UNIVERZITA J. E. PURKYNĚ V ÚSTÍ NAD LABEM
Přírodovědecká fakulta

Operační systémy

KI/OPS

Petr Kubera, Jiří Fišer

Ústí nad Labem 2016

Kurz:	Principy operačních systémů
Obor:	Informační systémy, Informatika (dvouoborové studium)
Klíčová slova:	Linux, Unix, správa OS, bash
Anotace:	Cílem kurzu je seznámení studentů se strukturou a prostředky moderních operačních systémů (OS) se zaměřením na OS typu Unix/Linux. Kurz je zaměřen na použití systémových shellů, psaní vlastních skriptů, konfiguraci a administraci systému.

Jazyková korektura nebyla provedena, za jazykovou stránku odpovídá autor.

Autor: Petr Kubera, Jiří Fišer

Obsah

Úvodní slovo

Cílem tohoto kurzu je uvést Vás do problematiky moderních operačních systémů se zaměřením na OS Linux/Unix. Kurz je zaměřen prakticky, maximální pozornost věnujte vlastnímu experimentování s OS a vyzkoušení a promyšlení v textu uvedených příkladů. V textu jsou sice uvedené příklady, ale celou řadu zde ukázaných věcí lze řešit i jiným způsobem.

Nainstalujte si některou z distribucí Linuxu (jsou jich dnes stovky), mezi asi nejznámější patří v době psaní tohoto textu (setříděno od uživatelsky nejjednodušších):

- Ubuntu a jeho klony
- Suse a jeho klony
- Fedora (Red Hat)
- Debian
- Arch linux
- Gentoo

Kurz je ukončen zápočtem, který představuje praktickou práci, při které ukážete schopnost práce se systémem a řešení některých problémů. Typické zápočtové úlohy:

- Správa uživatelů, souborů a procesů
- Zpracování textů (filtry a kolony)
- Automatizace úkolů (tvorba skriptů)
- Jednoduchá konfigurace systému a instalace programů
- Zálohování

Pro praktické předvedení budete mít k dispozici plně funkční OS (Debian, či Ubuntu) s připojením k internetu a dokumentací. Je nutné splnit 75% úloh.

1 Operační systémy a jejich struktura

Existují různé definice co to je operační systém. Uveďme si např. jednu z nich Operační systém je programové vybavení, jehož účelem je správa prostředků výpočetního systému a jejich efektivní přidělování uživatelským programům. Poskytuje uživatelským procesům služby (např. API). Úkolem operačního systému je oddělit hardware od uživatelských procesů.

Protože operační sytém zpravuje prostředky, tak si některé uveďme:

Prostředky:

- CPU
- Paměť
- Souborový systém
- V/V zařízení, periferie
- Spravuje procesy

Existují různé architektury operačních systémů, vždy se na něj však můžeme dívat tak, že OS je tvořen různými vrstvami, které spolu komunikují a jsou od sebe navzájem oddělené(víceméně). Jedna z nejdůležitějších částí je jádro OS,(kernel). Mezi typické úkoly jádra patří:

- Přidělování CPU a paměti procesům
- Přístup k V/V zařízení
- Podpora meziprocesové komunikace (IPC)

1.1 Unix a a Linux

Linux je volně šiřitelný systém odvozený z původního operačního systému Unix a velmi blízký různým v současnosti používaným verzím tohoto systému. Na druhou stranu jeho obliba a rozšíření vedou k přejímání některých jeho specifických vlastností i do ostatních operačních systémů unixovského typu.

Operační systémy unixovského typu (včetně Linuxu) vycházejí z původního Unixu, který vznikl na přelomu šedesátých a sedmdesátých let[1] a existoval jako jednotný systém po celou první polovinu let sedmdesátých. I když pak došlo k rozštěpení vývoje, jednotlivé unixovské systémy na sebe navzájem dále působily a působí nadále. Tato společná již více než pětatřicetiletá historie, se přirozeně projevuje na současných unixovských systémech a to jak v pozitivním tak negativním slova smyslu a proto si zaslouží trochu pozornosti.

[1] tj. například o více než deset let dříve než MS-DOS

Operační systém Unix vznikl jako určitá reakce na operační systém Multics. Tento operační systém vytvářený v druhé polovině šedesátých let několika velkými firmami byl projektován jako univerzální operační systém zaměřený na interaktivní zadávání a řízení úloh (v této době převažovaly systémy dávkově orientované). I když přinesl návrh tohoto systému mnoho na svou dobu přínosných řešení, projekt nakonec zkrachoval. Navržený a předběžně implementovaný systém měl vysoké nároky na hardware a nebyl proto v tehdejší době komerčně využitelný (náklady především hardwarové byly příliš velké). Proto většina firem z projektu odstoupila. Mezi těmito firmami byl i telekomunikační gigant AT&T, jehož laboratoře (Bell Labs) se na projektu také podíleli. Někteří programátoři z těchto laboratoří, kteří na tvorbě Multicsu podíleli, využili svých zkušeností na tvorbu malého interaktivního systému, který běžel na stroji[2], jehož parametry nebyly ani v kontextu dané doby nikterak závratné (měl např. pouze 32 KiB[3] operační paměti). Základem tohoto systému byl hierarchický souborový systém (souborový systém, jenž je největším přínosem Unixu je stále centrální komponentou systému) a správce procesů, jenž již téměř na samotném počátku umožňoval souběžné používaní systému vícero uživateli[4] (tj. mimo jiné poskytoval i preemptivní multitasking).

Po celou první polovinu 70. let se Unix používal pouze ve firmě AT&T a to jen v několika málo exemplářích. Už v této době byl však přepsán do jazyka C (jenž byl cíleně vytvořen jako implementační jazyk Unixu) a byl opatřen programem pro interaktivní ovládání systému tzv. *shellem*[5]. Na charakteru UNIXu se také projevilo jeho užívání v administrativním a právním oddělení firmy AT&T neboť byl opatřen souborem programů pro zpravování textů z nichž mnohé se stále používají.

V roce 1976 byl zdrojový kód Unixu uvolněn pro použití na univerzitách[6] a nedlouho potom (1977-78) portován na jinou architekturu (ta byla navíc na rozdíl od počítačů PDP 32-bitová). Od této chvíle přestává být Unix jednotným systémem, na druhou stranu se stává systémem otevřeným, jenž je k dispozici na vícero platformách a jeho vývoj nenní diktován jedinou organizací.

Při prvotním rozdělení vznikly dvě základní větvě Unixu, větev univerzitní, jež byla na počátku representována Berkeley Unix (známý spíše pod zkratkou BSD[7]), a větev komerční, jež byla zpočátku soustředěna v Bellových laboratořích AT&T (známá především pod označením Unix System III resp. System V). Komerční licence na systém však byly poskytnuty dalším softwarovým firmám, které začaly vytvářet vlastní rozšířené implementace (jádrem je vždy původní kód od AT&T). Mezi nejznámější patří AIX (IBM), HP/UX (HP), SCO Unix (SCO)[8].

Toto rozrůznění vedlo od počátku 90.let ke snahám o opětné sjednocení. Toto sjednocení nebylo snadné, neboť Unix již dávno nebyl tím malým a kompaktním operačním

[2] PDP-7

[3] v celém skriptu používám modernější zkratky KiB (1024 bytů) a MiB (1024 KiB), GiB (1024 MiB). Původní zkratky označují podle SI normy dekadické násobky. Správné čtení by mělo být kibi-, mebi-, gibi-, ale ještě se příliš nepoužívá.

[4] v tehdejší době byl strojový čas velmi drahý a jeho efektivnější využití si vynucovalo souběžnou práci více uživatelů na levných treminálech připojených k jedinému centrálnímu stroji (osobní počítače ještě neexistovaly)

[5] podobné řídící programy existovaly samozřejmě již dříve, ale buď byly orientovány na dávkové zpracování nebo byly velmi jednoduché (např. bez podpory skriptování)

[6] firma AT&T v té době procházela antimonopolním šetřením a nemohla Unix poskytovat komerčním subjektům.

[7] Berkeley Software Distribution

[8] Pouze SunOS (Solaris) je založen na BSD Unixu

systémem z počátku let sedmdesátých. Navíc sjednocení neprováděla jediná všemocná organizace. ale dělo se několika různými cestami. Z dnešního pohledu je nejdůležitější soubor standardů POSIX (IEEE) a X/OPEN (jenž POSIX převzal a dále jej rozvíjí). Většina unixovských systémů splňuje na určité úrovni oba standardy (včetně Linuxu) resp. i standardy další, emuluje další systémy (např. BSD) a navíc obsahuje i své rysy. Není divu, že mnohdy lze stejnou věc udělat v systému i několika různými (vzájemně nekompatibilními) způsoby, což bohužel systém výrazně zvětšuje a zesložiťuje.

Na dlouhou historii Unixu navazuje i kratší, ale již také relativně dlouhá historie Unixu jako skutečně otevřeného systému, jenž je ve formě zdrojových kódů volně k dispozici všem od jednotlivců po firmy, kteří jet tak mohou nejen bezplatně užívat, ale mohou jej i modifikovat podle svých požadavků. Nejstarším počinem v této oblasti je založení FSF (Free Software Foundation, 1985), jejímž plánem bylo vytvoření unixovského systému GNU (GNU's not Unix[9]). Při implementaci systému se začalo nejdříve s pomocnými programy a až poté mělo následovat jádro systému (označované jako Hurd). Během této doby však Linus Torvalds a posléze tisíce programátorů začalo vytvářet vlastní jádro pod označením *Linux*. Při tvorbě tohoto jádra se však používalo GNU překladače jazyka C (gcc) a systém byl rozšířen o velké množsví GNU aplikací (počínaje od shellu *bash*(1)). Jméno Linux nakonec zatlačilo označení GNU poněkud do pozadí (i když se společně vzniklý systém by se měl správně stále označovat jako *GNU/Linux*[10]).

Druhou větví tvorby open-source unixovského operačního systému jsou systémy odvozené z BSD Unixu jako je např. *FreeBSD*, *NetBSD* a *OpenBSD*. I když v některých oblastech *GNU/Linux* překonávají, jsou mnohem méně rozšířeny a především méně podporovány.

1.2 Vestavěná dokumentace Unixu

Unix byl již od počátku dodáván s obsáhlou vestavěnou dokumentací, tzv. manuálovými stránkami (*manpages*). Manuálové stránky i dnes zůstávají hlavním zdrojem informací o systému, avšak již nikoliv jediným.

Manuálové stránky jsou seskupeny do tzv. sekcí, které usnadňují orientaci ve schránkách a alespoň částečně brání kolizím identifikátorů (lze tak např. odlišit příkaz *printf* od knihovní funkce stejného jména). Počet sekcí i jejich význam se může v jednotlivých systémech lišit, ale v zásadě se příliš neliší od původního systému:

1. uživatelské příkazy
2. systémová volání
3. funkce (knihovní některých program. jazyků, mezi nimi i C)
4. zařízení
5. datové formáty
6. hry (nepříliš používáno)
7. různé
8. administrace systému (superuživatelské příkazy)

[9] tento rekurentní akronym je nutno chápat, tak že GNU není Unixem ve smyslu licence a autorských práv (je však přirozeně unixovským systémem)

[10] při formálním označení se tak děje téměř důsledně (viz např. výpis příkazu *uname -a*), v dokumentaci jen výjimečně a v neformální komunikaci snad jen u členů FSF.

Pro prohlížení manuálových stránek lze použít původní program ***man***(1) (dostupný vždy) resp. modernější hypertextové prohlížeče[11] jako např. *konqueror* (součást KDE). Příkaz *man* používá pro zobrazení tzv. stránkovač, jímž bývá v Linuxu program *less*(1) (lze změnit). Krátký popis ovládání tohoto stránkovače je na stránce 67 (zde jen tlačítko pro uzavření – q).

Například pro prohlížení manuálových stránek programu *ls* (výpis adresáře) vyvoláte buď příkaz `man ls`, nebo do Konqueroru zadáte URL *man:ls*. Pokud existuje více manuálových stránek stejného jména (v různých sekcích) je u příkazu *man* nutné zadat číslo sekce (jinak zobrazí pouze první nalezenou stranu, což nemusí být to co chcete), např. `man 3 printf` (C-funkce ze standardní knihovny). V případě Konqueroru vám bude nabídnut seznam všech stránek daného jména, ale i zde lze zadat jednoznačné URL *man:printf(3)*. Konqueror Vám také nabízí rejstřík sekcí (*man:*) resp. rejstřík jednotlivé sekce, např. *man:(1)*.

Manuálové stránky jsou kromě angličtiny dostupné i v jiných jazycích. K některým instalacím Linuxu se standardně dodávají i české manuálové stránky. Jejich počet je však omezen (u mé instalace je poměr 203/6887 ve prospěch angličtiny) a ne vždy bývají aktuální. O tom jaká jazyková verze stránky se implicitně zobrazí rozhoduje nastavení tzv. locales (=místní nastavení). Zobrazení anglické verze si lze vynutit např. takto `LANG="C" man ls`.

Modernějším systémem dokumentace je *info*. Je hierarchizované a je tak vhodnější pro rozsáhlejší dokumentace. Je však masivněji používán pouze v některých volně šiřitelných verzích Unixu (včetně Linuxu). Pro prohlížení lze použít buď původní program *info*, nebo modernější hypertextové prohlížeče (včetně Konqueroru). Jako příklad lze uvést čtení info dokumentace k programu date: příkaz `info date` resp. URL `info:date`.

Další zdrojem informací je dokumentace ve formě HTML (resp. PDF) dokumentů, dostupná pro většinu dodatečných aplikací ve adresáři */usr/share/doc*.

V souladu se zavedenou praxí uvádím u každé aplikace (programu, příkazu) manuálovou sekci (v oblých závorkách za příkazem). Tuto referenci však uvádím jen při prvním (přímém) uvedení dané aplikace (výjimečně i na jiném vhodném místě). Navíc používám i rozšiřující symboly pro info dokumentaci (*info*) a interní nápovědu shellu pro vestavěné příkazy (*int*), jež je v shellu *bash* dostupná příkazem *help*.

ÚKOLY

1. Vyzkoušejte si použití manuálových stránek: *man, info.*
2. Přečtěte si manuálové stránky k příkazům: *cd, ls, rm, rmdir, cp, mv, man,less.*

[11] manuálové stránky nepoužívají HTML (to v době jejich vzniku ještě neexistovalo) a pro jejich prohlížení musí být přítomen speciální formátovací program (*troff*).

2 Shell

Základní aplikací každého unixovského systému je tzv. *shell*, který poskytuje základní uživatelské rozhraní k systému, tj. interaktivní interpret příkazů a základní programovací prostředí pro tvorbu systémových i uživatelských programů (skriptů) orientovaných na správu operačního systému. V původním Unixu byl shell zcela nepostradatelný, dnes již některé jeho funkce přejaly další aplikace např. grafické desktopy (KDE, GNOME), resp. skriptovací jazyky (Perl, Python). Navzdory tomu se unixovské shelly stále používají, neboť v mnoha případech nabízejí nejefektivnější řešení dané problematiky a jsou v maximální míře přenositelné (a všobecně dostupné).

Navíc se stále vyvíjejí a nabízejí komfort, který uživatelé ostatních systémů neznají[1] (pokud mají vůbec něco podobného k dispozici). Shell je často skrytě používán i programy, které jsou buď zcela neinteraktivní (démoni) resp. užívají grafické uživatelské rozhraní (dále jen GUI programy).

Původní unixovský shell vznikl v polovině sedmdesátých let a je podle svého autora označován jako *Bourne shell* (resp. jen *sh* podle jména spustitelného souboru). Tento shell již obsahoval většinu typických rysů unixovských shellů (substituci, kolony, skriptování) a byl (resp. stále je) napodobován i v jiných operačních systémech (před Unixem existovaly pouze relativně jednoduché příkazové jazyky).

Druhým základním shellem je C-shell (*csh*) vzniklý v BSD (univerzitní) větvi Unixu na konci sedmdesátých let. Ten přinesl mnohé novinky (historii, úlohy, řádkovou editaci) je však nekompatibilní s Bourne shellem (přiklonil se totiž k syntaxi jazyka C, podle něhož má i název). Tato nekompatibilita se objevuje především v části skriptovací, ale projevuje se i u konstrukcí používaných interaktivně. Toto základní schizma unixovských shellů stále trvá, neboť i když se obě větve shellů vzájemně ovlivňovaly a přejímaly navzájem nové prvky, původní nekompatibilita zůstává.

