kde nebude objevena žádná závada, statistika bude mít tvar podle příkladu:

```
# fsck /dev/dsk/1s1

/dev/dsk/1s1
** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Count
** Phase 5 - Check Free List
1451 files 25572 blocks 2678 free
#
```

Řádky označené `** Phase` jsou komentářem o způsobu právě probíhající kontroly.

`** Phase 1` je kontrola alokovaných datových bloků podle informací uložených ve všech obsazených i-uzlech svazku. Dále je obsah i-uzlů kontrolován na typ, na nenulový počet odkazů na každý i-uzel atd.

`** Phase 2` kontrolu

je smysluplnost cest k souborům. Např. když neexistuje odpovídající i-uzel souboru nebo adresáře, přestože je na něj odkaz.

`** Phase 3` je kontrola návaznosti. Došlo-li k nestandardnímu zrušení adresáře, rozpadá se jím začínající podstrom. Tato fáze obsazených a současně nereferencovaných i-uzlů buď data uvolňuje do seznamu volné paměti, nebo je v případě smysluplnosti svazuje s adresářem `lost+found`, který je umístěn na úrovni začátku svazku (je v adresáři, kterým svazek začíná).

`** Phase 4` je kontrola počtu odkazů na i-uzel, porovnává se skutečný počet odkazů s hodnotou v i-uzlu.

`** Phase 5` se věnuje volné oblasti svazku. Porovnává seznam volných bloků se seznamem obsazených a požaduje disjunktnost těchto dvou množin bloků. Současně vyhledává bloky, které nejsou obsaženy v žádné množině, a připojuje je do seznamu volných bloků.

Poslední řádek výpisu je zpráva o celkové kapacitě a obsazení svazku.

### 9.3 Údržba

`fsck(8)` je interaktivní program. Nalezne-li v průběhu některé z uvedených kontrol nějaký nedostatek, před vlastní opravou svazku vypisuje informaci o chybě a k provedení opravy vyžaduje souhlas uživatele. Odpověď je buď `y(es)` nebo `n(o)`. Volbou `-y`

```
fsck -y ...
```

stanovujeme explicitně vykonání všech oprav, `fsck(8)` informace o chybě sice vypisuje, pro potvrzení opravy však sám sobě odpovídá `yes`. Obdobně

```
fsck -n ...
```

je možnost kontroly svazku bez oprav i v případě kolize.

Jsou případy, kdy `fsck(8)` na opravu svazku nestačí – předpokládá totiž alespoň určitým způsobem standardní strukturu svazku. Při ztrátě např. obsahu bloku popisu svazku `fsck(8)` upozorní na vážné problémy a navrhne raději svazek neopravovat. V části (5) svazku dokumentace `man` je uvedena postupně pod hesly `fs(5)`, `ino(5)`, ... struktura svazku. Zkušený a znalý systémový programátor proto dokáže i ze zhrouteného svazku kritická data získat. Představu o svazku získá výpisem obsahu speciálního souboru, např.:

```
# od -c /dev/rdisk/ls1 | more
```

a určitou část svazku může uložit do binárního souboru pomocí programu `dd(1)`, např.

```
# dd if=/dev/rdisk/ls1 of=/disk11 skip=1 count=1
```

získá do souboru `disk11` v kořenovém adresáři obsah bloku popisu svazku. Tento binární soubor lze zobrazovat (pomocí `od(1)`), ale také opravit (např. pomocí `adb(1)` – viz kap. 6), a příkazem

```
# dd if=/disk11 of=/dev/rdisk/ls1 seek=1 count=1
```

opět vsunout na patřičné místo na svazku.

Program `dd(1)`, se kterým se ještě v dalším textu setkáme, je určen pro přenos a konverzi dat souborů nezávisle na jejich typu. Má základní tvar

```
dd [volba=hodnota]
```

kde pomocí *volba* určujeme různé požadavky. Např.

volba=hodnota	význam
<code>if=ifile</code>	určení jména vstupního souboru jako <i>ifile</i>
<code>of=ofile</code>	jméno výstupního souboru bude <i>ofile</i>
<code>bs=n</code>	velikost přenosového bloku je <i>n</i> slabik
<code>skip=n</code>	před vlastní kopií je vynecháno prvních <i>n</i> bloků



<code>seek=n</code>	data jsou přenášena až po vynechání prvních <i>n</i> bloků výstupního souboru
<code>count=n</code>	je přenášeno pouze <i>n</i> bloků
<code>conv=way</code>	je provedena také konverze dat způsobem <i>way</i> ; je-li <i>way</i> např.
<code>ebcdic</code>	je konverze prováděna z ASCII interpretace do EBCDIC
<code>ascii</code>	opačně
<code>lcase</code>	velká písmena jsou převáděna na malá
	atd.

Po přenosu dat `dd (1)` vypíše na výstup diagnostiky (`stderr`) informaci o počtu skutečně transformovaných bloků, a to způsobem např.

```
1+0 records in
1+0 records out
```

což je informace o blocích načtených (`records in`) a zapsaných (`records out`). Počet je rozdělen na dvě části, část před znakem `+` je počet celých bloků a za znakem `+` je počet bloků, které se podařilo transformovat pouze částečně.

K opravě poškozeného svazku se také používá program `fsdb (8)`. Použití je

```
fsdb special
```

kde *special* je jméno znakového speciálního souboru odpovídajícího svazku. Pomocí `fsdb (8)` můžeme zobrazovat obsah *i*-uzlů nebo datových bloků, reference může být přitom daná absolutní adresou nebo číslem *i*-uzlu. Obsah dané lokality můžeme také měnit, a tak korigovat nedostatky struktury svazku. Nástroj je ovšem určen pro zkušené systémové programátory.

Důležité programy, které s údržbou svazků souvisejí, jsou `df (1)` a `mkfs (8)`. `df (1)` (disk free) je statistika volné oblasti připojených svazků. Např.

```
$ df
/                (/dev/hd01):    2676 blocks    2133 i-nodes
/usr/informix    (/dev/hd03):    10256 blocks    7123 i-nodes
/u               (/dev/hd11):    35891 blocks    21856 i-nodes
```

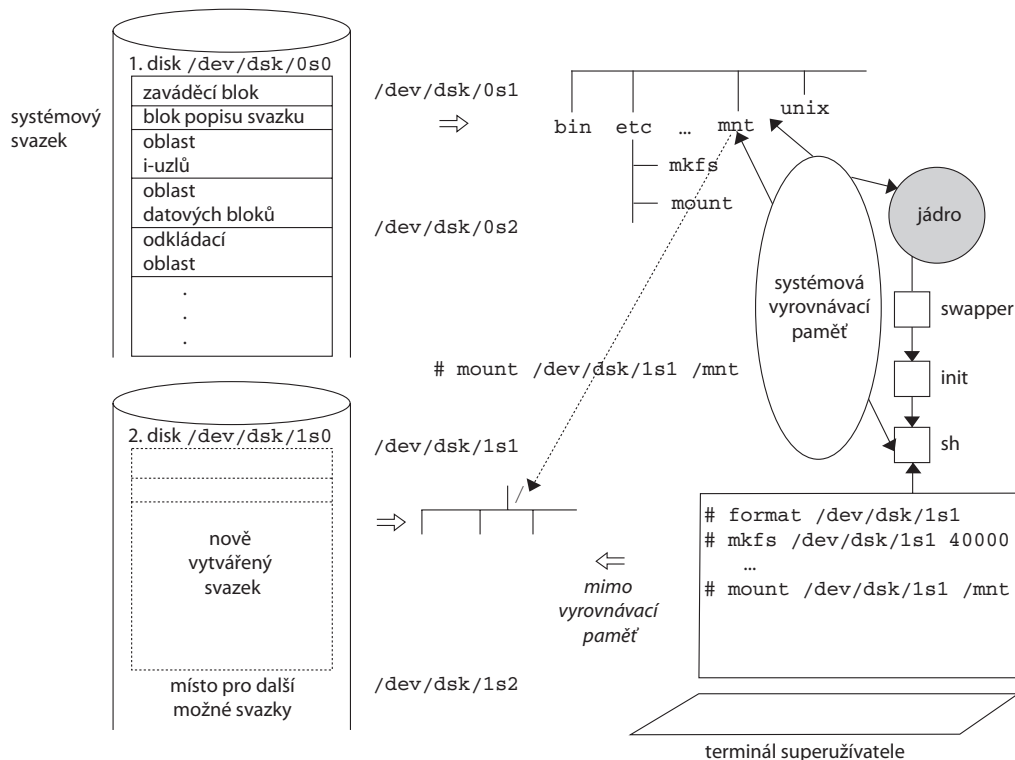
(v parametru `df (1)` můžeme určit speciální soubor svazku). Volbou `-v` získáme podrobnější informace:

```
$ df -v /dev/hd01
Mount Dir Filesystem blocks used free %used
/       /dev/hd01 28702 26026 2676 91%
```

Svazek je způsob organizace dat na trvalém magnetickém médiu s náhodným přístupem tak, jak bylo popsáno v čl. 2.3. Svazek je akceptován jádrem pro práce s daty a interpretován

### 9.3 Údržba

stromovou strukturou systému souborů a adresářů. Svazek ale musí být na médiu nějakým způsobem vytvořen, data musejí být zorganizována do prvotní podoby prázdného svazku. Za tímto účelem je superuživateli k dispozici program `mkfs` (8) (viz obr. 9.5).



Obr. 9.5 Vytvoření (`mkfs` (8)) a připojení (`mount` (8)) nového svazku

`mkfs` (8) pracuje se speciálním souborem části disku, na kterém vytváříme svazek. Jádro vynechává systémovou vyrovnávací paměť, protože svazek je teprve organizován. Teprve příkazem `mount` (8) vzniká vazba mezi systémem souborů kořenového svazku a nově vytvořeným svazkem, uživatelská data svazku po připojení procházejí vyrovnávací pamětí. `mkfs` (8) není nahrazením činnosti formátování magnetického média. Programové formátování souvisí se způsobilostí ovladače UNIXu (nebo i jiného operačního systému) pracovat s médiem jako s blokovým magnetickým zařízením s náhodným přístupem. `mkfs` (8) je organizace bloků na médiu do zvláštní struktury – svazku. Vzhledem k tomu nemá program `format` (8) v UNIXu standardní příkaz.

`mkfs` (8) můžeme použít způsobem

```
mkfs [-F fstyp] zařízení počet_bloků [:i-uzly] [díra_bloků/cyl]
      [-b velikost_bloku]
```

Na obr. 9.5 je na obrazovce terminálu použit jednoduchý případ pro vytvoření svazku na části disku speciálního souboru `/dev/dsk/ls1` (*zařízení*). Přitom je pro svazek dáno k dispozici 40 000 bloků (*počet\_bloků*). `mkfs(8)` uvažuje danou část disku a počet i-uzlů (*i-uzly*), tj. celkový možný počet souborů na svazku odvodí `mkfs(8)` implicitně z kapacity svazku. Rovněž uvažuje implicitně *velikost\_bloku* (1024B) a parametry související s otáčkami disku (*díra*, angl. *gap*) a počtem bloků na cylindr. Pro efektivnější chod systému doporučuji poslední uvedené parametry pozornosti systémových programátorů. Volba `-F` stanovuje typ svazku a nemusí být použita, vytváříme-li svazek typický pro danou implementaci.

`mkfs(8)` může mít ale i jiný způsob použití:

```
mkfs zařízení prototyp [díra bloků/cyl] [-b velikost_bloku]
```

kde *prototyp* je jméno souboru, v jehož obsahu určujeme parametry svazku, např.

```
# cat /etc/protobd
/stand/diskboot
40000 1100
d-777    3  1
$
#
```

Na prvním řádku obsahu souboru určujeme jméno souboru s programem, kterým bude naplněn zaváděcí blok svazku (jinak je v případě potřeby plněn programem `dd(1)`), na druhém řádku určujeme velikost svazku a počet i-uzlů. Na následujících řádcích můžeme stanovit výchozí strom adresářů, v našem případě stanovujeme přístupová práva výchozímu adresáři a jeho příslušnost k vlastníkovi a skupině. Tato definice je ukončena znakem `$`, můžeme v zápisu ale pokračovat a na dalších řádcích definovat vytvoření podadresářů nebo obyčejných souborů.

Kolize dat na systémovém svazku root nemusí vycházet z případu zhroucení struktury svazku. Poškození např. programu `/etc/init` nebo i `/bin/sh` může způsobit situaci, kdy nelze UNIX standardně spustit.

Pro tyto případy by měl mít správce systému k dispozici náhradní způsob zavedení operačního systému. Znamená to zálohu svazku, z něhož lze zavést a spustit operační systém. Takový svazek může být vytvořen na výměnném disku anebo na kapacitně dostačující disketě. Systémový svazek by mohl být také vytvořen na některém z dalších pevných disků, za předpokladu, že firmware, monitor technického vybavení, je schopen pak s takovým diskem pracovat jako se systémovým. Z hlediska operačního systému to znamená buď novou variantu jádra, anebo nutnost jádro při spouštění parametrizovat označením disku, který je při běhu jádra používán jako systémový. Další problémy nastávají s odkládací oblastí a dalšími funkcemi jádra, které jsou spojeny s diskem. Obvykle se tyto problémy řeší druhou uvedenou variantou – jádro je možné parametrizovat.

Disk se zálohou běhuschopného systému musí obsahovat:

- strukturu kořenového svazku vytvořenou pomocí `mkfs` (8)
- zaváděcí blok naplněný pomocí `dd` (1) programem, odpovídajícím typu disku
- základní standardní strukturu výchozího adresáře (tj. vytvořeny adresáře `bin`, `etc`, `dev`, `mnt` a `tmp`)
- soubor `/boot`, (program pro zavedení jádra) ve výchozím adresáři svazku a jádro `/unix`
- v adresáři `dev` pomocí `mknod` (8) vytvořeny všechny potřebné speciální soubory periférií
- všechny programy potřebné pro chod systému v adresářích `bin` a `etc`, zejména `/etc/init`, `/bin/sh`, `/etc/mount`, `/etc/umount`, `/etc/fsck`, `/bin/ls`, `/bin/pwd` atd.

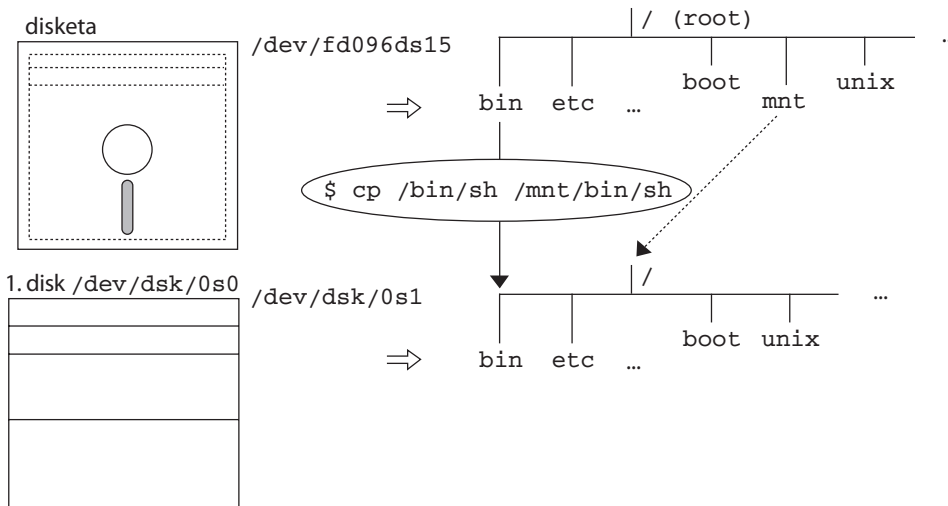
Uvedme si jednoduchý příklad vytvoření systémového svazku na disketě v prostředí operačního systému SCO XENIX/286, tedy v oblasti mikropočítačů:

```
( # format /dev/rfd096ds15 )
# mkfs /dev/fd096ds15 1122
isize = 272
m/n = 3 500
# dd if=/etc/fd96boot0 of=/dev/fd096ds15 count=1
0+1 records in
0+1 records out
# mount /dev/fd096ds15 /mnt
# cd /mnt
# cp /boot /xenix .
# mkdir bin etc dev mnt tmp
# cd /bin
# cp sh sync fsck ls pwd cp rm od tar dd /mnt/bin
# cd /etc
# cp init haltsys mount umount mknod mkfs passwd group /mnt/etc
# cd /mnt/dev
# mknod console c 3 1
# mknod clock c 8 0
# mknod cmos c 7 0
# mknod fd0 b 2 52
# ln fd0 root
# mknod hd00 b 1 0 ;: diský v SCO systemech
# mknod hd01 b 1 40
# mknod kmem c 4 1
# mknod mem c 4 0
# mknod ram00 b 31 0
# cd /
# sync
# umount /dev/fd096ds15
```

```
# fsck /dev/rfd096ds15
/dev/rfd096ds15
** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Count
** Phase 5 - Check Free List
36 files 1066 blocks 1140 free
# haltsys
```

A zavedení operačního systému z právě vytvořené diskety:

- po zapnutí počítače z uzavřené disketové mechaniky s disketou je zavlečen do paměti za pomoci obsahu zaváděcího bloku program **boot**. Je spuštěn, ohlásí se na operátorskou konzolu



Obr. 9.6 Oprava systémového svazku z diskety

```
Boot
:
```

– píšeme

```
Boot
: fd(52) xenix root=fd(52) pipe=fd(52)
```

Jádro je zavedeno do paměti a spuštěno. Operační systém se ohlásí, proces **swapper**, **init**, **sh** jsou vytvořeny, jádro připojuje disketu jako systémový svazek, **sh(1)** vypisuje na konzolu znak #. Můžeme testovat a opravovat poškozený systémový svazek na pevném disku pomocí

`fsck(8)`. V případě, že jej `fsck(8)` prohlásí za bezchybný, můžeme jej připojit k systému souborů:

```
# /etc/mount /dev/hd01 /mnt
```

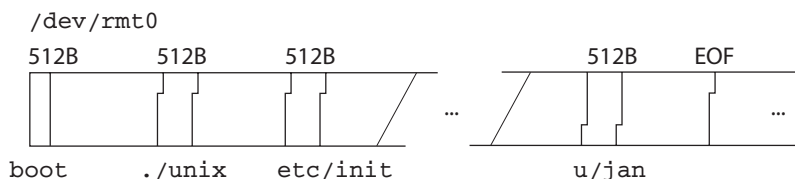
a vstupem do adresáře `/mnt` pracovat na opravě systémového svazku. Situace je zobrazena na obr. 9.6, kde je systémový svazek připojen na adresář `/mnt`, přes něj jsou dostupná všechna data porouchaného svazku. Jako příklad je uvedeno okopírování programu Bourne shell (`/bin/sh`) na systémový svazek disku.

Vytvoření zálohovaného systémového svazku je dnes už ve většině systémů automatizováno. Obvykle je to část programu údržby systému např. v SCO UNIX System V/386 s názvem **sysadmsh**, který disketu s operačním systémem dokáže vytvořit.

Pro úschovu uživatelských dat na externím médiu, např. magnetické pásce, nabízí UNIX několik možností. Nejstarší a dosud nejpoužívanější je archivace pomocí programu `tar(1)`. Provádí úschovu dat obsahu souborů včetně hlavních atributů z obsahu i-uzlu. `tar(1)` respektuje strukturu adresářů. Např.

```
# pwd
/
# tar cvfb /dev/rmt0 20 boot ./unix etc u
```

pracuje s obsahem pracovního adresáře. Hledá v něm soubory se jménem `boot`, `unix`, `etc` a `u` a jejich obsah a hlavní atributy ukládá na magnetickou pásku se speciálním souborem `/dev/rmt0`. Je-li některý z ukládaných souborů adresářem, `tar(1)` jeho obsah soubor po souboru rovněž archivuje (se jménem např. `etc/init` atd.), a to platí do libovolné hloubky vnoření adresářů. Bude-li k adresáři `u` připojen určitý svazek, `tar(1)` jeho obsah podle hierarchické struktury adresářů uloží na archivační médium.



Obr. 9.7 Způsob archivace programem `tar(1)`

Obsah takto vytvořené archivní magnetické pásky má velmi jednoduchou strukturu. Soubory jsou přeneseny vždy s označením souboru vytvořením hlavičky (o velikosti 512B), která obsahuje atributy souboru. Za hlavičkou následuje obsah souboru. Vzhledem k tomu, že hlavička také nese informaci o délce souboru, bezprostředně za poslední slabikou souboru může následovat hlavička dalšího archivovaného souboru.

V dokumentaci `man` je ve svazku (5) dostupný popis formátu `tar(5)` uložených dat. Uvedená struktura je nevýhodná ze dvou důvodů. Jednak při pouhém výpisu obsahu pásky

```
# tar -tvf /dev/rmt0
tar: blocksize = 20
r-----      3/3      24341 Apr 13 09:00 1987 boot
rw-r--r--      3/3      254995 May 15 10:30 1992 ./unix
rwx--x--x      3/3        2176 Apr 13 09:00 1987 etc/install
rwx-----      3/3        6640 Apr 13 09:00 1987 etc/haltsys
...
```

musí `tar(1)` přečíst celý obsah pásky, aby z jednotlivých hlaviček vyčetl jméno a ostatní atributy souborů. A dále jsou tato data ukládána bez jakékoli zefektivňující předchozí přípravy. Výhoda naopak spočívá v jednoduchosti uložených dat a z ní vyplývající snadné rekonstrukce souborů v případě zhoršení čitelnosti některé části archivního média.

`tar(1)` byl velmi záhy po vzniku doporučení SVID standardizován a dnes je základním programem přenosu dat mezi operačními systémy typu UNIX. Je-li médium s uloženými daty fyzicky čitelné v mechanice počítače, způsob uložení se ve všech systémech shoduje.

Program `tar(1)` vytváří (volba `c`) na médiu archivaci skupiny souborů, kterou označme jako balík dat. Balík dat je ukončen znakem konec souboru (na obr. 9.7 označen jako EOF). Zbytek média zůstává nevyužitý. Použijeme-li ale způsob

```
# tar cvfb /dev/nrmt0 20 boot ./unix etc u
# tar cvfb /dev/rmt0 20 usr/informix
```

můžeme vytvořit dva balíky dat na jedné magnetické pásce. Práce s balíky dat je dána typem archivního média. Magnetická páska mívá obvykle několik speciálních souborů, lišících se vedlejším číslem a ve jménu předponou `n`. Např. uvedený soubor `/dev/rmt0` má ekvivalent se jménem `/dev/nrmt0`. Předpona má význam no rewind, bez převinutí. Po přečtení balíku dat z pásky nechává ovladač magnetickou pásku v místě konce balíku dat, takže nové čtení pásky se vztahuje k následujícímu balíku dat. K manipulaci s páskou v rámci balíků dat obvykle slouží programy, které umí pásku nastavit na daný balík (např. `mt(8)`).

Obsah balíku dat nemusí mít formát `tar(5)`. Jiný způsob vytvoření archivu dat je pomocí programu `cpio(1)`.

`cpio(1)` má tři varianty použití, které určujeme volbou.

```
cpio -o
```

je první varianta a znamená vytvoření archivu. Archiv je předáván na standardní výstup, proto jej přesměrujeme, např.

```
... cpio -o > /dev/fd0
```

Podobně jako `tar(1)`, i `cpio(1)` respektuje strom adresářů. Vytváří archiv podle seznamu jmen souborů na standardním vstupu (jeden řádek, jedno jméno souboru). Např.

```
$ ls | cpio -o >/dev/fd0
```

### 9.3 Údržba

archivuje obsah pracovního adresáře (bez obsahu podadresářů!!) na disketu. Napíšeme-li např.

```
# find /u -mtime -7 -print | cpio -o >/dev/dsk/ls2
```

`find(1)` hledá všechny soubory od adresáře `/u`, jejichž obsah byl změněn v posledních 7 dnech. Archiv je vytvořen do nevyužité oblasti druhého disku podle obr. 9.2.

Druhá varianta

```
cpio -i
```

pracuje pro vyjmutí z archivu. Přitom může za volbou `-i` následovat seznam jmen souborů, které budou v archivu hledány. Není-li uvedeno žádné jméno souboru, obsah archivu je restaurován tak, jak byl vytvořen celý. I zde `cpio(1)` pracuje nad standardním vstupem, např.

```
$ cpio -i /u/petr/text < /dev/dsk/ls2
```

bude v archivu vyhledáván soubor s úplným jménem `/u/petr/text` a uložen na odpovídajícím místě stromu adresářů. Bude-li jméno souboru zapsáno při vytváření archivu relativní cestou (např. `cd /;find u ...`), při kopii z archivu bude `cpio(1)` vycházet z pracovního adresáře. Podadresáře, které na disku neexistují, na rozdíl od programu `tar(1)`, `cpio(1)` neumí implicitně vytvořit, explicitně tuto možnost určíme volbou `-d` (`cpio -id ...`). `cpio(1)` také nepřepisuje při vybírání z archivu existující obvyčejné soubory téhož jména (`tar(1)` ano). I to je možné nařídit volbou `-u` (`cpio -iu ...`). O tom, zda byl archiv vytvořen s relativními nebo absolutními jmény souborů, se můžeme přesvědčit výpisem obsahu archivu, např.

```
$ cpio -it </dev/dsk/ls2
```

Chceme-li přitom ve výpisu kromě jmen souborů také jejich základní atributy, přidáme volbu `-v`:

```
$ cpio -itv </dev/dsk/ls2
```

Volba `-v` je doplňující, při vytváření (nebo vybírání z) archivu zajistí výpis seznamu přenášených souborů, volba `-t` je dominující, provádí pouze výpis obsahu archivu a dá se použít pouze v kombinaci s `-i`.

Třetí varianta

```
cpio -p directory
```

je pouhá kopie souborů daných seznamem do adresáře *directory*. Pomocí



```
$ ls | cpio -p ../zaloha
```

okopírujeme všechny obyčejné soubory pracovního adresáře do adresáře `zaloha`. Adresář `zaloha` musí existovat a

```
# cd /
# find u/jan u/petr -print | cpio -pdu /zaloha
```

provede úschovu dat adresářů `/u/petr` a `/u/jan` do adresáře `/zaloha` (na který může být např. po instalaci svazku připojena nevyužitá část druhého disku `/dev/dsk/ls2` z obr. 9.2).

Formát archivovaných dat pomocí `cpio(1)` nemá obecnou možnost přenosu dat mezi různými implementacemi UNIXu. Zatímco `tar(1)` je určen pro úschovu (a přenos) lokálních dat uživatele, `cpio(1)` má charakter archivačního prostředku úschovy dat globálně. Pro

snadnější manipulaci je za pomoci `cpio(1)` v některých systémech naprogramován scénář `backup(8)`, který s magnetickou páskou umí pracovat ve smyslu doplnění archivu na základě změn od poslední úschovy. Principiálně lze provádět úschovu dat na deseti úrovních (0–9), přitom úroveň 0 je záznam všech dat (viz obr. 9.8).

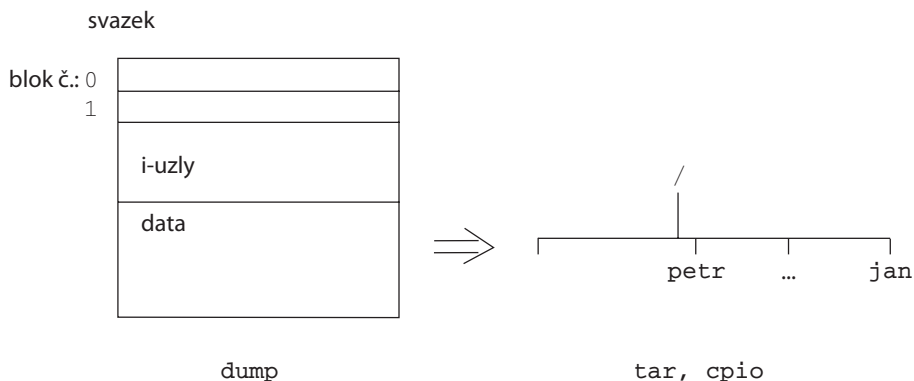
magnetická páska

			...
úroveň 0	úroveň 1, rozšíření o změny vzhledem k úrovni 0	úroveň 2, rozšíření o změny vzhledem k úrovni 0 a 1	...

Obr. 9.8 Program `backup(8)` nebo `dump(8)`

Např. denní používání přináší úschovu dat celého týdne včetně dostupnosti např. dat středních, které byly na svazcích ve čtvrtěk zrušeny.

Program opačný k `backup(8)` je scénář `restore(8)`.



Obr. 9.9 Úroveň archivace `dump(8)` a `tar(1)`, `cpio(1)`

Uvedená myšlenka zálohy dat v `backup(8)` je převzata ze syntaxe používání programů `dump(8)` a `restor(8)`. Byly původním vybavením UNIX version 7 pro minipočítače a dodnes jsou v mnohých verzích používány. `dump(8)` nepracuje se systémem souborů z pohledu uživatele, ale z pohledu jádra, tj. vnitřní struktury svazku. V příkladu

```
# dump -0 /dev/rk0
```

povinný argument (`/dev/rk0`) určuje speciální soubor média, na kterém bude `dump(8)` předpokládat strukturu svazku a který bude archivovat na magnetickou pásku. Archivační médium v příkladu je stanoveno implicitně, použitím volby `-f`

```
# dump -0f /dev/rmt0 /dev/rk0
```

je ale možné vynutit zápis explicitně. Použitá volba `-0` je stanovením úrovně úschovy, jejíž filozofie odpovídá popisu u `backup(8)`. Archiv bude vytvořen se všemi informacemi potřebnými pro úplnou rekonstrukci svazku včetně momentálního stavu vnitřní struktury (blok popisu svazku, každý soubor bude mít uschován i-uzel odpovídajícího čísla, je uschována alokace datových bloků atd.).

`restor(8)` se používá komplementárně, např.

```
# restor -rf3 /dev/rmt0 /dev/rk0
```

za pomoci volby `-r` z archivu na magnetické pásce `/dev/rmt0` třetí úrovně vytvoří dříve archivovaný svazek, a to na disku `/dev/rk0`. Při tomto použití je původní obsah média `/dev/rk0` ztracen. `restor(8)` dokonce nevyžaduje vytvoření svazku pomocí `mkfs(8)` na médiu, kam bude svazek nově vytvářet. Z archivu lze pomocí `restor(8)` vyjmout také pouze některé soubory, a to pomocí volby `-x`, např.

```
# restor -x petr/text jan/ztext /dev/rk0
```

Po volbě `-x` následuje seznam jmen požadovaných souborů. Aby nedošlo ke kolizi, při tomto nestandardním použití jsou jména souborů přejmenována na odpovídající čísla i-uzlů v archivu.

Úschova a obnova svazků musí být prováděna na médiu, které není evidováno v seznamu připojených svazků. Po ukončení obnovy se doporučuje použít pro kontrolu `fsck(8)`.

U systémového svazku dochází ke stejné situaci jako u kontroly pomocí `fsck(8)` (viz první část článku). Je proto nutné po obnovení systémového svazku vypnout počítač bez provedení `sync(2)`. `restor(8)` bývá také dostupný ve verzi samostatných (standalone) programů.