V současnosti převažují především nástupci Bourne shellu, které již plně vyrovnaly ba i překonaly prvotní náskok C-shellu v komfortu ovládání, zachovávajíce však původní syntaxi. Mezi nejdůležitější patří *Korn Shell* (*ksh*) užívaný u firemních unixů a *Bourne Again Shell* (*bash*(1)) vytvořený v projektu GNU užívaný na všech unixech, především však v Linuxu. Hlavním nástupcem C-shellu je v dnešní době shell *tcsh*(1) (je standardně dostupný i v Linuxu).

Z ostatních shellů mají jistý význam shelly odlehčené (pro systémy s omezenými prostředky[2]) a shelly specializované (například s důrazem na bezpečnost a ochranu systému).

[1] tím samozřejmě myslím MS Windows
[2] například *ash*

Pro stručnost a přehledností se dále zaměříme především na *Bourne Shell* resp. na *bash*, C-shell bude popisován na jeho pozadí a zmínky o něm budou vždy explicitně označeny.

2.1 Základní interakce se shellem

Základní (zjednodušený) princip práce shellu není složitý – shell zobrazí v prvém kroku tzv. *prompt* (výzvu) a čeká na zadání příkazu. Uživatel zadá příkaz pomocí řádkového editoru (editační řádky) a zadání ukončí stiskem klávesy *Enter* (kdekoliv v editačním řádku). Shell zajistí provedení příkazu a to buď ve své režii (tzv. interní příkaz) nebo jej svěří jinému procesu (který samozřejmě nejdříve vytvoří). Není-li určeno jinak, čeká na ukončení provádění příkazu a následně opět zobrazí výzvu.

Příkazy shellu mohou mít velmi různorodý tvar od jednoduchého jednopísmenného příkazu po mnohosetznakovou výceúrovňovou konstrukci (ty se však jen výjimečně používají interaktivně). Základem příkazu však většinou bývá volání externího programu se sadou přepínačů (voleb) a parametrů.

```
jméno-programu [volba ...] parametr ...
```

Jméno programu může být zadáno bez cesty (pak se hledá v adresářích uvedených v proměnné PATH, nikoliv však v aktuálním adresáři[3]) resp. s absolutní nebo relativní cestou (relativní cestou k atuálnímu adresáři je `./`).

Bohatý repertoár **voleb** (přepínačů) je pro unixovské programy typický. Klasické unixovské volby začínají pomlčkou a jsou jednopísmenné (tvořené malými písmeny) a mohou mít svůj vlastní parametr (zapisuje se po mezeře za volbu). Volby bez parametrů lze u většiny programů združovat tj. zapisovat několik voleb za pomlčku (poslední volba navíc může mít parametr). Volby stejně jako všechny ostatní syntaktické entity shellu musí být navzájem odděleny alespoň jedním bílým znakem[4] (typicky mezerou).

U některých aplikací nestačí pro volby repertoár minusek (malá písmena) a tak se používají i verzálky (velká písmena, v Unixu je velikost písmen vždy významná) nebo dokonce číslice.

U programů, jež podporují ještě vyšší počet voleb se používají i volby víceznakové (většinou ve formě celých slov nebo sousloví). Pro uvození dlouhých voleb programy buď používají standardní pomlčku (krátké volby pak přirozeně nelze združovat), nebo jiný odlišný znak. V Linuxu většina programů používá úzus GNU, jenž pro uvození používá dvě pomlčky a pro oddělení případného vlastního parametru rovnítko.

Relativně častá je i možnost použití několika různých voleb pro stejnou (resp. velmi podobnou) činnost, a to jak z důvodů zpětné nebo křížové kompatibility (viz historie Unixu), tak z důvodů snadnějšího použití (jednoznakové volby jsou často zdvojeny volbou víceznakovou, jež bývá více mnemotechnická).

Některé programy používají volby s odlišnou syntaxí, příčinou bývá převzetí programu z jiného operačního systému resp. specifické požadavky programu (detaily zjistíte z dokumentace programu).

Po nepovinných volbách následují (často opět nepovinné) parametry. Nejčastějšími parametry jsou soubory, ale mohou to být i výčty jiných objektů jako např. fontů, barev

[3] pokud není aktuální adresář uveden přímo v proměnné PATH (pod označením .), což však není z bezpečnostních důvodů standardem

[4] i oddělovač však lze v shellu předdefinovat, při interaktivním použití se tak však činí jen velmi zřídka

apod. Většina unixovských aplikací neomezuje počet souborových (ani jiných) parametrů, pokud není omezen sémantikou programu.

2.2 Nahrazování v shellu

Vstup nemusí být k dalšímu zpracování předán přesně ve tvaru v jakém je zadal uživatel, ale shell může nahrazovat (substituovat) některé jeho části části, aby usnadnil uživateli resp. aplikacím práci resp. jim poskytl dodatečnou funkčnost. Současné shelly znají podporují ne méně než deset různých druhů nahrazení se kterými se budeme seznamovat postupně (a na některé nepříliš často používané se ani nedostane). V této kapitole se zaměříme na ta nejjednodušší (a nejčastěji používané) a také na problematiku jak nahrazením případně zabránit.

Souborové nahrazení (pathname expansion)

Základními souborovými nahrazeními jsou tzv. souborové žolíky (wildcards), jež jsou ve zjednodušené podobě známy i z jiných operačních systémů. Unixovské žolíky jsou shrnuty v následující tabulce.

*	libovolná posloupnost znaků (i prázdná)
?	libovolný (jeden) znak
[*x-yab*]	libovolný (jeden) znak z intervalu *x* až *y* resp. znaky *a*, *b*
[^*x-yab*]	libovolný (jeden) znak neležící ve výše popsané množině

Zápisy používající žolíky (vzory) se rozvíjejí na jména všech souborů v adresáři, jehož cesta substutuci předchází (resp. v adresáři pracovním není-li cesta použita). Neexistuje-li žádný takový soubor není vzor substituován, tj. je předán nezpracovaný aplikaci, která soubor pravděpodobně skončí s chybou při přístupu k neexistujícímu souboru[5]. Rozvoj získaný s tímto nahrazením je abecedně setříděn.

Např. vzor `/etc/[A-Z]*` se rozvine na absolutní jména všech souborů v adresáři `/etc`, jejichž jméno začíná velkým písmenem (přihlíženo je pouze k verzálkám v ASCII tabulce, jež všechny leží v intervalu A-Z[6]). Chcete-li získat setříděný seznam všech souborů v aktuálním adresáři (vyjma skrytých) použijte vzor * (nikoliv *.* jenž popisuje pouze soubory s tečkou ve jméně).

Nahrazení složených závorek (brace expansion)

Nahrazení složených závorek je obecné nahrazení, ktreré se však nejčastěji používá pro specifikaci množiny souborů, kterou nelze popsat pomocí souborového nahrazení. Při tomto nahrazení jsou seznamy čárkami oddělených slov[7] uzavřené do složených závorek rozvinuty do seznamů odělených mezerami. Navíc je každé slovo v rozvinutém seznamu uvozeno prefixem původního seznamu (tj. slovem jež stálo před otvírací

[5] toto je chování standardní chování a lze je změnit pomocí volby shellu *nullglob* (je-li nastaven je výraz odstraněn)

[6] správnějším by tedy byl zápis `/etc/[[:upper:]]*`, jenž používá pojmenovanou znakovou množinu

[7] slovo = libovolná posloupnost znaků vyjma bílých (mezer, apod.)

složenou závorku) a následováno původním sufixem (slovo za složenou závorku). Například zápis `auto{mobil,mat}em` se rozvine v seznam (dvou slov) `automobilem automatem`. Nahrazení složených závorek lze kombinovat se souborovým nahrazením (souborové nahrazení se provede nakonec) a lze je libovolně vzájemně vnořovat.

Příklad: Nalezení řetězce *double* ve všech zdrojových a hlavičkových souborech jazyka C resp. C++ v podadresářích domovského adresáře se jmény *c_prog* (zde se očekávají soubory jazyka C) resp. *cpp_prog* (se soubory C++):

```
grep double ~/{c_prog/*.[ch],cpp_prog/*.{cpp,h,cc,cxx}}
```

po rozvinutí (zank ~ se rozvinul na můj domovský adresář, viz dále):

```
grep double /home/fiser/c_prog/*.[ch] /home/fiser/cpp_prog/*.cpp
/home/fiser/cpp_prog/*.h /home/fiser/cpp_prog/*.cc
/home/fiser/cpp_prog/*.cxx
```

Citování (quoting) – jak zabránit nahrazení

I když jsou nahrazení ve většině případů užitečná, někdy je jim nutno zabránit. Hlavním důvodem je potřeba použití řetězců s expanzními znaky na místě parametrů některých aplikací (ty musí být aplikaci předány bez změn tj. bez nahrazení). I když těmito (nebezpečnými) parametry mohou být i soubory, není to příliš časté, neboť většina uživatelů se vyhýbá použití většiny speciálních znaků v souborech[8]. Mnohem častěji to bývají textové řetězce (např. vyhledávaný resp. vypisovaný text).

Do citování se zahrnuje i ochrana dalších znaků, které mají nějakou speciální syntaktickou funkci v shellu, například mezer (při standardním nastavení oddělují jednotlivé parametry a volby navzájem i od jména programu resp. příkazu). Jsou-li mezery použity např. ve jménu souboru (což není rarita) musí být speciálně označeny, aby nebyly interpretovány jako oddělovače (tj. aby se jméno souboru nerozpadlo na více parametrů). Mezi znaky, které vyžadují citování patří i vlastní citovací znaky.

Existují tři druhy ochrany před nahrazením (resp. před jinou speciální interpretací).

prefixace znaku zpětným lomítkem — zabrání všem nahrazením, je však vhodný pouze pro jednotlivé znaky resp. kratší texty (je nezbytný pro ochranu citovacích uvozovek a apostrofů)

uvození v apostrofech — důsledná ochrana vhodná pro delší texty

uvození v uvozovkách — až na dvě (důležité) výjimky (proměnné shellu, vložení výstupu) brání nahrazení, lze je použít i pro ochranu mezer a dalších oddělovačů

Příklad: Nalezni všechny výskyty sousloví „Ústí nad Labem“ v souborech s maskou „Města 200?.*“ (?,* by měly být použity souborovým nahrazením).

```
grep "Ústí nad Labem" Města\ 200?.*
```

Ne vždy však citování pomůže. Mějme například soubor se jménem `-r` (pomlčka+r), což je platné jméno souboru v Unixu. Při pokusu o jeho smazání příkazem `rm -r`, však obržíte chybovou zprávu `rm: příliš málo argumentů`, neboť jméno souboru je interpretováno jako volba (zde rekurzivní mazání). Ochrana pomocí citování v tomto případě na věci nic nezmění, neboť pomlčka není interpretována shellem, ale cílovým programem (zde tedy programem *rm*). Naštěstí i zde existuje řešení na úrovni shellu[9], neboť

[8] často po vlive, MS Windows, kde je většina speciálních znaků ve jménech souborů zakázána.
[9] soubor lze přirozeně smazat i externím programem např. GUI správcem souborů.

většina programů[10] dodržují úmluvu, že samotná dvojitá pomlčka ukončuje seznam voleb. Správným zápisem je tedy `rm -- -r`.

2.3 Editace příkazové řádky

C-shell přinesl již na počátku 80-let řádkovou editaci a historii příkazů (v té době ještě MS-DOS neexistoval), později byl komfort unixovských shellů dále rozšířen například o kontextové doplňování nebo správu úloh (*jobs*). I když není znalost těchto rozšiřujících funkcí shellu pro práci s Linuxem nezbytná, může výrazně usnadnit a urychlit komunikaci se se systémem.

Editaci příkazové řádky u shellu bash zajišťuje knihovna *readline*(info), jenž je použita i u některých dalších řádkově orientovaných linuxovských aplikací. Základní klávesové zkratky (lze je přirozeně i redefinovat) vycházejí z editoru Emacs (viz podkapitola na straně **??**), jsou však přirozeně omezeny pouze na jediný rozměr. Nejčastěji se kromě kurzorových kláves používá přesun na počátek řádku (Ctrl+A, Home) nebo na jeho konec (Ctrl+E, End). S určitými omezeními lze použít i funkce schránky (viz tabulka **??**) a *undo* (Ctrl+_, Ctrl+X Ctrl+U).

Nestačí-li vám editační možnosti příkazové řádky lze pomocí klávesové zkratky Ctrl+X Ctrl+E vyvolat systémový editor, který Vám editaci může usnadnit. Systémový editor je určen hodnotou proměnné EDITOR (standardně je to *vi*). Pokud jej chcete trvale změnit přidejte do souboru *~/.bashrc* řádek:

`export EDITOR=`*`váš-oblíbený-editor`*

Kontextové doplňování usnadňuje zápis dlouhých jmen programů, souborů, adresářů, apod. Navíc je automaticky ověřována existence těchto objektů. Všechny druhy doplňování spočívají na stejném principu — stisk klávesy *Tab* na příslušném místě příkazového řádku spustí vyhledání všech objektů, jež lze na daném místě použít. Pokud existuje právě jeden použitelný objekt je jeho jméno vloženo do řádku, v opačném případě se buď neděje zdánlivě nic (žádný objekt neexistuje, jména objektů nemají žádný společný prefix) nebo se doplní nejdelší společný prefix přípustných jmen. Po opakovaném stisku tabelátoru se však zobrazí seznam všech přípustných jmen (pokud samozřejmě nějaká existují). Je-li jmen více (řádově stovky) musíte svůj zájem na zobrazení seznamu potvrdit.

Pokud je klávesa *Tab* stisknuta bezprostředně za znakem, který není bílým nebo speciálním znakem, je tento znak a znaky bezprostředně před ním interpretovány jako prefix jména objektu a při doplňování jsou zohledňovány pouze objektym jež tímto prefixem začínají (čímž se jejich počet ve většině případů výrazně sníží). Při doplňování tedy stačí napsat několik prvních znaků jména (dvě či tři) stisknout *Tab*, a pokud již není objekt jednoznačně určen, doplnit znaky další (nápovědou je doplněný prefix nebo vygenerovaný seznam) a pokusit se o doplnění znovu. Po určité době lze tento postup vyladit tak, že stačí jen několik stisků kláves i pro zadání dlouhých jmen.

Typy doplňování:

doplňování příkazů (aliasů, shellových funkcí) – na počátku řádku (prázdný řádek, první prefix na řádku), doplní externí program (podle proměnné PATH), resp. interní jméno (interní příkaz, alias, funkce).

[10] alespoň těch systémových

doplňování uživatelských jmen – po tildě, doplňuje jméno uživatele podle */etc/passwd*

doplňování jmen serverů – po zavináči (podle */etc/hosts* tj. nepříliš použitelné)

doplňování souborů – ve všech ostatních případech, doplňovat lze postupně jména adresářů (nejrychlejší cesta průchodu adresářovým stromem) a následně ostatních souborů

2.4 Historie

Veškeré zadané příkazy shellu jsou po svém potvrzení (a před většinou nahrazení) ukládány do operační paměti dané instance shellu[11] a po ukončení sezení připojeny do souboru historie (implicitně *~/.bash_history*), z níž jsou opětně po spuštění každého nového shellu načítány (uložení historie je sdílena všemi novými shelly). Maximální počet uložených příkazů v jednom sezení je standardně 500, lze jej však změnit pomocí proměnné shellu $HISTSIZE, podobně lze omezit i maximální velikost souboru historie pomocí proměnné $HISTFILESIZE (implicitně je neomezená), vždy by však mělo platit $HISTFILESIZE $\geq$ $HISTSIZE. Osobně doporučuji nastavit $HISTSIZE alespoň na 1000 a $HISTFILESIZE na 10000 (nebo nechat neomezenou) a to ve skriptu *~./bash_profile*.

Uloženou historii interakce uživatele se systémem lze užívat několika způsoby, od jednoduchého nahrazování po editaci a komplexní vyhledávání. Navíc je většina mechanismů spojených s historií ve výrazné míře konfigurovatelná (viz *bash*(1)), což však na druhou stranu znesnadňuje jejich popis (popsaná konfigurace je standardní, i když mírné rozdíly v jednotlivých instalacích nelze vyloučit).