### 9.4 INSTALACE

Instalace operačního systému UNIX je soubor činností, jejichž cílem je za pomoci přenosného (distribučního) média vytvořit na technickém vybavení počítače systémový svazek se všemi daty tak, aby byl UNIX schopen běžného provozu. Přenosné médium musí obsahovat data systémového svazku a prostředky pro jejich přenos na technické vybavení.

Obsah přenosného média (distribuce) musí odpovídat typu počítače.

Je zřejmé, že instalace bude poplatná typu počítače. UNIX lze provozovat na osobním počítači typu IBM PC/AT, na počítačích střední třídy typu VAX Digital Equipment Corp. nebo na střediskových počítačích např. CRAY. Instalace v uvedených kategoriích bude zřejmě odlišná, ale nikoliv principiálně jiná.

V článku vysvětlíme obecné principy instalace a důvody, proč tomu tak je. Přizpůsobení provedené instalace odpovídající konfiguraci souvisí s regenerací (znovuvytvořením) instalovaného jádra, která bude předmětem čl. 9.5. Výpočetní systém je totiž navržen a konfigurován podle typu a počtu provozovaných aplikací (informačního systému), které bude výpočetní systém provozovat a ke kterým je vybrán odpovídající typ počítače v odpovídající konfiguraci technického vybavení a operačního systému.

UNIX je distribuován na

- magnetické pásce
- disketách
- disketě a magnetické pásce
- výměnném disku a magnetické pásce

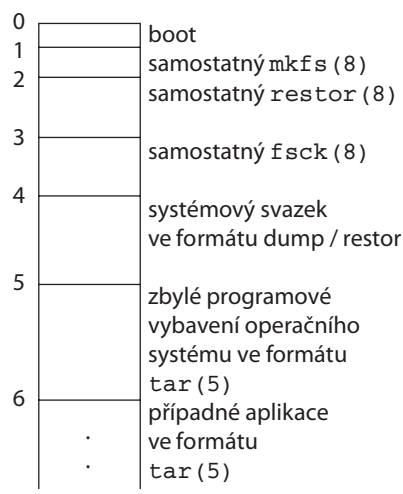
Strukturu distribuční magnetické pásky ukazuje obr. 9.10.

Tento způsob instalace je vhodný pro počítač s vybavením firmware, které akceptuje z operátorské konzoly pokyn pro zavedení samostatného programu také z magnetické pásky. Strojově závislý program **boot** uložený jako první v pořadí na pásce představuje analogii programu uloženého v zaváděcím bloku systémového disku. Po zavedení do paměti a spuštění na konzolu vypisuje (systémy BSD):

```
mtBoot
:
```

kdy dále očekává pokyny operátora. **boot** umí zpracovat odkaz na soubor pásky, jehož obsah uvažuje jako proveditelný program. Obsah souboru zavede do operační paměti a předá mu řízení. Další v pořadí za programem **boot** je soubor se samostatným programem **mkfs (8)**. Operátor provádějící instalaci jej použije pro vytvoření systému souborů, který bude později plnit funkci systémového svazku. Komunikace může mít např. tento tvar:

```
mtBoot
:tm(0,1)
mkfs
file system : rp(0,0)
file system size : 30000
```



Obr. 9.10 Instalační magnetická páska

kde `tm(0,0)` je odkaz na druhý soubor první magnetické pásky (UNIX počítá od 0), `rp(0,0)` operátor určuje zařízení pro systémový svazek (označení periferie, `rp` jako jeden z typu disků, je dáno instalační dokumentací) a číslem 30000 velikost svazku v KB.

Každý samostatný program interaktivně spolupracuje s operátorem. `mkfs(8)` si po vyžádání informací, potřebných pro vytvoření svazku, vytvoří svazek na odpovídajícím zařízení. Po skončení činnosti předá opět řízení programu **boot**. Odkazem např.

```
: tm(0,2)
```

zpřístupňujeme komunikaci se samostatným programem `restor(8)`, který pátý soubor pásky (`tm(0,4)`) uvažuje jako archivní záznam systémového svazku. Obnoví jej na daný disk.

`restor(8)` obvykle neumí naplnit zaváděcí blok. Vytvořený systémový svazek (jehož strukturu jsme mohli kontrolovat pomocí samostatného programu `fsck(8)` viz `tm(0,3)`) sám ještě nemá schopnost zaveditelnosti operačního systému. Tu ale má magnetická páska. Pomocí **boot**:

```
: rp(0,0)/unix
```

provádíme odkaz na svazek obsahující soubor `unix` v kořenovém adresáři. Jádro `/unix` je zavedeno do paměti a je mu předáno řízení.

Pro korektní dokončení instalace základního operačního systému je ještě zapotřebí:

- naplnit zaváděcí blok systémového svazku typu disku odpovídajícím programem `boot`
- nastavit parametry reálného času, tj. časovou zónu, automatické přepínání zimního a letního času
- vytvořit heslo privilegovaného uživatele

Po prvním zavedení jádra nově instalovaného systémového svazku probíhá obvykle automatizované pokračování instalace ve smyslu uvedených úprav. Zaváděcí blok je naplněn odpovídajícím samostatným programem pomocí `dd(1)`. Jsou vyžádány a v tabulkách evidovány informace o časové zóně, datum změny letního času na zimní.

Později můžeme s formátem výpisu data a času pracovat pomocí `zic(8)`. Stejně tak je interaktivně vyžádáno heslo superuživatele. Pokud tomu tak není, v rámci instalační dokumentace se doporučuje pro vytvoření nebo i pozdější změnu hesla použít příkaz `passwd(1)`, což je prostředek pro vytvoření nebo změnu hesla kteréhokoli uživatele. Použití je

```
passwd jméno
```

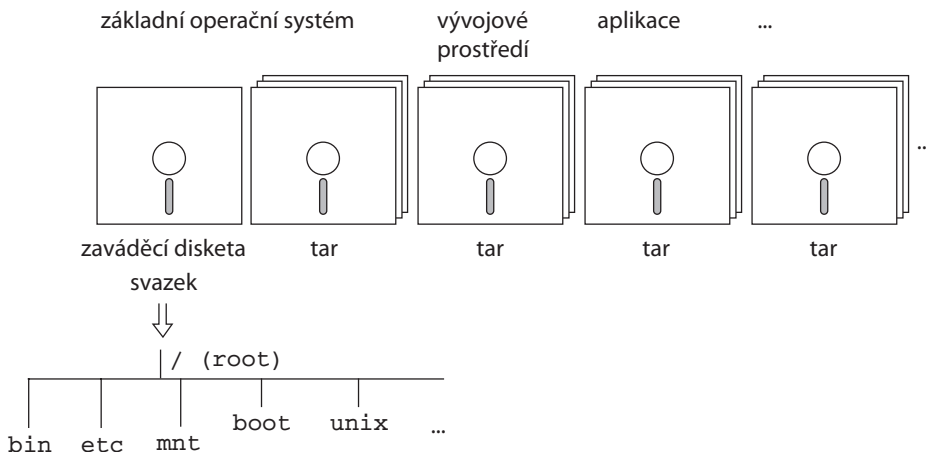
kde `jméno` je označení uživatele. Lze tedy měnit heslo i jinému uživateli než sobě, pokud k tomu máme oprávnění. Při vytváření hesla je uživatel po spuštění programu požádán o zápis hesla na klávesnici, heslo se na obrazovku neopisuje. V případě změny hesla je

vyžádáno nejprve doposud platné heslo. Nové heslo je vzápětí pro porovnání vyžádáno znovu. Např.:

```
# passwd root
Enter new password (minimum of 5 characters)
Please use a combination of upper and lowercase letters and numbers
New password:
Re-enter new password:
#
```

Po těchto prvotních a nutných úpravách systémového svazku je základní operační systém připraven k běžnému provozu. Soubor pásky označený jako **Zbylé programové vybavení operačního systému** obsahuje část nazývanou **Vývojové prostředí** (Development System), tj. ostatní standardní nástroje programátora, překladač jazyka C atd. Jeho instalace je provedena také automatizovaně pomocí instalačního scénáře pro shell. Instalační scénář je součástí archivu vývojové prostředí a je vyjmut za pomoci `tar(1)` speciálním prostředkem (např. programem `pkgadd(8)`, nebo dříve `installpkg(8)`), který jej pak interpretuje pomocí `sh(1)`. Obdobným způsobem je také instalován každý dodatečný programový celek umístěný jako následující soubory pásky nebo jako soubor jiné (další) pásky.

Instalační prostředek distribuovaného programového vybavení umí programy instalovat nebo vyjímát ze systému, současně vede evidenci o všech instalovaných produktech a dává tyto informace k dispozici, a to i na úrovni výpisu všech souborů, které s programovým vybavením souvisejí. Příkladem instalačního prostředku může být např. sada programů pro UNIX SYSTEM V: `installpkg(8)`, `deletepkg(8)`, `listpkg(8)`, nebo nově podle SVID3 definovaná sada programů `pkgadd(8)`, `pkgask(8)`, `pkgchk(8)`, `pkgrm(8)`, `pkginfo(8)`, `pkgtrans(8)` (viz Příloha F).



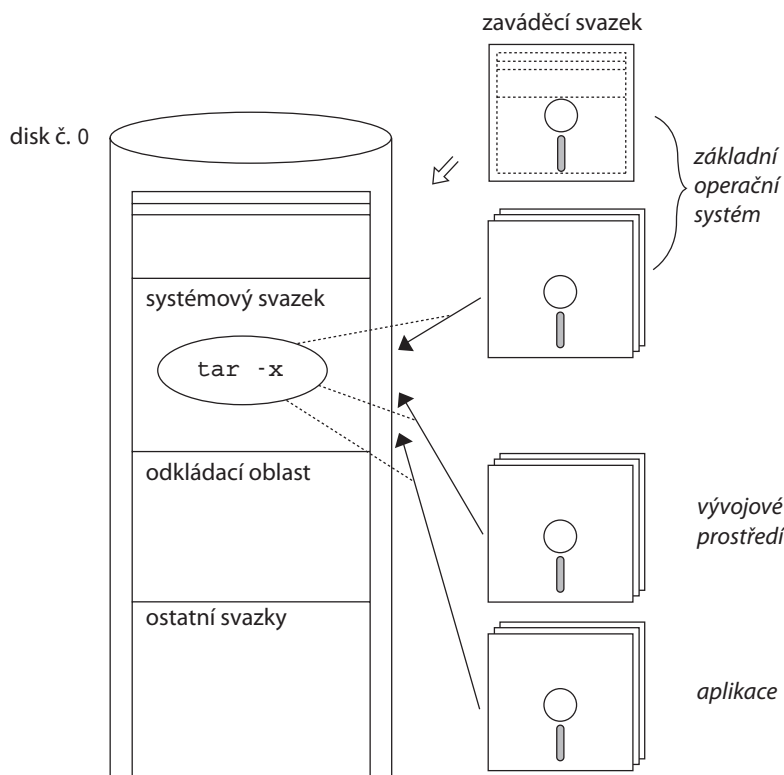
Obr. 9.11 Instalační sada disket

Způsobu distribuce na disketách se používá především v oblasti mikropočítačů (SCO UNIX, Interactive UNIX). Kupující obdrží větší množství disket, jejichž obsah je rozdělen podle schématu **Základní operační systém, Vývojové prostředí, Aplikace** (obr. 9.11). Obsah disket je uložen ve formátu `tar(5)` (nebo `cpio(5)`) vyjma první diskety **Základního operačního systému**. Bývá označena jako zaváděcí disketa (boot disk) a má formát systémového svazku. Instalace probíhá založením zaváděcí diskety do mechaniky počítače, odkud je možné zavést a startovat libovolný operační systém (na PC hlavně MS DOS). Jádro uzpůsobené pro práci se systémovým svazkem na disketě je spuštěno a je interpretován instalační scénář shellu. Za jeho pomoci v interakci s operátorem je inicializován disk, na který bude UNIX instalován a z něhož bude při běžném provozu zaváděn do paměti a spouštěn (obvykle disk č. 0). Je stanovena velikost systémového svazku, odkládací oblasti a případné další části pro další svazek. Instalační scénář rozdělí disk, vytvoří svazky (pomocí `mkfs(8)`), vybuduje hierarchickou strukturu stromu adresářů systémového svazku (pomocí `mkdir(1)`), okopíruje z diskety jádro, program boot a další základní prostředky programátora, naplní zaváděcí blok odpovídajícím programem. Poté scénář zastaví operační systém a vyzve operátora ke spuštění jádra z nově inicializovaného disku. Disketu vyjmeme z mechaniky a zavedeme a spustíme UNIX. Instalační scénář má své pokračování na novém systémovém svazku, odkud je po startu jádra interpretován a pokračuje v činnosti. Pomocí prostředku pro instalaci programového vybavení (`installpkg(8)` nebo `pkgadd(8)`), v SCO UNIX má jméno `custom(8)` je instalována zbylá část **Základního operačního systému**, přitom je možné pokračovat instalací **Vývojového prostředí** a části **Aplikace**. Metodiku instalace ukazuje obr. 9.12. Zaváděcí systémový svazek bývá někdy virtualizován ze dvou fyzických disket, obě mají formát svazku, první disketa obsahuje pouze zaváděcí mechanismus – strojově závislý program v zaváděcím bloku, soubor `/boot` a jádro `/unix`, druhá zbylé nutné vybavení prostředků programátora pro instalaci. Tento způsob se používá pro zvětšení kapacity zaváděcího svazku např. v SCO UNIX.

Způsob distribuce na disketě a magnetické pásce spojuje výhodu přímého využití zaváděcího svazku na disketě a odstraňuje nevýhodu znovuzavedení systému pro dokončení instalace, protože všechna ostatní data **Operačního systému, Vývojového prostředí a Aplikace** jsou čerpána z magnetické pásky. Odstraněna je také únavná práce s častou výměnou disket (operační systém a vývojové prostředí bývá distribuováno až na několika desítkách disket). Tento typ instalace bývá používán u počítačů střední třídy dodávaných s magnetickou páskou; příkladem může být např. distribuce WYSE UNIX. Celkově je instalace analogická předchozím případům (viz obr. 9.13).

Instalace z výměnného disku a (klasické) magnetické pásky je doménou střediskových počítačů, přestože bývá dnes používána i u některých typů pracovních stanic. Princip instalace však zůstává tentýž. Instalační disketa je zaměněna za výměnný disk, obsahující zaveditelný operační systém připravený pro běžný provoz. Instalace na jiný typ disku je prováděna odtud, přestože pracovně může být využíván upravený distribuční disk. Vytvoření dalších svazků na ostatních discích a přenos do jejich struktur balíků programového vybavení z magnetické pásky má zcela standardní průběh.

Upravený distribuční disk je pochopitelně vytvářen z kopie dodaného. Tato poznámka souvisí s běžnou rutinou vytvoření kopií všech distribučních médií. Provádí se před započítím instalace s cílem ochránit dodaný originál.



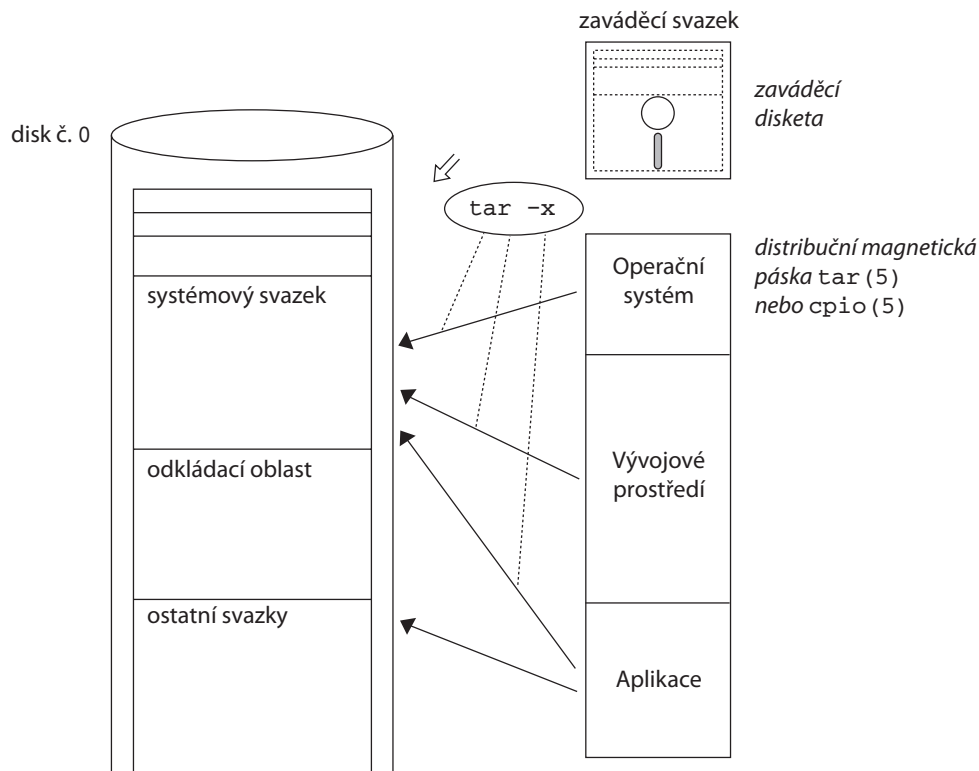
Obr. 9.12 Instalace ze sady disket

Distribuce operačního systému UNIX pro konkrétní typ počítače vždy obsahuje možnost komunikace se standardně dodávaným technickým vybavením výrobce. Ovladače všech takových periférií jsou distribuovány jako součást jádra. Při startu systému (viz čl. 9.1) jádro samo testuje konfiguraci (přítomnost různých typů a počtu periférií) a podle výsledků usměrní další chod systému. Je-li součástí konfigurace určitá speciální periferie, jejíž ovladač výrobce UNIXu nedodává v základní distribuci, je nutno ovladač vytvořit (nebo koupit) a vložit do jádra (viz čl. 9.5 a 9.6).

V rámci běžné instalace za standardní konfigurace je ještě důležitá instalace tiskárny. Zde jde pouze o nastavení typu tiskárny a způsob jejího ovládání. Programové vybavení je instalováno už dříve jako součást **Základního operačního systému**.

UNIX pro tisk dat na tiskárně podporuje klasickou metodu spooling. Uživatelé sdílejí tiskárnu tak, že pro zamezení několika smíchaných tisků slouží proces **lp sched** (proces démon, viz čl. 2.4), který přebírá od uživatelů jejich požadavky na tisk a postupně je tiskne v pořadí, v jakém přišly. Uživatel do fronty FIFO (bývá realizována pojmenovanou rourou `/usr/spool/lp/fifos/FIFO`) připojuje svůj požadavek příkazem `lp(1)`.

Služby obsluhy tiskárny (print services) jsou vždy startovány (podle `/etc/inittab`) přechodem do stavů víceuživatelského režimu. Instalace tiskárny probíhá pomocí příkazů



Obr. 9.13 Instalace z diskety a magnetické pásky

<code>/usr/lib/lpadmin</code>	pro nastavení typu a charakteristik tiskárny
<code>/usr/lib/accept</code>	připojení tiskárny k systému Služeb obsluhy tiskárny
<code>/usr/lib/lpsched</code>	je démon pro spooling Služeb obsluhy tiskárny
<code>/usr/lib/lpshut</code>	pro zastavení démona pro spooling
<code>/usr/lib/lpusers</code>	pro nastavení přístupových práv různých uživatelů
	vzhledem k využití Služeb obsluhy tiskárny

kteří modifikují obsahy tabulek (včetně `/etc/inittab`). Při běžném provozu jsou dále k dispozici příkazy

<code>/usr/lib/lpfilter</code>	což je vkládání programů pro filtrování dat směrem k tiskárně
<code>/usr/lib/lpstat</code>	vypisuje statistiku při problémech komunikace s tiskárnou
<code>/usr/lib/cancel</code>	zruší 1 nebo více požadavků pro tisk z fronty na tiskárnu
<code>lp(1) nebo lpr(1)</code>	tiskne obsah souboru na tiskárnu

Protože je možné mít připojeno více tiskáren a v současných verzích je možné přerušit styk Služeb obsluhy tiskárny na několika úrovních (tiskárna je připojena, odpojena, démon **lpsched** je zastaven, spuštěn atd.), doporučuji správci systému pro instalaci použít obrazovkové



orientovaný shell správce systému (viz úvod kap.), který v součinnosti s firemní dokumentací provede instalaci korektně, při instalaci se přitom obejdeme bez detailní znalosti různých úrovní Služeb obsluhy tiskárny.

Dalším nezbytným krokem pro běžné využití operačního systému je instalace registru uživatelů, která také nebývá součástí instalačního scénáře. V prvních komerčních systémech z počátku 80. let nebyl k dispozici žádný příkaz pro instalaci uživatele a v příručkách pro výuku UNIXu byl tento problém zadáván čtenářům k řešení za domácí úkol. Instalace uživatele je v principu opravdu sekvencí příkazů, které můžeme vložit do scénáře pro shell. S odvoláním na čl. 2.6 instalace uživatele znamená

- připojení nového řádku k souboru `/etc/passwd` s informacemi identifikujícími nového uživatele
- připsání uživatele do seznamu uživatelů patřících k vybrané skupině editací souboru `/etc/group`
- vytvoření domovského adresáře a předání jeho vlastnictví vytvářenému uživateli (příkazy `mkdir(1)` a `chown(1)` ve výchozím adresáři uživatelů)
- vytvoření poštovní schránky (mailbox) uživatele a předání jejího vlastnictví uživateli (v adresáři `/usr/spool/mail`)

Vzhledem k tomu, že správnost a bezchybnost takových akcí je diskutabilní a hlavně proto, že některé systémy registrují uživatele ještě v dalších tabulkách a konečně také proto, že podsystém elektronické pošty v síťovém provozu má složitější vazby, a tím i náročnější evidenci uživatelů, vznikly v některých současných systémech příkazy `mkuser(8)` pro zavedení a `rmuser(8)` pro zrušení uživatele. V poslední době se ale akce instalace nebo vypuštění uživatele ze systému stala součástí shellu správce systému. SVID3 definuje pro práci s registry uživatelů a skupin několik příkazů:

<code>useradd(8)</code>	vytvoří nového uživatele
<code>userdel(8)</code>	zruší uživatele
<code>usermod(8)</code>	provádí změny v registrech uživatelů
<code>groupadd(8)</code>	vytvoří novou skupinu
<code>groupdel(8)</code>	zruší skupinu
<code>groupmod(8)</code>	provádí změny v registru skupin
<code>logins(8)</code>	vypisuje informace o uživateli
<code>passwd(1)</code>	změna hesla uživatele

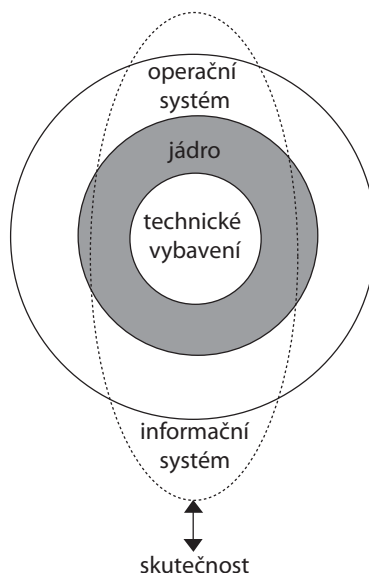
Jejich formát a volby jsou popsány v Příloze E a F.

## 9.5 GENERACE JÁDRA

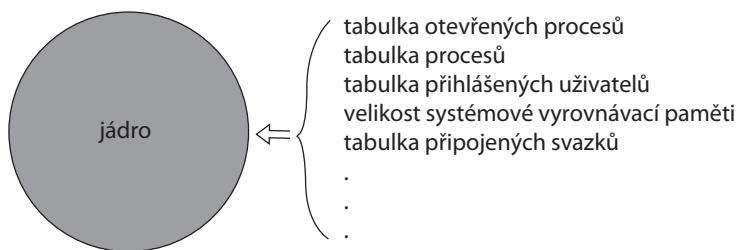
Návrh informačního systému je primární projekt pro využití modelu skutečnosti na počítači. Z něj vychází úvaha o typu a třídě výpočetního systému, který použijeme. Výpočetní systém zahrnuje operační systém a technické vybavení. UNIX na distribučním médiu je připraven v základní podobě pro základní technické vybavení použité konfigurace (je dnes běžné měnit velikost operační paměti nebo kapacity disků) vzhledem k vybranému typu počítače. Informační systém diktuje konfiguraci technického vybavení, a tím i operačního systému.

Jádro (obr. 9.15) obsahuje datové struktury, jejichž velikost souvisí s kapacitou průchodnosti operačního systému a s konkrétní konfigurací výpočetního systému.

Velikost datových struktur jádra nazýváme parametry jádra. Parametry jádra jsou pevné délky a znamenají horní hranici datových struktur při práci jádra. Změna některého parametru musí být provedena regenerací jádra. Součástí jádra jsou také ovladače periférií, které souvisejí s daným počtem různých nebo týchž periférií. Ovladače, jejich struktura a připojování k jádru budou předmětem čl. 9.6.



Obr. 9.14 Využití výpočetního systému



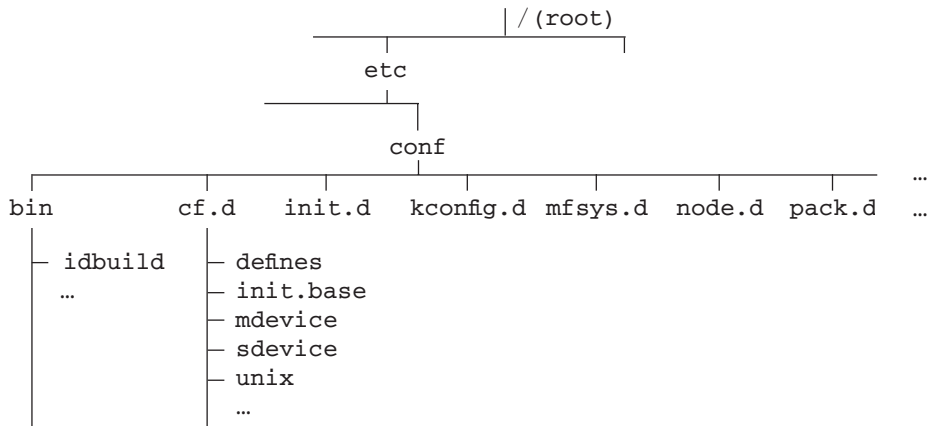
Obr. 9.15 Fragment struktur jádra

UNIX běžně umožňuje privilegovanému uživateli měnit parametry jádra. Toho je dosaženo změnou parametrů v dostupné tabulce a vytvořením nového jádra. Protože dnes už UNIX není běžně dodáván se zdrojovými texty, generace jádra není uskutečněna v podstromu zdrojových textů (`/usr/src/sys`), ale byla přesunuta do podstromu začínajícího adresářem `/etc/conf`. Struktura adresářů uvedená na obr. 9.16 a 9.17 je orientační, protože není obecně stanovena ani pevná struktura adresářů, ani způsob generace jádra.

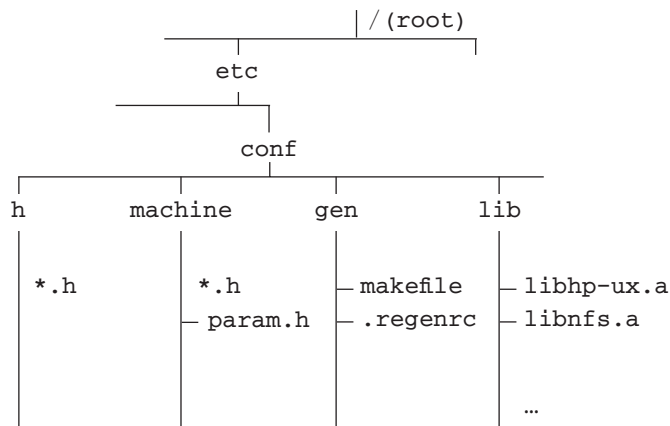
Obecný charakter přitom vychází z:

- adresářů, které obsahují knihovní moduly jádra (`/etc/conf/pack.d` nebo `/etc/conf/lib`)
- adresářů knihoven modulů jádra, ovladačů periférií a strojově závislé části

- ( /etc/conf/pack.d, /etc/conf/lib)
- ostatních adresářů, vztahujících se např. k připojování periférií ( /etc/conf/sdevice.d), popisu strojově závislé části ( /etc/conf/machine) nebo popisu struktur jádra ( /etc/conf/h) nebo adresářů pro generaci jádra ( /etc/conf/bin)



Obr. 9.16 Podstrom generace jádra UNIX SYSTEM V



Obr. 9.17 Podstrom generace jádra HP-UX

Definice parametrů jádra je obvykle uložena v adresáři pro generaci jádra ( /etc/conf/cf.d/config.h) nebo v adresáři popisu stroje ( /etc/conf/machine/param.h). Je to soubor definic, který bývá načten ve zdrojovém textu do `c.c` a využíván jako jeden z modulů pro sestavování jádra. Jádro je totiž programováno jako běžný program; sestavující prostředek `ld(1)` jej umí na základě vhodných voleb vytvořit tak, aby pracovalo na holém stroji (samostatná, standalone podoba). K vytvoření jádra dáváme pokyn např. v SCO UNIX

```
# ./link_unix
```

nebo v HP-UX

```
# make hp-ux
```

Rozdíl je v pojetí: `link_unix` je scénář shellu, `hp_ux` (jádro systému UNIX fy. HEWLETT PACKARD) je vytvořeno pomocí prostředku `make(1)` při popisu v souboru `/etc/gen/makefile`. Za příkazem `./link_unix` se skrývá scénář shellu obsahující sekvenci příkazů z adresáře `/etc/conf/bin`. Jméno scénáře `link_unix` firma SCO udržuje z historických důvodů, protože jádro v UNIX SYSTEM V se dnes vytváří pomocí sady příkazů z `/etc/conf/bin`. Některé z nich jsou

<code>id tune(8)</code>	se používá pro nastavení hodnoty parametrů jádra
<code>id config(8)</code>	pro vytvoření prostředí generace jádra ze stávajících tabulek
<code>id mkunix(8)</code>	pro sestavení jádra
<code>id build(8)</code>	zastřešuje prostředky <code>id config(8)</code> a <code>id mkunix(8)</code>

A k dispozici je správci systému dále ještě

<code>id check(8)</code>	pro kontrolu současné konfigurace.
--------------------------	------------------------------------

Uvedené příkazy nejsou součástí SVID3.

Jádro je sestaveno z nově definovaných parametrů, knihovních modulů ovladačů periferií, modulů strojově závislé části a ostatních modulů jádra (tzv. vlastní jádro). V adresáři generace jádra poté vzniká soubor se jménem `unix` (nebo `hp-ux` apod.), který je programem nového jádra. Překopírováním tohoto souboru do adresáře `/` je nové jádro připraveno pro denní provoz. Pro pocit jistoty je vždy doporučována úschova starého jádra:

```
# cd /
# mv unix unix.old
# cp /etc/conf/cf.d/unix .
# shutdown
```

Posledním příkazem zastavujeme operační systém s běžícím starým jádrem. Příští start už bude znamenat práci s novým jádrem. Dojde-li ke kolizi a je potřeba nové jádro pozměnit, zastavením systému a startem

```
Boot
: /unix.old
```

spouštíme operační systém se starým jádrem. Tímto způsobem měníme jádro ve fázi ladění jádra. Z důvodu časté výměny jádra v paměti se nedoporučuje generovat jádro ve víceuživatelském režimu.