Procházení historií

Pro rychlé prohlížení historie slouží kurzorové klávesy ↑ (předchozí příkaz, je možno použít klávesovou zkratkou Ctrl+P) a ↓ (následující příkaz, Ctrl+N). Cílenější a rychlejší procházení umožňuje inkrementální vyhledávání dostupné pomocí klávesové zkratky Ctrl+R. Po jejím stisku stačí postupně zadávat jednotlivá písmena hledaného řetězce, při němž se postupně objevují příkazy historie (vždy ten poslední obsahující již zadané znaky).

Nahrazování historií

Stejně jako ostatní nahrazování (kap. 2.2 na straně 12) spočívá nahrazování historií v substituci textu příkazového řádku po jeho potvrzení, ale před jeho provedením (nahrazování historií se provádí před ostatními nahrazeními). Specifickými znaky substituovaných řetězců jsou vykřičník resp. vokáň (^), substituující texty jsou přebírány z historie. Mezi nejdůležitější a nejužitečnější patří:

!*číslo* – nahrazeno příkazem s daným číslem v historii (jak toto číslo zjistit viz níže)

!*řetězec* – nahrazeno posledním příkazem začínajícím daným řetězcem (vhodné při cyklickém zadávání stále stejných příkazů)

!?*řetězec*? – nahrazeno posledním příkazem obsahujícím daný řetězec

[11] každá instance shellu si tak udržuje vlastní historii sezení

Vyhledávání v historii

Pro komplexnejší vyhledávání v historii lze použít vestavěného příkazu *history*(int), který vypíše celou historii (tj. obsah souboru *~/.bash_history* + historii aktuálního sezení), přičemž každý řádek je opatřen číslem příkazu. Tento výpis, jež může být velmi rozsáhlý lze prohlížet pomocí stránkovače (např. *less*) nebo filtrovat pomocí grepu (resp. zpracovávat jakýmkoliv jiným způsobem).

Příklad: prohlížení historie: `history | less`
nalezení všech příkazů pracujících se zdrojovými soubory C++[12]: `history | grep -E "\.cpp\b"`

2.5 Shell a terminál

Shell je terminálová aplikace tj. používá pro interakci s uživatelem zařízení označované v Linuxu jako terminál (zkráceně *tty*). Původně se jednalo o fyzické terminály (nepříliš inteligentní a tudíž levné) spojené s centrálním počítačem (na němž běžel operační systém) pomocí sériového kabelu. I když se v současnosti fyzické terminály téměř nepoužívají, zůstává jejich podpora (která je v jádru shodná s původním Unixem) důležitou součástí vstupně-výstupního systému Linuxu. Místo fyzických terminálů však využívá terminálů virtuálních (tj. softwarově emulovaných na jiných fyzických zařízeních).

Mezi základní typy virtuálních terminálů patří:

- virtuální terminály na systémové konzoli (například na textové obrazovce PC)
- terminálové programy užívající internetový protokol *telnet* resp. *ssh*
- terminálové aplikce v X-Windows (xterm a jeho následníci)

Virtuální terminály (resp. terminálové aplikace) lze rozdělit do dvou nevyhraněných skupin. Terminály první skupiny sůsledně emulují chování původních fyzických terminálů (např. terminálu VT-100) a nejsou tudíž příliš inteligentní. Druhá skupina sice vychází z původních terminálu (a z důvodů kompatibility mnohé jejich funkce emuluje) přináší však vlastní protokol (např. xterm, linux) a četná rozšíření.

Linux však zachovává i jisté prvky staršího stavu kdy se používaly jako terminály dálnopisy (*teletypes* → *tty*), což bylo zařízení které zadávané znaky tisklo (*print*) na papír. Protože však editační schopnosti dálnopisu jsou velmi omezené (nelze např. mazat znaky), podporoval původní Unix několik editačních a řídících příkazů přímo na úrovni terminálu. Většina těchto editačních možností se již nepoužívá (jsouce nahrazeny výše popsanou knihovnou *readline* resp. podporou na úrovni shellu) existují však výjimky.

Následující přehled shrnuje ty nejvýznamnější funkce poskytované terminálem včetně implicitní klávesových zkratek:

přerušení procesu (*intr*, Ctrl+C) — všem procesům spojeným s terminálem je poslán signál SIGINT (viz kapitola 5.4 na straně 56). Shell reaguje přerušením vstupu a vypsáním nové výzvy (promptu). Některé jiné aplikace mohou na signál reagovat svým ukončením.

[12] přesněji nalezne příkazy obsahující řetězec *.cpp* (za ním musí být hranice slova). Tento řetězec se však povětšinou vyskytuje jen v příponách zpracovávaných souborů (výjimkou je mimo jiné samotný příkaz pro filtrování historie).

ukončení vstupu (*eof*, Ctrl+D) — uzavření vstupů spojených s terminálem (vstupní proud se dostane do stejného stavu jako při dosažení konce souboru).

pozastavení terminálu (*stop*, Ctrl+S) — všechny procesy na terminálu jsou pozastaveny. Terminál tak zdánlivě zmrzne (tj. nic není vypisováno, ani nelze nic zadat).

restart pozastaveného terminálu (*start*, Ctrl+Q) — obnovení běhu všech pozastavených procesů na terminálu. Na některých terminálech stačí pro obnovení stisknout libovolnou klávesu.

suspendování procesu (*susp*, Ctr+Z) — pokud na popředí běží proces (nikoliv sám shell) je přesunut do pozadí (a pokud pracuje s terminálem je pozastaven), na popředí je následně vypsána výzva shellu. Další informace viz kapitola 8.1 na straně 97.

Pro konfiguraci terminálu resp. zjištění jeho aktuálního stavu lze použít program *stty*(1). Stav terminálu ovlivňuje několik desítek nastavení v zásadě se věk může nacházet ve dvou základních režimech:

cooked – vstup je bufferován po řádcích, řídící znaky jsou interpretovány v souladu s aktuálním nastavení. Tento režim je využíván interaktivním shellem i většinou programů užívajících interaktivní textový vstup nebo výstup.

raw — vstup není filtrován, řídící znaky nejsou interpretovány, vstup není bufferován. Tento stav je vhodný pro přímý přístup k zařízení (v shellu je užíván pouze omezeně v některých skriptech).

ÚKOLY

1. Vyzkoušejte si zde uvedené příklady.
2. Pomocí programu *less*, nebo editoru *nano* si prohlédněte následující soubory a zjistěte k čemu slouží
   - .bash_history
   - .bash_logout
   - .bashrc

3 Souborový systém

V Unixu (a tím přirozeně i v Linuxu) je souborový systém centrální částí operačního systému, neboť kromě poskytování místa pro persistentní[1] ukládání dat, ale i jako centrální jmenný prostor pro identifikaci prostředků operačního systému.

Základem souborového systému v Unixu jsou regulární soubory a adresáře. Regulární soubory slouží pro ukládání aplikačních a systémových dat, adresáře vytvářejí jednotnou stromovou hierarchickou strukturu identifikátorů souborů (resp. dalších prostředků).

Hierarchie adresářů tvoří v Unixu jediný strom s jediným kořenovým adresářem, jež je označen jako / (lomítko). Ostatní adresáře jsou buď přímými podadresáři kořenového adresáře nebo podadresářů nižších úrovní. V každém adresáři mohou být navíc umístěny i regulární soubory, jež tak tvoří listy celé hierarchie.

Adresáře spolu s regulárními soubory (a některými speciálními objekty jež mohou být umístěny v adresářové struktuře) jsou v Unixu označovány jako *soubory*. Tento termín je tak v Unixu značně širší než například v MS Windows, kde je omezen pouze na objekty, jež se v Unixu označují jako regulární soubory. Každý soubor je opatřen tzv. bázovým jmenem, což je řetězec libovolných znaků kromě znaku lomítko. Celkový repertoár znaků a maximální délka jména závisí na nízkoúrovňovém sytému (viz dále), modernější systémy umožňují obecně použít všechny znaky znakové sady Unicode[2] a maximální délku alespoň 250 bytů (což u nejčastěji používaného kódování UTF-8, jehož kódy mají různou šířku nemusí vždy znamenat 250 znaků[3]. Původní Unixy však podporovaly pouze maximálně čtrnáctiznakové ASCII řetězce.

Jména (bázová) souborů (zde a dále jen v širším unixovském smyslu) v adresáři musí být jedinečná, což však přirozeně nemusí platit v rámci celého souborového systému. V tomto globálním rámci je plně kvalifikovaným jménem tzv. úplné jméno obsahující (absolutní) cestu k souboru od kořenového adresáře. Jednotlivé soubory v úplné cestě jsou od sebe odděleny lomítkem (nikoliv zpětným lomítkem jako v MS Windows) a cesta sama začíná lomítkem, jež v této roli označuje kořenový adresář. Kromě posledního členu cesty musí být ostatní soubory v cestě adresáři (resp. symbolickými odkazy na adresáře). Systémová volání pracující se souborovým systémem navíc omezují maximální délku úplného jména, kterou jsou schopni akceptovat (v současnosti cca 4000 bytů). Delší cesty mohou sice existovat (hloubka zanoření není omezena), ale nelze je

[1] persistentní úložiště uchovávají data i po ukončení aplikací, přebootování systému resp. i po vypnutí počítače.

[2] platí pro distribuce vydané od roku 2003 (včetně)

[3] jednobytové jsou v UTF-8 jen ASCII znaky, znaky s českou diakritikou jsou dvoubytové, základní čínské znaky tříbytové, exotické znaky mohou být i čtyřbytové)

použít jako parametr systémových volání a tím i většiny příkazů (pokud už tato situaci nastane je nutno použít kratší relativní cesty[4]).

Každý proces si v souborovém systému udržuje tři záchytné body – adresáře. Prvním kořenový adresář procesu, jenž je u většiny procesů shodný s kořenovým adresářem souborového systému. U některých procesů je však z bezpečnostních důvodů nastaven na jiný adresář, což těmto procesům zabraňuje přistupovat k souborům mimo vymezený podstrom.

Dalším specifickým adresářem procesu je tzv. *pracovní adresář* (*working directory*). Význam pracovního adresáře spočívá v především v možnosti krácení úplných jmen souborů, neboť od pracovního adresáře začínají tzv. relativní cesty, které lze použít na místo absolutních ve jménech souborů. Relativní cesty nezačínají na rozdíl od absolutních cest lomítkem a často se v nich používá symbolické jméno bezprostředně nadřízeného adresáře `..` (dvě tečky)[5].

Poslední adresář není spojen přímo s procesem, ale s uživatelem-vlastníkem procesu. Je to domovský adresář uživatele-vlastníka. Na systémové úrovni nemá pro proces žádné speciální postavení, mnohé procesy však v tomto adresáři ukládají uživatelská nastavení a nabízejí jej jako východiskou prohledávání souborového systému (např. GUI aplikace ve svých *Open*/*Save as* dialogových oknech).

3.1 Základní hierarchie adresářů

Protože neexistuje žádný standardní Unix nebo Linux neexistuje ani žádná standardní hierarchie adresářů. Přesto však unixovské systémy sdílejí společnou kostru, která je ve svém jádru společným dědictvím z původního jednotného Unixu. Také se navzájem ovlivňují a některé standardy předpokládají jisté dílčí hierarchie (podstromy). V Linuxu je situace navíc příznivější existencí de facto standardu „The Linux filesystem standard“, který je detailnější a většina distribucí jej dodržuje (přirozeně nikoliv důsledně, neboť některé zásady jsou předmětem nekončících diskusí).

Rozsah tohoto standardu (a linuxovských souborových systému) je značný a proto na tomto místě uvedu jen ty nejdůležitější adresáře (které byste i jako začátečníci měli znát)[6]. Pro zajímavost uvádím i počty souborů v některých adresářích získané z mého domácího Linuxu, důležitá nejsou absolutní čísla (která se přirozeně mění) ale spíše poměry naznačující skutečné využití adresářů.

Spustitelné soubory (aplikace) a knihovny

/bin (96) – obsahuje základní programy a aplikace nutné pro základní nastavení či opravu systému (jsou dostupné i při částečné nefunkčnosti systému a lze je použít pro jeho záchranu v tzv. jednouživatelském systému kdy může být přihlášen pouze *root*). Při normálním běhu jsou však využívány i běžnými uživateli. V adresáři */bin* se nachází vždy alespoň jeden *shell*.

[4] tato situace se však v praxi téměř nevyskytuje (nejdelší cesta v mém Red Hat Linuxu má méně než 500 znaků)

[5] lze je formálně používat i v absolutních cestách, jsou však vždy delší než cesty bez zpětného chodu (a navíc narušují jednoznačnost absolutních cest)

[6] další informace viz *hier(7)*

/sbin (318) – podobně jako u /bin obsahuje základní programy dostupné i při částečné nefunkčnosti systému, nepředpokládá se však jejich využití běžnými uživateli.

/usr/bin (3414) – obsahuje zbývající programy užívané superuživatelem i běžnými uživateli dodané spolu se systémem (mnohdy obsahuje i hlavní spustitelné soubory přidaných balíků či alespoň odkazy na ně)

/usr/sbin (519) – obsahuje zbývající systémové programy (používané pouze superuživatelem) dodávané se systémem

/usr/local/bin (52) – spustitelné soubory doinstalovaných aplikačních balíků

/usr/local/sbin (0) – doinstalované systémové nástroje

/usr/X11R6/bin (193) – spustitelné soubory pro X-Window (GUI rozhraní). Většinou zde však bývá pouze starší vrstva programů (dnes již nepříliš užívaných). Programy pro nové desktopy (Gnome resp. KDE) jsou v */usr/bin*.

/opt/bin – spustitelné soubory doinstalovaných aplikací často variantních (instalovaných mimo */usr*, aby nedošlo ke kolizím)

~/bin – spustitelné soubory instalované nebo vytvořené uživatelem

/lib (1870) – hlavní knihovny (nutné i při částečné nefunkčnosti systému)

/usr/lib (32523) – systémové knihovny (včetně knihoven, jež mohou být potenciálně užívány více aplikacemi). Většina knihoven je dynamicky linkovaná (sdílená). Některé aplikace však do tohoto adresáře ukládají i dokumentaci nebo data.

/usr/local/lib (465) – lokální systémy (používané pouze jednotlivými doinstalovanými aplikacemi)

/opt/lib – variantní knihovny

/usr/X11R6/lib (169) – nízkoúrovňové knihovny X-Window (stále používané)

Datové a konfigurační soubory

/etc (2765) – systémové konfigurační soubory + konfigurační soubory démonů + konfigurační soubory X-Window (/etc/X11). Navíc zde jsou některé systémové skripty (především bootovací)

/usr/etc (2) – nastavení, jež mohou být přístupná z několika počítačů (v síti). Je na něj odkazováno z */etc*.

/usr/local/etc (4) – konfigurační systémy dodatečných aplikací

/usr/share – většina datových souborů jednotlivých aplikací (měly by to být ty systémové nebo alespoň dodané se systémem)

/usr/share/doc – dokumentace jednotlivých aplikačních balíků.

/usr/share/man – manuálové stránky (valná většina, ale nikoliv všechny neboť některé aplikace je mají ve svých adresářích). Dříve byly v adresáři /usr/man (pár jic je tam dodnes)

/usr/share/info – info soubory

/var – soubory, jejichž obsah se průběžně mění (logovací soubory, vyrovnávací paměti, apod). Tento adresář musí být na svazku, jež stále umožňuje zápis a čtení (ostatní adresáře mimo domovských mohou být na svazcích připojených pouze pro čtení)

/var/log – logovací soubory

/var/run – soubory s PID běžících služeb (démonů), viz níže

/var/www – oblíbené místo pro WWW stránky (*documentroot* WWW serveru Apache)

Další důležité adresáře

/boot – obsahuje mimo jiné jádro (resp. jádra) Linuxu, jež se načítá do paměti při bootování systému

/dev – obsahuje speciální soubory jednotlivých zařízení

/proc – virtuální adresář zobrazující ve své rozvětvené hierarchii základní informace o systému a procesech

/mnt – v jednotlivých adresářích jsou přípojné body dočasně připojovaných systémů (v nových systémech jen nevýměnných médií)

/media – v nových systémech přípojní body výměnných médií (disketových a CDROM mechanik, USB disků, apod.)

/home – implicitní místo domovských adresářů

/tmp – systémový adresář dočasných souborů. Do tohoto adresáře mohou zapisovat všichni uživatelé (ale soubory může mazat pouze jejich tvůrce). Obsah adresáře je cyklicky promazáván.

/var/tmp – variantní místo dočasných souborů, vhodné v systémech kde je na zapisovatelném médiu pouze adresář */var* (např. Live-distribuce linuxu)

3.2 Svazky

I když se souborový systém jeví běžnému uživateli jako jediný strom, mohou být jeho jednotlivé části umístěny na různých zařízeních a to jak lokálních (=umístěných na počítači na němž běží daní instance operačního systému) tak vzdálených (na jiných počítačích v síti nebo na Internetu).

Zařízení mohou být různého druhu, ale z hlediska správce souborového systému se jedná o bloková zařízení na nichž je vybudován dílčí souborový systém. Linux podporuje více typů souborového systému, např. FAT (MSDOS), VFAT (Windows32) nebo *ext2* resp. *ext3* (dnes de facto nativní souborové systémy Linuxu). Pro zařízení nesoucí dílčí souborové systémy se používá termín *svazek* (volume).

Z těchto dílčích souborových systémů je vybudován jednotný souborový systém pomocí mechanismu, jež je označován jako připojování (mounting) svazků.

Základem je tzv. kořenový svazek, jehož kořenový adresář se stává kořenovým adresářem výsledné hierarchie. Tento svazek je připojen již v raných fázích bootování systému a jeho přítomnost je nezbytná. Ostatní dílčí systémy se připojují standardním způsobem pomocí příkazu *mount*(8). Při připojování se kořenový adresář připojovaného svazku de facto ztotožní s některým z adresářů systémového svazku (může to však obecně i adresář jiného již připojeného svazku), jenž je pak označován jako přípojný bod (*mount point*). Původní obsah přípojného adresáře se stává neviditelným (povětšinou je však prázdný) a je překryt obsahem připojeného kořenového adresáře. Dílčí souborový systém je tak naroubován na původní strom a stává se tak jeho podstromem.

Každý připojený svazek je posléze nutno odpojit pomocí programu *umount*(8). U pevných disků a jiných nevyjímatelných zařízení se tak děje těsně před ukončením běhu systému, u vyjímatelných před jejich vyjmutím. Některá média např. CDROM bez odpojení ani nelze vyjmout, jiná sice vyjmout lze (např. USB disky) může však dojít k porušení dat.

Základní údaje o svazcích lze získat příkazem ***df***(1), jenž poskytuje následující výstup (ukázka z mého domácího systému):

```
/Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/hda1              8088120   6809064    868196  89% /
/dev/hda2              2071416   1853016    113176  95% /backup
/dev/hda5              8262036   6741780   1100560  86% /home
/dev/hda6               727624    712868     14756  98% /install
/dev/sda1                29808     28240      1568  95% /mnt/usb
```

První sloupec identifikuje blokové zařízení na němž je svazek umístěn (zde povětšinou oddíly na prvém IDE disku, pouze poslední identifikuje USB flash pamět přístupnou jako oddíl na SCSI zařízení). Další jeho velikost a využití (v pořadí celková velikost v 1KiB blocích[7], použito, volno, procento použitého). Poslední sloupec pak obsahuje absolutní cestu k přípojnému bodu (jak je vidět systémový svazek je uveden jako první).

Připojování a následné odpojování svazků je ještě víco zjednodušeno pomocí tabulky */etc/fstab*. Tato tabulka shrnuje všechna zařízení, která mohou obsahovat připojitelné svazky i základní infomace o nich a informace z ní využívá příkaz *mount* i mnohé další systémové nástroje (včetně různých GUI pomůcek).

Strukturu tabulky si ukažme na konktrétním příkladě:

```
/dev/hda5  /home        ext3         defaults               1 2
/dev/hda6  /install     jfs          defaults               1 2
none       /proc        proc         defaults               0 0
/dev/hda7  swap         swap         defaults               0 0
/dev/cdrom /mnt/cdrom   udf,iso9660  noauto,owner,ro        0 0
/dev/fd0   /mnt/floppy  auto         noauto,users           0 0
/dev/sda1  /mnt/usb     vfat         noauto,user,umask=0077 0 0
```

První sloupec identifikuje zařízení na němž je svazek uložen, podle jeho speciálního souboru v adresáři /dev. V našem případě jsou to rozšiřující oddíly na prvém IDE disku (počínají od *hda5*), CDROM mechanika, disketová mechanika (*fd0*) a SCSI zařízení zajišťující v tomto konkrétním případě připojení USB flash paměti (*sda1*[8]). Druhý sloupec určuje přípojný bod (ten musí v okamžiku připojení existovat). Třetí sloupec určuje typ připojovaného souborového systému. Zde lze uvést buď jediný typ (pokud se pomocí daného souboru připojuje jediné paměťové medium), seznam přípustných typů (ne všechny typy lze však jednoznačně rozlišit) resp. označení *auto* (typ je určen podle počátečních bloků svazku, u některých spíše exotických typů nelze použít).

Dalčí sloupec obsahuje základní volby připojení. Většinou musí být uvedena volba *defaults* u nevýměnných zařízení s unixovským souborovým systémem (při startu jsou automaticky připojeny) resp. *noato* u zařízení výměnných. U výměnných zařízení je také často nutno uvést jednu z voleb *owner*, *user*, *users* jinak svazek připojí pouze superuživatel (i když znáte heslo je to značně nepohodlné). Při volbě *owner* smí svazek

[7] přehlednější výstup velikostí lze získat přepínačem -h

[8] některé USB *mass storage* zařízení používají namísto souboru *sda1* (první oddíl naprvém SCSI zařízení) soubor *sda* (prvé zařízení jako celek).

připojit/odpojit vlastník příslušného speciálního souboru (vhodné pokud chcete připojení povolit jedinému běžnému uživateli), volba *user* umožňuje připojení všem uživatelům, odpojení však může provést jen ten kdo dílčí systém připojoval. Volba *users* toto omezení nemá (je víceuživatelských systémů je však nebezpečná).

Poslední důležitou volnou je *ro*, která si vynutí připojení zařízení pouze pro čtení (*read-only*). Zde je použita u CDROM, kde je přirozená (CDROM není zapisovatelným zařízením[9]), ale lze ji použít např. i u pevných disků pokid chcete zabránit zápisu do dané části souborového systému (na takováto zařízení nemůže zapisovat ani superuživatel, pokud je samozřejmě neodpojí a znovu nepřipojí bez této volby).

Poslední dva sloupce souvisí s programem *dump* (zálohování) a *fsck* (kontrola souborového systému). Pokud je nehodláte na daný svazek použít (u neunixovských souborových systémů to ani nemá valný smysl) použijte v těchto sloupcích nuly.

Tabulka */etc/fstab* se používá i pro speciální souborové systémy (jež často nemají fyzickou representaci) respektive pro odkládací prostory (*swap*). Tyto řádky jsou většinou nastaveny již při instalaci a pokud nemáte vážný důvod (např. připojení nového swapovacího prostoru) tak je neměnte (a především nemažte!).

Pokud máte potenciální souborová zařízení uvedena v */etc/fstab*[10], stačí pro jejich připojení uvést příkaz `mount přípojný_bod` (například `mount /mnt/usb`) nebo použít grafický nástroj z grafického desktopu (něco jako „Můj počítač“, „disky“ nebo „mount applet“). Podobně funguje i odpojení.

Svazky bez volby *noauto* (včetně těch s volbou *defaults*) jsou automaticky připojeny při startu systému a odpojeny před jeho ukončením.

3.3 Základní příkazy pro práci v souborovém systému

Příkazy pracující se souborovým systémem se v Linuxu na rozdíl od světa MS Windows stále používají a to jak pro interaktivní práci tak v systémových a uživatelský skriptech. Důvodem je jejich lepší návrh (původní příkazy MS DOSu byly jen jejich nepříliš povedenou napodobeninou) tak i podpora všech i těch nejmodernějších vlastností operačního a souborového systému (včetně dlouhých jmen, odkazů, pojmenovaných rour, správy úloh, apod.)

I v Linuxu však přirozeně existují vizuálně orientovaní správci souborů počínajíc obdobou Norton Commanderu (*Midnight Commander* – *mc*(1)) po mamuty typu *Konqueroru* (KDE) a *Nautila* (Gnome). Tyto vizuální správci jsou pro mnohé typy úloh mnohem efektivnější (například pro některé hromadné úlohy), nikoliv však pro všechny. Jejich ovládání zde nebude popsáno, neboť se v zásadě neliší od svých vzorů (mnohdy bohužel).

Nejdříve se pojďme detailněji podívat na dva základní příkazy *ls* a *cp*:

[9] to platí i pro CD vypalovačky, které pro zápis nepoužívají souborový systém.

[10] tabulka je naplněna již po instalaci systému odkazy na všechna automaticky identifikovaná zařízení, většinou stačí pouze její mírná modifikace nebo malé rozšíření

zobrazení obsahu adresáře – ls

Příkaz ***ls****(1)* bez voleb a parametrů zobrazuje obsah pracovního adresáře shellu. Pokud jde tento výstup na terminál je pro přehlednost rozdělen do několika sloupců a barevně jsou vyznačeny některé základní typy souborů (barevné vyznačování nemusí fungovat na všech terminálech a ve všech instalacích).

Parametrem příkazu *ls* bývá většinou jediný adresář (jeho zobrazen jeho obsah), ale může to být i více adresářů (je zobrazen obsah každého z nich) nebo dokonce jeden nebo několik souborů různého druhu (u souborů, které nejsou adresáří je zobrazeno jejich jméno resp. informace o nich).

Další informace o souborech resp. modifikace výstupního formátu lze získat pomocí velkého počtu dostupných voleb.

Mezi nejpoužívanější volby příkazu *ls* patří:

přepínač	**význam**
-l	detailní informace o souboru (viz dále)
-a	vypíše i skryté soubory
-d	vypíše informace o vlastním adresáři nikoliv o jeho souborech
-F	za jménem souboru vypíše znak určující jeho typ (ne vždy funguje barevné zvýraznění)
-t	třiď podle času (implicitně času změny)
-i	vypisuje i-číslo souborů
-s	zobrazuje alokované místo (v KB)

Přepínače lze přirozeně kombinovat (ne vždy je to však rozumné). Například volání `ls -alF /etc/X11` vyprodukovalo na mém počítači následující výstup:

```
total 96
drwxr-xr-x   17 root     root         4096 Apr 14  2003 ./
drwxr-xr-x   96 root     root         8192 Nov 15 22:27 ../
lrwxrwxrwx    1 root     root           27 Apr 14  2003 X -> ../../usr/X11R6/bin/XFree86*
-rw-rw-r--    1 root     root         3211 Apr 14  2003 XF86Config
-rw-rw-r--    1 root     root         3234 Apr 14  2003 XF86Config.backup
-rw-r--r--    1 root     root          372 Feb 27  2003 XftConfig.README-OBSOLETE
-rw-r--r--    1 root     root          613 Feb  5  2003 Xmodmap
-rw-r--r--    1 root     root          492 Feb  5  2003 Xresources
drwxr-xr-x    5 root     root         4096 Apr  9  2003 applnk/
drwxr-xr-x    2 root     root         4096 Apr  9  2003 desktop-menus/
drwxr-xr-x    2 root     root         4096 Apr  9  2003 fs/
drwxr-xr-x    6 root     root         4096 Feb 18  2003 gdm/
drwxr-xr-x    2 root     root         4096 Apr  9  2003 lbxproxy/
-rwxr-xr-x    1 root     root         1419 Mar 13  2003 prefdm*
drwxr-xr-x    2 root     root         4096 Apr  9  2003 proxymngr/
drwxr-xr-x    4 root     root         4096 Apr  9  2003 rstart/
drwxr-xr-x    2 root     root         4096 Jan 25  2003 serverconfig/
drwxr-xr-x    2 root     root         4096 Apr  9  2003 starthere/
drwxr-xr-x    2 root     root         4096 Apr  9  2003 sysconfig/
drwxr-xr-x    2 root     root         4096 Apr  9  2003 twm/
```

```
drwxr-xr-x    3 root     root         4096 Apr  9  2003 xdm/
drwxr-xr-x    3 root     root         4096 Feb 27  2003 xinit/
lrwxrwxrwx    1 root     root           27 Apr  9  2003 xkb -> ../../usr/X11R6/lib/X11/xkb/
drwxr-xr-x    2 root     root         4096 Apr  9  2003 xserver/
drwxr-xr-x    2 root     root         4096 Apr  9  2003 xsm/
```

Výpis obsahuje všechny soubory v adresáři (včetně skrytých), kromě jména zobrazuje i další informace a jméno souboru (v posledním sloupci) je doplněno o symbol určující jeho typ (zde trochu nadbytečně).

První znak každého řádku přináší informaci o typu souboru. V našem výpis se vyskytují znaky „-“ (regulární soubor), „d“ (adresář), a „l“ (symbolický odkaz). Kromě toho se používají znaky „p“ – pojmenovaná roura, „s“ – unixovský socket, „b“ – blokové zařízení a „c“ – znakové zařízení (soubory zařízení jsou vesměs v adresáři */dev*).

Další devítice znaků zobrazuje přístupová práva souborů (viz podkapitola 3.5 na straně 30).

Malé přirozené číslo v následujícím sloupci zobrazuje počet odkazů na fyzický soubor z adresářové hierarchie. U souborů bývá tento počet roven 1 (tj. soubor je dostupný pouze pod jedním jménem). Výjimkou jsou dnes již téměř nepoužívané pevné odkazy (viz podkapitola 3.4 na straně 27). Naopak u adresářů je počet odkazů vždy vyšší nebo roven dvěma, neboť každý adresář je odkazován z nadřízeného adresáře (pod vlastním jménem), sám na sebe odkazuje jménem „.“ (tečka) a ze všech podřízených adresářů je odkazován jménem „..“ (dvě tečky). Pokud tedy od počtů odkazů u adresáře odečteme dvojku získáme počet jeho podadreářů.

Další dva sloupce obsahují informaci o uživateli-vlastníkoví a oprávněné skupině (zde je to bez výjimky superuživatel a jeho pseudoskupina).

Další sloupec přináší informace o velikosti souboru (v bytech). Tato velikost však ne vždy odpovídá místu, jež soubor okupuje na disku. Nehledíme-li na využívaní celých alokačních jednotek (řádově jednotky KiB), jež alokovaný prostor mírně zvyšuje existují v Unixu i soubory, které zabírají na disku jen zlomek své velikosti, neboť Unix alokuje jen bloky na něž bylo zapisováno (a zapsat lze díky náhodnému přístupu pomocí služby *lseek*(2) na libovolnou adresu v souboru)[11].

Další část výpisu obsahuje čas poslední změny souboru ve zkráceném formátu vhodném pro přímé čtení (pro další strojové zpracování je vhodnější úplnější a pravidelnější formát dostupný pomocí dlouhé volby `--full-time`). Pro soubory starší 180 dní je zobrazován jen datum včetně roku pro novější i čas (bez vročení).

Poslední sloupec obsahuje jméno souboru. Jméno souboru je následováno znakem, jež symbolizuje typ souboru (přepínač -F). Pokud je implicitně podporováno barevné zvýraznění je základní typ souboru symbolizován i barvou textu (použité barvy lze konfigurovat pomocí souboru DIR_COLORS). Kromě běžných typů souborů jsou speciálně označeny spustitelné soubory (za jménem je hvězdička, standardní barva zelená) a v případě barevného zvýraznění i další typy souborů (archivní soubory aplikace *tar*(1), grafické soubory apod.).

U symbolických odkazů (viz dále) je kromě vlastního odkazu zobrazen i odkazovaný soubor, jehož typ je opět vyjadřen jak připojeným znakem tak případnou odlišnou barvou.

[11] tyto tzv. děravé soubory se nedoporučuje kopírovat pomocí standardního *cp*, neboť při kopírování se kopírují i nealokavané bloky (s hodnotou 0), čímž se výhoda jejich úsporné representace ztrácí. Podivně se mohou chovat i v dalších situacích. Naštěstí jich není mnoho.

kopírování souborů – cp

I když dnes lze v Linuxu kopírovat soubory i pomocí různých souborových managerů, zůstává původní příkaz ***cp***(1) v mnoha případech nejjednodušší a nejrychlejší volbou. Nezastupitelný je především ve skriptech a lze jej volat i z kódu v jiných jazycích (pomocí knihovní funkce *system*(3)). Navíc sdílí základní sadu přepínačů a parametrů s některými dalšími obdobnými příkazy – ***mv***(1) = přesouvání (přejmenování) souborů, ***rm***(1) = výmaz souborů, *ln*(1) vytváření odkazů.