Seznam parametrů jádra a způsob generace je vždy uveden v odpovídající části dodané firemní dokumentace.

Uvedme si ukázkou některých parametrů jádra a vazby mezi nimi:

NBUF	velikost systémové vyrovnávací paměti v diskových blocích
NFILE	největší možný počet současně otevřených souborů
NMOUNT	nejvyšší možný počet připojených svazků
NOFILES	nejvyšší možný počet otevřených souborů pro jeden proces
NPROC	maximální počet procesů
MAXUP	maximální počet procesů pro jednoho uživatele
MAXUSER	nejvyšší možný počet současně přihlášených uživatelů
MSGMAX	největší dovolená velikost předávané zprávy mezi procesy (IPC)
SEMMNI	maximální možný počet semaforů registrovaných jádrem (IPC)
SHMMAX	nejvyšší dovolená velikost sdílené paměti procesů (IPC)

Souvislost parametrů jádra je např. zřejmá u `NPROC`, `NFILE` a `NOFILES`, největší možný počet otevřených souborů souvisí s počtem otevřených souborů pro jeden proces. Součin `MAXUSER` a `MAXUP` nesmí překročit `NPROC` atd. Složitost vztahů mezi parametry jádra předpokládá její detailní studium při změně některého z parametrů, což je obvykle dobře popsáno v dodané dokumentaci výrobce operačního systému. Předmětem obchodních machinací je parametr `MAXUSER`. Jeho změnu si nechává výrobce dobře zaplatit, např. při přechodu z 32 na 64 uživatelů. Drahá je verze neomezeného počtu uživatelů vstupujících do systému. V rámci dodaného operačního systému přitom `MAXUSER` není viditelný ani měnitelný.

`SVID3` definuje příkaz výpisu parametrů jádra jako `sysdef(8)`. Dále definuje pro sledování skutečných dosažených hodnot parametrů jádra, tj. pro ladění operačního systému v konkrétním provozu prostředek `sar(8)` (`system activity reporter`). Jeho práce je podporována sadou programů uložených v adresáři `/usr/lib/sa`. Je to např. program `sa1(8)` a `sa2(8)`. Jejich spuštěním je vytvořen záznam o hodnotách parametrů jádra v daném okamžiku. Pro získání určité statistiky jsou spouštěny cyklicky pomocí démonu **cron**. `sar(8)` pak využívá získaných záznamů a dokáže uživateli zpřístupnit tabulky aktivity operačního systému za uplynulé časové období. Pro **cron** je po instalaci operačního systému naplněna tabulka `/usr/spool/cron/crontabs/sys` záznamem

```
0 * * * 0-6 /usr/lib/sa/sa1
20,40 8-17 ** 1-5 /usr/lib/sa/sa1
5 18 * * 1-5 /usr/lib/sa/sa2 -s 8:00 -e 18:01 -i 1200 -A
```

na jehož základě bude **cron** spouštět program `sa1(8)` v době od 8 do 17 hodin každých 20 minut, jinak každou hodinu. `sa2(8)` vytváří zprávu každou hodinu v rozmezí od 8 do 18 hodin.

Program `sar(8)` na základě vytvořených záznamů `sa1(8)` a `sa2(8)` podle pokynů je schopen podat zprávu o:

- aktivitě systému vyrovnávací paměti pomocí  
# `sar -b`
- zařazování do front procesů při soupeření o operační paměť

```
# sar -q
- práci procesoru
# sar -u
- stavu tabulek jádra (tabulce procesů, i-uzlů, sdílené paměti atd.)
# sar -v
- aktivitě odkládacího prostoru
# sar -w
```

Úhrnnou zprávu pro uvedené kategorie od daného dne v měsíci můžeme obdržet:

```
# cd /usr/adm/sa ;# zde jsou uloženy zaznamy sa1 a sa2
# sar -A -f sa15 ;# vypis ode dne 15. tohoto mesice
```

### 9.6 OVLADAČE PERIFERIÍ

Styk operačního systému s periferiemi zajišťují ovladače (drivers). Jsou součástí jádra, které je využívá jako sadu funkcí. Pro styk uživatele s periferiemi potom slouží speciální soubory v adresáři `/dev`. Z atributů speciálního souboru jsou hlavní a vedlejší číslo (major and minor numbers) styčné body uživatele s jádrem. Jak bylo řečeno, hlavní číslo určuje typ periferie (disk, tiskárna, magnetická páska, terminál, ...), je to tedy selekce sady funkcí ovladačů pro styk s daným typem periferie (bývá obsahem souboru např. `c.c` v adresáři pro generaci jádra ve strukturách `bdevsw` a `cdevsw`). Běžně je konfigurace výpočetního systému rozšiřována o nové typy periférií, systémový programátor má proto k dispozici volnou oblast konce tabulky seznamu periférií. Při generaci jádra je mu umožněno obsadit následující položku tabulky, a tím alokovat nové hlavní číslo. Toto hlavní číslo je dále klíčové při vytváření speciálního souboru. Speciální soubor vzniká pomocí příkazu `mknod(8)`, zaniká příkazem `rm(1)`, standardně se doporučuje speciální soubory vytvářet v adresáři `/dev`. Formát `mknod(8)` je

```
mknod name [ b | c ] major minor
```

kde *name* je jméno speciálního souboru, *major*, *minor* jsou hlavní a vedlejší číslo vytvářeného speciálního souboru. *b* nebo *c* označuje přístup k periférii. Ten je buď blokový (*b*lock) nebo znakový (*c*haracter). V případě blokového přístupu je nutné, aby ovladač měl k dispozici funkci, která optimálně provádí přenos mezi periférií a systémovou vyrovnávací pamětí (`xx_strategy()` v následujícím seznamu). Je-li odkaz na periférii vytvořen a uskutečňován znakově, přenos z periférie je po skupinách znaků a je s vynecháním systémové vyrovnávací paměti (viz obr. 2.9).

Ovladače jsou programovány v jazyce C. Nově vznikající ovladač systémový programátor pojmenuje zatím v instalaci nepoužitým dvouznakovým jménem. Identifikace je použita na několika místech při vkládání ovladače do jádra, zejména je ale identifikací ovladače. V dalším textu budeme jako synonyma obecně používat jména *xx*. Ovladač je vytvořen především z těchto funkcí:

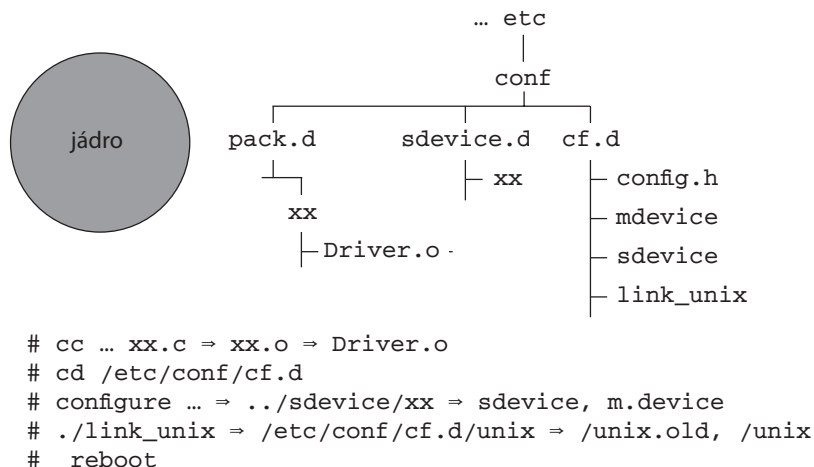
```
xx_init()    je inicializační funkce; jádro při spuštění testuje přítomnost periferie
             pomocí této funkce. Ohlásí-li funkce jádru kolizi, jádro v další činnosti
             periférii považuje za neexistující.
```

<code>xx_open()</code>	zpřístupnění periferie; je realizací volání jádra <code>open(2)</code> pro periferii (nikoliv pro obyčejný soubor)
<code>xx_close()</code>	ukončení činnosti s periferií; analogicky odpovídá volání jádra <code>close(2)</code>
<code>xx_read()</code>	přenos dat z periferie do datové oblasti procesu, který o data požádal voláním jádra <code>read(2)</code>
<code>xx_write()</code>	přenos dat z oblasti dat procesu na periferii na žádost procesu pomocí volání jádra <code>write(2)</code>
<code>xx_strategy()</code>	souvisí s blokovým přenosem dat mezi periferií a vyrovnávací pamětí, funkce optimalizuje dobu přenosu bloku vzhledem k typu periferie
<code>xx_intr()</code>	ošetřuje příchod přerušení od periferie; rutina je po příchodu přerušení vsunuta do zásobníku nestandardně, funkce není volána, ale vložena do zásobníku jádra, které ji provede
<code>xx_ioctl()</code>	zajišťuje speciální požadavky na periferie, které jsou jiného druhu než přenosu dat (obvykle nastavení periferie do určitého stavu), souvisí s voláním jádra <code>ioctl(2)</code>
<code>xx_select()</code>	je zamykací funkcí pro práci s periferií z více míst systému současně; vrací logickou pravdu, je-li periferie přístupná pro zápis nebo čtení

Při programování funkcí ovladače jsou k dispozici rutiny jádra, podporující např. přenos znaku z určité adresy sběrnice atd. Seznam a způsob jejich použití je součástí dokumentace konkrétního operačního systému, protože jednotlivé implementace podporují připojování ovladačů různým způsobem. Stejně tak je součástí firemní dokumentace způsob programování ovladače, jeho překlad a generace jádra s nově připojeným ovladačem. Obvykle dokumentace také obsahuje konkrétní ukázky.

Před generováním jádra s novým ovladačem se pro obohacení tabulek periferií používá program `configure(8)`. Uživatel v příkazovém řádku stanoví hlavní a vedlejší číslo periferie, adresu, vektor přerušení a jiné parametry, `configure(8)` se postará o konkrétní rozšíření tabulek jádra. Způsobem uvedeným ve čl. 9.5 pak generujeme nové jádro.

Popsané principy ukazuje obr. 9.18.



Obr. 9.18 Příklad jádra a ovladače xx v SCO UNIX

## Příloha A – Tabulka kódu ASCII

## Reprezentace v osmičkové soustavě

000	nul	001	soh	002	stx	003	etx	004	eot	005	enq	006	ack	007	bel
010	bs	011	hl	012	nl	013	vt	014	np	015	cr	016	so	017	si
020	dle	021	dc1	022	dc2	023	dc3	024	dc4	025	nak	026	syn	027	etb
030	can	031	em	032	sub	033	esc	034	fs	035	gs	036	rs	037	us
040	sp	041	!	042		043	#	044	\$	045	%	046	&	047	'
050	(	051	)	052	*	053	+	054	,	055	-	056	.	057	/
060	0	061	1	062	2	063	3	064	4	065	5	066	6	067	7
070	8	071	9	072	:	073	;	074	<	075	=	076	>	077	?
100	@	101	A	102	B	103	C	104	D	105	E	106	F	107	G
110	H	111	I	112	J	113	K	114	L	115	M	116	N	117	O
120	P	121	Q	122	R	123	S	124	T	125	U	126	V	127	W
130	X	131	Y	132	Z	133	[	134	\	135	]	136	^	137	_
140	`	141	a	142	b	143	c	144	d	145	e	146	f	147	g
150	h	151	i	152	j	153	k	154	l	155	m	156	n	157	o
160	P	161	q	162	r	163	s	164	t	165	u	166	v	167	w
170	x	171	y	172	z	173	{	174		175	}	176	~	177	del

## Reprezentace v šetnáčkové soustavě

00	nul	01	soh	02	stx	03	etx	04	eot	05	enq	06	ack	07	bel
08	bs	09	ht	0a	n1	0b	vt	0c	np	0d	cr	0e	so	0f	si
10	dle	11	dc1	12	dc2	13	dc3	14	dc4	15	nak	16	syn	17	etb
18	can	19	em	1a	sub	1b	esc	1c	fs	1d	gs	1e	rs	1f	us
20	sp	21	!	22		23	#	24	\$	25	%	26	&	27	'
28	(	29	)	2a	*	2b	+	2c	,	2d	-	2e	.	2f	/
30	0	31	1	32	2	33	3	34	4	35	5	36	6	37	7
38	8	39	9	3a	:	3b	;	3c	<	3d	=	3e	>	3f	?
40	@	41	A	42	B	43	C	44	D	45	E	46	F	47	G
48	H	49	I	4a	J	4b	K	4c	L	4d	M	4e	N	4f	O
50	P	51	Q	52	R	53	S	54	T	55	U	56	V	57	W
58	X	59	Y	5a	Z	5b	[	5c	\	5d	]	5e	^	5f	_
60	`	61	a	62	b	63	c	64	d	65	e	66	f	67	g
68	h	69	i	6a	j	6b	k	6c	l	6d	m	6e	n	6f	o
70	P	71	q	72	r	73	s	74	t	75	u	76	v	77	w
78	x	79	y	7a	z	7b	{	7c		7d	}	7e	~	7f	del



**Část pro kód národní abecedy latin2 (ISO 8859)**

80		81		82		83		84		85		86		87
88		89		8a		8b		8c		8d		8e		8f
90		91		92		93		94		95		96		97
98		99		9a		9b		9c		9d		9e		9f
a0		a1		a2		a3		a4		a5	Ĺ	a6		a7
a8		a9	Š	aa		ab	Ť	ac		ad		ae	Ž	af
b0		b1		b2		b3		b4		b5	Í	b6		b7
b8		b9	š	ba		bb		bc		bd		be	ž	bf
c0	Ř	c1	Á	c2		c3		c4	Ä	c5	Ľ	c6		c7
c8	Č	c9	É	ca		cb		cc	Ě	cd	Í	ce		cf
d0		d1		d2	Ň	d3	Ó	d4	Ö	d5		d6	Ö	d7
d8	Ř	d9	Ů	da	Ú	db		dc	Ü	dd	Ý	de		df
e0	ř	e1	á	e2		e3		e4	ä	e5	í	e6		e7
e8	č	e9	é	ea		eb		ec	ě	ed	í	ee		ef
f0		f1		f2	ň	f3	ó	f4	ô	f5		f6	ö	f7
f8	ř	f9	ů	fa	ú	fb	ť	fc	ü	fd	ý	fe		ff

**Část pro kód národní abecedy pc latin2 (ČSN 369103)**

80		81	ü	82	é	83		84	ä	85	û	86		87
88		89		8a		8b		8c		8d		8e	Ä	8f
90	É	91	Ľ	92	Ĺ	93	ô	94	ö	95	Ľ	96	Í	97
98		99	Ö	9a	Ü	9b	Ť	9c	ť	9d		ge		9f
a0	á	a1	í	a2	ó	a3	ú	a4		a5		a6	Ž	a7
a8		a9		aa		ab		ac	Č	ad		ae		af
b0		b1		b2		b3		b4		b5	Á	b6		b7
b8		b9		ba		bb		bc		bd		be		bf
c0		c1		c2		c3		c4		e5		c6		c7
c8		c9		ca		cb		cc		cd		ce		cf
d0		d1		d2	Ď	d3		d4	d'	d5	Ň	d6	Í	d7
d8	ě	d9		da		db		dc		dd		de	Ů	df
e0	Ó	e1		e2	Ô	e3		e4		e5	ň	e6	Š	e7
e8	Ř	e9	Ú	ea	ř	eb		ec	ý	ed	Ý	ee		ef
f0		f1		f2		f3		f4		f5		f6		f7
f8		f9		fa		fb		fc	Ř	fd	ř	fe		ff

# Příloha B – Textové editory

Základním vybavením je editor `ex(1)`. Je řádkově orientován, ale umožňuje plynulý přechod do obrazkově orientované editace editoru `vi(1)`. `ex(1)` je mocný editor a jeho schopnosti jsou doplněny možností využívání ostatních nástrojů systému např. k filtraci označených dat editovaných souborů.

`vi(1)` pracuje při zobrazení části souboru na obrazovce terminálu, pomocí kláves zvláštního významu se pohybuje po obrazovce a mění nebo rozšiřujeme editovaný text. Z obrazkového režimu můžeme plynule přecházet do režimu řádkového a využívat tak plnohodnotně funkcí editoru `ex(1)`. `vi(1)` pracuje na libovolném typu terminálu, jehož lokální obrazkové funkce lze popsat v konvencích položky databáze terminálů `terminfo(4)`.

`ed(1)` je řádkově orientovaný editor, je předchůdcem `ex(1)`, se kterým má mnohé základní funkce shodné. Nemá žádný obrazkový režim a neumožňuje spolupráci s `vi(1)`. Používá se stále tam, kde je operační systém z hlediska kapacity systémového svazku velmi omezen nebo při haváriích.

## B.1 `ed(1)`

Editor `ed(1)` má příkazový řádek:

```
ed [-] [-p string] [file]
```

Editor pracuje s načteným textem ze souboru `file` ve své datové oblasti, oblasti editace. Není-li `file` uveden, je tato oblast editace prázdná. Editor je řádkově orientován, nemá ale žádnou výzvu ke vstupu příkazu uživatele. Uživatel může řetězec výzvy stanovit při spuštění volbou `-p`. Je-li vše v pořádku, editor pracuje bez komentáře. Chyby komentuje výpisem znaku `?`, některé významné akce výpisem krátké statistiky. Využitím volby `-` potlačíme při práci i tyto výpisy.

Při vstupu do editoru je nastaven režim prohlížení. Odkazem (adresací) na určitou část oblasti editace (označením mezních řádků) pracujeme s editovaným textem. Text zobrazujeme vnitřním příkazem `p` a měníme příkazem `s`.

Režim vsouvání textu do určitého místa oblasti editace získáme vnitřním příkazem `i` nebo `a`. Následující text zapisovaný na terminál je pak vložen do oblasti editace až po řádek, který obsahuje pouze v prvním sloupci znak `.`.

Po celou dobu editace je změněný text pouze v paměti editoru. Teprve použitím příkazu `w` zapisujeme provedené změny do souboru.

Příkazem `q` ukončíme práci editoru.

Adresujeme oblast příkazu, a to buďto označením jednoho řádku nebo dvojicí řádků. Při vstupu do editoru je nastavena současná adresa na první řádek oblasti editace. Adresace má za následek změnu současné adresy.

**Adresujeme**

.	současný řádek
\$	poslední řádek textu v editované oblasti
<i>n</i>	číslo desítkové soustavy je pořadí řádku editované oblasti
' <i>x</i>	dříve označený řádek příkazem <i>k</i> jako <i>x</i> (musí být malé písmeno)
+	následující řádek
–	předchozí řádek
. + <i>n</i>	relativní odkaz na řádek vzdálený <i>n</i> řádků kupředu
. – <i>n</i>	odkaz na řádek vzdálený <i>n</i> řádků zpět

V adresaci můžeme používat také regulární výraz, který označuje textový řetězec. Při adresaci můžeme použít regulární výraz k vyhledání prvního řádku s určeným textem. Toho dosáhneme uzavřením regulárního výrazu mezi znaky / nebo ?

/regulární_výraz/	prohledávání textu ve vzestupném pořadí řádků od nastavené adresy; je-li dosaženo konce editované oblasti, editor pokračuje od prvního řádku
?regulární_výraz?	je totéž, ale prohledávání je sestupné; je-li dosaženo prvního řádku editované oblasti, postupuje se od posledního řádku nazpět

*regulární\_výraz* je textový řetězec, který může obsahovat znaky speciálního významu:

.	nahrazuje libovolný znak textu
*	nahrazuje opakování předchozího znaku (včetně žádného opakování)
. *	libovolný textový řetězec, který může obsahovat i znak binární nuly
[ ]	jeden z výčtu znaků, může být použit znak – pro uvedení výčtu podle pořadí v tabulce ASCII
[ ^ ]	jeden z neuvedených znaků ve výčtu
\	vypínáme zvláštní význam znaku, následující znak je nyní uvažován jako obyčejný znak konstruovaného textového řetězce
^	začátek řádku
\$	konec řádku

Adresace změny současné nastavení na adresovaný řádek. Ve dvojici jsou adresy odděleny znakem , (čárka) např.

1, \$ je adresace celé oblasti editace.

Nastavení pracovního řádku je podle druhé uvedené adresy. Použijeme-li znak ; (středník) namísto , (čárka) jde o stejnou adresaci, pracovní řádek je ale nastaven podle první uvedené adresy.

Následující seznam příkazů editoru má vždy před vlastním příkazem v kulatých závorkách uvedenu implicitní adresaci. Ze zápisu také vyplývá formát možné adresy příkazu:

## B.1 Textové editory

- (.)a vložení textu *text* za uvedenou adresu  
*text*  
.
- (.)c adresovaný řádek je nahrazen následujícím textem *text*  
*text*  
.
- (.,.)d zrušení řádků podle adresace
- e *file* obsah oblasti editace bude zrušen a namísto něj bude načten obsah souboru *file*
- E *file* totéž co e, editor ale neupozorňuje uživatele na případnou ztrátu doposud editovaného textu
- f [*file*] je nastaveno nové jméno souboru pro zápis oblasti editace na disk, bez uvedení jména souboru *file* získáme současné informace o nastavení
- (1,\$)g/*regulární\_výraz/příkazy*  
příkaz globální platnosti, nejprve vyhledá řádky odpovídající regulárnímu výrazu a pak provede uvedené příkazy editoru tak, že každý vybraný řádek je uvažován jako pracovní
- (1,\$)G/*regulární\_výraz/*  
interaktivní příkaz globální platnosti
- h nápověda chybových zpráv  
H zapíná (vypíná) další zobrazování chybových zpráv k výpisu znaku ? při práci uživatele
- (.)i vložení textu *text* před uvedenou adresu  
*text*  
.
- (.,.+1)j spojení řádků
- (.)kx označení řádku pro použití v další adresaci (x může být pouze malé písmeno)
- (.,.)l výpis řádků podle adresace, na rozdíl od příkaz p budou vypisovány všechny nezobrazitelné znaky textu v konvencích znakových konstant jazyka C
- (.,.)ma příkaz přemístění adresované části za adresu *a*, na místě *a=0* znamená přemístění na začátek editované oblasti
- (.,.)n výpis řádků podle adresace obohacený o číslo řádků v pořadí editované oblasti

- (. . .)P výpis řádků podle adresace
- P od této chvíle bude editor vypisovat znak výzvy, a sice \* ; příští použití P toto nastavení ruší
- q ukončení práce editoru
- Q ukončení práce editoru bez upozornění na ztrátu nepřenesených změn z oblasti editace do souboru
- ( $\$$ )r *file*  
načtení dalšího textu do oblasti editace ze souboru *file*, adresace určuje, za který řádek oblasti editace bude text souboru vložen, 0 je vsunutí na začátek oblasti editace
- (. . .)s/*regulární\_výraz/nový\_text/*  
(. . .)s/*regulární\_výraz/nový\_text/g*  
(. . .)s/*regulární\_výraz/nový\_text/n*  
(*n* je v rozsahu 1–512) substituce nalezeného textu v oblasti adresace s obsahem podle regulárního výrazu, regulární výraz je nahrazen novým textem, doplňující příkaz *g* určuje globální platnost v rámci nalezeného řádku, všechny odpovídající texty jsou vyměněny; je-li uveden počet *n*, dojde pouze k *n* výměnám
- (. . .)ta kopie adresovaného textu za adresu *a*, 0 na místě *a* je začátek souboru
- u návrat ke stavu oblasti editace před provedením posledního příkazu
- (1, $\$$ )v/*regulární\_výraz/příkazy*  
má tentýž význam jako příkazy *g*, globální platnost je ale vztažena na všechny řádky, které neobsahují text daný regulárním výrazem
- (1, $\$$ )V/*regulární\_výraz/*  
má tentýž význam jako příkaz *G*, globální platnost je ale vztažena na všechny řádky, které neobsahují text daný regulárním výrazem
- (1, $\$$ )w [*file*]  
zápis obsahu oblasti editace na disk do souboru *file*; neuvedeme-li *file*, zápis je do nastaveného souboru (načteného souboru při vstupu do editace nebo příkazem *f*), nebude-li *file* určen při vstupu do editace a soubor tohoto jména existuje, budeme muset pro přepis jeho obsahu připojit bezprostředně za *w* znak !
- (1, $\$$ )W *file*  
stejný význam jako příkaz *w*, ale obsah oblasti editace je k souboru připojen

## B.2 Textové editory

- (`$`) = výpis čísla současného řádku editace
- !*příkaz* provedení příkazu *příkaz* interpretem shell
- (`+.1`) adresace řádku sama o sobě způsobí výpis adresovaného řádku, stisk klávesy Enter je totéž jako `+.1p`, používá se pro výpis obsahu krokováním

### B.2 `ex(1)`

Editor `ex(1)` má příkazový řádek:

```
ex [-] [-v] [-r] [-R] [+command] [-l] [file ...]
```

Editor `ex(1)` je sice řádkově orientován, ale pomocí jeho příkazu `visual` můžeme přecházet do editace editorem `vi(1)`. Volbou `-v` v příkazovém řádku přecházíme do `vi(1)` ihned po startu editace. `ex(1)` udržuje historii editačních prací uživatele v pomocném souboru, který zůstává v pracovním adresáři i po havárii systému. Pro obnovu ztracené práce s editorem je možné po obnovení práce systému použít volbu `-r`. Editor může z důvodů větší bezpečnosti při pouhém prohlížení textu pracovat v režimu ochrany proti přepisu, což vyžadujeme volbou `-R`. V průběhu takové práce ale můžeme příkazem `noreadonly` ochranu proti přepisu zrušit. Volba `-` potlačí jakýkoliv výpis statistik nebo zpráv pro uživatele a používá se tam, kde je `ex(1)` využíván pro neinteraktivní úkony (ve scénářích). Volba `+command` je požadavek na provedení příkazu editace při startu editoru. `ex(1)` může pracovat v režimu programovacího jazyka LISP. K tomu slouží volba `-l`. Jako poslední v seznamu pozičních argumentů příkazového řádku `ex(1)` jsou použita jména všech souborů, které budeme editovat. Do oblasti editace je přitom načten první z nich. Uživatel může při práci s textem používat odkládací prostory identifikované písmeny malé abecedy (`a ... z`).

Po startu se `ex(1)` ohlásí výpisem znaku `:` a čeká na příkazy uživatele. Rovněž jako u `ed(1)` je nastaven režim prohlížení oblasti editace, který měníme na režim vsouvání příkazy `append`, `insert` nebo `change`. Poslední řádek vsouvaného textu, který obsahuje pouze v prvním sloupci znak `.`, končí režim vsouvání.

Regulární výrazy lze používat při dodržování stejné syntaxe jako u `ed(1)`, v konstrukci je možné navíc použít znak `~`, který nahrazuje textový řetězec výměny při posledním použití příkazu `substitute`.

Rovněž adresace řádků oblasti editace je stejná, navíc je možné psát

```
%          zkráceně namísto 1, $ (celá oblast editace)
/regulární_výraz/
?regulární_výraz?
            jednou zapsaný regulární výraz může být opakován pouze zápisem // nebo
            ?? při hledání dalšího takového textu
'x          odkaz na označený řádek znakem x, k označení slouží příkaz mark
```

V dále uvedených příkazech editoru `ex(1)` platí adresace jednoho řádku (u příkazů je označena slovem **line**) nebo adresace dvojice řádků (ve formátu příkazů označena slovem **range**). Dále je použito slova **flag**, které označuje jeden nebo několik z možných příkazů výpisu `#`, `p`, a `l`. Je-li použito **count**, ztrácí **range** smysl, protože **count** je počet opakování zadaného příkazu (není-li explicitně řečeno jinak), adresace při opakování je dána novým nastavením současného řádku. **buffer** je jedna z oblastí editace, každá oblast odpovídá jinému editovanému souboru.