Příkaz cp má dva hlavní tvary. Nejčasteji se používá tvar, jež umožňuje kopírování jednoho nebo vícero souborů do jediného cílového adresáře (bázová jména souborů se nemění):

cp *zdrojový-soubor* [*zdrojové-soubory...*] *cílový-adresář*

Pokud je pořeba měnit i bázové jméno je nutno použít druhého tvaru, jež však umožňuje jen kopírování jediného souboru.

cp *zdrojový-soubor cílový-soubor*

Mezi základní volby příkazu *cp* patří:

přepínač	**význam**
-i	interaktivní tj. bude například vyžadovat potvrzení přepsání souboru
-f	hrubá síla (na nic se neptá a kopíruje)
-r	rekurzivní kopírování (u adresářů kopíruje celý jejich obsah)
-v	upovídané kopírování (vypisuje kopírované soubory a jejich kopie)

Volby -i a -f se přirozeně vylučují, volbu -i (*interactive*) použijte v případě, že chcete detailněji řídit proces kopírování menšího počtu souborů, volba -f (*force*) se používá ve skriptech a při kopírování většího množství souborů, kdy nechcete být zatěžováni velkým množstvím potvrzování. Kopírování hrubou silou však může být i nebezpečné (můžete přepsat i to co jste nechtěli).

Rekurzivní kopírování umožňuje jedním příkazem kopírovat celé podstromy adresářové hierarchie (resp. dokonce celý les podstromů). Při jeho použití lze na místě zdrojových souborů uvádět i adresáře jež se pak na cílové místo určení kopírují i s celým svým obsahem (jinak se adresáře nekopírují vůbec). Rekurzivní zpracování není bohužel podporováno u přesunů příkazem *mv*.

Přepínače lze i kombinovat, například příkaz `rm -rf ~/texty` smaže rekurzívně a bez zbytečného obtěžování operátora celý adresář (a jeho případné podadresáře libovolné úrovně). Bohužel smazané soubory již nelze obnovit (pouze u některých typů souborových systémů existují nízkoúrovňové nástroje, které mohou někdy pomoci[12]).

[12] na druhou stranu lze soubor příkazem *shred*(1) smazat skutečně důkladně

další důležité příkazy

Ostatní příkazy nutné (resp. alespoň užitečné) pro práci se souborovým systémem shrneme jen stručně:

cd *adresář*
adresář se stane novým pracovním adresářem shellu (ostatní procesy tím nejsou ovlivněny)

cd
pracovním adresářem se stane domovský adresář uživatele

pwd
vypsání aktuálního adresáře

mkdir *adresář*
vytváří nový adresář (-p = popřípadě vytváří i mezilehlé adresáře)

stat *soubory...*
detailní informace o souboru (použitím formátovacích řetězců si lze vyžádat konkrétní údaj)

file *soubory...*
pokusí se zjistit konkrétní formát souboru. Každý identifikátor typu obsahuje alespoň jedno klíčové slovo určující obecnější typ. Mezi nejdůležitější klíčová slova patří: *text* (textový soubor), *data* (netextový soubor), *executable* (spustitelný soubor včetně skriptů).

3.4 Pevné a symbolické odkazy

Odkazy (links) jsou prostředkem jak jediný fyzický soubor umístit na vícero míst v adresářové hierarchii. Odkazy (alespoň v podobě pevných odkazů) jsou v Unixu od jeho počátků a jsou využívány na všech úrovních operačního systému (na rozdíl např. od tzv. zástupců v MS Windows, které se objevily relativně pozdě a jejich použití je limitované).

Pevné odkazy

Pevné odkazy úzce souvisejí s mechanismem budování adresářové hierarchie a organizací souborů na svazcích. Každý fyzický soubor je v Unixu representován datovou strukturou, jež se nazývá i-uzel (*i-node*). I-uzly jsou organizovány do tabulek (polí struktur) a index i-uzlu v této tabulce je v rámci svazku unikátním identifikátorem souboru. Tento index (identifikátor) je označován jako i-číslo (*i-number*).

Adresáře jsou pak datovými soubory, které zajišťují překlad absolutních jmen na čísla i-uzlu. Pro tento překlad stačí, aby každý adresář obsahoval tabulku s dvěma hodnotami pro každy soubor resp. podadresář, jenž je v daném adresáři obsažen. Jednou hodnotou je i-číslo fyzického souboru, druhou bázové jméno souboru.

Jeden fyzický soubor však může být pod svým i-číslem uveden v několika různých adresářových tabulkách. U adresářů je to běžné, neboť pomocí vícenásobných odkazů je vybudován hierarchický strom (obousměrně průchozí, neboť odkazy na adresář jsou uvedeny jak v adresáři rodičovském tak i v synovských podadresářích).

Podobný mechanismus lze použít i pro regulární soubory, které se tak mohou zrcadlit na několika místech souborového systému. Jednotlivé výskyty souboru v adresářovém

systému se pak označují jako pevné odkazy (*hard links*)[13]. Všechny pevné odkazy na soubor jsou rovnocenné, plně transparentní a vztah mezi nimi je zcela symetrický. Konkrétně řečeno žadný z odkazů nelze označit jako primární a aplikace ani nemohou zjistit, že na soubor odkazuje více pevných odkazů (pokud by neprohledaly celý adresářový strom a nesrovnávaly použitá i-čísla). Při výmazu souboru se nejdříve mažou pouze odkazy a až teprve po smazání posledního se odstraní jeho fyzická data.

Tato dokonalá transparentnost však může být i na překážku (uživatel může nechtěně pozměnit obsah souboru prostřednictvím jiného pevného odkazu, soubor je dvakrát archivován, po výmazu se nemusí uvolnit prostor na disku, apod.). Navíc pevné odkazy nelze vést mezi svazky (i-čísla jsou jedinečná pouze v rámci svazku!). To je velmi nepříjemné omezení, neboť souborový systém je téměř na všech systémech rozdělen na více svazků a toto rozdělení by mělo být pro běžného uživatele neviditelné (a ani superuživatel by se jím neměl příliš často zabývat).

Oba tyto nedostatky odstraňují tzv. symbolické odkazy, které jsou na Linuxu dostupné od jeho počátků (v Unixu byly poprvé implementovány v polovině osmdesátých let). Z tohoto důvodu se pevné odkazy již téměř nepoužívají (jsou však samozřejmě stále k dispozici).

Symbolické odkazy

Symbolické odkazy (*symbolic links*) jsou speciální datové soubory (v i-uzlu mají speciální příznak), které ve své datové části obsahují absolutní či relativní jméno jiného souboru v adresářové struktuře. Při pokusu o jejich otevření je však na úrovni systémové služby *open*(2) provedeno přesměrování na odkazovaný soubor (to se může i několikrát opakovat, pokud symbolický odkaz míří na další symbolický odkaz atd.). Stejně tak je přesměrování provedeno i u dalších systémových služeb (např. *stat*(2), *chmod*(2), apod.). Existují však i služby, které přistupují přímo k symbolickému odkazu (*readlink*(2), *lstat*(2), apod.) Symbolický odkaz není proto zcela transparentní a programy, které jej potřebují rozlišit (např. archivační, *ls*(1), GUI správci souborů, apod.), tak mohou snadno učinit.

Pro symbolické linky je také typický asymetrický vztah mezi odkazem a odkazovaným souborem. Pokud je smazán odkaz, je vše v pořádku (odkazovaný soubor přirozeně existuje dále), při smazání odkazovaného souboru však vzniká tzv. odkaz-sirotek (*orphans*, *dangling link*). Při pokusu o jeho otevření končí služby s chybou. Tato chyba je však shodná s chybou při pokusu o otevření neexistujícího souboru, což může být matoucí (aplikace hlásí neexistující soubor, i když ten je pro výpisu obsahu adresáře zobrazován).

Symbolické odkazy mohou mířit i na adresáře. Pro většinu aplikací se pak symbolický odkaz chová jako adresář (lze do něj například vstoupit pomocí příkazu *cd*(int)).

Symbolické odkazy však mají i své nedostatky. Zaujímají relativně velké místo na disku (typicky 2-4 KiB) a lze je zacyklit (symbolický link může odkazovat přímo či nepřímo sám na sebe). Odkazy na adresáře narušují stromovou hirerachii adresářů včetně možnosti adresářových cyklů (symbolický odkaz odkazuje na adresář, jenž je předkem adresáře v němž se odkaz nachází). Toto narušení hierarchie musí zohlednit programy,

[13] de facto i jedinečný odkaz na soubor (= absolutní jméno) je pevným odkazem, ale zde může být použití tohoto termínu mírně dezorientující (nikoliv však zavádějící).

jež prochází adresářový strom (například při vyhledávání souborů), nejlépe tím, že symbolické odkazy ignorují.

Problematickou se také může jevit existence dvou typů symbolických odkazů: absolutních (s absolutní cestou) a relativních (s relativní cestou). Rozdíly mezi nimi se projeví při kopírování nebo přesouvání odkazů mezi adresáři. Odkazy s absolutní cestou odkazují po přesunu na stále stejné absolutní jméno souboru a hodí se tedy při odkazech na soubory s pevným umístěním (typicky systémové soubory vně domovského adresáře). Naopak při použití relativní cesty se absolutní jméno odkazovaného souboru po přesunu odkazu mění (je vztaženo k nové pozici odkazu). To však může být výhodou pokus se zároveň s odkazem přesouvá i celá adresářová větev, v níž je jak odkaz tak odkazovaný soubor. To je žádoucí chování pro lokální odkazy uvnitř uzavřené adresářové větve, u niž lze očekávat přesuny např. mezi více instalacemi Linuxu (např. při šíření a instalaci aplikačních balíků pomocí archívních resp. instalačních souborů) nebo jen v rámci jediného systému (uživatelské projekty tvořené vícero soubory).

vytváření odkazů

Pro vytváření odkazů slouží program ***ln***(1). Základní rozhraní tohoto programu odpovídá programu *cp*(1), s výjimkou rekurzivního kopírování (odkazování), jež není k dispozici. Program implicitně vytváří pevné odkazy (důvodem zpětná kompatibilita), symbolické jsou vynuceny přepínačem `-s`.

Základní činnost programu a intrepretace parametrů je stejně jako u kopírování určena typem posledního parametru. V všech případech je však jako poslední parametr uvedeno místo, na němž resp. z něhož bude vytvořen odkaz resp. odkazy.

a) posledním parametrem je (existující) adresář

v adresáři jsou vytvořeny odkazy na všechny soubory určené ostatními parametry (tj. nejdříve se uvádí cíle výsledných odkazů!). Bázové jméno odkazů odpovídá bázovému jménu souborů. Odkazy mohou ukazovat na libovolné soubory včetně adresářů a dalších symbolických odkazů (počet rekurzivních odkazů však může být na aplikační úrovni omezen na řádově jednotky).

Příklad: `ln -s /mnt/cdrom /mnt/floppy ~`

Vytvoří dva symbolické odkazy v domovském adresáři, které odkazují na dva adresáře (přípojné body). Odkazy mají bázová jména `cdrom` a `floppy` (absolutní jména jsou `~/cdrom` a `~/floppy`). Oba odkazy jsou absolutní tj. po svém případném přesunu nebo kopírování budou stále ukazovat na stejné adresáře (přípojném bodě).

Speciálním případem je uvedení symbolického odkazu na adresář na místě posledního parametru. Implicitně je interpretován jako adresář tj. odkaz (odkazy) jsou vytvořeny v odkazovaném adresáři. Při použití volby `-n` (resp. `--no-dereference`) je interpretován jako odkaz a tudíž může být přesměrován (viz bezprostředně následující případ).

b) posledním (druhým) parametrem je existující odkaz

Tento případ vede k chybě „Soubor existuje“. U symbolických odkazů lze však uvedením volby `-f` (spolu s volbou `-s`) provést přesměrování odkazu (tj. existující odkaz začne odkazovat na jiný cíl). Na místě nového cíle (první parametr) je možno uvést jen jediný soubor.

c) posledním (druhým) parametrem je identifikátor neexistujícího souboru

Tímto způsobem lze vytvářet odkazy, jejich bázové jméno se liší od bázového jména cíle odkazu (první parametr).

Při vytváření symbolických linků je ve všech výše uvedených případech nutno zohlednit rozdíl mezi relativními a absolutními linky. Tvorba absolutních linků je jednodušší, stačí pouze důsledně používat absolutní cesty na místě počátečních parametrů (tj. očekávaných cílů odkazů).

Co se však stane pokud na jejich místě použijete relativní cesty? Odpověď je jednoduchá. Veškeré parametry označující cíle odkazů se použijí tak jak jsou uvedeny, tj. jsou beze změny (nepočítaje běžná nahrazení shellu) uloženy do datové části odkazu. Nekontroluje se zda existují soubory daných jmen, ani se neupravují relativní cesty vzhledem k umístění nově vytvářeného odkazu (pokud se liší od aktuálního adresáře shellu v okamžiku volání příkazu *ln*). Důsledkem je skutečnost, že relativní cesty uvedené v počátečních parametrech musí být vztaženy k umístění odkazu a **nikoliv** k aktuálnímu pracovnímu adresáři (jak je tomu u kopírování a spol.). To je značně zavádějící a nepraktické (např. nelze použít automatické doplňnování pomocí klávesy *Tab*) a proto používejte buď jen absolutní cesty, nebo **se před použitím** příkazu **přesuňte do adresáře, v němž chcete odkaz vytvořit**.

Příklad: Například, chcete-li vytvořit relativní odkaz z domovského adresáře na soubor *~/myconf/.emacs* použijete následující sekvenci příkazů:

```
cd
ln -s myconf/.emacs .
```

kde tečka označuje pracovní adresář, jež je zároveň i adresářem nového odkazu (tj. *myconf/.emacs* je správným relativním jménem pro oba adresáře). Pokud by pracovní adresář ležel mimo domovského adresáře, vznikl by v tomto pracovním adresáři odkaz, jenž by však nebyl platný (byl by to hned od svého vzniku sirotek).

Otázkou je, proč byl zvolen relativní odkaz. V adresáři *myconf* shromažďuji konfigurační soubory aplikací, u nichž jsem provedl změny implicitních nastaveních. Při přechodu na jiný systém (jiný počítač nebo novou verzi Linuxu) stačí zkopírovat adresář a všechny odkazy na jeho obsah (jež jsou všechny v domovském adresáři). Protože jsou odkazy relativní (a lokální v podstromu domovského adresáře) jsou funkční bez ohledu na umístění mého domovského adresáře (jež se na různých systémech liší).

3.5 Přístupová práva

Současné Linuxy mohou používat hned několika systémů přístupových práv. Základním však zůstává klasický unixovský systém trojice práv pro tři *ad hoc* kategorie uživatelů (tj. klasická devítice práv).

Do prvé kategorie uživatelů, jimž lze přidělit konkrétní práva k souboru patří pouze uživatel-vlastník souboru (anglicky označován jako *user*, zkratka *u*). Do druhé patří všichni uživatelé patřící do oprávněné skupiny (*group*, *g*), ať již ji mají nastavenu jako primární či nikoliv (primární skupina se uplatní jen při vytváření souborů). Do třetí kategorie patří všichni ostatní uživatelé.

Mezi základní práva, jež lze nastavit patří právo ke čtení (*read*, *r*), zápisu (*write*, *w*) a ke spuštění (*execute*, *x*). Jejich význam je u regulárních souborů zřejmý (jen u práva ke

spuštění je nutno zdůraznit, že právo neimplikuje skutečnou schopnost souboru být přímo spuštěn). Složitější situace však nastává u ostatních základních typů souborů.

U adresářů právo *read* povoluje čtení tabulky souborů tj. možnost výpisu obsahu adresáře. Právo *write* umožňuje změnu tabulky tj. vytváření a výmaz souborů v adresáři (při výmazu souboru není rozhodující právo zápisu u tohoto souboru, ale právo zápisu u jeho adresáře!). Právo *execute* nemá u adresáře smysl (adresář nelze vykonat jako program) a proto bylo využito k jinému účelu. Pokud je nastaveno lze adresář používat v relativních a absolutních cestách, jinak nikoliv (včetně cest, v nichž je adresář zmiňován implicitně). Tj. pokud je nastaveno, lze přistupovat k souborům a podadresářům daného adresáře. V opačném případě je jeho obsah i obsah jeho podadresářů (tj. celý podstrom s kořenem v daném adresáři) nepřístupný (jako by byl odříznut).

Právo *execute* u adresářů úzce souvisí s právem *read,* nejsou však totožná. Rozdíly se projeví, pokud je nastaveno jen jedno z nich. Je-li nastaveno jen právo *read*, lze vypsat obsah adresáře, nelze však otevřít žádný ze souborů v něm uložených (tj. nelze ani vstoupit do žádného z jeho podadresářů resp. vypsat jejich obsah). Nelze dokonce zjistit ani jejich atributy, jako je velikost, vlastník či přístupové časy. Tato konstelace práv však není příliš častá (pro ověření jejích projevů jsem musel vytvořit testovací adresář).

Naopak, je-li u adresáře nastaveno jen právo *execute*, nelze zobrazit jeho obsah, ale pokud známe jméno některého souboru v něm uloženého, lze tento soubor otevřít (např. vypsat jeho obsah)[14]. Podobně lze přetraverzovat do jeho podadresářů s dostatečnými právy, ale jen přímým uvedením jejich jména v cestě (to však téměř brání přístupu k těmto podadresářům v různých GUI souborových manažerech[15], které nezobrazí jejich jména či ikonky).

U symbolických odkazů nehrají práva žádnou roli, neboť při přístupu se testují práva odkazovaných souborů. Formálně však mají symbolické odkazy nastavena všechna práva pro všechny kategorie.

Kromě těchto základních práv mohou mít soubory nastavena i další specifická práva na této úrovni. V Linuxu jsou to následující práva:

propůjčení identity (*set user/group id*, *setuid/setgid*, *s*) – spolu s právem *execute* u regulárních souborů, zajišťují že po spuštění souboru bude efektivní[16] vlastník procesu odpovídat vlastníkovi souboru. I když skutečný vlastník procesu zůstane nezměněn (ten je zděděn z procesu typu *login*) budou práva vstahována k efektivnímu vlastníku (tj. vlastník souboru *de facto* propůjčí procesu svou identitu). Právo lze nastavit v kategorii vlastníka (pak je propůjčena identita vlastníka-uživatele, musí být nastaveno i právo *execute* pro tuto kategorii) nebo v kategorii oprávněné skupiny (je propůjčena identita této skupiny, opět musí být nastaveno právo *execute* pro tuto kategorii). Nejčastějším případem je propůjčení identity superuživatele (*roota*) u programů, které musí být dostupné všem uživatelům (resp. jejich skupině), ale musí kontrolovaně využívat práva superuživatele (typicky např. program *passwd*, jež mění heslo a musí mít tedy přístup k souboru */etc/passwd* resp. */etc/shadow*).

propůjčení skupinového vlastníka adresáře – pokud je u adresáře nastaveno propůjčení identity skupině, jsou všechny soubory vytvořené v adresáři přiděleny oprávněné

[14] musíte však přirozeně mít příslušná práva k tomuto souboru

[15] včetně dialogových okének pro otvírání souborů v GUI aplikacích

[16] efektivní vlastník je používán při ověření práv při přístupu k prostředkům (podle nastavení práv pro jednotlivé kategorie uživatelů)

skupině adresáře (tj. skupinové vlastnictví je převzato z adresáře nikoliv z primární skupiny procesu-tvůrce). To je vhodné pro adresáře, jež obsahují data sdílená členy skupiny.

sticky bit (t) – původně jím byly označovány spustitelné soubory, jejichž data a kód zůstávaly v operační paměti i po ukončení programu (ten pak mohl být rychle restartován). Dnes v době vyspělejších mechanismů správy paměti ztratil význam. Je však používán u adresářů jako tzv. *restricted deletion flag*. Adresáře s tímto příznakem umožňují výmaz souborů jen procesům, jejichž vlastník je shodný s vlastníkem souboru (pokud však má vůbec právo na výmaz). Užíváno u adresářů sdílených všemi uživateli (kteří v něm mají všechna práva), jako je např adresář */tmp*, neboť jen tak lze zabránit vzájemným výmazům souborů.

závazný zámek (*mandatory lock*) – u souboru je nastaveno propůjčení práv skupině bez odpovídajících práva *execute* (tato jinak nesmyslná kombinace byla zvolena z důvodů zpětné kompatibility). Při uzamčení souborů prostřednictvím služby *fcntl*(3) jsou ostatní procesy zablokovány při pokusu o přístup k souborům (detaily viz soubor *mandatory.txt* v dokumentaci jádra systému). V praxi není prakticky využíváno.

příkaz *chmod*

Pro nastavení přístupových práv slouží na úrovni shellu příkaz externí program ***chmod***(1). Základní syntaxe tohoto příkazu je jednoduchá:

chmod [-Rc] *specifikace-práv soubory...*

Přepínač *-c* zajistí vypsání zprávy o nově nastavených právech u každého dotčeného souboru, přepínač *-R* aktivuje rekurzivní nastavení práv v adresářích (parametrem by v tomto případě měl být adresář resp. několik adresářů).

Specifikace práv používá dvojí základní syntaxe — symbolické a oktalové. Symbolická syntaxe je přehlednější a především pružnější. Oktalová je stručnější avšak méně přehledná. Protože se však používá v Unixu na více místech měly by ji znát i středně pokročilí uživatelé.

Symbolická specifikace se skládá z minimálně jednoho specifikátoru ve tvaru:

`[ugoa] [+-=] [rwxst | ugoa]`

kde první skupina znaků určuje kategorie pro něž jsou práva nastavována (*a = all* tj. všechny kategorie zároveň), druhá určuje zda jsou práva přidávána (+), odebírána (-) nebo nastavována (=). Třetí skupina označuje příslušná práva (viz přehled výše), nebo kategorii, od níž jsou práva zkopírována. Ve specifikátorů lze použit označení více kategorií a práv (avšak pouze jeden znak určující funkci), některé kombinace však postrádají význam. Navíc lze použít více různých specifikátorů, kde oddělovačem je čárka. V celé specifikaci nesmí být žádná mezera či jiný bílý znak.

Pronlematiku nejlépe osvětlí pár praktických příkladů:

Příklad: `o=r`
nastaví právo čtení pro kategorii ostatních (uživatelé v této kategorii však nebudou mít žádná další práva). Práva v ostatních kategoriiích se nemění.

`ug+w`
přidá právo zápisu v kategorii vlastník a oprávněná skupina. Ostatní práva a kategorie zůstávají nezměněna.

`a-xw`
ve všech kategoriích je odebráno právo ke spuštění a zápisu (ostatní práva nezměněna)

`go=` nebo `go-rwx`
v kategorii oprávněné skupiny a ostatních ruší všechna práva (kategorie vlastníka není ovlivněna)

`o=g`
ostatní získají stejná práva jako oprávněná skupina (kopírování práv)

`u+s,g+s,o+t,a+x`
nastaví propůjčení identity vlastníka (setuid), ověřené skupiny (setgid) a sticky bit. Nastavení všech těchto speciálních práv najednou je sice možné avšak nepříliš praktické (zde jen pro stručnost). Ve všech kategoriích musí být nastaveno také právo *execute* (proto je použit specifikátor *a+x*)

`u=rw,g=r,o=`
úplné nastavení všech kategorií. Vlastník má práva zápisu a čtení, oprávněná skupina je čtení, ostatní pak nemají práva žádná.

Oktalový specifikátor se sestává z jediné osmičkové číslice, kterou lze obdržet součtem (aritmetickým nebo logickým) jednotlivých bitových příznaků práv (musí být specifikovány všechny kategorie!). Tyto příznaky se opět snadno vyjadřují osmičkově, neboť práva tvoří bitové trojice. Oktalovým specifikátorem lze práva jen nastavovat, nikoliv přidávat nebo odebírat.

Oktalově vyjádřené bitové příznaky jednotlivých práv jsou shrnuty v následujících dvou tabulkách.

běžná práva:

	read	write	execute
user	0400	0200	0100
group	0040	0020	0010
others	0004	0002	0001

speciální práva a atributy:

setuid	4000
setgid	2000
sticky	1000

Například práva vyjádřená poslední symbolickou specifikací v předchozím příkladě (`u=rw,g=r,o=`) lze vyjádřit oktalovou číslicí `0640` = 0400+0200 + 0040.

příkaz umask

Jaká práva však jsou přidělena nově vytvořeným souborům? Při vytváření souborů se práva ve většině případu explicitně nespecifikují (například při ukládání souborů pomocí akce „Save as“ v GUI aplikacích jsou uživatelé dotázáni pouze na jméno souboru). Práva nově vytvářených souborů jsou ovlivněna dvěma hodnotami, za prvé právy jež jsou implicitně vyžadována aplikací, za druhé uživatelskou maskou práv.

Implicitně jsou požadována všechna běžná práva, která mohou mít pro soubor smysl. Pro běžné nespustitelné soubory jsou to práva 0666 (všichni kategorie uživatelů mohou

mít potenciálně práva čtení nebo zápisu). U spustitelných souborů a adresářů přibývá i právo *execute* pro všechny kategorie (tj. jsou požadována všechna běžná práva = 0777).

V případě skutečného uplatnění těchto práv, by se však zcela eminoval zásadní požadavek daný nutností vzájemného oddělení uživatelů — uživatelé by neměly mít možnost vzájemné modifikace souborů, resp. vzájemného vymazávání souborů. Všeobecně nebezpečné je i právo vytváření souborů pro kategorii ostatních (ostatní uživatelé by Váš domovský adresář používaly pro odkládání rozsáhlých souborů). U některých souborů je navíc nutné zajistit důvěrnost uloženého obsahu (na úrovni vlastníka nebo oprávněné skupiny).

Z tohoto důvodu každý proces udržuje tzv. masku uživatelských práv (*user mask*, *umask*). Tato maska je typicky děděna z procesů shellů nebo jiných správců sezení, které nabízí prostředky pro jejich nastavení. Maska definuje práva, které **nesmí** být u nově vytvářených souborů nastaveny (lze je však nastavit explicitním voláním programu *chmod*). Maska tedy určuje jistou část bezpečnostní politiky uživatele. Uživatelskou masku lze nastavit resp. zobrazit pomocí interního příkazu *chmod*. Většinou se tak děje v přihlašovacích skriptech, jež jsou vykonávány při každém přihlášení uživatele a během spouštění jeho přihlašovacího shellu (resp. jiného správce sezení). Některé z těchto skriptů jsou sdíleny všemi uživateli a mohou být editovány jen superuživatelem, což umožní nastavit i univerzální počáteční politiku na úrovni celého systému.

Výsledná práva nových souborů jsou určena pomocí logického součinu požadovaných práv a negace uživatelské masky V zápise programovacího jazyka C: `výsledná-práva = požadovaná-práva & ~umask`.

Nejčastěji se používají uživatelské masky práv uvedené v následující tabulce (druhý a třetí sloupec popisuje výsledná práva pro standardní požadavky):

maska	běžné soubory (požad. 0666)	spust. soubory, adresáře (0777)	komentář
002	0664 `rw- rw- r--`	0775 `rwx rwx r-x`	standardní nastavení u systému bez skut. skupin (např. implicitní RedHat)
022	0644 `rw- rw- r--`	0755 `rwx r-x r-x`	standardní nastavení u systému se skuteč. skupinami
006	0660 `rw- rw- ---`	0771 `rwx rwx --x`	pro uživatele, jež chtějí tajit všechna svá data (vyjma oprávněných skupin)
077	0600 `rw- --- ---`	0700 `rwx --- ---`	můj domovský adresář je můj hrad, má pevnost

Pokud chcete systémové (zděděné) nastavení změnit, lze použít vnitřní příkaz shellu *umask*(int). Parametrem je nová maska buď v symbolickém nebo oktalovém tvaru. Změna se však týká jen shellu a aplikací, které jsou spuštěny po změně masky. Aplikací spuštěných dříve nebo spouštěných mimo shell (například pomocí ikony v GUI prostředích) se změna nedotkne. Proto je vhodné umístit volání příkazu do některého z uživatelských přihlašovacích shellů (nejlépe do *~/.bashrc*).

další přístupová práva a atributy

Další (a mnohem detailnější) přístupová práva lze nastavit pomocí tzv. ACL seznamů, jež jsou však prozatím nejsou plnohodnotně podporovány v distribucích *Fedora Core* (podporovány jsou pouze vlastnosti kompatibilní s původním systémem práv). Podrobnější informace lze získat v manuálových stránkách *acl*(5), *setfacl*(1) resp. *getfacl*(1).

U souborového systému *ext2* (resp. *ext3*) lze nastavit další práva a příznaky pomocí příkazu ***chattr***(1) a vypsat je pomocí ***lsattr***(1). Bohužel jen menší část atributů skutečně funguje (například nefunguje *online* komprese pomocí atributu *c*). V praxi užívám pouze atribut *i* (může jej nastavit jen superuživatel pomocí `chattr +i soubor`), jehož nastavení zabrání smazání souboru. Označuji tak soubory, jejichž náhodné smazání by mohlo být nepříjemné (navzdory občas prováděnému zálohování).

3.6 Prohledávání souborového systému

Souborový systém v běžné úplné instalaci operačního systému Linux obsahuje několik set tisíc souborů (v mé instalaci je to téměř čtyřista tisíc souborů). Nalezení konkrétního souboru resp. všech souborů splňujících jistá kritéria proto patří mezi relativně časté činnosti každého uživatele (a superuživatele obzvláště).

Pro vyhledávání lze použít jednak různé grafické nadstavby (oba hlavní linuxovské desktopy je nabízejí v hlavním menu), ale jejich možnosti jsou dosti omezené (přestože překonávají starší verze MS Windows). Mnohé komplexní dotazy nelze zadat vůbec, jiné jen s vynaložením enormního úsilí. Na druhou stranu KDE aplikace *kfind* podporuje vyhledávání metainformací a podporuje i některé kompromované a strukturované formáty (např. *OpenOffice*). V oblasti vyhledávání multimediálních dat (včetně strukturovaných textů) však Linux poněkud zaostává.

Standardním textovým nástrojem pro prohledávání souborového systému je *find*. Funkce tohoto programu není omezena na pouhé prohledávání, ale lze jej použít i pro provádění hromadných akcí se soubory.

find

Program ***find***(1) prohledává rekurzivně celé větve operačního systému a pro každý soubor vyhodnocuje platnost vyhledávací podmínky. Podmínkou může být buď jednoduchý predikát (= test určité vlastnosti soboru) nebo složitější logický výraz, v němž jsou predikáty spojeny pomocí logických spojek. Pokud soubor podmínku splní, je provedena tzv. akce. Implicitně je to vypsání jména souboru, ale může to být téměř libovolná činnost, jejímž parametrem je nalezený soubor.

Rozhraní programu na příkazové řádce odpovídá výše uvedené funkci. Skládá se ze tří základních částí (všechny jsou nepovinné).

```
find [adresáře] [podmínka] [akce]
```

Prvním krokem je specifikace adresářů, které budou rekurzivně prohledávány (specifikované adresáře jsou kořenovými adresáři prohledávaných podstromů). Nejčastěji se na tomto místě uvádí kořenový adresář (/), domovský adresář (~), resp adresář pracovní (*tečka*). Pokud není stanoven žádný prohledávaný adresář je použit adresář pracovní.

Druhou sekcí je podmínka, kterou musí splňovat soubory, na něž bude aplikována akce (dále jen vybrané soubory). Není-li uvedena, jsou vybrány všechny soubory prohledávaných podstromů.

Jednotlivé jednoduché podmínky (predikáty) mají tvar víceznakových voleb (avšak opatřených jen jedinou pomlčkou) a lze je podle formy rozdělit do tří skupin:

bezparamerické : `-predikát` (například -empty)

parametrické : `-predikát nečíslený-parametr` (například *-user root*)

kvantitativní (trojcestné) : -predikát [+-]číselný-parametr (například *-size +1014k*)

Trojcestné parametry se používají u kvantitavních hodnot (například velikost souboru). Pokud jsou použity s parametrem bez znaménka, budou vybrány jen soubory u nichž nabývá testovaná veličina přesně nabývá dané hodnoty (například velikost je přesně rovna specifikovanému počtu bytů). To však většinou není příliš praktické (např. pro většinu možných velikostí [typicky 0-2GiB] se nepodaří nalézt ani jeden soubor v souborovém systému [statisíce souborů]). Téměř veskrze se proto hodnota opatřuje znaménkem minus (jsou nalezeny všechny soubory s menší hodnotou dané veličiny než je specifikována, např. soubory s menší velikostí) respektive minus (soubory s větší hodnotou dané veličiny).