Seznam příkazů `ex(1)` ukazuje následující tabulka, kde jsou uvedeny také jejich zkrácené ekvivalenty, které je možné používat:

abbrev	ab	next	n	unmap	unm
append	a	number #	nu	version	ve
args	ar	preserve	pre	visual	vi
change	c	print	p	write	w
copy	co	put	pu	xit	x
delete	d	quit	q	yank	ya
edit	e	read	re	(window)	z
file	f	recover	re	(escape)	!
global	g v	rewind	rew	(lshift)	<
insert	i	set	se	(rshift)	>
join	j	shell	sh	(resubst)	& s
list	l	source	so	(scroll)	^d
map	map	substitute	s	(line no)	=
mark	k ma	unabbrev	una		
move	m	undo	u		

implicitní hodnoty:

**line**  
**range** (., .)  
**count** 1  
**flag** nic

*ab word rhs*

vytvoření nové zkratky pro definovaný textový řetězec

**line a** vsouvání textu za adresaci line

**ar** výpis seznamu argumentů

**range c count**

adresovaná oblast je nahrazena nově zapisovaným textem, nezapišeme-li žádný text, jde o stejný efekt jako u příkazu `d`

**range co line flags**

kopie adresované oblasti za řádek adresace **line**

### **range d buffer count**

zrušení adresované oblasti z oblasti editace **buffer**

### **e +line file**

editace nového textu, který bude načten ze souboru *file*, **line** v tomto případě určuje nastavení současného řádku v nově editovaném textu

### **f**

výpis informací o souboru, ke kterému patří právě editovaný text

### **range g/regulární\_výraz/příkazy**

nejprve jsou vybrány všechny řádky oblasti editace podle regulárního výrazu a pak jsou pro ně prováděny uvedené příkazy editoru

### **range v/regulární\_výraz/příkazy**

nejprve jsou vybrány všechny řádky oblasti editace, které neodpovídají regulárnímu výrazu, a pro ně jsou pak prováděny uvedené příkazy editoru

### **line i**

vsouvání textu před adresaci **line**

### **range j count flags**

spojení všech řádků adresované oblasti do jednoho řádku

### **range l count flags**

zobrazení adresované oblasti; znaky, jejichž zobrazení na obrazovce nemá žádný efekt, jsou uvedeny v konvencích zápisu konstant jazyka C

### **map x rhs**

definice makra pro obrazovkový režim; na místě *x* může být *#n*, kde *n* je číselné označení funkční klávesy (F1, F2, ... F12), *rhs* je příkaz, který bude proveden po stisku klávesy; připojíme-li k příkazu znak **!** makro lze použít nikoliv jako příkaz pro editor, ale jako text, který lze využívat v režimu vsouvání

**line ma x** označení řádku oblasti editace podle znaku malého písmena ASCII *x*, jedná se o alternaci příkazu **k ed(1)**

### **range m line**

přemístění oblasti podle adresace **range** za řádek daný **line**

### **n**

požadavek editace následujícího souboru podle seznamu příkazového řádku

### **range nu count flags**

výpis řádků adresované oblasti, každý řádek bude očíslován podle svého pořadí v souboru, příkaz **#** alternuje příkaz **number**

### **pre**

úschova současného stavu editace pro případ havárie systému



**range p count**

výpis obsahu oblasti editace podle adresace, na obrazovce nezobrazitelné znaky budou vypisovány v souvislosti s klávesou Ctrl

**line pu buffer**

restaurace všech zrušených řádků textu oblasti editace

q ukončení práce editoru

**line r file**

načtení textu ze souboru *file* do oblasti editace za řádek daný adresací **line**

rec *file* restaurace akcí editace podle souboru, který byl vytvořen při havárii systému

rew editace opět prvního v pořadí uvedených souborů podle příkazového řádku

**set [parameter]**

modifikuje práci editoru; k dispozici je řada způsobů, např. lze požadovat zarovnávání řádků na definovanou délku atd., v dalším textu je uveden seznam možností při použití příkazu **set**; bez argumentů příkaz vypíše všechna nastavení, která byla změněna, při použití argumentu **all** bude výpis nastavení úplný

sh spuštění příkazového interpretu, po jeho ukončení návrat do editoru

so *file* načtení a provedení příkazů pro editor ze souboru *file*

**range s/regulární\_výraz/text/options count flags**

výměna textu daného regulárním výrazem za *text*; na místě *options* může být použit příkaz **g**, který rozšiřuje platnost výměny na celý řádek, při použití **c** na místě *options* bude editor požadovat před každou výměnou potvrzení stiskem klávesy **y**

**una word**

ruší platnost zkratky slova *word*

u návrat ke stavu před poslední provedenou akcí, nevztahuje se na příkazy typu **write**, **edit** a **next**

unm *x* ruší platnost dříve definovaného makra *x*

ve výpis verze editoru

**line vi [type] count**

vstup do editoru **vi** (1), na místě *type* může být buď **.** nebo **-**, kterými

určujeme vstupní pozici `vi(1)` převzatou z `ex(1)`, **count** určuje velikost okna, implicitní je velikost daná nastavením `window` (viz změna práce editoru), příkaz `Q` končí práci editoru `vi(1)`

### **range** `w[file]`

zápis do souboru *file*, jinak do nastaveného souboru; pokud na místě *file* uvedeme `!command`, bude spuštěn shell, kterému bude předán příkaz *command*, tomuto příkazu bude obsah oblasti editace předán na standardní vstup, příkaz `wq` je totéž jako `w a pak q`, `wq!` je totéž co `w!` a pak `q`

x pokud byly v textu provedeny změny, editovaný soubor je přepsán obsahem oblasti editace, poté je ukončena práce editoru

### **range** `ya` [**buffer**] **count**

vsune adresovaný text do odkládacího prostoru; není-li **buffer** uveden, odkládací oblast je nepojmenovaná

### **line** `z` [*type*] **count**

výpis obsahu oblasti editace od současného řádku; **count** určuje počet vypsaných řádků, není-li uveden, je využito nastavení `window`, *type* určuje umístění výpisu na obrazovce, `-` je od prvního řádku obrazovky, `.` je od jejího středu

`!command`

*command* je interpretován pomocí shellu

### **range** `!` *command*

adresované řádky textu (adresace zde nemá implicitní hodnotu, viz předchozí příkaz) budou vyslány na standardní vstup příkazu shellu *command*, adresované řádky jsou po ukončení shellu vyměněny za standardní výstup *command*

### **range** `<` **count**

znaky adresovaných řádků posune o počet mezer (nebo tabulátorů) daných nastavením `shiftwidth` doleva

### **range** `>` **count**

znaky adresovaných řádků posune o počet mezer (nebo tabulátorů) daných nastavením `shiftwidth` doprava

### **range** `&` [*options*] **count flags**

opakuje poslední provedený příkaz

`Ctrl-d` (v ASCII `eot`), výpis následujících *n* řádků, *n* je dáno obsahem nastavení `scroll`

**line=** výpis čísla řádku daného adresací

Změny práce editoru je dosaženo jiným nastavením jeho vnitřních proměnných. Vnitřní proměnné jsou logické nebo číselné, nastavují se příkazem `set`, kterým lze také zjišťovat jejich současný obsah. Uživatel může mít ve svém domovském adresáři soubor se jménem `.exrc`, který je editorem načten a interpretován při každém startu. V tomto souboru může inicializovat jiné prostředí editace než je standardní. Uvádíme následující seznam vnitřních proměnných, včetně jejich zkratk:

`autoindent, ai`  
nastavuje doplňování řádku mezerami na délku stanovenou předchozím řádkem

`autoprint, ap`  
po provedení každého příkazu je vypsán současný řádek

`autowrite, aw`  
oblast editace je přepsána do nastaveného souboru vždy, dojde-li ke změně nebo při použití příkazů `next`, `rewind` nebo `!`

`beautify, bf`  
všechny řídicí znaky vyjma tabulátorů, znaků nového řádku a znaku nové stránky jsou ze vstupního textu vynechávány

`directory, dir`  
obsah proměnné určuje adresář, ve kterém bude umístěna odkládací oblast editoru

`edcompatible, ed`  
při použití doplňujících příkazů `global` a `change` u příkazu `substitute`, bude zapamatována jejich platnost pro příští příkaz `substitute`

`ignorecase, ic`  
u regulárních výrazů není rozdíl mezi malými a velkými písmeny textů

`lisp`  
je nastavena proměnná `autoindent` a příkazy `()` `{ }` `[[ ]]` v obrazovkovém režimu `vi` (`1`) mají význam ve smyslu programovacího jazyka LISP

`list`  
při vypisovaném textu oblasti editace jsou tabulátory vypisovány jako `^I` a konec je označen znakem `$`

`magic`  
změna při konstrukci regulárních výrazů

`number, nu`  
vypisované řádky budou číslovány

## B.2 Textové editory

---

<code>paragraphs, para</code>	obsahuje text s označením makra podle konvencí maker v <code>nroff(1)</code> , kterým bude začínat každý paragraf editovaného textu
<code>prompt</code>	určuje, zda bude vypisován znak výzvy :
<code>redraw</code>	simulace inteligentního terminálu
<code>remap</code>	je-li tato proměnná nastavena, umožňuje uživateli definovat vnořená makra (příkazem <code>map</code> )
<code>report</code>	obsahuje počet řádků, které musí být v oblasti editace změněny, než editor vypíše zprávu o změnách
<code>scrool</code>	obsahuje počet řádků pro příkazy <code>CTRL-d</code> a <code>z</code>
<code>sections</code>	obsahuje jméno makra <code>nroff(1)</code> , kterým bude začínat každý odstavec
<code>shiftwidth, sw</code>	obsahuje nastavení tabulátoru nebo určuje počet mezer pro příkazy posunu
<code>showmatch, sm</code>	v obrazkovém editoru <code>vi(1)</code> , bude-li napsán znak ( nebo { , bude vyhledán jeho protiznak ) nebo }
<code>slowopen, slow</code>	ve <code>vi(1)</code> zabrání přepisu obrazovky v režimu vsouvání
<code>tabstop, ts</code>	nastavuje tabulátor pro text vstupního souboru
<code>terse</code>	je-li nastavena, chybové zprávy jsou zkráceny
<code>window</code>	obsahuje počet řádků okna, ve kterém bude pracovat <code>vi(1)</code>
<code>wrapscan, ws</code>	je-li nastavena, při prohledávání pomocí // nebo ?? pokračuje při dosažení konce nebo začátku souboru
<code>wrapmargin, wm</code>	je-li nastavena na hodnotu jinou než 0, bude při vsouvání textu po dosažení sloupce tohoto čísla do textu vnucen znak nového řádku
<code>writeany, wa</code>	ruší kontrolu přepisu souboru při použití příkazu <code>w</code>

**B.3 vi(1)**

Editor `vi(1)` má příkazový řádek:

```
vi [-rfile] [-l] [-wn) [-R] [+command] file ...
```

Obrazkově orientovaný editor `vi(1)` je část `ex(1)` editoru, je ale možné jej startovat z příkazového řádku, jehož volby jsou podobné významem formátu příkazového řádku `ex(1)`. Volba `-r` zajistí načtení souboru s informacemi akcí editace; pokud došlo k havárii systému v průběhu předchozí editace; `-l` je nastavení konvencí jazyka LISP a `-R` je pro pouze zobrazení obsahu souborů. Volbou `-w` definujeme počet řádků okna, ve kterém bude `vi(1)` pracovat. Konečně `+command` je příkaz pro editor `ex(1)`, který bude vykonán před započítím editace. Současně můžeme editovat více souborů.

Použijeme-li v editoru klávesu se znakem `:` vstupujeme dočasně (pro provedení jednoho příkazu) do řádkového režimu editoru `ex(1)`. Trvalý přechod do `ex(1)` zajistí stisk klávesy `Q`. Z `ex(1)` trvale do `vi(1)` pak přecházíme příkazem `visual`.

I pro `vi(1)` platí inicializace způsobu editace ze souboru `.exrc` jak tomu je u `ex(1)`. Protože `vi(1)` využívá lokálních vlastností terminálu z hlediska práce s obrazovkou, je nutné, aby typ terminálu byl nastaven v proměnné shellu `TERM` uživatele sezení, tento obsah přitom musí odpovídat správnému popisu v databázi `terminfo` jak bylo popsáno v kap. 7.

Konec souboru při zobrazení poslední stránky oblasti editace uživatel rozpozná podle znaků `~`, které v prvním sloupci na jinak prázdném řádku označují prozatím textem neobsazenou část oblasti editace. U některých terminálů, které nemají všechny potřebné obrazkové operace, jsou řádky, které uživatel na obrazovce zrušil, označeny znakem `@` v prvním sloupci opět na jinak prázdném řádku. Přepis obrazovky vypuštěním těchto řádků v textu uživatel dosáhne kombinací kláves `Ctrl-r`.

`vi(1)` po spuštění pracuje v zobrazovacím režimu. Všechny dále uvedené příkazy, pokud nebude řečeno jinak, mají význam pro tento režim. V režimu vsouvání jsou uvažovány jako jakýkoliv jiný znak. Uživatel prohlíží text a teprve stiskem kláves, které odpovídají příkazům pro vstup do režimu vsouvání textu (klávesy `i`, `a`, `o`, `c` atd.), rozšiřuje editovaný text. Rušení textu se provádí v zobrazovacím režimu. V následujícím seznamu příkazů `vi(1)` je v syntaxi použit zkrácený zápis pro klávesu `Ctrl` jako `^`.

Příkazy se na obrazovce nikde nezobrazují, pokud nebude při popisu řečeno jinak. Většinou příkazů může předcházet zápis čísla, které budeme v následujícím textu označovat jako **počet** a které představuje velikost nebo adresaci řádku oblasti editace (u příkazů zobrazování nebo přesunu) nebo počet opakování příkazu (u příkazů, které mění text). Stránkou rozumíme text o počtu řádků shodném s velikostí okna. Příkazy adresace se rozumí `c`, `d`, `y`, `<`, `>`, `!` a `=`

- `^b` (ASCII `\002`) zobrazení předchozí stránky, **počet** stanovuje, o kolik stránek se vrátit zpět
- `^d` (ASCII `\004`) zobrazení textu o další polovinu stránky, **počet** zadává, kolik řádků je polovina stránky, počet je zapamatován pro příští použití `^d` nebo `^u`
- `^e` (ASCII `\005`) posun o jeden řádek vpřed, pozice kurzoru vůči textu (pokud je to možné) zůstává zachována, v režimu vsouvání je eliminován počet mezer (nebo tabulátorů), které byly vloženy příkazem `^t`

### B.3 Textové editory

---

<code>^f</code>	(ASCII \006) zobrazení následující stránky, počet říká, o kolik stránek vpřed se posunout
<code>^g</code>	(ASCII \007) výpis atributů nastaveného souboru, (ekvivalent příkazu <code>file editor ex(1)</code> )
<code>^h</code>	(ASCII \010) přesun pozice kurzoru o znak doleva, <b>počet</b> stanovuje, o kolik znaků doleva se přesunout, (má tentýž význam jako příkaz <code>h</code> )
<code>^j</code>	(ASCII \012) přesun pozice kurzoru ve stejném sloupci o řádek dolů, <b>počet</b> udává, o kolik řádků vpřed se přesunout, (je totožný s příkazem <code>^n a j</code> )
<code>^l</code>	(ASCII \014) vymaže obrazovku a opět zobrazí tutěž stránku
<code>^m</code>	(ASCII \015) přesun kurzoru na první zobrazitelný znak následujícího řádku, <b>počet</b> udává, o kolik řádků vpřed se přesunout
<code>^n</code>	(ASCII \016) totéž jako <code>^j a j</code>
<code>^p</code>	(ASCII \020) přesun kurzoru o řádek zpět na tutěž pozici sloupce, <b>počet</b> určuje, o kolik řádků zpět se vrátit
<code>^r</code>	(ASCII \022) nově zobrazí tutěž stránku, vynechává přitom řádky označené znakem <code>@</code> (u méně inteligentních terminálů)
<code>^t</code>	(ASCII \024) v režimu vsouvání, jsme-li na začátku řádku nebo řádku pouze s mezerami či tabulátory, vsune počet mezer odpovídající <code>ex(1)</code> proměnné <code>shiftwidth</code> ; jejich zrušení je pak možné pouze příkazem <code>^d</code>
<code>^u</code>	(ASCII \025) zobrazení textu zpět o polovinu stránky, <b>počet</b> zadává, kolik řádků je polovina stránky, <b>počet</b> je zapamatován pro příští použití <code>^d</code> nebo <code>^u</code>
<code>^v</code>	(ASCII \026) v režimu vsouvání umožňuje vsunout následující zapsaný znak zvláštního významu do editovaného textu (s výjimkou Esc)
<code>^w</code>	(ASCII \027) v režimu vsouvání ruší naposledy vsunuté slovo
<code>^y</code>	(ASCII \031) posun zobrazeného textu o řádek zpět, kurzor zůstává na stejném místě, pokud je to možné
<code>^[</code>	(ASCII \035) zruší částečně zapsaný příkaz, v režimu vsouvání končí tento režim, v příkazu pro <code>ex(1)</code> (příkaz <code>:</code> ) končí vstupní řádek, následuje provedení příkazu
mezera	přesun pozice kurzoru o znak doprava (končí na konci řádku), <b>počet</b> udává, o kolik znaků kurzor přesunout, (totéž jako příkaz <code>l</code> )

- ! editor pošle adresované řádky oblasti editace na standardní vstup daného příkazu pro shell a vymění tyto adresované řádky za text standardního výstupu příkazu, ! je následován příkazem adresace; je-li **počet** stanoven, je předán příkazu adresace, zápisem !! jako příkazu opakujeme předchozí akci příkazu !
- " předchází označení odkládací oblasti, uživatel má k dispozici číslované odkládací oblasti 1–9, do kterých editor ukládá zrušené texty; pojmenované odkládací oblasti a–z jsou uživateli (jako u `ex(1)`) dostupné pro odkládání jím určených textů
- \$ přesun pozice kurzoru na konec současného řádku, **počet** určí, o kolik řádků kupředu, 2\$ znamená přesun na konec následujícího řádku
- % přesun kurzoru na znak kulaté nebo složené závorky, která kontextově odpovídá kulaté nebo složené závorce, na kterou ukazuje kurzor
- & opakuje poslední provedenou substituci
- ' následuje-li za ním opět znak ' vracíme se na začátek řádku předchozího nastavení kurzoru, následuje-li některý ze znaků a–z, příkaz nastavuje kurzor na začátek řádku označený příkazem m
- ` je-li následován opět znakem ` vracíme kurzor na řádek předchozí pozice kurzoru, následuje-li některý ze znaků a–z, kurzor je nastaven podle obsahu těchto pojmenovaných odkládacích oblastí na znak dříve označený
- [ [ nastavení kurzoru na začátek sekce; sekce je definována proměnnou `sections`, řádky, které začínají znakem `Ctrl-l` nebo {, končí platnost příkazu
- ] ] nastavení kurzoru na konec sekce, (viz příkaz ]])
- ^ přesun kurzoru na první zobrazitelný znak současného řádku
- ( nastavení kurzoru na začátek věty; věta končí znaky . ! nebo ?, které jsou následovány znakem nového řádku nebo dvěma mezerami, **počet** určí, o kolik vět se vrátit zpět
- ) nastavení kurzoru na začátek následující věty, **počet** stanovuje, o kolik vět kupředu
- { nastavení kurzoru na začátek předchozího paragrafu, paragraf je definován hodnotou proměnné `paragraphs`, **počet** stanoví, o kolik paragrafů zpět

}	nastaví kurzor na začátek následujícího paragrafu, <b>počet</b> stanoví, o kolik paragrafů kupředu
	vyžaduje striktně <b>počet</b> , kurzor je nastaven na pozici sloupce <b>počet</b>
+	přesun pozice kurzoru na první znak následujícího řádku, <b>počet</b> stanoví, o kolik řádků, (totéž jako $\wedge M$ )
,	opak posledního z příkazů $f$ , $F$ , $t$ nebo $T$ , prohledávání na řádku je v opačném pořadí, <b>počet</b> určuje opakování
–	přesun pozice kurzoru na první znak předchozího řádku, <b>počet</b> udává, o kolik řádků zpět
.	opakuje naposledy provedený příkaz, <b>počet</b> je opakování
/	(je zobrazeno na dolním řádku obrazovky) následující regulární výraz (je vypisován na dolním řádku obrazovky) je čten editorem a odpovídající textový řetězec je vyhledán od pozice kurzoru vpřed, prohledávání lze přerušit stiskem klávesy Del, regulární výraz je definován v <code>ex ( 1 )</code>
0	přesun kurzoru na první znak současného řádku
:	následuje zadání příkazu pro <code>ex ( 1 )</code> ; znak <code>:</code> i následující příkaz pro <code>ex ( 1 )</code> je vypisován na posledním řádku
;	opakuje naposledy zadaný příkaz $f$ , $F$ , $t$ nebo $T$ , <b>počet</b> stanoví kolikrát
<	posune současný řádek o <b>počet</b> znaků daných obsahem proměnné <code>shiftwidth</code> , <b>počet</b> je předán příkazu <code>adresace</code> , který může následovat
~	změna velkých písmen na malá a opačně, <b>počet</b> udává, na kolik znaků se příkaz vztahuje
>	posune současný řádek doprava o <code>shiftwidth</code> (viz <code>&lt;</code> )
=	při nastavení volby <code>lisp</code> , provede vsunutí potřebných mezer pro zarovnání pro stanovenou část textu
?	opačné prohledávání podle daného regulárního výrazu (viz <code>/</code> )
A	připojení dále vsouvaného textu za poslední znak současného řádku, vsouvání je ukončeno klávesou Esc, (totéž jako <code>\$a</code> )



- B přesun kurzoru o slovo zpět na první znak slova, **počet** určí, o kolik slov se přesunout zpět
- C výměna zbytku současného řádku za dále vsouvaný text, vsouvaný text je ukončen klávesou Esc, (totéž jako c\$)
- D zrušení zbytku současného řádku (totéž jako d\$)
- E přesun kurzoru kupředu na konec slova, **počet** stanovuje, o kolik slov se posunout
- F za příkazem musí následovat znak, který je vyhledán v současném řádku od pozice kurzoru zpět, **počet** stanovuje opakování příkazu
- G přesun na řádek daný předcházejícím argumentem nebo na konec souboru bez argumentu
- H přesun kurzoru na první řádek obrazovky, **počet** stanoví počet řádků, které jsou připočteny
- I vsunutí textu na začátek řádku
- J spojení současného a následujícího řádku, **počet** určuje, kolik následujících řádků bude spojeno
- L přesun kurzoru na začátek posledního řádku obrazovky, **počet** určuje kolik řádků od posledního řádku
- M přesun kurzoru na první znak řádku uprostřed obrazovky
- N hledání následujícího textu podle regulárního výrazu naposledy zadaného příkazem / nebo ?, ale v opačném pořadí (příkaz je z hlediska prohledávání opačný k n)
- O vytvoření prázdného řádku před současným řádkem a vstup do režimu vsouvání na jeho začátek
- P vložení naposledy zrušeného textu před pozici kurzoru, může předcházet jménu nebo číslu oblasti odložení
- Q ukončení vi ( 1 ) a vstup do ex ( 1 )
- R výmění znak na pozici kurzoru za bezprostředně zapsaný na klávesnici, **počet** udává, kolik následujících znaků bude za zapsaný znak vyměněno

### B.3 Textové editory

---

S	vymění celý řádek (totéž jako <code>cc</code> ), <b>počet</b> je výměna několika následujících řádků
T	musí být následován znakem a editor pak na současném řádku směrem zpět hledá zapsaný znak, kurzor je umístěn za nalezený znak, <b>počet</b> je opakování příkazu
U	obnoví stav současného řádku před poslední manipulací přesunu
W	přesun kurzoru na začátek následujícího slova na současném řádku, <b>počet</b> říká, o kolik slov se posunout
X	ruší znak před pozicí kurzoru, <b>počet</b> opakuje příkaz, ale ruší se pouze znaky na současném řádku
Y	umístí kopii současného řádku do nepojmenované oblasti odložení, <b>počet</b> stanoví kopii více řádků, může předcházet jménu oblasti odložení, která je potom cílovým místem příkazu
ZZ	ukončení práce editoru, všechny provedené změny jsou přitom přepsány do souboru na disk (totéž jako <code>xu ex(1)</code> )
a	za současnou pozici kurzoru bude vkládán text v režimu vsouvání, vsouvaný text je ukončen znakem Esc
b	přesun kurzoru o slovo zpět na první znak slova, <b>počet</b> udává, o kolik slov se vrátit zpět
c	zruší adresovanou část textu a vstupuje do režimu vsouvání, kdy nahrazuje zrušený text; je-li více než jeden řádek textu zrušen, je tento text uschován v číselné oblasti odložení, <b>počet</b> je předán příkazu adresace, který následuje; následuje-li opět znak <code>c</code> , zbytek řádku je zrušen a nahrazen vstupním textem
d	zruší adresovanou část editovaného textu, <b>počet</b> je předán příkazu adresace, který následuje; následuje-li opět <code>d</code> , je zrušen současný řádek
e	přesun kurzoru na konec následujícího slova, <b>počet</b> opakuje příkaz
f	znak, kterým je příkaz následován, je hledán ve zbytku řádku od kurzoru, <b>počet</b> opakuje příkaz
h	přesune kurzor o jeden znak doleva (stejně jako <code>Cntrl-h</code> ), <b>počet</b> opakuje příkaz

i	před pozici kurzoru je vložen následující text režimu vsouvání
j	přesun kurzoru na následující řádek téhož sloupce (totéž jako $\wedge J$ a $\wedge N$ )
k	přesun kurzoru o řádek zpět (totéž jako $\wedge P$ )
l	přesun kurzoru o jeden znak doprava (totéž jako příkaz mezera)
m	musí za ním následovat znak malé abecedy $x$ , označí tím současnou pozici (tj. řádek); na označenou pozici se lze později odkazovat způsobem $\backslash x$
n	opakuje poslední příkaz / nebo ?
o	vytvoření prázdného řádku za současným řádkem a vstup do režimu vsouvání na začátek vytvořeného řádku
p	vložení textu za pozici kurzoru tak, jak je popsáno u $P$
r	musí následovat znak, kterým je nahrazen znak na pozici kurzoru, <b>počet</b> je náhrada znaků následujících za tentýž znak
s	zruší znak pozice kurzoru a vstupuje do režimu vsouvání, vsouváný text ukládá na místo zrušeného znaku, <b>počet</b> stanoví, kolik bude zrušených znaků
t	musí následovat znak, na zbytku řádku hledá zapsaný znak, pozice kurzoru je přenesena za nalezený znak, pokud je nalezen; <b>počet</b> je opakování příkazu
u	obnova posledních změn, které jsou uloženy v současné oblasti odložení
w	přesune pozici kurzoru na začátek následujícího slova, <b>počet</b> udává, o kolik slov se posunout vpřed
x	zruší znak na pozici kurzoru, <b>počet</b> stanovuje počet znaků, které budou zrušeny za pozicí kurzoru, ale pouze v rámci současného řádku
y	musí následovat příkaz adresace, adresovaný text je okopírován do nepojmenované oblasti odložení, předchází-li příkazu označení jména oblasti odložení ve tvaru $x$ , text je umístěn do ní
z	znovu vypíše zobrazený text, současný řádek ale umístí podle následující klávesy, Return znamená umístění na první řádek stránky, . je střed stránky a – je dolní řádek obrazovky, <b>počet</b> za příkazem určuje velikost obrazovky v řádcích, <b>počet</b> před $z$ stanoví, o kolik řádků je zobrazení mimo střed obrazovky

## Příloha C – Bourne shell

### C.1 ZNAKY ZVLÁŠTNÍHO VÝZNAMU

#### Přesměrování kanálů:

`[kanál]>jméno` výstup deskriptoru č. *kanál* (implicitně 1) je přepnut do souboru *jméno*

`[kanál]<jméno` vstup deskriptoru č. *kanál* (implicitně 0) je přepnut na soubor *jméno*

`[kanál]>>jméno` výstup deskriptoru č. *kanál* (implicitně 1) je připojen k souboru *jméno*,

`<<[-]text` vstup z textu scénáře od prvního znaku nového řádku, řetězec *text* samostatně na novém řádku končí přesměrování

`[kanál1]>&[kanál2]` deskriptor č. *kanál1* je spojen s deskriptorem č. *kanál2*

`<&-` uzavření kanálu č. 0, analogicky `>&-` pro kanál č. 1

*příkaz* | *příkaz* roura

#### Jména souborů:

`*` jakýkoliv textový řetězec (včetně řetězce nulové délky)

`?` jakýkoliv znak

`[! -]` jakýkoliv znak z uvedeného seznamu `!` je negace výčtu a `-` označení rozsahu v pořadí podle tabulky ASCII

#### Řízení toku:

`;` oddělovač příkazů

`&` spuštění příkazového řádku na pozadí

`&&` logický součin příkazů

`||` logický součet příkazů

`for PROMĚNNÁ [in text ...] do příkazy done`  
proměnná s identifikátorem *PROMĚNNÁ* bude postupně vždy pro každé opakování cyklu sekvence příkazů *příkazy* obsahovat jeden z textových řetězců ze seznamu *text* ...

`case text in [vzorek [| vzorek] ...] příkazy;;]... esac`  
podle obsahu proměnné *text* bude provedena pouze jedna ze sekvencí příkazů *příkazy*, a to podle shody textových řetězců *text* a *vzorek*

`if příkazy then příkazy [elif příkazy then příkazy]...[else příkazy] fi`  
podle hodnoty návratového statutu posledního příkazu ze sekvence *příkazy* bude provedena buďto sekvence příkazů za `then` nebo za

případným `else`; v případě nepravdy návratového statutu lze použít další testování větví `elif`

`while příkazy` *do* `příkazy` *done*  
 sekvence příkazů mezi `do` a `done` bude opakovaně prováděna v případě pravdivého stavu posledního ze sekvence příkazů za `while`; namísto `while` lze použít `until`, tělo cyklu je pak prováděno v případě nepravdy návratového statutu

(*příkazy*) sekvence příkazů *příkazy* je interpretována nově vzniklým procesem příkazu `sh(1)`

{ *příkazy* ; } skupina sekvence příkazů

*jméno* () { *příkazy*; }  
 definice funkce *jméno*; tělo funkce definované sekvencí příkazů *příkazy* bude provedeno vždy v rámci následující sekvence příkazů

### Výluka:

\  
 'řetězec'  
 "řetězec"

znaku bezprostředně následujícímu bude vypnut jeho zvláštní význam, všechny znaky zvláštního významu v textovém řetězci *řetězec* budou uvažovány jako jakékoliv jiné znaky

totéž, znaky zvláštního významu pro substituci obsahu proměnné (`$ { }`) a příkazu (`` ``) si ale ponechají svůj zvláštní význam

### Proměnné:

*PROMĚNNÁ=obsah* [*PROMĚNNÁ=obsah*]...  
 naplnění (definice) obsahu proměnné *PROMĚNNÁ* textovým řetězcem *obsah*

`$PROMĚNNÁ` substituce obsahem proměnné *PROMĚNNÁ*

`${PROMĚNNÁ}` totéž

`${PROMĚNNÁ:-text}`  
 je-li proměnná *PROMĚNNÁ* definována a má obsah nenulové délky, je substituován její obsah; v opačném případě je konstrukce nahrazena textovým řetězcem *text*

`${PROMĚNNÁ:=text}`  
 je-li proměnná *PROMĚNNÁ* definována a má obsah nenulové délky, je substituován její obsah; v opačném případě je definována a její obsah bude mít hodnotu textového řetězce *text*, konstrukce je pak nahrazena textovým řetězcem *text*

`${PROMĚNNÁ:?text}`  
 je-li proměnná *PROMĚNNÁ* definována a má obsah nenulové délky, je substituován její obsah; v opačném případě je na obrazovku vypsán textový řetězec *text* a je ukončena práce `sh(1)`; je-li *text* vynechán, na obrazovku je vypsán `text parameter null or not set` a rovněž `sh(1)` končí činnost

## C.1 Bourne shell

`$ {PROMĚNNÁ:+text}`

je-li proměnná *PROMĚNNÁ* definována a má obsah nenulové délky, je konstrukce nahrazena textovým řetězcem *text*; není-li definována není nahrazeno nic

`PROMĚNNÁ='příkaz'`

obsah proměnné *PROMĚNNÁ* je naplněn standardním výstupem příkazu *příkaz*

### Jména implicitních proměnných:

1, 2, 3, 4, ...	poziční parametry scénáře
*	obsahuje poziční parametry příkazového řádku scénáře, a to způsobem " <code>\$1 \$2 \$3 ...</code> "
@	totéž, ale způsobem " <code>\$1</code> " " <code>\$2</code> " " <code>\$3</code> " ...
#	počet pozičních parametrů scénáře
-	volby příkazového řádku <code>sh (1)</code> , který scénář provádí, nebo volby dané příkazem <code>set</code>
?	návratový status posledního provedeného příkazu
\$	identifikační číslo procesu (PID) aktuálního <code>sh (1)</code>
!	PID procesu naposledy provedeného na pozadí
HOME	domovský adresář; vnucený parametr příkazu <code>cd</code> , je-li <code>cd</code> bez parametru
PATH	definice adresářů (jsou odděleny znakem <code>:</code> ), ve kterých budou hledány soubory vnějších příkazů,
CDPATH	definice adresářů (obdobně jako u <code>HOME</code> ) pro použití příkazu <code>cd</code>
IFS	oddělovač textů příkazového řádku, obvykle nastaven na znak mezery tabulátor a znak nového řádku
MAIL	je-li nastavena, jedná se o soubor poštovní schránky a <code>sh (1)</code> reaguje upozorněním uživatele, objeví-li se zde nová pošta; <code>MAILPATH</code> přitom nesmí být definována
MAILCHECK	počet vteřin, po jejichž uplynutí <code>sh (1)</code> vždy zjišťuje, zda se v poštovní schránce neobjevila nová pošta
MAILPATH	seznam poštovních schránek, které <code>sh (1)</code> zajímají ohledně nové pošty; soubory poštovních schránek jsou odděleny znakem <code>:</code> a za cestou souboru může po uvedení znaku <code>%</code> následovat text, kterým <code>sh (1)</code> uživatele upozorňuje na příjem pošty
PS1	znak výzvy komunikace uživatele s <code>sh (1)</code>
PS2	2. znak výzvy komunikace uživatele s <code>sh (1)</code> , objeví se, pokračuje-li uživatel příkazem na novém řádku
SHACCT	soubor, do něhož jsou ukládány účtovací informace uživatele související s <code>sh (1)</code>
SHELL	jméno souboru s programem shellu; je rozhodující při spouštění, protože podle obsahu této proměnné z prostředí provádění je vybírán buďto <code>sh (1)</code> nebo <code>rsh (1)</code>