Přehled nejdůležitějších predikátů přináší následující tabulka.

volba	význam
-type `{bcdpfls}`	typ souboru (jeden znak z množiny), viz podkapitola 3.3 na straně 24
-empty	prázdný
-size `[+-]n[ckMG]`	velikost souboru v bytech(c), KiB (k), MiB (M), GiB (G)
-name `soubor`	soubor má dané jméno, lze použít zástupné znaky souborového nahrazení (viz 2.2), musí však být chráněny před shellem (např. v uvozovkách)
-iname `soubor`	stejně jako výše, avšak nehledí se na velikost písmen
-regex `reg-výraz`	(celé) jméno souboru musí splňovat regulární výraz
-regex `reg-výraz`	stejně jako předchozí, nehledí se na velikost písmen
-user `uživatel`	soubor je vlastněn daným uživatelem
-uid `UID`	soubor je vlastně uživatelem s daným UID
-group `skupina`	soubor je vlastněn danou skupinou
-gid `GID`	soubor je vlastněn skupinou s daným GID
-mtime `[+-]n`	k poslední modifikaci souboru došlo před $24 \times n$ hodinami

-mmin [+-]n	k poslední modifikaci souboru došlo před n minutami
-newer soubor	k poslední modifikaci souboru došlo až po modifikaci souboru, jehož jméno je v parametru (je tedy novější).
-{ac}time [+-]n	podobně jako *-mtime*, ale podle času posledního přístupu (*atime*) nebo změny atributů i-uzlu (*ctime*)
-{ac}min [+-n]	obdoba -mmin pro časy přístupu a změny
-{ac}newer file	obdoba -newer pro časy přístupu a změny
-perm [+-]maska	soubory, jejichž práva přesně odpovídají oktalové masce (bez znaménka), obsahují všechna práva masky (znaménko -) nebo jakékoliv právo z masky (znaménko +)

Pokud uvedete vícero predikátů, jsou implicitně spojeny logickou spojkou „a zároveň“. Pro složitější podmínky lze použít i spojky „nebo“ (volba *-or* resp. *-o*), negace (!) nebo explicitně uvedené spojky a zároveň (*-and* resp. *-a*). Kolem spojek musí být stejně jako u ostatních voleb mezery, zohledňuje se však jejich běžná priorita. Z tohoto důvodu lze v podmínce používat i (oblých) závorek pro změnu pořadí vyhodnocování. Protože však závorky používá i shell, musí být před shellem chráněny (typicky zpětným lomítkem) a jelikož se jedná o parametry příkazu *find* musí být od okolí odděleny mezerami (po obou stranách).

Poslední hlavní skupinu parametrů tvoří akce (tj. jedna nebo vícero akcí). Typově je rozdělit do dvou skupin. Za prvé jsou to vestavěné akce pro výpis informací o souboru.

`-print` – standardní výpis jména souboru (je použit pokud není požadována žádní akce)

`-ls` – detailnější výpis (stejný jako u příkazu *ls -ils*). Výpis začíná i-číslem souboru a jeho skutečnou velikostí (tj. počtem alokovaných bloků o velikosti 1KiB).

`-printf format` – formátovaný výpis jména souboru. Ve formátu lze specifikovat pomocí popisovačů vypisované informace. Přehled popisovačů naleznete v manuálových stránkách *find*(1).

Obecné operace nad vybranými soubory lze provádět pomocí voleb *-exec* a *-ok*. Obě volby mají shodné parametry a liší jen tím, že akce uvozené pomocí *-ok*, vyžadují před svým provedením potvrzení ze strany uživatele (u každého vybraného souboru!).

Akce je u obou voleb specifikována pomocí běžného příkazu. Lze použít libovolného příkazu včetně příkazů s volbami a parametry, přesměrováním apod. Tento vložený příkaz však musí být vždy zakončen středníkem, jenž zároveň ukončuje volbu akce (po středníku lze dále pokračovat ve volbách příkazu *find*, např. lze uvést další akci). Ve vloženém příkazu lze navíc použít speciální dvojznak „{}“, jenž je při každém vyvolání akce substituován za jméno vybraného souboru (tímto způsobem je tedy příkaz parametrizován).

U vloženého příkazu je však nutné chránit některé specialní znaky před shellem (aby nebyly vyhodnoceny resp. subtituovány dříve než jsou předány příkazu *find*). To však

neplatí pro mezery, které musí být ponechány bez ochrany (tj. každý oddělený parametr vnořeného příkazu je zároveň i parametrem příkazu *find*). Jmenovitě musí být chráněn jak dvojznak parametru (zápisem `"{}"` resp. `'{}'` nebo `\"\"`) tak středník na konci vloženého příkazu (nejčastěji zápisem `\;`)[17].

Příklad: Nalezení všech souborů s příponou *cpp* (zdrojových textů jazyka C++) v domovském adresáři (zobrazena jsou absolutní jména souborů):

```
find ~ -name "*.cpp"
```

Nalezení všech prázdných souborů vlastněných uživatelem root v adresáři *etc* a *bin* a jejich podadresářích (vypsána jsou relativní jména vzhledem k pracovnímu adresáři)

```
find etc bin -empty -user root
```

Výpis detailního typu všech souborů v celém souborovém systému, které jsou větší než 10MiB, nebylo k nim přistupováno v posledním týdnu a nejsou vlastněny superuživatelem. Přesné vyjádření časové podmínky: od poslední modifikace souboru (*mtime*) uběhlo více než (+) 7×24 hodin. Pro každý vybraný soubor bude zavolán program *file*.

```
find / -size +10M -mtime +7 ! -user root -exec file '{}' \;
```

Zkopírování určených souborů z pracovního adresáře do zálohy v adresáři /media/dvdram/zaloha. Kopírovány jsou jen neprázdné regulární soubory (nikoliv tedy adresáře, odkazy a speciální soubory) a to jen tehdy pokud k nim nebylo přistupováno více než 30 dnů (u souborů s velikostí větší než 1MiB) nebo 90 dnů (jsou-li menší). Zálohování se však netýká souborů v podadresáři *./dulezite*. Příkaz musí být uveden na jediném řádku.

```
find -empty -a type f -a \( -size +1M -a -atime +30 -o -atime +90 \)
-a ! -regex '\./dulezite/.*' -exec copy '{}' /media/dvdram/zaloha \;
```

Zrušení práva zápisu pro ostatní u všech běžných souborů ve vašem domovském adresáři (a jeho podadresářích atd.), které jsou vlastněny vámi (jinak by se vám to ani nepodařilo) a mají toto právo a zároveň i právo ke spuštění pro ostatní. Soubory, která mají tato práva, představují závažné bezpečnostní riziko, neboť jejich prostřednictvím vám může být podstrčen libovolný kód, který následně můžete aktivovat (i nechtěně). Podstrčený kód pak běží s vaší identitou a s vašimi právy. Pro účely kontroly běhu budou o každém dotčeném souboru vypisovány detailní informace.

```
find ~ -type f -user $USER -perm -003 -ls -exec chmod o-wx '{}' \;
```

Interaktivní smazání všech záložních souborů v adresáři *~/projekty* (tj. každý výmaz bude muset být potvrzen), pokud nebyly aktualizovány resp. vytvořeny v aktuálním sezení. Pro účely tohoto příkazu je nutno najít soubor, jenž je modifikován při každém spuštění login shellu (a v žádném jiném případě). Bohužel soubor splňující obě tyto podmínky nemusí vůbec existovat (a v mé instalaci a nastavení opravdu neexistuje). Jediným dokonalým řešením je vytvoření specializované souboru, jehož jedinou funkcí je udržování časové informace (tzv. *timestamp file*). Pro tyto účely stačí do souboru *.bash_profile* (jenž je spuštěn při každém spuštění login shellu) doplnit řádek `touch .login.timestamp` (jméno souboru je přirozeně volitelné, jen nesmí kolidovat s ostatními soubory). Příkaz ***touch***(1) buď soubor vytvoří (neexistuje-li) resp. změní všechny jeho časy (soubor bude stále prázdný).

```
find ~/projekty -name "*~" ! -newer ~/.login.timestamp -exec rm '{}' \;
```

[17] ochranu nelze spojit tj. například zápis `"{} ;"` není platný

- Zjištění počtu všech spustitelných souborů v adresářích určených proměnnou PATH.

  ```
  find ‘echo $PATH | tr ":" " "‘ -maxdepth 1 -perm +111 | wc -l
  ```

  V příkladu je použita globální volba *-maxdepth*, pomocí níž lze omezit hloubku rekurze.

locate

Vyhledávání pomocí příkazu *find*(1) je velmi pružné, avšak nepříliš rychlé. Prohledávání celého adresářového stromu může trvat i několik několik desítek minut (většinou je však v řádu minut).

Pro rychlejší prohledávání je možné použít předběžné indexování. Klasickým nástrojem pro vytváření indexových (lokačních) databází a jejich dotazování je program ***locate***(1) resp. jeho bezpečnější verze ***slocate***(1). *Slocate* navíc udržuje informaci o přístupových právech a zobrazuje jen soubory, které jsou pro daného uživatele viditelné.

I když lze lokační databáze vytvářet i ručně (nejjednodušeji příkazem *locate -u*, ale jen superuživatel), ale na většině jsou vytvářeny automaticky pomocí mechanismu *cron* resp. *anacron*. U systémů, které jsou spouštěny jen na krátkou dobu (méně než hodina denně), však automaticky vytvářené databáze nemusí odrážet aktuální stav systému (při vyšším stáří databáze je vypisováno varování).

Pro vyhledávání lze použít zápis `locate řetězec` nebo `locate -r základní-regulární-výraz`. V prvém případě jsou vypsány všechny soubory, jejichž jméno obsahuje daný řetězec, v druhém všechny lze jméno popsat regulárním výrazem (i zde stačí jen shoda s libovolným podřetězcem).

Příklad: `locate .txt` – výpis všech souborů s příponou *.txt* (nemusí však být poslední)

`locate -r ".*\.txt$"` – výpis všech souborů jejichž poslední přípona je .txt

- `locate -r "/$USER/.*\.pl$" | xargs head -n 2 >/dev/null` – výpis prvních dvou řádků všech perlovských skriptů v domovském adresáři (adresáře ani soubory nesmí obsahovat mezery)

ÚKOLY

1. Vyzkoušejte si zde uvedené příklady (!).
2. Vytvořte si adresář a vyzkoušejte vliv různě nastavených práv.
3. Navrhněte sekvenci příkazů, která zjistí celkovou velikost všech souborů daného uživatele.
4. Vykoušejte co dělají příkazy *chattr -i, lsattr*

4 Uživatelé, skupiny a jejich správa

Unix je od svých počátků víceuživatelský (*multiuser*) systém. I když dnes je Linux (resp. Unix) často užíván jako systém s jediným uživatelem, například je-li používán jako desktopový systém na osobním počítači resp. jako dedikovaný WWW server, původní víceuživatelský návrh systému zůstává zachován a nelze jej ignorovat. Touto víceuživatelskou podstatou se Unix výrazně liší od např. od MS Windows, které naopak začínaly jako systém jednouživatelský a víceuživatelský režim je pouhou nadstavbou.

Při používání unixovského systému je nutno rozlišovat fyzického uživatele (což je nejčastěji člověk můžu to však být i externí aplikace, tj. aplikace běžící mimo operační systém např. WWW robot) a uživatele jako vlastníka prostředků v rámci OS (počínaje identifikátorem a záznamem v tabulce uživatelů po vlastnictví souborů a procesů). Uživatel-vlastník jako někdy označován jako (uživatelský) účet (konto), ale v Unixu se častěji používá pouze termín uživatel (*user*), neboť u většiny účtů se předpokládá relace 1:1 mezi osobou a účtem. Výjimkou však nejsou situace, kdy jedna osoba vlastní více uživatelských účtů resp. jeden účet užívá více osob (resp. spíše externích aplikací resp. externích[1] uživatelů[2]).

4.1 Root

Nejdůležitějším uživatelem v systému (a jediným, jenž musí být definován) je *root*[3]. Tento uživatel má neomezená (a neomezitelná) práva ke všem prostředkům systému. Rozhodující jsou neomezená práva k souborovému systému (*root* může např. mazat či číst všechny soubory a adresáře), procesům (může ukončit [= zabít] libovolný proces libovolného uživatele) a externím zařízením (včetně sítě). Nepřímo může přistupovat i k adresovým prostorům procesů resp. k heslům uživatelů (zde však musí použít externí program pro luštění [*crackování*]).

Pod uživatelem *root* běží většina systémových programů a démonů (včetně síťových a internetových služeb), neboť jen root má v Unixu dostatečná privilegia (např. může naslouchat na privilegovaných TCP portech). V případě internetových (resp. intranetových) služeb však mohou být neomezená práva na škodu, neboť usnadňují průnik do systému (využitím chybného kódu může získat narušitel plnou kontrolu nad systémem). Proto většina současných síťových démonů programů využívá vlastního (de-

[1] externí uživatelé – uživatelé mimo organizaci provozující systém nebo lokální síť (často jsou de facto neidentifikovatelní)

[2] užívání jednoho účtu více (interními) fyzickými osobami není z hlediska bezpečnostní správy příliš vhodné a mělo by být administrativně-organizačními prostředky vyloučeno.

[3] typické jméno tohoto uživatele, je odvozeno od kořenového (root) adresáře, jenž je kořenem stromové hierarchie adresářů (termín z teorie grafů), což byl původní domovský adresář tohoto uživatele.

dikovaného) a především neprivilegovaného uživatele po většinu svého běhu a s právy roota běži jen při startu a při přístupu k privilegovaným prostředkům.

Privilegovaný účet využívá i fyzická osoba — administrátor systému (i on jen označován jako *root*). I když se většinou předpokládá, že v této roli je zkušený uživatel Linuxu (resp. Unixu), musí tuto roli hrát v jednouživatelském desktopovém systému i začátečník (jinak není schopen systém nakonfigurovat a dlohodoběji spravovat).

Nomezená práva *roota* i když jsou pro správu systému nutná, přinášejí mnohá nebezpečí (především pro začátečníky, ale u zkušení uživatelé mohou být mnohdy nepříjemně překvapeni). Jediný neopatrně použitý příkaz nebo jen pouhý překlep mohou vést k neodvratné destrukci systému (tím nemyslím jen ukončení jeho běhu, ale i nemožnost dalšího nabootování). Nejčastější příčinou bývá výmaz systémových souborů, ale lze nebezpečná může být i pouhá nešťastná modifikace systémových souborů. Navíc příčinou nemusí být jen chyba superuživatele, ale i chyby aplikací (které se u rozsáhlejších či univezálnějších programů vyskytují téměř vždy) resp. dokonce záměrná destruktivní čonnost programu (trójský kůň).

Z tohoto důvodu by měl správce systému, pokud možno co nejvíce omezit své působení s právy roota a to jak v čase tak v použitých prostředcích. Rozhodně by neměl spouštět aplikace, jež nesouvisí se správou (kancelářské aplikace, programátorské nástroje, X-Window) a neověřené programy (například stažené z Internetu) Také by měl s pečlivě kontrolovat zadávané příkazy (a jejich parametry resp. volby). Pokud superuživatelská práva nepotřebuje měl by být přihlášen jako běžný uživatel (nutně např. při přístupu k WWW stránkám, jež mohou obsahovat viry resp. trojské koně).

Pro větší zabezpečení systému většina linuxovských aplikací nedovoluje přímé vzdálené přihlášení superuživatele (tj. *root* se může přihlásit pouze na konzoli) resp. nedovoluje spustit X-Window sezení s právy roota. Obě omezení lze obejít dodatečnou změnou uživatele (vlastníka interaktivního shellu) [viz *su*(1) níže], to však lze bezpečně učinit jen při použití chráněného (šifrovaného) komunikačního kanálu (u některých protokolů včetně oblíbeného telnetu se heslo roota přenáší internetem nešifrované).