<code>return</code> [ <i>n</i> ]	ukončení funkce s návratovou hodnotou <i>n</i> ; je-li <i>n</i> vynechána, je použit návratový status posledního provedeného příkazu
<code>set</code> [-aefhktuvx] [ <i>arg</i> ...]	bez voleb a parametrů vypíše seznam všech definovaných proměnných a jejich obsah; volby při použití + namísto - mají za důsledek navrácení do původního stavu; následující význam uvedených voleb je analogický při použití voleb v příkazovém řádku <code>sh(1)</code> <ul style="list-style-type: none"> <li>-a označení proměnných, jejichž obsah byl modifikován nebo byly označeny pro export</li> <li>-e ukončení procesu <code>sh(1)</code> jakmile některý provedený příkaz vrací nepravdivý návratový status</li> <li>-f vypíná význam expanzních znaků u jmen souborů</li> <li>-h pamatuje si místo definice funkce</li> <li>-k do prostředí provádění vnějšího příkazu jsou umístěny všechny klíčové parametry, nejen ty, které předcházejí příkazový řádek; klíčové parametry jsou definovány v příkazovém řádku <code>sh(1)</code> jako definice proměnných (viz <code>sh(1)</code> Příloha E)</li> <li>-n čte příkazy, ale neprovádí je</li> <li>-t ukončí <code>sh(1)</code> po přečtení a provedení jednoho příkazu</li> <li>-u substituce neexistujících proměnných je považována za chybu</li> <li>-v příkazové řádce bude <code>sh(1)</code> opisovat na standardní výstup tak, jak byly načteny</li> <li>-x <code>sh(1)</code> opisuje příkazové řádce na standardní výstup tak, jak byly prováděny</li> <li>-- je nastavena výchozí hodnota všech uvedených voleb</li> </ul>
<code>shift</code> [ <i>n</i> ]	posun pozičních parametrů, od <code>\$n+1</code> je přejmenován na <code>\$n</code> atd., není-li <i>n</i> stanoveno, je 1
<code>test</code>	vyhodnocení podmíněného výrazu (viz <code>test(1)</code> v Příloze E)
<code>times</code>	výpis spotřebovaného strojového času synovských procesů, a to pro supervizorový i uživatelský režim
<code>trap</code> [ <i>arg</i> ] [ <i>n</i> ]	příkazový řádek <i>arg</i> je proveden, zachytí-li proces <code>sh(1)</code> signál <i>n</i>
<code>type</code> [ <i>jméno</i> ...]	oznámí cesty souborů externích příkazů <i>jméno</i> ...; jedná-li se o vnitřní příkaz, komentuje to
<code>ulimit</code> [-[HS]] [ <i>a c d f n s t v</i> ]]	
<code>ulimit</code> [-[HS]] [ <i>c d f n s t v</i> ]] <i>limit</i>	nastavuje omezení; bez <i>limit</i> vypisuje nastavená omezení, volbou <ul style="list-style-type: none"> <li>-a všechna omezení; -H je tvrdé omezení (<u>h</u>ard) a -S měkké omezení (<u>s</u>oft); volby mají význam</li> <li>-c největší možná velikost souboru <code>core</code> (v blocích o velikosti 512 slabik)</li> <li>-d největší možná velikost datového segmentu nebo haldy (v blocích o velikosti 1024 slabik)</li> <li>-f největší možná velikost obsahu souboru (v blocích o velikosti 512 slabik)</li> <li>-n největší možná velikost tabulky deskriptorů souborů zvýšená o 1</li> <li>-s největší možná velikost zásobníku (v blocích o velikosti 1024 slabik)</li> </ul>



- t nejdelší přípustný čas centrální jednotky (v sekundách)
- v největší přípustná velikost virtuální paměti (v blocích o velikosti 1024 slabik)

**umask** [*nnn*] nastavení masky *nnn* přístupových práv pro vytváření souborů; bez *nnn* je nastavená maska zobrazena

**unset** [*PROMĚNNÁ ...*] zruší definici proměnných *PROMĚNNÁ ...*; proměnné *PATH*, *PS1*, *PS2*, *MAILCHECK* a *IFS* nemohou být zrušeny

**wait** [*n*] čekání na dokončení synovského procesu s *PID=n*, který byl spuštěn na pozadí; bez uvedení *n* *sh(1)* čeká na všechny procesy spuštěné na pozadí

### C.3 ŘÍZENÍ PRACÍ (PLATÍ POUZE PRO *jsh(1)*)

*%práce* identifikace práce, kde *práce* může být následující

*%* nebo *+* pro aktuální práci

*-* pro předchozí práci

*% řetězec* pro práci, která ve jménu obsahuje textový řetězec *řetězec*

*n* práce č. *n*

*předpona* práce, jejíž jméno začíná textovým řetězcem *předpona*

#### Vnitřní příkazy:

**bg** [*%práce*] převedení pozastavené práce na pozadí; je-li *%práce* vynecháno, na pozadí je převedena aktuální práce

**fg** [*%práce*] převedení pozastavené *práce* nebo *práce* běžící na pozadí na popředí

**jobs** [*-p | -l*] [*-n*] [*%práce*] výpis pozastavených prací nebo prací běžících na pozadí; je-li *%práce* vynecháno, výpis se vztahuje na všechny takové práce; volby mají význam

**jobs -x** *příkaz* [*argumenty*] *-l* výpis bude také obsahovat PGID a pracovní adresář prací

*-p* výpis bude obsahovat pouze PGID prací

*-n* výpis pouze těch prací, jejichž stav se od posledního výpisu změnil

*-x* provede *příkaz* s *argumenty*, předtím nahradí identifikaci práce v *příkaz* a *argumenty* za odpovídající GID

**kill** [*-signal*] *%práce* verze *kill(1)* pro *jsh(1)*; procesům práce *%práce* bude zaslán signál *signal*

**stop** *%práce ...* pozastavení prací *práce ...* běžící na pozadí

**suspend** pozastavení provádění *jsh(1)* (ale jen v případě, že *jsh(1)* není interpret startovaný s přihlášením uživatele)

### C.3 Bourne shell

---

```
wait [%práce ...]
```

rozšíření vnitřního příkazu `wait`; `jsh(1)` čeká na dokončení prací.  
`%práce ...` bez argumentů čeká na dokončení všech prací běžících na pozadí

U `jsh(1)` je důležité mít nastavenou možnost pozastavení práce běžící na popředí. Je zvykem používat klávesu `Ctrl-z`, kterou definujeme příkazem `stty(1)` takto:

```
$ stty susp ^z
```

## Příloha D – Volání jádra

Příloha nezahrnuje žádné síťové rozhraní. Volání jádra jsou v kontextu kap. 5 a doporučení SVID3.

### **access**

```
#include <unistd.h>
int access(const char *path, int amode);
        ověření přístupových práv souboru
souvisí s chmod(2), stat(2)
```

### **acct**

```
#include <sys/types.h>
int acct(const char *path);
        zapnutí (vypnutí) účtování
souvisí s exec(2), exit(2), fork(2), signal(2)
```

### **adjtime**

```
#include <sys/time.h>
int adjtime(struct timeval *delta, struct timeval *olddelta);
        oprava času vzhledem ke stavu hodin
souvisí s date(1), gettimeofday(2)
```

### **alarm**

```
#include <unistd.h>
unsigned int alarm(unsigned int sec);
        vyslání signálu SIGALRM po uplynutí zadaného intervalu reálného času
souvisí s exec(2), fork(2), pause(2), signal(2)
```

### **atexit**

```
#include <stdlib.h>
int atexit(void (*func) (void));
        připojení další funkce do registru funkcí, které budou provedeny v době
        ukončení procesu
souvisí s exit(2)
```

### **brk**

```
int brk(char *endds);
        posunutí hranice dat na endds
char *sbrk(int incr);
        posunutí hranice dat o incr
        není uvedeno v SVID3
souvisí s exec(2), shmop(2), ulimit(2), end(3), malloc(3)
```

### **chdir**

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

```
int chdir(const char *path);  
změna pracovního adresáře  
souvisí s chroot(2), open(2)
```

```
chmod  
#include <sys/types.h>  
#include <sys/stat.h>  
int chmod(const char *path, int protect);  
změna přístupových práv  
souvisí s chmod(1), chown(2), creat(2), fcntl(2), mknod(2), open(2),  
read(2), stat(2), write(2)
```

```
chown  
#include <unistd.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
int chown(const char *path, int uid, int gid);  
změna vlastníka  
souvisí s chmod(2), chown(1)
```

```
chroot  
int chroot(const char *path);  
změna kořenového adresáře  
souvisí s chdir(2)
```

```
close  
#include <unistd.h>  
int close(int fd);  
uzavření souboru  
souvisí s creat(2), dup(2), exec(2), fclose(3), fcntl(2), open(2), pipe(2),  
signal(2)
```

```
creat  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
int creat(const char *path; int protect);  
vytvoření souboru  
souvisí s chmod(2), close(2), dup(2), fcntl(2), lseek(2), open(2), read(2),  
umask(2), write(2)
```

```
dup  
#include <unistd.h>  
int dup(int fd);  
int dup2(int fd, int fd2);
```

přidělení duplikátu k deskriptoru souboru

```
souvisí s creat(2), close(2), exec(2), fcntl(2), lockf(2), open(2),
      pipe(2)
```

**exec**

```
#include <unistd.h>
int execl(const char *path, const char *arg0, const char *arg1,
      ..., const char *argn, (char *)0);
int execlv(const char *path, char *const *argv);
int execlp(const char *path, const char *arg0, const char
      *arg1, ..., const char *argn, (char *)0, char *const
      *envp);
int execve(const char *path, char *const *argv, char *const
      *envp);
int execlp(const char *file, const char *arg0, ... const char
      * argn, (char *)0);
int execvp(const char *file, char *const *argv);
```

výměna textu a dat procesu

```
souvisí s alarm(2), exit(2), fork(2), priocntl(2), sigaction(2),
      signal(2), sigpending(3), sigprocmask(2), times(2), ulimit(2),
      umask(2)
```

**exit**

```
#include <stdlib.h>
void exit(int status);
#include <unistd.h>
void _exit(int status);
```

ukončení procesu a ukončení procesu bez úklidu

```
souvisí s atexit(3), catopen(3), fclose(3), signal(2), termios(3),
      wait(2), waitid(2)
```

**fattach**

```
int fattach(int fd, const char *path);
      spojení PROUDu se jménem souboru
souvisí s fdetach(2), isastream(3)
```

**fcntl**

```
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
int fcntl (int fd, int cmd [, arg]);
      řídící operace se souborem (typ arg závisí na cmd)
souvisí s access(2), close(2), exec(2), open(2), lockf(2), read(2),
      signal(2), write(2)
```

### **fdetach**

```
int fdetach(const char *path);  
    zrušení spojení PROUDu a jména souboru  
souvisí s fattach(2)
```

### **fork**

```
#include <unistd.h>  
#include <sys/types.h>  
int fork (void);  
    vytvoření procesu  
souvisí s alarm(2), exec(2), directory(2), fcntl(2), lockf(2),  
    priocntl(2), sigaction(2), signal(2), system(3), times(2),  
    ulimit(2), umask(2), wait(2)
```

### **fsync**

```
int fsync(fd);  
    přepis dat souboru ze systémové vyrovnávací paměti na disk  
souvisí s sync(2)
```

### **getcontext**

```
#include <ucontext.h>  
int getcontext(ucontext_t *ucp);  
int setcontext(ucontext_t *ucp);  
    přepnutí kontextu procesu  
souvisí s setjmp(3), sigaction(2), sigprocmask(2), sigsetjmp(3)
```

### **getitimer**

```
#include <sys/time.h>  
int getitimer(int which, struct itimerval *value);  
int setitimer(int which, struct itimerval *value,  
    struct itimerval *ovalue);  
    nastavení nebo získání intervalu reálného času  
souvisí s gettimeofday(2)
```

### **getmsg**

```
#include <stropts.h>  
int getmsg(int fd, struct strbuf *ctlptr, struct strbuf  
    *dataptr, int *flagsp);  
int getpmsg(int fd, struct strbuf *ctlptr, struct strbuf  
    *dataptr, int *bandp, int *flagsp);  
    příjem následující zprávy z PROUDu  
souvisí s poll(2), putmsg(2), read(2), write(2)
```

### **getpgid**

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

```
#include <unistd.h>
pid_t getpgid(pid_t pid);
    získání PGID procesu s PID pid
souvisí s exec(2), fork(2), getpid(2), getsid(2), setpgid(2), setsid(2)
```

**getpid**

```
#include <unistd.h>
#include <sys/types.h>
int getpid(void);
    získání identifikačního čísla procesu (PID)
int getpgrp(void);
    získání identifikačního čísla procesu vedoucího skupiny (PGID)
int getppid(void);
    získání identifikačního čísla procesu otce (PPID)
souvisí s exec(2), fork(2), setpgrp(2)
```

**getrlimit**

```
#include <sys/time.h>
#include <sys/resource.h>
int getrlimit(int resource, struct rlimit *rlp);
int setrlimit(int resource, const struct rlimit *rlp);
    získání a nastavení limitních hodnot zdrojů procesu
souvisí s malloc(3), open(2), signal(2)
```

**getsid**

```
#include <sys/types.h>
pid_t getsid(pid_t pid);
    získání identifikace sezení
souvisí s exec(2), fork(2), getpid(2), setpgid(2), setsid(2)
```

**gettimeofday**

```
#include <sys/time.h>
int gettimeofday(struct timeval *tp);
int settimeofday(struct timeval *tp);
    získání nebo nastavení data a času
souvisí s adjtime(2), ctime(3)
```

**getuid**

```
#include <unistd.h>
#include <sys/types.h>
unsigned short getuid(void);
    získání reálné identifikace vlastníka (UID)
unsigned short geteuid(void);
    získání efektivní identifikace vlastníka (EUID)
unsigned short getgid(void);
```

získání reálné identifikace skupiny (GID)

```
unsigned short getegid(void);
```

získání efektivní identifikace skupiny (EGID)

```
souvisí s cuserid(3), setuid(2)
```

**ioctl**

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

```
int ioctl(int fd, int cmd [, arg]);
```

řídící operace se znakovým zařízením (typ `arg` závisí na `cmd`)

```
souvisí s termios(3)
```

**kill**

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

```
#include <signal.h>
```

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

vyslání signálu procesu nebo skupině procesů

```
souvisí s getpid(2), signal(2)
```

**link**

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

```
int link(const char *path, const char *newpath);
```

další jméno souboru

```
souvisí s rename(3), symlink(2), unlink(2)
```

**lockf**

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

```
int lockf(int fd, int function, long size);
```

zamknutí souboru nebo jeho části

```
souvisí s alarm(2), chmod(2), close(2), creat(2), fcntl(2), open(2),  
read(2), write(2)
```

**lseek**

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

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

```
long lseek(int fd, long int offset, int from);
```

přestavení ukazovátka souboru

```
souvisí s creat(2), dup(2), fcntl(2), fseek(3), open(2)
```

**memcntl**

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

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

```
int memcntl(caddr_t addr, size_t len, int cmd, int arg, int  
attr, int mask);
```

správa paměti procesu



```
souvisí s sysconf(2), mlock(2), mlockall(3), mmap(2), mprotect(2),
msync(2), plock(2)
```

### mlock

```
#include <sys/types.h>
int mlock(caddr_t addr, size_t len);
int munlock(caddr_t addr, size_t len);
        zamknutí (nebo zrušení zámku) stránky v operační paměti
souvisí s memcntl(2)
```

### mkdir

```
#include <sys/types.h>
#include <sys/stat.h>
int mkdir(const char *path, int mode);
        vytvoření adresáře
souvisí s chmod(2), umask(2)
```

### mkfifo

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *path, mode_t mode);
        vytvoření nové fronty FIFO
souvisí s chmod(2), exec(2), mkdir(2), mknod(2), umask(2)
```

### mknod

```
#include <sys/types.h>
#include <sys/stat.h>
int mknod(const char *path, int mode, int dev);
        vytvoření adresáře, speciálního nebo obyčejného souboru
souvisí s chmod(2), exec(2), mkdir(2), mkfifo(2), stat(2)
```

### mmap

```
#include <sys/types.h>
#include <sys/mman.h>
caddr_t mmap(caddr_t addr, size_t len, int prot, int flags, int
        fd, off_t off);
        mapování stránek paměti
souvisí s fcntl(2), fork(2), lockf(2), mlockall(2), munmap(2),
mprotect(2), plock(2), sysconf(2)
```

### mount

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

```
int mount(const char *spec, const char *dir, int rw,
          const char *fstype, const char *dataptr,
          int datalen);
```

připojení svazku

souvisí s `umount(2)`

**mprotect**

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

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

```
mprotect(caddr_t addr, size_t len, int prot);
```

nastavení přístupových práv mapování paměti

souvisí s `mmap(2)`, `memcntl(2)`, `mlock(2)`, `mlockall(2)`, `plock(2)`, `sysconf(2)`

**msgctl**

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

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

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

```
int msgctl(int id, int cmd, struct msqid_ds *mstatbuf);
```

řídící operace nad frontou zpráv

souvisí s `msgget(2)`, `msgop(2)`

**msgget**

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

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

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

```
int msgget(key_t key, int flag);
```

zpřístupnění fronty zpráv,

souvisí s `msgctl(2)`, `msgop(2)`

**msgop**

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

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

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

```
int msgsnd(int id, void *msg, size_t count, int flag);
```

zaslání zprávy do fronty zpráv

```
int msgrcv(int id, void *msg, int count, long type, int flag);
```

příjem zprávy z fronty zpráv

souvisí s `msgctl(2)`, `msgget(2)`, `signal(2)`

**msync**

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

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

```
int msync(caddr_t addr, size_t len, int flags);
```

aktualizace dané stránky paměti na disk

souvisí s `mmap(2)`, `sysconf(2)`

**munmap**

```
#include <sys/types.h>
#include <sys/mman.h>
int munmap(caddr_t addr, size_t len);
```

uvolnění stránek paměti

souvisí s `mmap(2)`, `sysconf(2)`

**nice**

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

změna priority

souvisí s `exec(2)`, `priocntl(2)`

**open**

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *path, int oflag [, int protect]);
```

otevření souboru

souvisí s `chmod(2)`, `close(2)`, `creat(2)`, `dup(2)`, `fcntl(2)`, `fopen(3)`,  
`lseek(2)`, `read(2)`, `umask(2)`

**pause**

```
#include <unistd.h>
int pause (void);
```

pozastavení procesu do příchodu signálu

souvisí s `alarm(2)`, `kill(2)`, `sigaction(2)`, `signal(2)`, `sigsuspend(3)`,  
`wait(2)`

**pipe**

```
#include <unistd.h>
int pipe(int pd[2]);
```

vytvoření kanálu komunikace dvou procesů (roury)

souvisí s `read(2)`, `write(2)`

**poll**

```
#include <poll.h>
int poll (struct pollfd fds[], unsigned long nfds, int timeout);
```

obsluha více v/v kanálů

souvisí s `getmsg(2)`, `putmsg(2)`, `read(2)`, `write(2)`

**plock**

```
#include <sys/lock.h>
int plock(int op);
```

uzamknutí segmentů procesu v paměti

souvisí s `exec(2)`, `exit(2)`, `fork(2)`, `memcntl(2)`

**prctl**

```
#include <sys/types.h>
#include <sys/procset.h>
#include <sys/prctl.h>
#include <sys/tspriocntl.h>
long prctl(idtype_t idtype, id_t id, int cmd, ... [arg]);
```

**správa procesů**

souvisí `exec(2)`, `fork(2)`, `nice(2)`, `prctl(8)`

**profil**

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

**monitorování procesu**

**ptrace**

```
#include <sys/types.h>
int ptrace(int cmd, int pid, int addr, int data);
```

**ladění synovského procesu**

souvisí `exec(2)`, `signal(2)`, `wait(2)`

**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);
```

**zaslání zprávy do PROUDu**

souvisí `getmsg(2)`, `poll(2)`, `read(2)`, `write(2)`

**read**

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/uio.h>
int read(int fd, char *buf, unsigned count);
int readv(int fd, struct iovec *iov, unsigned count);
```

**čtení souboru**

souvisí `creat(2)`, `dup(2)`, `fcntl(2)`, `fopen(3)`, `fread(3)`, `ioctl(2)`,  
`lseek(2)`, `open(2)`, `popen(3)`

**readlink**

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

**čtení hodnoty nepřímého odkazu**

souvisí `stat(2)`, `symlink(2)`

**rmdir**

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

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

zrušení adresáře

```
souvisí s mkdir(2)
```

### **semctl**

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

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

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

```
int semctl(int id, int number, int cmd,
```

```
union semun {
```

```
    int val;
```

```
    struct semid_ds *buf;
```

```
    ushort *array;
```

```
    } arg);
```

řízení semaforů

```
souvisí s semget(2), semop(2)
```

### **semget**

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

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

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

```
int semget(key_t key, int count, int flag);
```

zpřístupnění skupiny semaforů

```
souvisí s semctl(2), semop(2)
```

### **semop**

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

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

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

```
int semop(int id, struct sembuf *oplist, unsigned count);
```

operace nad semaforem

```
souvisí s exec(2), exit(2), fork(2), semctl(2), semget(2)
```

### **setpgid**

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

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

```
int setpgid(pid_t pid, pid_t pgid);
```

nastavení identifikace skupiny

```
souvisí s exex(2), fork(2), getpid(2), kill(2), setsid(2)
```

### **setpgrp**

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

```
pid_t setpgrp(void);
```

prohlášení procesu za vedoucí proces skupiny

```
souvisí s exec(2), fork(2), getpid(2), kill(2), setsid(2)
```

### **setsid**

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

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

```
pid_t setsid(void);
```

nastavení identifikace sezení

```
souvisí s exec(2), exit(2), fork(2), getpid(2), getpgid(2), getsid(2),  
setpgid(2)
```

### **setuid**

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

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

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

nastavení reálné i efektivní identifikace vlastníka (UID, EUID)

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

nastavení reálné i efektivní identifikace skupiny (GID, EGID)

```
souvisí s exec(2), getuid(2),
```

### **shmctl**

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

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

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

```
int shmctl(int id, int cmd, struct shmid_ds *buf);
```

řídící operace nad sdílenou pamětí

```
souvisí s shmget(2), shmop(2)
```

### **shmget**

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

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

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

```
int shmget(key_t key, int size, int flag);
```

získání přístupu ke sdílené paměti

```
souvisí s shmctl(2), shmop(2)
```

### **shmop**

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

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

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

```
void *shmat(int id, void *addr, int flag);
```

```
int shmdt(void *addr);
```

operace nad sdílenou pamětí

```
souvisí s exec(2), exit(2), fork(2), shmctl(2), shmget(2)
```

### **sigaction**

```
#include <signal.h>
```

```
int sigaction(int sig, const struct sigaction *act,
              struct sigaction *oact);
    podrobná správa signálů mezi procesy
souvisí s exit(2), kill(2), pause(2), siginfo(2), signal(2),
    sigprocmask(2), sigsetopts(2), sigsuspend(3), ucontext(2),
    wait(2)
```

**signal**

```
#include <signal.h>
void (* signal(int sig, void (*func) (int))) (int);
void (* sigset(int sig, void (*func) (int)))(int);
    změna reakce na signál
souvisí s kill(2), pause(2), wait(2), waitid(2),
```

**sigprocmask**

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set,
                sigset_t *oact);
    získání nebo změna masky příjmu signálů
souvisí s sigaction(2), signal(2), sigsetopts(2)
```

**sigsetopts**

```
#include <signal.h>
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);
    manipulace se skupinou signálů
souvisí s pause(2), sigaction(2), signal(2), sigprocmask(2)
```

**stat**

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *path, struct stat *buf);
    získání informací o souboru
int lstat(const char *path, struct stat *buf);
    získání informací o nepřímém odkazu
int fstat(int fd, struct stat *buf);
    získání informací o otevřeném souboru
souvisí s chmod(2), chown(2), creat(2), link(2), lockf(2), mknod(2),
    pipe(2), read(2), time(2), unlink(2), utime(2), write(2),
    statvfs(2)
```

### **statvfs**

```
#include <sys/types.h>
#include <sys/statvfs.h>
int statvfs(const char *path, struct statvfs *buf);
int fstatvfs(int fildes, struct statvfs *buf);
        získání informací o svazku
souvisí s chmod(2), chown(2), creat(2), dup(2),fcntl(2),link(2),
        mknod(2),open(2),pipe(2),read(2),time(2),unlink(2),ustat(2),
        utime(2),write(2)
```

### **stime**

```
#include <sys/types.h>
#include <time.h>
int stime(const long *tpttr);
        nastavení systémového času
souvisí s adjtime(2),time(2)
```

### **swapctl**

```
#include <sys/stat.h>
#include <sys/swap.h>
int swapctl(int cmd, void *arg);
        správa odkládací oblasti
```

### **symlink**

```
int symlink(const char *path, const char *newpath);
        vytvoření nepřímého odkazu
souvisí s link(2),readlink(2),stat(2),unlink(2)
```

### **sync**

```
void sync(void);
        zápis systémových vyrovnávacích pamětí na disk
souvisí s fsync(2)
```

### **sysconf**

```
#include <unistd.h>
long sysconf(int name);
        získání informací o parametrech jádra
```

### **time**

```
#include <sys/types.h>
#include <time.h>
time_t time(time_t *tloc);
        získání systémového času
souvisí s stime(2)
```



**times**

```
#include <sys/types.h>
#include <sys/times.h>
long int times(struct tms *tbuf);
```

získání časů spotřebovaných procesem na procesoru a doby trvání procesu

souvisí s `exec(2)`, `fork(2)`, `time(2)`, `times(2)`, `wait(2)`

**ulimit**

```
#include <ulimit.h>
long ulimit(int cmd, long new);
```

získání (nastavení) horní meze délky souboru a posunutí hranice dat v paměti

souvisí s `getrlimit(2)`, `write(2)`

**umask**

```
#include <sys/types.h>
#include <sys/stat.h>
int umask(int mask);
```

nastavení masky přístupových práv

souvisí s `chmod(2)`, `creat(2)`, `mkdir(2)`, `mkfifo(2)`, `mknod(2)`, `open(2)`

**umount**

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

odpojení svazku

souvisí s `mount(2)`

**uname**

```
#include <sys/utsname.h>
int uname(struct utsname *name);
```

získání jména provozní verze UNIXu

**unlink**

```
#include <unistd.h>
int unlink(const char *path);
```

odstranění položky adresáře (souboru)

souvisí s `close(2)`, `open(2)`, `remove(3)`, `unlink(2)`

**ustat**

```
#include <sys/types.h>
#include <ustat.h>
int ustat(int dev, struct ustat *buf);
```

získání informací o svazku

souvisí s `stat(2)`, `statvfs(2)`

### utime

```
#include <sys/types.h>
#include <utime.h>
int utime(const char *path, const struct utimbuf *times);
    získání časů posledního přístupu a zápisu do souboru
souvisí s stat(2)
```

### wait

```
#include <sys/types.h>
#include <sys/wait.h>
int wait(int *stat_loc);
int wait((int *)0);
    čekání na ukončení synovského procesu
souvisí s exec(2), exit(2), fork(2), pause(2), signal(2)
```

### write

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/uio.h>
int write(int fd, const char *buf, unsigned count);
int writev(int fd, const struct iovec *iov, int iovcnt);
    zápis do souboru
souvisí s chmod(2), creat(2), dup(2), getrlimit(2), lseek(2), open(2),
    pipe(2), ulimit(2)
```

## Příloha E – Vnější příkazy

Příloha je sestavena s ohledem na současný stav UNIX SYSTEM V a doporučení SVID3. Příkazy v knize použité jako příklad ze systémů jiných výrobců se zde nemusí vyskytovat.

### **adb**

```
adb [a.out [core]]
```

dynamický ladící prostředek na úrovni zdrojového textu v assembleru,  
není uvedeno v SVID3

```
souvisí s cc(1), ld(1), sdb(1)
```

### **admin**

```
admin [-n] -i [name] [-rrel] [-tname] [-fflag[flag-val] ...]
```

```
[-a[!]login ...] [-m[mrlist]] [-y[comment]] file
```

```
admin -n [-tname] [-fflag[flag-val] ...] [-a[!]login ...]
```

```
[-m[mrlist]] [-y[comment]] file ...
```

```
admin [-t[name]] [-fflag[flag-val] ...] [-dflag[flag-val] ...]
```

```
[-a[!]login ...] [-elogin ...] file ...
```

```
admin -h file
```

```
admin -z file
```

vytvoření a údržba souboru SCCS

```
souvisí s delta(1), get(1), prs(1), what(1)
```

### **ar**

```
ar option [posname] afile [name] ...
```

údržba a archivace knihoven

```
souvisí s ld(1), strip(1)
```

### **as**

```
as [-oobjfile] [-m] [-V] file
```

běžný assembler

```
souvisí s cc(1), ld(1), m4(1)
```

### **at**

```
at [-f script] [-m] time [date] [+incremental]
```

```
at -l [job ...]
```

```
at -r job ...
```

```
at -d job ...
```

```
at -z job ...
```

```
at -Z job ...
```

```
batch
```

provedení příkazu ve stanoveném čase

```
souvisí s cron(1), date(1)
```

## E Vnější příkazy

---

**atq**

`atq [-c] [username ...]`

`atq [-n] [username ...]`

výpis fronty požadavků na provedení příkazů ve stanoveném čase

souvisí s `at(1)`, `atrm(1)`, `cron(1)`

**atrm**

`atrm [-f |-i] [-a] job-number [[job-number | username] ...]`

`atrm [-f |-i] [-a] username [[job-number | username] ...]`

zrušení požadavku na provedení příkazu ve stanoveném čase

souvisí s `at(1)`, `atq(1)`, `cron(8)`

**awk**

`awk [-F c] [-f progfile] ['program'] [parameters] [file ...]`

programovací jazyk pro práci s texty

souvisí s `nawk(1)`

**banner**

`banner strings`

konstrukce plakátových písmen

souvisí s `echo(1)`

**basename**

`basename string [suffix]`

`dirname string`

vyjmutí holého jména souboru nebo adresáře z cesty

souvisí s `sh(1)`

**cal**

`cal [[month] year]`

tisk kalendáře daného měsíce daného roku

**calendar**

`calendar`

jednoduchý elektronický diář

souvisí s `ctime(3)`

**cat**

`cat [-s] [file ...]`

spojení a výpis obsahu souborů

**cc**

`cc [options] file ...`

spuštění překladače jazyka C

souvisí s `ld(1)`, `prof(1)`, `sdb(1)`, `adb(1)`, `exit(2)`, `monitor(3)`

**cflow**

`cflow [-c] [-r] [-ix] [-i_] [-dnum] [-V] files`

vytváří graf toku řízení

souvisí s `cc(1)`, `lex(1)`, `lint(1)`, `yacc(1)`

**chgrp**

`chgrp [-R] [-h] group filename ...`

změna vlastnictví skupiny souboru

souvisí s `chown(1)`, `groups(1)`, `passwd(1)`

**chmod**

`chmod [-R] mode files`

změna přístupových práv souboru

souvisí s `chmod(2)`, `ls(1)`

**chown**

`chown [-R] [-h] owner file ...`

změna vlastníka souboru

souvisí s `chmod(1)`, `chown(1)`

**cmp**

`cmp [-l] filename1 filename2 [skip1] [skip2]`

`cmp [-s] filename1 filename2 [skip1] [skip2]`

porovnání obsahu dvou souborů slabiku po slabice

souvisí s `comm(1)`, `diff(1)`

**col**

`col [-bfpx]`

filtr pro úpravu textu ve smyslu zpětných odkazů

souvisí s `nroff(1)`

**comm**

`comm [-[123]] file1 file2`

vyhledání společných řádků v textu dvou setříděných souborů

souvisí s `cmp(1)`, `diff(1)`, `sort(1)`, `uniq(1)`

**compress**

`compress [-cfv] [-b bits] [filename ...]`

`uncompress [-c] [filename ...]`

`zcat [filename ...]`

kódování a dekódování obsahu souboru do zhuštěné podoby

souvisí s `ln(1)`, `pack(1)`

**cp**

`cp [-ipr] file1 [file2 ...] target`

## E Vnější příkazy

`ln [-f] [-s] file1 [file2 ... ] target`

`mv [-fi] file1 [file2 ... ] target`

vytvoření kopie, vytvoření nového odkazu nebo přejmenování souboru

souvisí s `chmod(2)`, `chmod(1)`, `cpio(1)`, `rm(1)`

**cpio**

`cpio -o [acBL] [-C size] [-H header]`

`cpio -i[Bcdkmrtuvf] [-C size] [-H header] [-R id] [patterns]`

`cpio -p[adlLmruv] [-R id] directory`

vytváření archivu, výběr z archivu

souvisí s `cpio(4)`, `ar(1)`, `find(1)`, `ls(1)`, `sh(1)`, `tar(1)`

**cpp**

`$LIBDIR/cpp [option ... ] [ifile [ofile]]`

textový makroprocesor jazyka C

souvisí s `m4(1)`,

**crontab**

`crontab [file]`

`crontab -e [username]`

`crontab -r [username]`

`crontab -l [username]`

soubor požadavků uživatele na provedení příkazů v daném čase

souvisí s `sh(1)`, `cron(1)`

**csplit**

`csplit [-s] [-k] [-fprefix] file arg1 [ ... argn]`

rozdělení souboru na části uložené v nových souborech podle kontextu

souvisí s `ed(1)`, `sh(1)`

**ctags**

`ctags [-aBFtuwx] (-f tagsfile) filename ...`

vytvoření odkazového souboru pro editor `ex(1)`

souvisí s `ex(1)`, `find(1)`, `vi(1)`, `lex(1)`, `yacc(1)`

**cu**

`cu [-sspeed] [-l line] [-h] [-t] [-d] [-o | -e] [-n] telno`

`cu [-sspeed] [-h] [-d] [-o | -e] -lline`

`cu [-h] [-d] [-o | -e] systemname`

vstup uživatele do vzdáleného uzlu sítě

souvisí s `cat(1)`, `echo(1)`, `stty(1)`, `uname(1)`, `uucp(1)`

**cut**

`cut -clist [file1 file2 ... ]`

`cut -flist [-dchar] [-s] [file1 file2 ...]`

rozdělení souboru podle polí v každém řádku do jednotlivých souborů

```
souvisí s grep(1), paste(1), sh(1)
```

### cxref

```
cxref [options] files
```

program křížových odkazů

```
souvisí s cc(1)
```

### date

```
date [-u] [-a [-] sss.ff] mmdhmm[[cc]yy]
```

```
date [+format]
```

výpis nebo nastavení data a času

```
souvisí s printf(3)
```

### dd

```
dd [option=value] ...
```

konverze typů a obsahů souborů

### delta

```
delta [-rSID] [-s] [-n] [-glist] [-m[mrlist]] [-y[comment]]
```

```
[-p] file ...
```

vytvoření nové verze v souboru SCCS

```
souvisí s admin(1), get(1), prs(1), rmdel(1)
```

### df

```
df [-F FSType] [-begklntV] [-o specific_options]
```

```
[directory | special ... ]
```

výpis počtu volných bloků a i-uzlů svazku

```
souvisí s mount(8)
```

### diff

```
diff [-bitw] [-c | -e | -f | -h | -n] file1 file2
```

```
diff [-bitw] [-C num] file1 file2
```

```
diff [-bitw] [-D string] file1 file2
```

```
diff [-bitw] [-c | -e | -f | -h | -n] [-l] [-r] [-s] [-S name]
```

```
directory1 directory2
```

výpis rozdílu obsahu souborů

```
souvisí s cmp(1), comm(1), diff3(1), ed(1)
```

### diff3

```
diff3 [-exEX3] filename1 filename2 filename3
```

výpis řádků, ve kterých se tři soubory liší

```
souvisí s diff(1), ed(1)
```

### **dircmp**

```
dircmp [-d] [-s] [-wn] dir1 dir2  
porovnání obsahů dvou adresářů  
souvisí s cmp(1), diff(1)
```

### **dis**

```
dis [-o] [-V] [-L] [-F function] [-l string] files  
zpětný assembler  
souvisí s as(1), cc(1)
```

### **du**

```
du [-ars] [file ...]  
výpis využití disku
```

### **echo**

```
echo [arg] ...  
opis argumentů příkazu na standardní výstup  
souvisí s sh(1)
```

### **ed**

```
ed [-] [-p string] [file]  
red [-] [-p string] [file]  
textový editor a zúžený textový editor  
souvisí s grep(1), sed(1), sh(1)
```

### **egrep**

```
egrep [options] [expression] [files]  
fgrep [options] [strings] [files]  
vyhledání textového řetězce v souboru  
souvisí s ed(1), grep(1), sed(1)
```

### **env**

```
env [-] [name=value ...] [command]  
výpis seznamu proměnných prostředí provádění procesu nebo nastavení  
další proměnné do prostředí  
souvisí s sh[1]
```

### **eqn**

```
eqn [-dxy] [-pn] [-sn] [-ffont] [file ...]  
neqn [-dxy] [-pn] [-sn] [-ffont] [file ...]  
zpracování matematických vzorců pro přípravu dokumentace pomocí  
nroff(1)  
není uvedeno v SVID3  
souvisí s nroff(1)
```



**ex**

`ex [-] [-v] [-r] [-R] [+command] [-l] [file]`  
 textový editor

`souvisí s ed(1), vi(1), terminfo(4)`

**expr**

`expr expression`  
 vyhodnocení výrazu

`souvisí s ed(1), sh(1)`

**file**

`file [-f file] [-h] file`  
 určení obsahu souboru

**find**

`find path-name-list expression`  
 vyhledání souboru

`souvisí s chmod(1), cpio(1), sh(1), stat(2), test(1)`

**fmt**

`fmt [-cs] [-w width] [inputfile ... ]`  
 jednoduchá úprava dokumentace

`souvisí s mail(1), nroff(1), vi(1)`

**fmtmsg**

`fmtmsg [-e classification] [-u subclass] [-l label]`  
`[-s severity] [-t tag] [-a action] text`  
 zobrazení zprávy ve standardním tvaru na standardní chybový výstup a na  
 systémovou konzolu

`souvisí s fmtmsg(3)`

**ftp**

`ftp [-g] [-i] [-n] [-v] [server-host]`  
 program přenosu souborů počítačovou sítí  
není uvedeno v SVID3

`souvisí s sh(1), rcp(1)`

**gcore**

`gcore [-o filename] PID ...`  
 získání obsahu paměti běžícího procesu

`souvisí s kill(1), ptrace(2)`

**get**

`get [-rSID] [-ccutoff] [-e] [-b] [-ilist] [-xlist] [-k]`  
`[-l[p]] [-p] [-s] [-m] [-n] [-g] [-t] file ...`

vyjmutí verze zdrojového textu z souboru SCCS  
`souvisí s admin(1), delta(1), prs(1), what(1)`

`gettxt`  
`gettxt msgfile:msgnum [dflt_msg]`  
obnova textového řetězce z databáze zpráv  
`souvisí s gettxt(3), mkmsg(1), srchtxt(1)`

`grep`  
`grep [options] expression [files]`  
vyhledání textového řetězce v souboru  
`souvisí s ed(1), egrep(1), sed(1)`

`groups`  
`groups [user]`  
výpis skupin uživatele

`head`  
`head [-n] [filename ...]`  
zobrazí začátek souboru  
`souvisí s cat(1), tail(1)`

`id`  
`id [-a]`  
výpis číselné a slovní identifikace uživatele a dalších informací  
`souvisí s logname(1), getuid(2)`

`join`  
`join [-an] [-e string] [-jn m] [-o list] [-tc] file1 file2`  
spojení souborů na základě shody polí  
`souvisí s awk(1), comm(1), sort(1), uniq(1)`

`kill`  
`kill [-signal] PID ...`  
`kill -1`  
zaslání signálu procesu nebo výpis možných symbolických označení signálů  
`souvisí s kill(2), ps(1), signal(2)`

`ld`  
`ld [options] file ...`  
sestavující program  
`souvisí s ar(1), cc(1), strip(1)`

**lex**

`lex [-ctvn] [file] ...`  
 prostředek lexikální analýzy textu  
 souvisí s `cc(1)`, `yacc(1)`

**line**

`line`  
 ze standardního vstupu čte jeden řádek a zapisuje jej na standardní výstup  
 souvisí s `sh(1)`

**lint**

`lint [options] file ...`  
 podrobnější kontrola zdrojových textů jazyka C z hlediska syntaktické analýzy  
 souvisí s `cc(1)`, `cpp(1)`, `make(1)`

**listusers**

`listusers [-h] [-v]`  
`listusers [-g groups] [-l logins]`  
 výpis informací o registrovaném uživateli

**lorder**

`lorder file ...`  
 získání informací o třídění obsahu knihovny přeložených modulů  
 souvisí s `ar(1)`, `ld(1)`, `tsort(1)`

**lp**

`lp [printing-options] files`  
`lp -i request-IDs printing-options`  
`cancel [request-IDs] [printers]`  
`cancel -u login-IDs [printers]`  
 vytvoření požadavku ve frontě na tisk nebo zrušení požadavku z fronty tiskárny  
 souvisí s `lpstat(1)`, `mail(1)`, `terminfo(4)`

**lpstat**

`lpstat [options]`  
 výpis informací o stavu fronty požadavků na tiskárnu  
 souvisí s `lp(1)`

**ls**

`ls [options] [file ...]`  
 výpis obsahu adresáře  
 souvisí s `chmod(1)`, `find(1)`, `terminfo(4)`

## E Vnější příkazy

---

### **m4**

m4 [*options*] [*file* ...]  
textový makroprocesor

souvisí s cc(1), cpp(1)

### **mail**

mail [-epqr] [-f*file*]  
mail [-t] *name* ...  
rmail [-t] *name* ...  
pošle nebo čte příslou poštu

souvisí s sh(1), mailx(1)

### **mailx**

mailx [-e]  
mailx [-f*filename*] [-H] [-N] [-u*user*] [-i] [-n]  
mailx [-F] [-h*number*] [-r*address*] [-s*subject*] [-i] [-n] *name* ...  
podsystem příjmu a odesílání pošty

souvisí s ed(1), mail(1), pg(1), ls(1), vi(1)

### **make**

make [-f *makefile*] [-p] [-i] [-k] [-s] [-r] [-n] [-e] [-t]  
[-q] [*name* ...]  
údržba rozsáhlých programových celků

souvisí s cc(1), lex(1), sh(1), yacc(1)

### **mesg**

mesg [y | n]  
uzavření (nebo otevření) sezení pro příjem zpráv

souvisí s wall(8), write(1)

### **man**

man [-afbcw] [-t*proc*] [-p*pager*] [-d*dir*] [-T*term*] [*section*]  
[*title*]

výpis určené části referenční příručky

není uvedeno v SVID3

souvisí s nroff(1), tbl(1), troff(1)

### **mkdir**

mkdir [-m *mode*] [-p] [-M] [-l *level*] *dirname* ...  
vytvoření adresáře

souvisí s rm(1)

**more**

```
more [-cdfisu] [-lines] [+linenumber] [+/pattern] [filename ...]
page [-cdfisu] [-lines] [+linenumber] [+/pattern] [filename ...]
```

stránkování souboru na obrazovce terminálu

souvisí s `cat(1)`, `sh(1)`, `terminfo(4)`, `vi(1)`

**nawk**

```
nawk [-F fs] [prog] [inputfile ...]
nawk [-F fs] [-f progfile] [inputfile ...]
```

programovací jazyk pro práci s texty

souvisí s `awk(1)`, `grep(1)`, `lex(1)`, `printf(3)`, `sed(1)`

**newgrp**

```
newgrp [-] [group]
```

změna aktivní skupiny uživatele

souvisí s `sh(1)`

**news**

```
news [-a] [-n] [-s] [items]
```

výpis nových zpráv

**nice**

```
nice [-incremental] command
```

spuštění procesu s nižší prioritou

souvisí s `nice(2)`, `prctl(2)`

**nl**

```
nl [-htype] [-btype] [-ftype] [-vstart#] [-iincr] [-p]
    [-lnum] [-ssep] [-wwidth] [-nformat] [-ddelim] [file]
```

očíslování řádků vstupního souboru

souvisí s `pr(1)`

**nm**

```
nm [options] file ...
```

výpis tabulky symbolů přeloženého modulu

souvisí s `cc(1)`, `ld(1)`

**nohup**

```
nohup command [arguments]
```

spuštění procesu v ochraně při zrušení otce

souvisí s `sh(1)`, `signal(2)`

**nroff**

```
nroff [option ...] [file ...]
```

```
troff [options ...] [files]
```

příprava dokumentace, příprava textu pro práci s fotosázecím strojem  
není uvedeno v SVID3

`souvisí s col(1), eqn(1), tbl(1)`

**od**

`od [-bcDdFfosvx] [file] [[+]offset[.][b]]`

výpis obsahu souboru v kódování podle ASCII v osmičkové soustavě

**pack**

`pack [-] [-f] name ...`

`pcat name ...`

`unpack name ...`

sbalení a rozbalení obsahu souboru

`souvisí s cat(1)`

**passwd**

`passwd [name]`

změna uživatelského hesla

**paste**

`paste file1 file2`

`paste -d list file1 file2 ...`

`paste -s [-d list] file1 file2`

spojení více souborů do jednoho spojením řádků jednotlivých souborů

`souvisí s cut(1), grep(1), pr(1)`

**pg**

`pg [-number] [-p string] [-cefns] [+linenumber] [+pattern/]  
[files ...]`

zobrazení souborů na obrazovce po stránkách

`souvisí s ed(1), grep(1), more(1)`

**pr**

`pr [[-column] [-wwidth] [-a]] [-e[c]k] [-i[c]k] [-dtrfp] [+page]  
[-n[c]k] [-offset] [-llength] [-sseparator] [-hheader] [-F]  
[file ...]`

`pr [[-ml [-wwidth] [-e[c]k] [-i[c]k] [-dtrfp] [+page] [-n[c]k]  
[-offset] [-llength] [-sseparator] [-hheader]  
[-F] [file ...]`

úprava obsahu souboru před tiskem na tiskárně

`souvisí s lp(1)`

**printf**

`printf format [arg ...]`

výpis textu na standardním výstupu v určeném formátu

**souvisí** `sprintf(3)`

**prof**

`prof [-tcan] [-ox] [-q] [-z] [-m mdata] [prog]`  
 zobrazí data vytvořená trasováním jiného procesu  
**souvisí** `s cc(1), exit(2), profil(2), monitor(3), mark(3)`

**prs**

`prs [options] files`  
 výpis souboru SCCS  
**souvisí** `s admin(1), delta(1), get(1), what(1)`

**ps**

`ps [options]`  
 výpis stavu procesů

**pwd**

`pwd`  
 výpis jména pracovního adresáře

**rcp**

`rcp [-r] file1 [file2 ...] target`  
 kopie souborů mezi vzdálenými uzly sítě  
není uvedeno v SVID3  
**souvisí** `s cp(1), ftp(1), remsh(1)`

**remsh**

`remsh host [-l username] [-n] command`  
 provedení příkazu ve vzdáleném uzlu sítě  
není uvedeno v SVID3  
**souvisí** `s rlogin(1)`

**rlogin**

`rlogin rhost [-e c] [-7] [-8] [-l username]`  
 vstup uživatele do vzdáleného systému v síti  
**souvisí** `s remsh(1), telnet(1), sh(1), stty(1), termio(5)`

**rm**

`rm [-fri] file ...`  
`rmdir dir ...`  
 zrušení položky souboru nebo adresáře  
**souvisí** `s unlink(2)`

**rmdel**

`rmdel -rSID files`

## E Vnější příkazy

---

zruší verzi ze souboru SCCS  
`souvisí s delta(1), get(1), prs(1)`

`ruptime`  
`ruptime [-a] [-rl [-l] [-t] [-u]`  
výpis stavu lokálního uzlu sítě  
není uvedeno v SVID3  
`souvisí s rwho (1)`

`rwho`  
`rwho [-a]`  
stav přihlášených uživatelů místní sítě  
není uvedeno v SVID3  
`souvisí s ruptime(1)`

`sact`  
`sact file ...`  
výpis současné aktivity editace souboru SCCS  
`souvisí s delta(1), get(1), unget(1)`

`sdb`  
`sdb [objfile [corfile [directory-list]]]`  
symbolický dynamický ladicí prostředek  
`souvisí s adb(1), cc(1), ed(1), sh(1)`

`sed`  
`sed [-n] [-e script] [-f sfile] [files]`  
neinteraktivní editor  
`souvisí s awk(1), ed(1), grep(1)`

`sh`  
`sh [flags] [args]`  
`jsh [flags] [args]`  
`rsh [flags] [args]`  
shell, příkazový interpret, zúžený příkazový interpret  
`souvisí s dup(2), echo(1), exec(2), fork(2), kill(1), pipe(2), pwd(1),`  
`signal(2), system(3), test(1), ulimit(2), umask(2), umask(1),`  
`wait(2)`

`shl`  
`shl`  
správce více současně aktivních shellů  
`souvisí s ioctl(2), sh(1), signal(2), stty(1)`



**size**

```
size [-o] [-x] [-V] file ...
```

výpis velikostí sekcí přeložených modulů

```
souvisí s cc(1), ld(1)
```

**sleep**

```
sleep time
```

pozastavení provádění po daný interval

```
souvisí s alarm(2), sleep(3)
```

**sort**

```
sort [-cmu] [-ooutput] [-y[kmem]] [-zrecsz] [-dfinr] [-btx]
      [+pos1 [-pos2]] [files]
```

třídění obsahu souboru

```
souvisí s comm(1), join(1), uniq(1)
```

**spell**

```
spell [-v] [-b] [-x] [+local_file] [files]
```

vyhledání neexistujících slov v textu podle slovníku

**split**

```
split [-n] [file [name]]
```

rozdělení souboru na části uložené v nových souborech

```
souvisí s csplit(1)
```

**strchg**

```
strchg -h module1 [, module2 ...]
```

```
strchg -p [-a | -u module]
```

```
strchg -f file
```

```
strconf [-t | -m module]
```

změna PROUDu (vsunutí nebo vyjmutí dalšího modulu) nebo získání informací o PROUDu

```
souvisí s ioctl(2)
```

**strip**

```
strip [-x] [-r] [-V] file ...
```

zrušení tabulky symbolů z přeloženého modulu

```
souvisí s ar(1), cc(1), ld(1)
```

**stty**

```
stty [-a] [-g] [options]
```

nastavení charakteristik terminálu

```
souvisí s termio(5)
```

## E Vnější příkazy

---

### **su**

`su [-] [name [arg ...]]`

nastavení práv superuživatele nebo jiného uživatele v rámci sezení

souvisí s `sh(1)`

### **sum**

`sum [-r] file`

výpis kontrolního součtu a počtu bloků souboru

souvisí s `wc(1)`

### **sync**

`sync`

přepis systémové vyrovnávací paměti na připojené svazky, aktualizace dat na fyzickém médiu

souvisí s `sync(2)`

### **tabs**

`tabs [tabspec] [+mn] [-Ttype]`

nastavení klávesy tabulátoru na pozici příkazového řádku

### **tail**

`tail [--[number] [lbcrl]] [file]`

`tail [--[number] [lbcfl]] [file]`

výpis konce souboru

### **tar**

`tar [options] [file ...]`

archivace na externí médium

### **tbl**

`tbl [files]`

program vytváření tabulek pro zpracování programem `nroff(1)`

není uvedeno v SVID3

souvisí s `eqn(1)`, `nroff(1)`

### **tee**

`tee [-i] [-a] [file] ...`

kopie obsahu roury do souboru

### **telnet**

`telnet [host [port]]`

virtualizace terminálu do vzdáleného uzlu sítě

není uvedeno v SVID3

souvisí s `sh(1)`, `stty(1)`, `termio(5)`

**test**`test expr`

vrací návratový status podle vyhodnocení výrazu

`souvisí s find(1), sb(1)`**time**`time command`

měření uživatelského a systémového času spotřebovaného příkazem

**touch**`touch [-amc] [mmddhhmm[yy]] [file ...]`

změna času přístupu a času modifikace souboru

**tty**`tty [-s]`

výpis jména speciálního souboru uživatele terminálu

**tr**`tr [-cds] [string1 [string2]]`

záměna znaků

**true**`true``false`

vrací pravdivý (nebo nepravdivý) návratový status

`souvisí s sh(1)`**truss**`truss [-pfcae] [-t[!]syscall[, syscall ...]] [-v[!]syscall  
[, syscall ...]] [-x[!]syscall[, syscall ...]] [-s[!]signal  
[, signal ...] ] [-m[!]fault[, fault ...]] [-r[!]fd[, fd ...]]  
[-w[!]fd[, fd ... ]]  
[-o outfile] command`

sledování volání jádra procesu

**tsort**`tsort [file]`

topologické třídění

`souvisí s lorder(1)`**umask**`umask [ooo]`

změna masky přístupových práv při vytváření souborů

`souvisí s chmod(1)`

## E Vnější příkazy

---

### **uname**

`uname [-snrvma]`

výpis jména provozní verze UNIXu

`souvisí s uname(2)`

### **unget**

`unget [-rSID] [-s] [-n] files`

provede restauraci stavu souboru SCCS před naposledy provedeným

`get(1)`

`souvisí s delta(1), get(1), sact(1)`

### **uniq**

`uniq [-udc] [+n] [-n]] [input [output]]`

zpráva o opakujících se řádcích v souboru

`souvisí s comm(1), sort(1)`

### **uucp**

`uucp [-c] [-C] [-d] [-f] [-j] [-m] [-nuser] [-r] source_files  
destination_file`

`uulog [-ssystem]`

`uname [-l]`

kopie souborů mezi různými uzly sítě sériovým rozhraním

`souvisí s mail(1), uustat(1), uux(1)`

### **uustat**

`uustat [-m]`

`uustat [-q]`

`uustat [-kjobid]`

`uustat [-rjobid]`

`uustat [-ssys] [-uuser]`

výpis stavu fronty požadavků na přenos podsystému UUCP

`souvisí s uucp(1)`

### **uux**

`uux [options] command-string`

provedení příkazu ve vzdáleném uzlu sítě

`souvisí s mail(1), sh(1), uucp(1), uustat(1)`

### **val**

`val -`

`val [-s] [-rSID] [-mname] [-ytype] file ...`

test správnosti obsahu souboru SCCS

`souvisí s admin(1), delta(1), get(1), prs(1)`

**vi**

`vi [-rfile] [-l] [-wn] [-R] [+command] file ...`

obrazkově orientovaný editor

souvisí s `ex(1)`

**wc**

`wc [-lwc] [files]`

výpočet počtu znaků, slov a řádků v souboru

**what**

`what [-s] files`

zjištění obsahu souboru SCCS

souvisí s `get(1)`

**who**

`who [-uTlHqpdbrtas] [file]`

výpis seznamu přihlášených uživatelů a informace o stavu systému

souvisí s `getut(3)`

**write**

`write user [terminal]`

zápis zprávy přihlášenému uživateli

souvisí s `getut(3)`, `msg(1)`, `wall(B)`, `who(1)`

**xargs**

`xargs [options] [command [initial-arguments]]`

vytvoření seznamu argumentů a provedení příkazu

souvisí s `echo(1)`, `sh(1)`

**yacc**

`yacc [-vdlr] grammar`

prostředek pro vytvoření automatu k popsání gramaticy umělého jazyka

souvisí s `lex(1)`

## Příloha F – Vnější příkazy správce systému

Příloha odráží situaci v UNIX SYSTEM V a SVID, protože ostatní systémy byly komentovány dříve. Všechny příkazy odpovídají SVID3.

```
acct
accton [file]
acctwtmp "reason"
chargefee login-name number
ckpacct [blocks]
dodisk [files]
lastlogin
omacct number
prdaily [-l] [-c] [mmd]
prtacct file ["heading"]
shutacct ["reason"]
startup
turnacct on
turnacct off
turnact switch
```

příkazy pro podporu účtování

```
souvisí s acct(2), acctcms(8), acctcom(8), acctcon(8), acctmerg(8),
      acctprc(8), cron(8), diskusg(8), fwtmp(8), runacct(8)
```

```
acctcms
acctcms [-a [-p] [-o]] [-c] [-j] [-n] [-s] files
      výpis stavu účtování
```

```
souvisí s acct(8), acct(2), acctcom(8), acctcon(8), acctmerg(8),
      acctprc(8), fwtmp(8), runacct(8)
```

```
acctcom
acctcom [[options] [file]] ...
      zpracování účtovacích informací vytvořených acct(8)
```

```
souvisí s acct(2), acct(8), acctcms(8), acctcon(8), acctmerg(8),
      acctprc(8), ed(1), fwtmp(8), regcmp(3), runacct(8)
```

```
acctcon
acctcon1 [-p] [-t] [-l file] [-o file]
acctcon2
prctmp
```

zpracování záznamů o vstupu uživatelů do systému

```
souvisí s acct(2), acct(8), acctcms(8), acctcom(8), acctmerg(8),
      acctprc(8), fwtmp(8), runacct(8)
```

**acctmerg**

acctmerg [-a] [-i] [-p] [-t] [-u] [*file* ...]

slučování informací o účtování z několika souborů

souvisí s acct(2), acct(8), acctcms(8), acctcom(8), acctcon(8),  
acctprc(8), fwtmp(8), runacct(8)

**acctprc**

acctprc1 [*ctmp*]

acctprc2

zobrazení účtovacích informací v podobě tabulek

souvisí s acct(2), acct(8), acctcms(8), acctcom(8), acctcon(8),  
acctmerg(8), fwtmp(8), runacct(8)

**adv**

adv [[-r] [-d *description*] *resource* *pathname* [*client* ...]]

adv -m *resource* -d *description* [*client* ...]

adv -m *resource* -d *description* *client* ...

nabídka podstromu adresáře pro počítačovou síť

souvisí s mailx(1), mount(8), rfstat(8), share(8), unadv(8), unshare(8)

**backup**

backup -i [-s] [-t *table*] [-o *orig*] [-e | n] [-c *week:day*] [-m *user*]

backup -i [-s] [-t *table*] [-o *orig*] [-e | n] [*demand*] [-m *user*]

backup -i [-v] [-t *table*] [-o *orig*] [-e | n] [-c *week:day*] [-m *user*]

backup -i [-v] [-t *table*] [-o *orig*] [-e | n] [*demand*] [-m *user*]

backup -a [-t *table*] [-o *orig*] [-e | n] [-c *week:day*] [-m *user*]

backup -a [-t *table*] [-o *orig*] [-e | n] [*demand*] [-m *user*]

backup -S [-u *user*]

backup -S [-A]

backup -S [-j *jonbid*]

backup -R [-u *user*]

backup -R [-A]

backup -R [-j *jonbid*]

backup -C [-u *user*]

backup -C [-A]

backup -C [-j *jonbid*]

vstup do podsystému zálohy dat

souvisí s bkhistory(8), bkoper(8), bkreg(8), bkstatus(8)

**bkexcept**

bkexcept -a *pattern* ... [-t *exception-list*]

bkexcept -r *pattern* ... [-t *exception-list*]

bkexcept [-d *pattern* ...] [-t *exception-list*]

změny nebo jen zobrazení v seznamu souborů, které budou při úschově  
dat vynechávány

`souvisí s backup(8), cpio(1), ed(1), incfile(8), sh(1)`

### **bkhistory**

`bkhistory [-lh] [-d dates] [-o origs] [-t tags] [-f c]`

`bkhistory -p period`

*zpráva o úschově dat*

`souvisí s backup(8), bkreg(8), date(1), ls(1)`

### **bkoper**

`bkoper [-u users]`

*uživatelsky příjemné řízení úschovy a obnovy dat*

`souvisí s bkreg(8), bkstatus(8), getvol(8), mailx(1)`

### **bkreg**

`bkreg -p period [-w cweek] [-t table]`

`bkreg -a tag -o orig -d ddev -c weeks:days -m method  
[-b moptions] [-t table] [-P prio] [-D depend]`

`bkreg -a tag -o orig -d ddev demand -m method [-b moptions]  
[-t table] [-P prio] [-D depend]`

`bkreg -e tag [-o orig] [-c weeks:days] [-m method] [-d dev]  
[-b moptions] [-t table] [-P prio] [-D depend]`

`bkreg -e tag [-o orig] [demand] [-m method] [-d dev]  
[-b moptions] [-t table] [-P prio] [-D depend]`

`bkreg -r tag [-t table]`

`bkreg -C fields [-hv] [-t table] [-f c] [-c weeks:days]`

`bkreg -C fields [-hv] [-t table] [-f c] [demand]`

`bkreg -A [-hsv] [-t table] [-c weeks:days]`

`bkreg -A [-hsv] [-t table] [demand]`

`bkreg -O [-hsv] [-t table] [-c weeks:days]`

`bkreg -O [-hsv] [-t table] [demand]`

`bkreg -R [-hsv] [-t table] [-c weeks:days]`

`bkreg -R [-hsv] [-t table] [demand]`

*práce s tabulkou úschovy dat*

`souvisí s backup(8), fdisk(8), fdp(8), file(8), fimage(8), incfile(8),  
mkfs(8), mount(8), restore(8), volcopy(8)`

### **bkstatus**

`bkstatus [-a] [-h] [-f c] [-j jobids] [-u users]`

`bkstatus [-s states] [-h] [-f c] [-j jobids] [-u users]`

`bkstatus -p period`

*výpis stavu operace zálohování dat*

`souvisí s backup(8), bkhist(8), bkreg(8)`



**captoinfo**

`captoinfo [-v ...] [-V] [-l] [-w width] file ...`

převod popisu terminálu z dříve používané databáze terminálů  
termcap(4) do terminfo(4)

`souvisí s infocmp(8), terminfo(4)`

**chroot**

`chroot newroot command`

provedení příkazu se změnou kořenového adresáře

`souvisí s chdir(1)`

**cron**

`cron`

démon sledující čas

`souvisí s at(1), crontab(1), sh(1)`

**defadm**

`defadm`

`defadm [filename [name [=value]][name[=value]] [...]]`

`defadm [-d filename name [name] [...]]`

zobrazení nebo změna implicitních hodnot daných obsahem adresáře  
/etc/default

`souvisí s cron(8), passwd(1), su(1), useradd(8), userdel(8)`

**devnm**

`devnm pathname`

zjistí jméno speciálního souboru připojeného svazku

**dfmounts**

`dfmounts [-F fstype] [-h] [-o specific-options] [restriction ...]`

zobrazení připojených systémů souborů nabídnutých do sítě

`souvisí s fumounts(8), dfshares(8), mount(8), share(8), unshare(8)`

**dfshares**

`dfshares [-F fstype] [-h] [-o specific-options] [server ...]`

seznam dostupných systémů souborů vzdálených uzlů sítě

`souvisí s dfmounts(8), mount(8), share(8), unshare(8)`

**diskusg**

`diskusg [-s] [-v] [-i fnmlist] [-p file] [-u file] [special-file ...]`

`acctdisk`

výpis obsazení svazku uživatelem

`souvisí s acct(8), volcopy(8)`

### **dnname**

`dnname [-D domain] [-N netspec] [-dna]`

výpis nebo nastavení jmen sítě a domén vzdálených systémů

souvisí s `nsquery(8)`, `rfstat(8)`

### **fdisk**

`fdisk -B [-dovAENV] bkjobid odname opartdev odlab descript`

`fdisk -RC [-dovAENV] odname opartdev descript refsname redev  
rsjobid`

archivace a obnova dat celého systému souborů, je používán jako syn

`backup(8)`

souvisí s `backup(8)`, `dump(8)`, `fdp(8)`, `file(8)`, `fimage(8)`, `incfile(8)`,  
`restore(8)`, `rsoper(8)`, `volcopy(8)`

### **fdp**

`fdp -B [-dovAENSJ [-c count] bkjobid odpname odpdev odplab  
descript`

`fdp -RC [-dovAENS] [-c count] odpname odpdev descript refsname  
redev rsjobid`

vytvoření nebo obnova dat, úschova celého svazku, je používán jako syn

`backup(8)`,

souvisí s `backup(8)`, `bkoper(8)`, `fimage(8)`, `incfile(8)`, `restore(8)`,  
`rsoper(8)`, `volcopy(8)`

### **file**

`file -B [-dlmortvAENSV] bkjobid ofsname ofsdev ofslab descript`

`file -RC [-dlmortvAENSV] ofsname ofsdev descript refsname redev  
rsjobid`

`file -RF [-dlmortvAENSV] ofsname ofsdev descript`

`rsjobid:uid:date:type:name[:[rename]:[inode]] ...`

archivace a obnova archivu dat systému souborů, je používán jako syn

`backup(8)`

souvisí s `backup(8)`, `bkoper(8)`, `cpio(1)`, `fdp(8)`, `fimage(8)`, `incfile(8)`,  
`ls(1)`, `restore(8)`, `rsoper(8)`, `time(2)`, `urestore(8)`, `volcopy(8)`

### **fimage**

`fimage -B [-dlmotuvAENS] bkjobid ofsname ofsdev ofslab descript`

`fimage -RC [-dlmotuvAENS] ofsname ofsdev descript refsname redev  
rsjobid`

`fimage -RF [-dlmotuvAENS] ofsname ofsdev descript rsjobid:uid:`

`date:type:name[:[rename]:[inode]] ...`

vytvoření nebo obnova dat z archivu celého systému souborů, je používán  
jako syn `backup(8)`

souvisí s `backup(8)`, `bkoper(8)`, `fdp(8)`, `file(8)`, `incfile(8)`, `ls(1)`,  
`restore(8)`, `rsoper(8)`, `time(2)`, `urestore(8)`, `volcopy(8)`

**fsck**

```
fsck [-F FSType] [-V] [-m [-o specific-options] [special ...]
```

kontrola a oprava svazku

souvisí s `mkfs(8)`

**fsdb**

```
fsdb [-F FSType] [-V] [-o specific-options] special
```

nástroj pro ladění struktury svazku

souvisí s `fsck(8)`

**fstyp**

```
fstyp [-v] special
```

určení typu svazku

souvisí s `fsck(8)`

**fumount**

```
fumount [-w sec] resource
```

násilné odpojení lokálního systému souborů nabízeného do sítě

souvisí s `adv(8)`, `mount(8)`, `share(8)`, `umount(8)`, `unshare(8)`, `unadv(8)`

**fusage**

```
fusage
```

```
fusage mount-point ...
```

```
fusage advertised-resource ...
```

```
fusage blk-special-dev ...
```

výpis využití nabízeného disku v síti klienty

souvisí s `adv(8)`, `mount(8)`, `share(8)`

**fuser**

```
fuser [-cfku] files [-[cfku] files] ...
```

výpis procesů, které pracují s danými soubory

souvisí s `kill(2)`

**fwtmp**

```
fwtmp [-ic]
```

```
wtmpfix [files]
```

práce se záznamy účtování vstupu uživatelů do systému

souvisí s `acct(2)`, `acct(8)`, `acctcms(8)`, `acctcom(8)`, `acctcon(8)`,  
`acctmerg(8)`, `acctprc(8)`, `runacct(8)`

**groupadd**

```
groupadd [-g gid [-o]] group
```

vytvoří novou skupinu

souvisí s `groupdel(8)`, `groupmod(8)`

### **groupdel**

*groupdel group*

zruší skupinu

souvisí s *groupadd(8),groupmod(8)*

### **groupmod**

*groupmod [-g gid [-o]] [-n name] group*

provádí změny v registru skupin

souvisí s *groupadd(8),groupdel(8)*

### **idload**

*idload [-n] [-g g-rules] [-u u-rules] [directory]*

*idload -k*

vytvoření transakční tabulky identifikace uživatelů v rámci různých uzlů sítě

souvisí s *mount(8)*

### **incfile**

*incfile -B [-dilmortvxAENSV] [-b days] bkjobid ofsname ofsdev  
ofslab descript [exception-list]*

*incfile -B [-dilmortvxAENSV] [-b incl] bkjobid ofsname ofsdev  
ofslab descript [exception-list]*

*incfile -T bkjobid tocfname descript*

*incfile -RC [-dilmortvxAENSV] ofsname ofsdev refsname redev  
rsjobid descript*

*incfile -RF [-dilmortvxAENSV] ofsname ofsdev descript  
rsjobid:uid:date:type:name[:[rename]:[inode]] ...*

úschova nebo obnova částí systému souborů, je používán jako syn  
backup(8)

souvisí s *backup(8),bkoper(8),bkreg(8),cpio(1),fdp(8),file(8),  
fimage(8),ls(1),restore(8),rsoper(8),time(2),volcopy(8)*

### **infocmp**

*infocmp [-d] [-c] [-n] [-I] [-L] [-C] [-r] [-u]*

*[-s d | i | l | c] [-v] [-V] [-1] [-w width]*

*[-A directory] [-B directory] [termname ...]*

porovnání nebo výpis z binární podoby do různých formátů položky  
databáze terminfo(4)

souvisí s *captoinfo(8),terminfo(4)*

### **init**

*init [0123456sSqQ]*

změna úrovně práce systému

souvisí s *who(1)*

**installf**

```
installf [-c class] pkginst pathname [ftype [[major minor]
mode owner group]]
```

```
installf [-c class] pkginst -
```

```
installf -f [-c class] pkginst
```

instalace souboru dat v rámci instalace programového vybavení

```
souvisí s pkgadd(8), pkgask(8), pkgchk(8), pkginfo(8), pkgmk(8),
pkgparam(8), pkgproto(8), pkgrm(8), pkgtrans(8), removef(8)
```

**ipcrm**

```
ipcrm [-q msqid] [-m shmid] [-Q msgkey] [-M shmkey] [-S semkey]
```

zrušení identifikačního klíče fronty zpráv, sdílené paměti nebo semaforů

```
souvisí s ipcs(8), msgctl(2), msgget(2), msgop(2), semctl(2), semget(2),
semop(2), shmctl(2), shmget(2), shmop(2)
```

**ipcs**

```
ipcs [options]
```

výpis zprávy o stavu komunikace mezi procesy (IPC)

```
souvisí s msgop(2), semop(2), shmop(2)
```

**killall**

```
killall [signal]
```

vyslání signálu všem procesům

```
souvisí s kill(1), kill(2), signal(2)
```

**last**

```
last [-number] [-f filename] [name | tty] ...
```

zjistí informace o posledním přihlášení uživatele nebo informace  
o přihlášení na terminálu

**link**

```
link file1 file2
```

```
unlink file
```

vytvoření dalšího odkazu nebo zrušení odkazu na soubor

```
souvisí s link(2), unlink(2)
```

**lfmt**

```
lfmt [-cl [-f flags] [-l label] [-s severity] [-g catalog:msgnum]
format [args]
```

zobrazení chybových zpráv na standardní výstup

**logins**

```
logins [-abdhmopstuvx] [-g groups] [-l logins]
```

vypisuje informace o uživateli

```
souvisí s passwd(1), useradd(8), usermod(8), userdel(8)
```

### logkeeper

```
logkeeper [-o] [logfile]  
logkeeper -a [-n num] [-s size] [-k nlines] [-f dest] logfile  
logkeeper -r [-k nlines] [-f dest] logfile  
logkeeper -r logfile  
logkeeper -c
```

sledování souboru přihlášení uživatele

```
souvisí s at(1), cron(8), crontab(1), defadm(1), msgalert(8), msglog(8),  
msggrpt(8)
```

### migration

```
migration -B [-AENS] bkjobid ofsname ofsdev ofslab descript  
přenos dat mezi různými skupinami archivu dat, je používán jako syn  
backup(8)
```

```
souvisí s awk(1), backup(8), bkoper(8), grep(1), ls(1), restore(8),  
rsoper(8), sed(1), time(1), urestore(8), volcopy(8)
```

### mkfifo

```
mkfifo path...
```

vytváří soubor pojmenované roury

```
souvisí s mkfifo(2)
```

### mkfs

```
mkfs [-F FSType] [-V] [-m] [-o specific-options] special  
[operands]
```

vytvoření svazku

### mkmsgs

```
mkmsgs [-o] [-i locale] inputstrings msgfile
```

vytvoření zprávy pro použití pomocí `gettext(1)`

```
souvisí s gettext(1), gettext(3), setlocale(2), srchtxt(8)
```

### mknod

```
mknod name b major minor
```

```
mknod name c major minor
```

```
mknod name p
```

vytvoření speciálního souboru

```
souvisí s mknod(2)
```

### mount

```
mount [-v]
```

```
mount [-p]
```

```
mount [-F FSType] [-V] [-r] [-o specific-options] special
```

```
mount [-F FSType] [-V] [-rl] [-o specific-options] mount-point
```

```
mount [-F FSType] [-V] [-r] [-o specific-options]
    special mount-point
umount [-V] [-o specific-options] special
umount [-V] [-o specific-options] mount-point
    připojení a odpojení svazku nebo části systému souborů nabízeného sítí
souvisí s setmnt(8), mount(2), umount(2)
```

```
mvdir
mvdir dirname name
    změna jména nebo umístění adresáře
```

```
ncheck
ncheck [-F FSType] [-V] [-o specific-options] (special ...)
    výpis seznamu jmen daného svazku
souvisí s fsck(8)
```

```
nsquery
nsquery [-h] [name]
    poskytnutí informací o lokálních nebo vzdálených systémech souborů v síti
souvisí s adv(8), share(8), unadv(8), unshare(8)
```

```
pkgadd
pkgadd [-d device] [-r response] [-n] [-a admin]
    [pkginst1[,pkginst2 ...]]
pkgadd -s spool [-d device] [pkginst][pkginst2 ...]
    instalace dalšího programového vybavení
souvisí s pkgask(8), pkgchk(8), pkgrm(8), pkginfo(8), pkgtrans(8)
```

```
pkgask
pkgask [-d device] -r response pkginst [pkginst ...]
    vytváření souboru s texty pro interakci při instalaci dalšího programového
    vybavení
souvisí s installf(8), pkgadd(8), pkgchk(8), pkginfo(8), pkgmk(8),
    pkgparam(8), pkgproto(8), pkgrm(8), pkgtrans(8), removef(8)
```

```
pkgchk
pkgchk [-l | -acfqv] [-nx] [-p path1[,path2 ...]] [-i file]
    [pkginst ...]
pkgchk -d device [-l|v] [-p path1[,path2, ...]] [-i file]
    [pkginst ...]
pkgchk -m pkgmap [-e envfile] [-l | -acfqv] [-nx] [-i file]
    [-p path1[,path2 ...]]
    kontrola správnosti provedené instalace
souvisí s pkgadd(8), pkgask(8), pkginfo(8), pkgrm(8), pkgtrans(8)
```

### pkginfo

```
pkginfo [-q|x|l] [-p|i] [-a arch] [-v version] [-r]
        [-c category1[,category2 ...]] [pkginst[,pkginst ...]]
pkginfo [-d device] [-q|x|l] [-a arch] [-v version]
        [-c category1[,category2 ...]] [pkginst[,pkginst ...]]
```

výpis informací o instalovaném programovém vybavení

```
souvisí s pkgadd(8), pkgask(8), pkgch(8), pkginfo(8), pkgrm(8),
        pkgtrans(8)
```

### pkgmk

```
pkgmk [-o] [-d device] [-r rootpath] [-b basdir] [-l limit]
        [-a arch] [-v version] [-p pstamp] [-f prototype]
        [variable=value ...] [pkginst]
```

vytváří distribuční verzi programového vybavení akceptovatelnou  
příkazem pkgadd(8)

```
souvisí s df(1), installf(8), pkgparam(8), pkgproto(8),
        pkgtrans(8), removef(8) uname(1)
```

### pkgparam

```
pkgparam [-v] [-d device] pkginst [param ...]
pkgparam -f file [-v] [param ...]
```

zobrazí hodnotu daných parametrů podle instalace

```
souvisí s installf(8), pkgmk(8), pkgparam(3), pkgproto(8), pkgtrans(8),
        removef(8)
```

### pkgproto

```
pkgproto [-i] [-c class] [path1[=path2] ...]
```

vytváří vstup popisu souborů pro pkgmk(8)

```
souvisí s installf(8), pkgmk(8), pkgparam(8), pkgtrans(8), removef(8)
```

### pkgrm

```
pkgrm [-n] [-a admin] [pkginst1[,pkginst2[, ...]]]
pkgrm -s spool [pkginst]
```

zrušení instalovaného programového vybavení

```
souvisí s installf(8), pkgadd(8), pkgask(8), pkgchk(8), pkginfo(8),
        pkgmk(8), pkgparam(8), pkgproto(8), pkgtrans(8), removef(8)
```

### pkgtrans

```
pkgtrans [-i] [-o|n] device1 device2 [pkginst1[,pkginst2[, ...]]]
```

převod dat vhodných pro instalaci do jiného formátu

```
souvisí s installf(8), pkgadd(8), pkgask(8), pkginfo(8), pkgmk(8),
        pkgparam(8), pkgproto(8), pkgrm(8), removef(8)
```



**priocntl**

```
priocntl -l
priocntl -d [-i idtype] [idlist]
priocntl -s [-c class] [class specific options] [-i idtype] [idlist]
priocntl -e [-c class] [class specific options] command [arguments]
```

řízení práce procesů

```
souvisí s exec(2), fork(2), nice(1), priocntl(2), ps(1)
```

**prtconf**

```
prtconf
```

výpis konfigurace systému

**pwck**

```
pwck [file]
grpck [file]
```

kontrola registru uživatelů a skupin

**removef**

```
removef pkginst path1 [path2 ...]
removef -f pkginst
```

zrušení souboru z instalovaného programového vybavení

```
souvisí s installf(8), pkgadd(8), pkgask(8), pkgchk(8), pkginfo(8),
      pkgmk(8), okgparam(8), pkgproto(8), pkgtrans(8), removef(8)
```

**restore**

```
restore [-o target] [-d date] [-m] [-s] -P partdev
restore [-o target] [-d date] [-n] [-s] -P partdev
restore [-o target] [-d date] [-m] [-v] -P partdev
restore [-o target] [-d date] [-n] [-v] -P partdev
restore [-o target] [-d date] [-m] [-s] -S odevice
restore [-o target] [-d date] [-n] [-s] -S odevice
restore [-o target] [-d date] [-m] [-v] -S odevice
restore [-o target] [-d date] [-n] [-v] -S odevice
restore [-o target] [-d date] [-m] [-s] -A partdev
restore [-o target] [-d date] [-n] [-s] -A partdev
restore [-o target] [-d date] [-m] [-v] -A partdev
restore [-o target] [-d date] [-n] [-v] -A partdev
```

obnova dat z archivace, systému souborů, částí dat nebo svazků

```
souvisí s at(1), fdisk(8), file(8), getdate(3), incfile(8), ls(1), mail(1),
      rsoper(8), rsstatus(8), urestore(8), ursstatus(8)
```

**rfadmin**

```
rfadmin
rfadmin -a hostname
rfadmin -r hostname
```

`rfadmin -o option`

`rfadmin -p`

`rfadmin -q`

údržba síťové aplikace Remote File Sharing

`souvisí s passwd(8), rfpasswd(8), rfstart(8)`

**rfpasswd**

`rfpasswd`

změna hesla uzlu v síti

`souvisí s passwd(8), rfstart(8)`

**rfstart**

`rfstart [-v] [-p primary-addr]`

spuštění síťové aplikace Remote File Sharing

`souvisí s dname(8), rfadmin(8), rfpasswd(8), rfstop(8)`

**rfstop**

`rfstop`

zastavení síťové aplikace Remote File Sharing

`souvisí s adv(8), dfmounts(8), fumount(8), mount(8), rfadmin(8),  
rfstart(8), rmntstat(8), share(8), unadv(8), unshare(8)`

**rmntstat**

`rmntstat [-h] [resource]`

zobrazí informace o připojených systémech souborů, které poskytuje síť

`souvisí s dfmounts(8), fumount(8), mount(8)`

**rpcgen**

`rpcgen infile`

`rpcgen [-Dname=value]] [-T] infile`

`rpcgen -c | -h | -l | -m | -t [-o outfile] [infile]`

`rpcgen -s nettype [-o outfile] [infile]`

`rpcgen -n netid [-o outfile] [infile]`

překladač síťových aplikací protokolu RPC

`souvisí s cc(1)`

**rsoper**

`rsoper -d ddev [-j jobids] [-u users] [-m method] [-s] [-t]  
[-o oname[:odevice]]`

`rsoper -d ddev [-j jobids] [-u users] [-m method] [-v] [-t]  
[-o oname[:odevice]]`

`rsoper -r jobids`

`rsoper -c [jobids]`

uživatelsky příjemné řízení obnovy dat z archivace

**souvisí s** `fimage(8)`, `file(8)`, `fdp(8)`, `fdisk(8)`, `getdate(3)`, `incfile(8)`, `mail(1)`,  
`restore(8)`, `rsnotify(8)`, `rsstatus(8)`, `urestore(8)`, `ursstatus(8)`

#### **rsstatus**

`rsstatus [-h] [-d ddev] [-j jobids] [-u users] [-f c]`

výpis zpráv vytvořených při obnově dat z archivace

**souvisí s** `restore(8)`, `urestore(8)`, `ursstatus(8)`, `volcopy(8)`

#### **runacct**

`runacct [mmd] [state]`

denní zpracování výsledků účtování

**souvisí s** `acct(2)`, `acct(8)`, `acctcms(8)`, `acctcom(8)`, `acctcon(8)`,  
`acctmerg(8)`, `acctprc(8)`, `cron(8)`, `fwtmp(8)`

#### **sa1**

`sadc [t n] [ofile]`

`sa1 [t n]`

`sa2 [options] [-s time] [-e time] [-i sec]`

programy zjišťující využitelnost jádra

**souvisí s** `sar(8)`

#### **sadp**

`sadp [-th] [-d device [-drive]] s [n]`

zjišťování využitelnosti disku

#### **sar**

`sar [options] [-o file] t [n]`

`sar [options] [-s time] [-e time] [-i sec] [-f file]`

výpis zpráv aktivity jádra vzhledem k vytvořeným datům `sa(1)`

**souvisí s** `sa1(8)`

#### **setmnt**

`setmnt`

vytvoření tabulky připojených svazků podle požadavků správce systému

**souvisí s** `mount(8)`

#### **setuname**

`setuname -s name [-n node] [-t]`

`setuname [-s name] -n node [-t]`

změna jména systému nebo jména uzlu v síti

**souvisí s** `uname(2)`, `uname(1)`

#### **share**

`share [-F fstype] [-a specific-options] [-d description]`

`[pathname] [resourcename]`

uvolnění lokálního systému souborů pro síť  
`souvisí s unshare(8)`

**sysdef**  
`sysdef [opsys [master]]`  
výpis informací o konfiguraci jádra

**tic**  
`tic [-v[n]] [-c] file`  
překladač položek databáze `terminfo(4)`  
`souvisí s captainfo(8), terminfo(4)`

**timex**  
`timex [-pos] command`  
`timex -p[fhkmrt] command`  
měření systémového času spotřebovaného příkazem i kontrola procesem  
využitých dat  
`souvisí s acctom(8), sar(8)`

**unadv**  
`unadv resource`  
vyjmutí nabízeného systému souborů ze seznamu pro síť  
`souvisí s adv(8), dfshares(8), fumount(8), nsquery(8), share(8),`  
`unshare(8)`

**unshare**  
`unshare [-F fstype] [-o specific-options] pathname unshare`  
`[-F fstype] [-o specific-options] resource`  
odpojení lokálního systému souborů z veřejné sítě  
`souvisí s share(8)`

**urestore**  
`urestore [-o target] [-d date] [-m] [-s] -F file ... urestore`  
`[-o target] [-d date] [-n] [-s] -F file ... urestore`  
`[-o target] [-d date] [-m] [-v] -F file ... urestore`  
`[-o target] [-d date] [-n] [-v] -F file ... urestore`  
`[-o target] [-d date] [-m] [-s] -D dir ... urestore`  
`[-o target] [-d date] [-n] [-s] -D dir ... urestore`  
`[-o target] [-d date] [-m] [-v] -D dir ... urestore`  
`[-o target] [-d date] [-n] [-v] -D dir ... urestore`  
`-c jobId`  
obnova souborů a adresářů  
`souvisí s file(8), getdate(3), incfile(8), ls(1), mail(1), restore(8),`  
`ursstatus(8)`

**ursstatus**

```
ursstatus [-f c] [-h] [-j jobids]
```

zpráva o zaslané poště při obnově uživatelských souborů a adresářů

```
souvisí s restore(8), rsstatus(8), urestore(8)
```

**useradd**

```
useradd [-u uid [-o]] [-g group] [-G group[, group ...]]
        [-d dir] [-s shell] [-c comment] [-m [-kskel-dir]]
        login
```

vytvoření nového uživatele

```
souvisí s logins(8), passwd(1), sh(1), usermod(8), userdel(8)
```

**userdel**

```
userdel [-r] login
```

zrušení uživatele

```
souvisí s logins(8), passwd(1), useradd(8), usermod(8)
```

**usermod**

```
usermod [-u uid [-o]] [-g group] [-G group[, group ...]]
        [-d dir [-m]] [-s shell] [-c comment] [-l new-logname]
        login
```

provede změny v registrech uživatelů

```
souvisí s logins(8), passwd(1), useradd(8), userdel(8)
```

**uuencode**

```
uuencode [source-file] file-label
```

```
uudecode [encoded-file]
```

kódování binárního souboru pro přenos poštou do jiného uzlu sítě

```
souvisí s mail(1), uucp(1), uux(1)
```

**volcopy**

```
volcopy [options] fsname special1 volname1 special2 volname2
        labelit special [fsname volume [-n]]
```

kopie svazku s kontrolou návěští

**wall**

```
wall
```

zpráva všem přihlášeným uživatelům

```
souvisí s msg(1), write(1)
```

**who**

```
who [-h] [-l] [user]
```

výpis aktivit přihlášených uživatelů

```
souvisí s ps(1), who(1)
```

## **F** *Vnější příkazy správce systému*

---

**zdump**

`zdump [-v] [-c cutoffyear] [zonename ...]`

úschova časové zóny

souvisí s `ctime(3)`

**zic**

`zic [-v] [-d directory] [-l localtime] [filename ...]`

převod času do aktuální časové zóny

souvisí s `ctime(3)`, `time(1)`

## Příloha G – Knihovna CURSES

Uvedené funkce tvoří základní sadu funkcí pro práci s obrazovkou terminálu podle doporučení SVID3.

### **addch**

```
#include <curses.h>
addch(chtype ch);
waddch(WINDOW *win, chtype ch);
mvaddch(int y, int x, chtype ch);
mvwaddch(WINDOW *win, int y, int x, chtype ch);
echochar(chtype ch);
wechochar(WINDOW *win, chtype ch);
        vsunutí znaku do okna
souvisí s attr(3), clear(1), inch(3), outopts(3), refresh(3), putc(3)
```

### **addchstr**

```
#include <curses.h>
int addchstr(chtype *chstr);
int addchnstr(chtype *chstr, int n);
int waddchstr(WINDOW *win, chtype *chstr);
int waddchnstr(WINDOW *win, chtype *chstr, int n);
int mvaddchstr(int y, int x, chtype *chstr);
int mvaddchnstr(int y, int x, chtype *chstr, int n);
int mvwaddchstr(WINDOW *win, int y, int x, chtype *chstr);
int mvwaddchnstr(WINDOW *win, int y, int x, chtype *chstr, int n);
        vsunutí textového řetězce do okna
```

### **addstr**

```
#include <curses.h>
int addstr(char *str);
int addnstr(char *str, int n);
int waddstr(WINDOW *win, char *str);
int waddnstr(WINDOW *win, char *str, int n);
int mvaddstr(y, int x, char *str);
int mvaddnstr(y, int x, char *str, int n);
int mvwaddstr(WINDOW *win, int y, int x, char *str);
int mvwaddnstr(WINDOW *win, int y, int x, char *str, int n);
        vsunutí textového řetězce do okna a přesun pozice kurzoru
souvisí s addch(3)
```

### **attr**

```
#include <curses.h>
int attroff(int attrs);
int wattroff(WINDOW *win, int attrs);
```

```
int attron(int attrs);
int wattron(WINDOW *win, int attrs);
int attrset(int attrs);
int wattrset(WINDOW *win, int attrs);
int standend(void);
int wstandend(WINDOW *win);
int standout(void);
int wstandout(WINDOW *win);
```

rutiny nastavení atributů práce s oknem (zvýrazněné písmo, inverze  
zobrazení atd.)

souvisí s `addch(3)`, `addstr(3)`, `printw(3)`

### **beep**

```
#include <curses.h>
int beep (void);
int flash(void);
```

akustický nebo optický signál pro uživatele

### **bkgd**

```
#include <curses.h>
void bkgdest(chtype ch);
void wbkgdset(WINDOW *win, chtype ch);
int bkgd(chtype ch);
int wbkgd(WINDOW *win, chtype ch);
```

manipulace s pozadím okna

souvisí s `addch(3)`, `outopts(3)`

### **border**

```
#include <curses.h>
int border(chtype ls, chtype rs, chtype ts, chtype bs,
           chtype tl, chtype tr, chtype bl, chtype br);
int wborder(WINDOW *win, chtype ls, chtype rs, chtype ts,
            chtype bs, chtype tl, chtype tr, chtype bl,
            chtype br);
int box(WINDOW *win, chtype verch, chtype horch);
int hline(chtype ch, int n);
int whline(WINDOW *win, chtype ch, int n);
int vline(chtype ch, int n);
int wvline(WINDOW *win, chtype ch, int n);
```

vytvoření rámečku kolem okna

souvisí s `outopts(3)`



**clear**

```
#include <curses.h>
int erase(void);
int werase(WINDOW *win);
int clear(void);
int wclear(WINDOW *win);
int clrtoebot(void);
int wclrtoebot(WINDOW *win);
int clrtoeol(void);
int wclrtoeol(WINDOW *win);
```

vypřázdňení okna nebo současného řádku okna

souvisí s `outopts(3)`, `refresh(3)`

**color**

```
#include <curses.h>
int start_color(void);
int init_pair(short pair, short f, short b);
int init_color(short color, short r, short g, short b);
bool has_colors(void);
bool can_change_color(void);
int color_content(short color, short *r, short *g, short *b);
int pair_content(short pair, short *f, short *b);
```

funkce práce s barvami

souvisí s `initscr(3)`, `attr(3)`

**delch**

```
#include <curses.h>
int delch(void);
int wdelch(WINDOW *win);
int mvdelch(int y, int x);
int mvwdelch(WINDOW *win, int y, int x);
```

zrušení znaku na pozici kurzoru v okně

**deleteln**

```
#include <curses.h>
int deleteln(void);
int wdelete(WINDOW *win);
int insdelln(int n);
int winsdelln(WINDOW *win, int n);
int insertln(void);
int wininsertln(WINDOW *win);
```

zrušení a vsunutí řádku v okně

**getch**

```
#include <curses.h>
int getch (void);
int wgetch(WINDOW *win);
int mvgetch(int y, int x);
int mvwgetch(WINDOW *win, int y, int x);
int ungetch(int ch);
```

převzme znak z klávesnice v daném okně nebo jej vrátí zpět do okna  
souvisí s `inopts(3)`, `move(3)`, `refresh(3)`

**getstr**

```
#include <curses.h>
int getstr(char *str);
int wgetstr(WINDOW *win, char *str);
int mvgetstr(int y, int x, char *str);
int mvwgetstr(WINDOW *win, int y, int x, char *str);
int wgetnstr(WINDOW *win, char *str, int n);
```

převzme textový řetězec z klávesnice  
souvisí s `getch(3)`

**getyx**

```
#include <curses.h>
void getyx(WINDOW *win, int y, int x);
void getparyx(WINDOW *win, int y, int x);
void getbegyx(WINDOW *win, int y, int x);
void getmaxyx(WINDOW *win, int y, int x);
```

určení souřadnic kurzoru a okna

**inch**

```
#include <curses.h>
chtype inch(void);
chtype winch(WINDOW *win);
chtype mvinch(int y, int x);
chtype mvwinch(WINDOW *win, int y, int x);
```

vrací znak a jeho atributy z okna

**inchstr**

```
#include <curses.h>
int inchstr(chtype *chstr);
int inchnstr(chtype *chstr, int n);
int winchstr(WINDOW *win, chtype *chstr);
int winchnstr(WINDOW *win, chtype *chstr, int n);
int mvinchstr(int y, int x, chtype *chstr);
int mvwinchnstr(int y, int x, chtype *chstr, int n);
```

```
int mvwinchstr(WINDOW *win, int y, int x, chtype *chstr);
int mvwinchnstr(WINDOW *win, int y, int x, chtype *chstr, int n);
```

vrací řetězec znaků a jeho atributy z okna

souvisí s `inch(3)`

### **initscr**

```
#include <curses.h>
WINDOW *initscr(void);
int endwin(void);
int isendwin(void);
SCREEN *newterm(char *type, FILE *outfd, FILE *infd);
SCREEN *set_term(SCREEN *new);
void delscreen(SCREEN *sp);
```

inicializace obrazovky

souvisí s `kernel(3)`, `refresh(3)`, `slk(3)`

### **inopts**

```
#include <curses.h>
int cbreak(void);
int nocbreak(void);
int echo(void);
int noecho(void);
int halfdelay(int tenths);
int intrflush(WINDOW *win, bool bf);
int keypad(WINDOW *win, bool bf);
int meta(WINDOW *win, bool bf);
int nodelay(WINDOW *win, bool bf);
int notimeout(WINDOW *win, bool bf);
int raw(void);
int noraw(void);
void noqiflush(void);
void qiflush(void);
void timeout(int delay);
void wtimeout(WINDOW *win, int delay);
int typeahead(int fd);
```

funkce řízení vstupu a výstupu

souvisí s `getch(3)`, `initscr(3)`, `termio(5)`

### **insch**

```
#include <curses.h>
int insch(chtype ch);
int winsch(WINDOW *win, chtype ch);
int mvinsch(int y, int x, chtype ch);
int mvwinsch(WINDOW *win, int y, int x, chtype ch);
```

vloží znak před pozici kurzoru

**insstr**

```
#include <curses.h>
int insstr(char *str);
int insnstr(char *str, int n);
int winsstr(WINDOW *win, char *str);
int winsnstr(WINDOW *win, char *str, int n);
int mvinsstr(int y, int x, char *str);
int mvinsnstr(int y, int x, char *str, int n);
int mvwinsstr(WINDOW *win, int y, int x, char *str);
int mvwinsnstr(WINDOW *win, int y, int x, char *str, int n);
```

vloží textový řetězec před pozici kurzoru

souvisí s `clear(3)`, `inch(3)`

**instr**

```
#include <curses.h>
int instr(char *str);
int innstr(char *str, int n);
int winstr(WINDOW *win, char *str);
int winnstr(WINDOW *win, char *str, int n);
int mvinstr(int y, int x, char *str);
int mvinnstr(int y, int x, char *str, int n);
int mvwinstr(WINDOW *win, int y, int x, char *str);
int mvwinnstr(WINDOW *win, int y, int x, char *str, int n);
```

převezme řetězec znaků z okna

**kernel**

```
#include <curses.h>
int def_prog_mode(void);
int def_shell_mode(void);
int reset_prog_mode(void);
int reset_shell_mode(void);
int resetty(void);
int savetty(void);
int getsyx(int y, int x);
int setsyx(int y, int x);
int ripoffline(int line, int (*init) (WINDOW *win, int));
int curs_set(int visibility);
int napms(int ms);
```

funkce nejnižší úrovně CURSES

souvisí s `initscr(3)`, `outopts(3)`, `refresh(3)`, `scr_dump(3)`, `slk(3)`

**move**

```
#include <curses.h>
int move (int y, int x) ;
int wmove(WINDOW *win, int y, int x);
```

**přesun pozice kurzoru**

souvisí s `refresh(3)`

**outopts**

```
#include <curses.h>
int clearok(WINDOW *win, bool bf);
int idlok(WINDOW *win, bool bf);
void idcok(WINDOW *win, bool bf);
void immedok(WINDOW *win, bool bf);
int leaveok(WINDOW *win, bool bf);
int setscrreg(int top, int bot);
int wsetscrreg(WINDOW *win, int top, int bot);
int scrollok(WINDOW *win, bool bf);
int nl (void);
int nonl (void);
```

**funkce řízení výstupu znaků na obrazovku**

souvisí s `addch(3)`, `clear(3)`, `initscr(3)`, `scroll(3)`, `refresh(3)`

**overlay**

```
#include <curses.h>
int overlay(WINDOW *scrwin, WINDOW *dstwin);
int overwrite(WINDOW *scrwin, WINDOW *dstwin);
int copywin(WINDOW *scrwin, WINDOW *dstwin, int sminrow,
            int smincol, int dminrow, int dmincol,
            int dmaxrow, int dmaxcol, int overlay);
```

**překrývání oken**

souvisí s `pad(3)`, `refresh(3)`

**pad**

```
#include <curses.h>
WINDOW *newpad(int nlines, int ncols);
WINDOW *subpad(WINDOW *orig, int nlines, int ncols,
               int begin_y, int begin_x);
int prefresh(WINDOW *pad, int pminrow,
             int pmincol,
             int sminrow, int smincol, int smaxrow,
             int smaxcol);
int pnoutrefresh(WINDOW *pad, int pminrow, int pmincol,
                 int sminrow, int smincol, int smaxrow,
                 int smaxcol);
int pechochar(WINDOW *pad, chtype ch);
```

**vytvoření virtuálního okna a jeho zobrazení**

souvisí s `refresh(3)`, `touch(3)`, `addch(3)`

**printw**

```
#include <curses.h>
int printw(char *fmt [, arg] ...);
int wprintw(WINDOW *win, char *fmt [, arg] ...);
int mvprintw(int y, int x, char *fmt [, arg] ...);
int mvwprintw(WINDOW *win, int y, int x, char *fmt [, arg] ...);
```

formátovaný výstup do okna

souvisí s `printf(3)`, `vprintf(3)`

**refresh**

```
#include <curses.h>
int refresh(void);
int wrefresh(WINDOW *win);
int wnoutrefresh(WINDOW *win);
int doupdate(void);
int redrawwin(WINDOW *win);
int wredrawln(WINDOW *win, int beg_line, int num_lines)
```

aktualizace okna na obrazovce

souvisí s `outopts(3)`

**scanw**

```
#include <curses.h>
int scanw(char *fmt [, arg] ...);
int wscanw(WINDOW *win, char *fmt [, arg] ...);
int mvscanw(int y, int x, char *fmt [, arg] ...);
int mvwscanw(WINDOW *win, int y, int x, char *fmt [, arg] ...);
```

formátovaný vstup z okna

souvisí s `getstr(3)`, `printw(3)`, `scanf(3)`

**scr\_dump**

```
#include <curses.h>
int scr_dump(char *filename);
int scr_restore(char *filename);
int scr_init(char *filename);
int scr_set(char *filename);
```

zápis (čtení) okna do (ze) souboru

souvisí s `initscr(3)`, `refresh(3)`, `util(3)`, `system(3)`

**scroll**

```
#include <curses.h>
int scroll(WINDOW *win);
int scl (int n);
int wscrl(WINDOW *win, int n);
```

průchod zobrazovaných řádků oknem

souvisí s `outopts(3)`

**slk**

```
#include <curses.h>
int slk_init(int, fmt);
int slk_set(int labnum, char *label, int fmt);
int slk_refresh(void);
int slk_noutrefresh(void);
char *slk_label(int labnum);
int slk_clear(void);
int slk_restore(void);
int slk_touch(void);
int slk_attron(chtype, attrs);
int slk_attrset(chtype, attrs) int slk_attroff(chtype attrs);
```

**funkce zobrazování značek na obrazovce terminálu**

**souvisí s attr(3), initscr(3), refresh(3)**

**termattrs**

```
#include <curses.h>
int baudrate(void);
char erasechar(void);
int has_ic(void);
int has_il(void);
char killchar(void);
char *longname(void);
chtype termattrs(void);
char *termname(void);
```

**funkce práce s atributy okna**

**souvisí s initscr(3), outopts(3)**

**termcap**

```
#include <curses.h>
#include <term.h>
int tgetent(char *bp, char *name);
int tgetflag(char id[2]);
int tgetnum(char id[2]);
char *tgetstr(char id[2], char **area); char *tgoto(char *cap,
int col, int row);
int tputs(char *str, int affcnt, int (*putc) (void));
```

**emulace funkcí základní úrovně dřívější databáze terminálů TERMcap**

**souvisí s terminfo(3)**

**terminfo**

```
#include <curses.h>
#include <term.h>
int setupterm(char *term, int fildes, int *errret);
int setterm(char *term);
```

```
int set_curterm(TERMINAL *nterm);
int del_curterm(TERMINAL, *oterm);
int restartterm(char *term, int fildes, int *errret);
char *tparm(char *str, long int p1, long int p2, long int p3,
            long int p4, long int p5, long int p6,
            long int p7, long int p8, long int p9);
int tputs(char *str, int affcnt, int (*putc) (char));
int putp(char *str);
int vidputs(chtype attrs, int (*putc) (char));
int vidattr(chtype, attrs);
int mvcur(int oldrow, int oldcol, int newrow, int newcol);
int tigetflag(char *capname);
int tigetnum(char *capname);
int tigetstr(char *capname);
```

**základní úroveň práce s databází terminálů**

**souvisí s** `initscr(3)`, `kernel(3)`, `termcap(3)`, `putc(3)`, `terminfo(4)`

**touch**

```
#include <curses.h>
int touchwin(WINDOW *win);
int touchline(WINDOW *win, int start, int count);
int untouchwin(WINDOW *win);
int wtouchln(WINDOW *win, int y, int n, int changed);
int is_linetouched(WINDOW *win, int line);
int is_wintouched(WINDOW *win);
```

**zobrazení okna na obrazovce**

**souvisí s** `refresh(3)`

**util**

```
#include <curses.h>
char *unctrl(chtype c);
char *keyname(int c);
int filter(void);
void use_env(char bool);
int putwin(WINDOW *win, FILE *filep);
WINDOW *getwin(FILE *filep);
int delay_output(int ms);
int flushinp(void);
```

**další různé funkce práce s okny**

**souvisí s** `initscr(3)`, `scr_dump(3)`

**window**

```
#include <curses.h>
WINDOW *newwin(int nlines, int ncols, int begin_y, int begin_x);
int delwin(WINDOW *win);
```



```
int mvwin(WINDOW *win, int y, int x);
WINDOW *subwin(
WINDOW *orig, int nlines, int ncols, int begin_y, int begin_x);
WINDOW *derwin(WINDOW *orig, int nlines, int ncols,
                int begin_y, int begin_x);
int mvderwin(WINDOW *win, int par_y, int par_x);
WINDOW *dupwin(WINDOW *win);
void wsyncup(WINDOW *win);
int syncok(WINDOW *win, bool bf);
void wcursyncup(WINDOW *win);
void wsyncdown(WINDOW *win);
                vytvoření nového okna
souvisí s refresh(3), touch(3)
```

## Literatura

- [1] MADNICK, STUART E.; DONOVAN JOHN J. *Operating Systems*. New York : McGraw Hill Book Company, 1974. ISBN 0070394555, česky *Operační systémy*. Přel. Helena Fendrychová, František Plášil. Praha : SNTL, 1983. 589 s.
- [2] KERNIGHAN, BRIAN W.; PLAUGER P. J. *Software Tools*. Reading : Addison-Wesley, 1976. 338 pp. ISBN 201-03669-X
- [3] KERNIGHAN, BRIAN W.; PIKE, R. *The UNIX Programming Environment*. Englewood Cliffs: Prentice Hall, 1984. 357 pp. ISBN 0-13-937699-2, česky *Programové prostředí operačního systému UNIX*. Přel. Pavel Just, Jana Viktorínová. Veletiny : Science, 1996. 300 s. ISBN 80-901475-69
- [4] BRODSKÝ, J.; SKOČOVSKÝ, L. *Operační systém Unix a jazyk C*. Praha : SNTL, 1989. 368 s. ISBN 80-03-00049-1
- [5] BACH, MAURICE J. *The Design of the UNIX Operating System*. Englewood Cliffs : Prentice Hall, 1986. ISBN: 0-13-201799-7, česky *Principy operačního systému UNIX*. Přel. Jiří Felbáb. Praha : SA&S, 1993. 532 s. ISBN 80-901507-0-5
- [6] KERNIGHAN, BRIAN W.; RITCHIE, DENIS M. *The C Programming Language*. Englewood Cliffs : Prentice Hall, 1978. ISBN: 0-13-110163-3, slovensky *Programovací jazyk C*. Přel. Vladimír Benko. 1. vyd. Bratislava : Alfa, 1988. 249 s. ISBN 80-901475-69
- [7] McKUSIC, M. K.; JOY, W. N.; FABRY, R. S. A Fast File System for UNIX. In *ACM Trans. Computer Systems*. No. 2,3 (Aug. 1984), pages 181-197. University of California, Berkeley, 1984.
- [8] RITCHIE, DENIS M. A Stream Input Output System. In *Technical Journal*. Oct. 1984, No. 8, pages 1897–1910. AT&T Bell Laboratories, 1984.
- [9] DIJKSTRA, E. W. Cooperating Sequential Processes. In: Genuys, F. (ed.): *Programming languages*. New York : Academia Press, 1968.
- [10] LEFFLER, S. J.; FABRY, R. S.; JOY, W. N. et al. *An Advanced 4.3BSD Interprocess Communication*. University of California, Berkeley, 1987.
- [11] KATSEFF, HOWARD P. *Sdb: A Symbolic Debugger*. University of California, Berkeley, 1987.
- [12] NEMETH, E.; SNYDER, G.; SEEBASS, S. *UNIX System Administration Handbook*. Englewood Cliffs : Prentice Hall, 1989. ISBN: 0 -13-933441-6

*Doporučená literatura*

- [13] BOURNE, A. S. *The UNIX System*. London : Addison-Wesley, 1982. 351 pp.
- [14] EGAN, JANET I.; TEXEIRA, THOMAS J. *Writing a UNIX Device Driver*. New York : John Wiley&Sons, Inc., 1988. 366 pp. ISBN-13: 978-0471628590
- [15] *System V Interface Definition*. Third Edition, Volume I, II, III, IV. UNIX Software Operation AT&T, 1989. Volume V. UNIX System Laboratories, Addison-Wesley, 1991. ISBN 0-201-56656-7. Volume VI. UNIX System Laboratories, Addison-Wesley, 1991. ISBN 0-201-52480-5
- [16] STEVENS, RICHARD W. *UNIX Network Programming*. Englewood Cliffs : Prentice Hall, 1990. 772 pp. ISBN 0-13-949876-1
- [17] SCHEIFLER, ROBERT W.; GETTYS, J. *X Window System: The Complete Reference to Xlib, X Protocol, ICCCM, XLFD*. Bedford : Digital Press, 1990. ISBN 0-13-972050-2
- [18] ASENTE, PAUL J.; SWICK RALPH R. *X Window System Toolkit: The Complete Programmer's Guide Specification*. Bedford : Digital Press, 1990. 966 pp. ISBN 1-55558-051-3

## Rejstřík

#define 109, 110  
 #include 109, 110  
 #pragma 109  
 . 16, 19, 20  
 .. 16, 19, 20  
 .profile 40, 64, 132  
 /bin 45, 62, 80, 172  
 /bin/csh 39, 76  
 /bin/sh 38, 50, 62, 63  
 /boot 48, 158, 172, 182  
 /dev 23, 45, 60, 190  
 /dev/null 45, 60  
 /etc 46, 172, 187  
 /etc/conf 46, 187  
 /etc/ctab 166, 167  
 /etc/group 38, 85, 87, 185  
 /etc/inittab 121, 162, 183  
 /etc/passwd 38, 50, 85, 87, 137, 185  
 /etc/rc 161, 164, 166  
 /lib 47, 77, 111  
 /mnt 30, 47, 172  
 /tmp 48, 69  
 /unix 48, 158, 172, 180, 182, 188  
 /usr 20, 47, 190  
 /usr/bin 48, 51, 62  
 /usr/include 47, 48, 110  
 /usr/lib 111  
 2.xBSD 11  
 4.xBSD 11, 135

### A

a.out 41, 65, 109, 111, 113  
 accept(2) 103  
 access(2) 87  
 adb(1) 110, 113, 114  
 admin(1) 115  
 adresář (directory)  
   domovský (home) 18, 62  
   kořenový (root) 19, 159  
   nadřazený 42, 94  
   pracovní (current) 18, 38, 42, 55  
 adv(8) 156  
 alarm(2) 84  
 ar(1) 40, 111  
 archivace 25, 174  
 ARP 149  
 ARPA 135, 147, 152  
 assembler 40  
 asociace  
   asociace 148  
   poloviční 148  
 AT&T 4, 10

### B

backup(8) 177  
 basename(1) 41  
 BBN 135  
 bg 59, 74, 217  
 binární editor 114  
 bind(2) 103, 104  
 blok popisu svazku 26, 29, 165, 170  
 boot 158, 173, 179, 182  
 boot disk 182  
 BPCL 10  
 brána 148  
 brk(2) 86  
 BSD 11, 103, 114, 135, 147

### C

C  
 C  
   hlavičkové soubory 110  
   makroprocesor 40, 48, 109, 110, 117  
   překladač 40, 109, 181  
   programovací jazyk 9, 10, 14, 40, 48, 53, 54, 63, 74,  
     90, 109, 113, 117, 181  
   streams 90  
 C-shell 59, 64, 74  
 case 70, 212  
 cat(1) 18, 21  
 cb(1) 54  
 cc(1) 40, 109, 113  
 cd 19, 51, 56, 107  
 cesta k souboru (path)  
   absolutní 20, 43  
   relativní 20, 42  
 chdir(2) 93  
 child 31  
 chmod(1) 26, 34, 63  
 chmod(2) 87  
 chown(1) 25  
 chown(2) 87  
 close(2) 88, 98, 139  
 configure(8) 191  
 connect(2) 103  
 cooked 124  
 core 83, 113, 216  
 cp(1) 20, 43, 60  
 cpio(1) 175  
 creat(2) 88  
 cron 189  
 cu(1) 142  
 CURSES 126  
 cxref(1) 113

### D

DARPA 103, 135, 149  
 dd(1) 168, 171, 172  
 DEC 4, 133  
 delta(1) 115

démon (daemon) 32, 101, 159, 161, 166  
df(1) 169  
dfshares(8) 156  
DNS 154  
dump(8) 177  
dup(2) 90, 97

## E

echo(1) 55, 60, 68, 215  
ed(1) 194  
EGID 85  
EIA (Electronics Industry Association) 135  
EUID 85  
ex(1) 21, 108, 194  
exec(2) 49, 80  
exit(2) 64, 79, 107  
export 63, 109, 215  
expr(1) 66

## F

fcntl(2) 28, 92  
fg 59, 74, 217  
FIFO 24, 36, 48, 54, 95, 98  
file(1) 41  
file system (systém souborů) 16  
find(1) 22, 107, 176  
for 67, 69, 70, 212, 215  
fork(2) 31, 34, 49, 78, 97  
fs(5) 168  
fsck(8) 46, 159, 167, 172, 179  
fsdb(8) 169  
ftp(1) 135, 146, 153

## G

gateway (brána) 148  
get(1) 115  
getgid(2) 85  
getmsg(2) 140  
getpgrp(2) 84  
getpid(2) 82, 100  
getppid(2) 82  
getty 32, 49, 121, 142, 161  
getuid(2) 85  
GID 85, 217  
groupadd(8) 185  
groupdel(8) 185  
groupmod(8) 185

## H

HP-UX 4, 11, 125, 157, 187

## I

i-uzel (i-node) 17, 26, 42, 90, 94, 167  
ICCCM 133, 283  
ICMP 149  
idbuild(8) 188  
idcheck(8) 188  
idconfig(8) 188

idmkunix(8) 188  
IEEE (Institute of Electrical Electronic Engineers) 12, 147  
init 31, 46, 49, 85, 121, 159, 162  
interaktivní práce 9, 10  
ioctl(2) 93, 103, 119, 123, 139  
IP (Internet Protocol) 146, 149  
IPC (Interprocess Communication)  
IPC 36, 37, 98, 109, 133  
předávání zpráv (messages) 37, 98, 101, 140  
schránka (socket) 139  
sdílená paměť (shared memory) 101  
semafore (semaphores) 102, 103  
ISO (International Standards Organization)  
125, 146, 193

## J

jádro (kernel)  
generace 186  
jádro (kernel) 10, 16, 28, 30, 37, 77, 78, 104, 119, 124, 139, 157, 158, 160, 172, 180  
parametry 186, 189  
reálný čas 37  
jednouživatelská úroveň (single user mode) 31, 159, 162

## K

KERMIT 135  
kernel (jádro) 10, 16, 28, 30, 37, 77, 78, 104, 119, 124, 139, 157, 158, 160, 172, 180  
kill(1) 36, 58, 109, 261  
kill(2) 36, 81, 84  
klient 104, 133, 148, 156  
kolona 54

## L

LAN (Local Area Network) 135, 147  
ld(1) 109, 111, 187  
link(2) 90  
lint(1) 113  
listen(2) 103, 104  
lockf(2) 93  
login : 49, 121, 142  
logins(8) 185  
lp(1) 24, 45, 183  
lpr(1) 184  
lpsched 32, 161, 183  
lseek(2) 31, 91  
lyrix 125

## M

mail(1) 62, 135, 137  
mailx(1) 137, 139  
major number (hlavní číslo speciálního souboru) 23, 95, 190  
make(1) 115, 116, 188  
man 12, 13, 48  
MAN (Metropolitan Area Network) 148

**mesg**(1) 137  
**minor number** (vedlejší číslo speciálního souboru) 23, 95, 190  
**MIT** (The Massachusetts Institute of Technology) 133  
**mkdir**(1) 20, 44  
**mkfs**(8) 28, 46, 169, 170, 179, 182  
**mknod**(2) 93, 94  
**mknod**(8) 25, 172, 190  
**mkuser**(8) 185  
**monitor**(3) 114  
**Motif** 134  
**mount**(2) 95  
**mount**(8) 30, 44, 156, 166, 170, 172  
**mountall**(8) 166  
**msgctl**(2) 99, 101  
**msgget**(2) 99  
**msgrcv**(2) 99  
**msgsnd**(2) 99  
**MULTICS** 10  
**mv**(1) 20, 41, 188

## N

**nástroje programátora** (Tools) 10, 76, 107  
**nepřímý odkaz** (symbolic link), viz **i-uzel** 17, 28  
**NFS** (Network File System) 135, 146, 147, 155  
**NIC** (Network Information Center) 150  
**nice**(1) 33  
**nice**(2) 86  
**nm**(1) 112  
**nsquery**(8) 156

## O

**oblast**  
     **datových bloků** 26, 27, 29, 31, 167, 178  
     **i-uzlů** 26, 28, 167, 190  
     **odkládací** 34, 166, 190  
**obyčejný soubor** (regular file) 17, 18, 94  
**ođ**(1) 18, 60, 168  
**odkaz na soubor** (link to the file) 28, 43, 90  
**open**(2) 87, 98, 122, 139  
**operační systém** (Operating System) 9, 14, 31, 33, 180, 186  
**OSF** (Open System Foundation) 134  
**OSI** (Open Systems Interconnection) 135, 146  
**otevřenost** 10  
**ovladač** 9, 14, 37, 45, 46, 94, 103, 104, 119, 124, 139, 183, 186, 190

## P

**paket** (packet) 148, 149  
**parametry jádra** 186, 189  
**passwd**(1) 180, 185  
**pause**(2) 84  
**periferie** 9, 24, 37, 45, 46, 77, 94, 139, 183, 190  
**PGID** 84, 217  
**PID** 32, 57, 68, 78, 82, 84, 95, 136, 214, 217  
**ping**(1) 149

**pipe**(2) 96, 97, 141  
**pipe** (roura) 36, 37, 52, 95, 97  
**pkgadd**(8) 181, 182  
**plock**(2) 86  
**poll**(2) 141  
**POSIX** 12  
**práce, řízení prací** 59, 75, 189  
**přesměrování**  
     **diagnostika**, **stderr** 53, 74  
     **kanály** 53, 212  
     **standardní vstup**, **stdin** 22, 52  
     **standardní výstup**, **stdout** 22, 52, 74  
**příkazový interpret** (shell) 9, 38, 45, 49, 76, 96, 107, 108, 212  
**proces**  
     **návratový status**, viz také **test**(1) 65, 68, 79, 214, 216  
     **popředí** 57, 58, 59, 74, 217  
     **pozadí** 57, 59, 68, 74, 212, 214, 217  
     **priorita**  
         **dynamická** 33, 86  
         **uživatelská** 33, 86  
         **výchozí** 33  
     **prostředí provádění** 109, 214  
     **rodič, otec** (parent) 31, 35, 49, 71, 78, 80, 85, 97  
     **rodina** 38, 97  
     **segment**  
         **datový** 35, 78, 80, 86, 112  
         **textový** 35, 77, 78, 80, 86, 112, 119, 215  
         **zásobník** 35, 78, 112, 114, 216  
     **stav** 34, 75, 79, 83  
     **syn** (child) 31, 78, 81, 97, 107  
**profil**(2) 106, 114  
**programovací jazyk C** (C programming language) 9, 10, 14, 40, 48, 53, 54, 63, 74, 90, 109, 113, 117, 148, 181  
**PROUD** (STREAM)  
     **pojmenovaný** 49  
     **POP** 94, 140  
     **PROUD** (STREAM) 94, 139  
     **PUSH** 94, 140  
**ps**(1) 57, 165  
**PS1** 62, 214  
**PS2** 62, 65, 214  
**ptrace**(2) 106, 114  
**putmsg**(2) 140  
**pwd** 19, 61, 174, 215

## R

**RARP** (Reverse Address Resolution Protocol) 149  
**raw** 124  
**rc**, viz také **/etc/rc** 161, 164, 166  
**rcp**(1) 147, 154  
**read** 61, 66, 120, 215  
**read**(2) 77, 89, 139  
**readlink**(2) 90  
**recv**(2) 103, 104  
**regular file** (obyčejný soubor) 17, 18, 94

remsh(1) 147, 155  
 restor(8) 178  
 rexec 147  
 rexec(3) 155  
 RFS (Remote File System) 147, 155  
 rlogin(1) 146, 154  
 rm(1) 21, 28, 52, 60, 190  
 rmdir(1) 20  
 rmuser(8) 185  
 root 15, 19, 45, 136, 157, 159, 173, 181  
 roura  
   pojmenovaná (named) 37, 94, 97  
   roura (pipe) 36, 37, 52, 95, 97  
 router (směrovač) 148  
 RPC (Remote Procedure Call) 146, 147  
 RS 232 C 120  
 rsh(1) 214

## S

s-bit 35, 85, 94  
 sa1(8) 189  
 sa2(8) 189  
 sar(8) 189  
 sbrk(2) 86  
 SCCS (Source Code Control System) 115  
 SCO UNIX 13, 113, 125, 157, 182, 191  
 sdb(1) 113  
 seek, viz také lseek(2) 31, 91  
 segment 35, 78, 80, 92, 150  
 semctl(2) 102, 103  
 semget(2) 102  
 semop(2) 102  
 send(2) 103, 105  
 server 99, 104, 133, 148  
 set 61, 73, 216  
 setpgrp(2) 35, 85  
 setuid(2) 85  
 sezení (session) 9, 15, 40, 118, 120  
 sh 32, 38, 49, 213  
 share(8) 155  
 shell (příkazový interpret)  
   Bourne shell 32, 38, 212  
   C-shell 39, 59, 74  
   KornShell 109  
   příkaz  
     příkaz 51  
     vnější 45, 51  
     vnější lokální 51  
     vnitřní 45, 56, 216  
   proměnná 61, 62, 68, 131, 139, 212  
   scénář 63, 108, 120, 161, 181, 212  
 shmat(2) 101  
 shmctl(2) 101  
 shmget(2) 101  
 shutdown 161, 188  
 shutdown(2) 105  
 signal(2) 36, 83, 84  
 signál (signal) 36, 57, 81, 101, 109, 113, 164, 216, 217

směrovač (router) 148  
 SMTP (Simple Mail Transfer Protocol) 147, 153  
 SNA (System Network Architecture) 146  
 socket(2) 103  
 soubor  
   adresář 17, 26, 42, 55, 94  
   cesta  
     absolutní 20, 43  
     relativní 20, 42  
   deskriptor 88  
   obyčejný 17, 18, 26, 35, 93  
   přístupová práva  
     přístupová práva 17, 26, 35, 85, 87, 94  
     s-bit 35, 85, 94  
     t-bit 34, 94  
     x-bit 35, 85  
   speciální  
     hlavní číslo (major number) 23, 94, 190  
     speciální 12, 17, 23, 45, 94, 167, 190  
     vedlejší číslo (minor number) 23, 94, 190  
 standardní chybový výstup (standard error output) 52, 88, 107  
 standardní vstup (standard input) 22, 36, 52, 73, 88  
 standardní výstup (standard output) 22, 36, 52, 88  
   stat(2) 91  
 stav systému  
   jednouživatelský 31, 159, 162  
   stav systému 49, 164  
   víceživatelský 32, 47, 49, 121, 162  
   stime(2) 105  
 STREAM (PROUD) 94, 139  
 strip(1) 112  
 stty(1) 118, 119, 218  
 super block (blok popisu svazku) 26, 29, 165, 170, 178  
 superuživatel (superuser) 15, 24, 30, 85, 157  
 supervizorový režim 9, 34, 37, 77, 106, 114  
 svazek  
   AFS 30  
   archivace 174  
   FFS 30, 166  
   FSS 30, 166  
   kořenový 159  
   svazek 26, 42, 95, 158, 165, 170  
   vytvoření 28, 29, 46, 170, 172, 178, 179  
 SVID (System V Interface Definition) 11, 125, 135, 147  
 swapper, sched 31, 34, 50, 159, 162, 173  
 symbolic link (nepřímý odkaz), viz i-uzel 17, 28, 90  
 symlink(2) 90  
 sync(1) 31, 166  
 sync(2) 105, 160, 166  
 sysdef(8) 189  
 System Calls (volání jádra) 12, 16, 34, 77, 109, 219  
 systém souborů (file system) 16, 26, 179

## T

t-bit 34, 94  
 tar(1) 24, 146, 174

## Restřík

TCP (Transmission Control Protocol) 103, 133, 135, 146, 156  
TCP/IP (Transmission Control Protocol/Internet Protocol) 103, 104, 135, 148, 149, 151, 156  
telnet(1) 135, 147, 152  
TERM 62, 126, 131, 205  
termcap 132  
terminfo 131, 194, 205  
termio(5) 50, 122  
test(1) 65, 216  
tic(8) 131, 132  
time(1) 53, 114  
time(2) 105  
times 216  
times(2) 106  
Tools (nástroje programátora) 10, 76, 107  
trap 58, 69, 109  
truss(1) 114

## U

UDP (User Datagram Protocol) 103, 146, 147, 150, 152  
umask 217  
umask(2) 88  
umount(2) 95  
umount(8) 30, 156  
unadv(8) 156  
uname(1) 143  
uname(2) 106  
University of California, Berkeley (UCB) 11, 74, 103, 135  
UNIX SYSTEM V 11, 38, 125, 139, 162, 187, 235  
unlink(2) 90  
unset 62, 74, 217  
untic(8) 132  
until 67, 213  
**update** 32, 161, 166  
useradd(8) 185  
userdel(8) 185  
usermod(8) 185  
ustat(2) 95  
uucico 50, 145  
UUCP 135, 142  
uucp 50  
uucp(1) 144, 145  
uustat(1) 145  
uux(1) 146  
uživatel  
obyčejný 15, 35  
privilegovaný, superuživatel 15, 24, 30, 35, 85, 157  
prostředí u terminálu 107

## V

vi(1) 21, 108, 124, 194, 205  
víceuživatelský režim (multi user mode) 32, 47, 49, 121, 160  
volání jádra (System Call) 12, 16, 34, 77, 109, 219  
vyrovnávací paměť systému 30, 37, 88, 105, 160, 166, 189  
Vývojové prostředí (Development System) 14, 181

## W

wait 59, 109, 217, 218  
wait(2) 79  
wall(8) 137  
WAN (Wide Area Network) 135, 148  
while 67, 69, 213, 215  
who(1) 51, 135, 164  
WM (Window Manager) 133  
write(1) 135, 136  
write(2) 89, 139

## X

X (X-WINDOW SYSTEM)  
Motif 134  
WM (Window Manager) 133  
X (X-WINDOW SYSTEM) 133  
X-Consortium 133  
X-server 133  
X-terminál 133  
X-Toolkit 133  
Xlib 133, 134  
Xt 133  
x-bit 35, 85  
X/OPEN 12  
XDR (eXternal Data Representation) 146, 147  
XENIX 11, 13, 172  
XNS (Xerox Networking Systems) 146

## Z

zaváděcí blok (boot block) 30, 158, 165