Uživatel *root* je v operačním systému interně vždy representován hodnotou 0 (interní číslo uživatele je v Unixu označováno jako UID (*user-id*). Žádný neprivilegovaný uživatel nemůže mít UID rovno 0 (čistě teoreticky lze uživatele s UID rovným nule označit i jiným jménem, ale na jeho neomezených právech se tím nic nemění).

4.2 Běžní uživatelé

Běžně (neprivilegované) uživatele lze v zásadě rozdělit do dvou skupin. Do první skupiny patří tzv. systémový uživatelé tj. uživatelé, které využívají interní systémové procesy resp. jejich prostřednictvím uživatelé externí (včetně externích programů). Hlavním důvodem existence těchto formálních uživatelů je bezpečnost (nemusí běžet s právy roota [viz výše] a navzájem se neovlivňují).

Druhou skupinu tvoří uživatelé spojení s fyzickými osobami. Ve většině linuxovských je jejich UID větší nebo rovni 500. I když čistě teoreticky nemusí v systému existovat žádný účet fyzického uživatele, měl by být již při instalaci vytvořen běžný účet správce systému (u jednouživatelských systémů je to běžný účet příslušného jediného uživatele).

Základní informace o uživatelích systému jsou u jednouživatelských systémů uloženy v textovém souboru */etc/passwd*. To mnohdy platí i pro některé systémy víceuživatelské, u sítí s více počítači však bývají často ukládány centrálně a údaje z nich distribuovány pomocí různých síťových služeb (soubor */etc/passwd* pak obsahuje jen některé systémové uživatele).

Zkrácený obsah souboru */etc-passwd* (u mého počítače) ukazuje následující výpis:

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
...
fiser:x:500:500:Jiri Fiser:/home/fiser:/bin/bash
```

Každý záznam v databázi uživatelů je uložen na jedné řádce a jeho pole jsou oddělena dvojtečkou. V prvním poli je identifikátor uživatele (nejčastěji obsahuje jen malá písmena anglické abecedy resp. tečky). Druhé pole dříve obsahovalo digitální otisk (digest) hesla. Protože však je soubor /etc/passwd veřejně přístupný bylo možno hesla luštit hrubou silou (testováním všech permutací znaků dané délky). Z tohoto důvodů je preferováno uložení otisků hesel v souboru */etc/shadow*[4] (tzv. stínová hesla) k němuž má přístup pouze *root*. Namísto otisku pak */etc/passwd* obsahuje jen znak `x`.

Třetí sloupec obsahuje UID uživatele, čtvrtý identifikátor jeho počáteční primární skupiny (viz níže). Páté pole historicky označované jako GECOS obsahuje volitelný komentář. Obsah tohoto pole není standardizován[5], u fyzických osob zde však bývá skutečné jméno uživatele resp. kontakt na něj (telefon, e-mail apod).

Poslední dvě pole obsahují informace o domovském adresáři uživatele (viz níže) a o jeho implicitním interaktivním shellu (tento shell je spuštěn po jeho přihlášení resp. otevření nového terminálového okna). U fyzických uživatelů v Linuxu to bývá */bin/bash* resp. */bin/tcsh*. U systémových uživatelů to je nejčastěji program */sbin/nologin*, který pouze vypíše chybové hlášení a skončí, neboť interaktivní přihlášení postrádá u většiny systémových uživatelů smysl.

Počet uživatelů v systému není *de iure* omezen (omezením je pouze dostupná paměť a časové nároky při správě jejich databáze).

4.3 Domovský adresář

Domovský adresář určený šestým polem databáze */etc/passwd*, hraje v Unixu velmi důležitou roli. Tento adresář (resp. jeho případné podadresáře) jsou hlavním úložištěm uživatelských souborů. Na většině systémů je to také jediné místo, kde uživatelé mohou vytvářet vlastní adresáře a soubory (nepočítaje veřejně přístupná výměnitelná média). Na druhou stranu k němu mají ostatní uživatelé pouze omezená práva (nejvýše čtení). Přístupu superuživatele nelze sice zabránit, ale ten by měl jistá omezení dodržovat

[4] kromě otisků obsahuje soubor */etc/shadow* i položky související s časovou platností hesla
[5] některé aplikace však očekávají jistou strukturu tohoto pole, například *finger* viz *chfn*(1).

dobrovolně (neměl by například číst data uživatelů resp. vytvářet v jejich adresářích své soubory).

Významné postavení domovského adresáře pro uživatele je ještě zdůrazněno jeho preferencí v některých unixovských aplikacích. Například po přihlášení uživatele se stává počátečním pracovním adresářem shellu, manažeři pracovní plochy v X-Windows v něm vytvářejí svou implicitní plochu resp. vkládají odkaz na tento adresář do plochy apod.

Umístění domovských adresářů v adresářovém stromu není pevné. I když dnes bývají domovské adresáře uživatelů umístěny ve větvi */home* (na nejvyšší úrovni resp. seskupené do skupin v podadresářích), nebylo tomu tak vždy (prvotně byly v adresáři */usr*) a nemusí to platit pro všechny uživatele (domovským adresářem superuživatele bývá nejčastěji */root*).

Pro uživatele shellu (a jiných aplikací) může proměnlivost umístění domovských adresářů přinášet problémy. Jak například napsat přenostitelný skript pracující s obsahem domácího adresáře?

Jedním z řešení je použití příslušného pole v */etc/passwd*. To je však příliš složité a pomalé. Druhým řešením dostupným již od počátků Unixu je použití proměnné $HOME, jež se rozvíjí na absolutní jméno domovského adresáře aktuálního uživatele (tj. uživatele, jež vlastní shell v němž je proměnná použita). Nejjednodušší je však použití speciálního nahrazování pomocí znaku vlnka (tilda), jež je dostupný ve všech moderních Unixech (včetně Linuxu).

Nahrazení vlnky

Znak vlnka použitý v příkazu shellu se rozvíjí na absolutní jméno domovského adresáře aktuálního uživatele (tj. uživatel jež vlastní proces shellu). Avšak pokud za vlnkou následuje písmenný znak, pak se slovo za vlnkou interpretuje jako uživatelské jméno a existuje-li takový uživatel, je výsledkem rozvoje absolutní cesta k jeho domovskému adresáři (neexistuje-li, pak se dané nahrazení neprovede).

Vlnkové nahrazení navíc podporuje i tzv adresářový zásobník (directory stack), který může výrazně usnadnit přechody mezi adresáři (detaily viz *pushd*(int), *popd*(int), *dirs*(int), *bash*(1)).

4.4 Uživatelé a procesy

Při bootování (natahování) systému existuje v systému z počátku jen jediný proces[6] označovaný jako *init*(8), jehož vlastníkem je přirozeně uživatel *root*. Tento vlastník je uveden u daného procesu v tabulce procesů a určuje přístup (resp. práva) uživatele k prostředkům operačného systému. Proces *init* postupně vytváří procesy systémových a internetových služeb, které alespoň běží jako procesy (roota). Jak bylo řečeno výše mohou dočasně běžet pod jiným systémovým uživatelem (z bezpečnostních důvodů při přístupu k neprivilegovaným prostředkům).

[6] proces je instance programu (resp. aplikace)

Pro spojení procesů s fyzickými uživateli měly v původním Unixu rozhodující roli procesy *getty*/*login*. Proces *getty*[7] (vlastněný stále rootem) je v Unixu vytvářen pro každý textový terminál v systému (procesem *init*). Po aktivaci daného terminálu program *getty* vypíše přihlašovací prompt a čeká na zadání identifikátoru uživatele.

Poté co je zadáno jméno, začne původní proces běžet podle jiného programu, jímž je program *login*. Ten si vyžádá heslo uživatele a a jeho základě jej autentifikuje. Pokud se autentifikace úspěšně provede (tj. bylo zadáno heslo příslušné danému uživatelskému účtu[8]), mění proces *getty*/*login* svého vlastníka podle přihlašovacího jména a začíná běžet podle programu příslušného shellu (viz poslední pole v */etc/passwd*). Všechny programy spouštěné ze shellu pak dědí vlastníka, čímž je uživatelská identita zachována po celou dobu sezení. Podle této identity se ověřují přístupová práva k souborům a ostatním prostředkům operačního systému.

V modernějších Unixech (včetně Linuxu) zůstává v zásadě tento mechanismus identifikace uživatelů s procesy zachován, pouze je rošířeno spektrum programů, jež zajišťují inicializaci identifikaci uživatelů (obdoby *getty*) resp. jejich autentifikaci (obdoby *login*). Obdobou *getty* jsou např. procesy vyvolávané servery internetových služeb vzdáleného přihlašování *telnet* resp. *ssh* (*ssh-agent*(1)) nebo správce sezení X-Window (*xdm*(1), *gdm*). Zatímco např. telnet využívá pro autentifikaci standardní program *login*, a správce sezení *xdm* jeho grafickou obdobu *xlogin*, může *ssh-agent* užívaný u služby *ssh* užívat i rozmanitější autentifikační schémata jako je asynchronní šifra nebo hesla na jedno použití.

4.5 Skupiny

V Linuxu má každý systémový prostředek právě jediného vlastníka. Vlastník má *de facto* k prostředku všechna práva (i když mohou být aktuálně omezena, může si je uživatel vždy přidělit), ostatním uživatelům musí práva propůjčit práva vlastník (některá práva však přirozeně propůjčit, nejde např. propůjčit právo změny práv). Tato práva však v Unixu nejde propůjčit jednotlivým (ostatním) uživatelům, ale v zásadě pouze všem ostatním uživatelům jako celku. Toto hrubé rozdělení světa na já a všichni ostatní ve většině případů stačí (např. u soukromých dokumentů uživatelů nebo u většiny systémových souborů), ne však vždy. Výjimkou jsou prostředky dostupné jen omezené skupině uživatelů (např. tiskárna nebo WWW připojení) resp. sdílené skupinou uživatelů (např. dokumenty při týmové spolupráci).

Pro tyto případy umožnuje Linux definovat tzv. skupiny uživatelů. Každý uživatel může být členem libovolného počtu skupin (vždy však alespoň jedné) a každá skupina může obsahovat libovolný počet členů (vždy však alespoň jednoho).

Navíc má každý prostředek (typicky soubor) kromě uživatele-vlastníka přidělenu právě jednu tzv. oprávněnou skupinu. Pro tuto skupinu (tj. pro všechny její členy-uživatele) lze definovat specifická přístupová práva (tzv. skupinová) odlišná od práv vlastníka i práv ostatních uživatelů.

Je-li uživatel členem jen jedné skupiny je zřejmé, že oprávněnou skupinou se u nově vytvářených prostředků stane tato skupina. Co se však stane pokud je uživatel členem

[7] v Linuxu se pro virtuální konzole používá minimální implementace *mingetty(8)*, jež neobsahuje řízení sériového portu.

[8] tím však samozřejmě není ověřena skutečná identita uživatele

více skupin? Z tohoto důvodu má každý uživatel přidělenu tzv. primární skupinu, jež se explicitně stává oprávněnou skupinou u nově vytvářených prostředků (souborů). Při kontrole přístupových práv se však zohledňují všechny skupiny, jejímž je vlastník procesu členem (nikoliv jen primární). Primární skupina je uživateli přidělena při přihlášení (viz čtvrté pole */etc/passwd*). Primární skupinu lze změnit příkazem ***newgrp***(1), přičemž uživatel musí být přirozeně členem této skupiny[9].

Informace o skupinách a jejích členech jsou uloženy v souboru */etc/group*.

Skupiny u desktopových instalací Linuxu nehrají příliš velkou roli. Po instalaci jsou většinou nastaveny pouze tzv. pseudoskupiny, které existují pouze z formálního důvodů, neboť každý uživatel resp. každý prostředek musí mít nastavenu primární resp. oprávněnou skupinu. Buďto je vytvořena jediná skupina obsahující všechny uživatele (skupinová práva splývají s právy pro ostatní) nebo je pro každého uživatele vytvořena jednočlenná skupina, jejíž jméno i mnohdy i identifikační číslo (tzv. GID) je shodné s identifikací uživatele (zde naopak skupinová práva splývají s právy uživatele-vlastníka). Ostatní skupiny se vytvářejí až podle požadavků (a pouze u skutečně víceuživatelských systémů).

Informace o vlastní identitě včetně členství ve skupinách lze získat pomocí příkazu ***id***(1). Výpis z mého domácího počítače ukazuje, že jsou použity jednočlenné pseudoskupiny (hodnota *gid* označuje primární skupinu).

```
uid=500(fiser) gid=500(fiser) skupiny=500(fiser)
```

4.6 Zobrazování informací o uživatelích

Unix je víceuživatelský systém a tak již od počátků nabízel jednoduché nástroje pro zobrazování informací o uživatelích.

Začněme programy, které zobrazují informace o uživatelích, kteří jsou právě přihlášeni. Nejjednodušší je program ***w***(1). Stejně jako ostatní programy nezobrazuje pouze jména přihlášených uživatelů, ale i další užitečné informace. V tomto případě je to terminál přihlašovacího shellu (TTY), označení počítače z něhož jsou přihlášeni (jen u vzdálených přihlášení), čas přihlášení (LOGIN@), čas od poslední zaznamenané aktivity uživatele (IDLE) a dva časy (JCPU je čas skutečného běhu všech existujících procesů na dané konzoli, PCPU je čas běhu procesu z posledního sloupce). Všechny časy jsou není-li určeno jinak v minutách. Poslední sloupec ukazuje jméno aktuálně prováděného příkazu (běžícího na popředí).

Příkaz *w* vznikl v dobách, kdy všechny programy byly textové a uživatelé se systémem komunikovaly převážně jen pomocí shellu. Proto program zobrazoval uživatele podle záznamů, jež vytvářejí procesy *getty* a *login* v souboru */var/run/utmp*(5) a preferuje procesy běžící na popředí (tj. ty které používají terminál pro svůj interaktivní vstup a výstup).

V současnosti, kdy většina uživatelů používá graficky orientovaných aplikací na platformě X-Window mohou být některé údaje zavádějící a jiné neúplné. Za prvé je zcela dobrovolné zda se proces s funkcí původního přihlašovacího shellu (správce interaktivního terminálu) zaznamená svou existenci do */var/run/utmp*(5), což je soubor z něhož

[9] to není zcela pravda, neboť původní Unix umožňoval přihlášení i do cizí skupiny (přes heslo). Dnes se však tato možnost již nepovoluje.

program *w* další čerpají informace. U některých programů (jako jsou virtuální terminály typu *xterm*) lze zápisy do */var/run/utmp* konfigurovat.

Důsledkem je chování, kdy některé relevantní programy nejsou vypisovány (ve výjimečném případě nemusí být vypisována žádná informace o aktivním uživateli), respektive jsou naopak vypisovány neinteraktivní shelly (například shelly, jejichž výstup uživatel vůbec nevidí) apod.

Sníženou informační hodnotu má v dnešní době i pole s aktuálním procesem resp. čas zahálky (*idle time*), neboť obě hodnoty nezohledňují procesy na pozadí, mezi něž patří i GUI programy spouštěné pomocí ikon nebo ze shellu prostřednictvím ampersandu.

Alternativním programem zobrazujícím přihlášené uživatele je ***who***(1). Zdroj i zobrazené informace jsou obdobné programu *w* (pouze o něco stručnější a jinak formátované).

Operační systém si v souboru zaznamenává */var/log/wtmp*(5) i starší přihlášení a odhlášení. Pro zobrazení obsahu tohoto souboru slouží aplikace ***last****(1)*. Pokud ji spustíte bez parametrů, zobrazí informace o přihlášení a odhlášení všech uživatelů včetně záznamů u bootování (zavádění) systémů a celkové době běhu systému (pod pseudouživatelským jménem *reboot*). Na konci výpisu se zobrazí datum, od kterého jsou data dostupná (soubor se cyklicky maže pomocí mechanismu *logrotate*(8)).

Jestliže chcete získat informace o jen o jediném uživateli (např. o sobě) resp. několika vybraných uživatelích stačí zadat jejich jména jako parametry příkazové řádky. Pro zjištění údajů o bootování a dobách běhu systému použijte pseudouživatele *reboot*.

Pokud chcete údaje o dobách přihlášení v jednotlivých uživatelů vidět sumárně, použijte program ***ac***(1) (doporučuji vyzkoušet přepínač *-d*). Bohužel tento program není příliš inteligentní. Nedokáže například počítat se skutečnými uživateli, kteří mají najednou otevřeno více terminálů, a proto není divu, že pro některé dny mi vrací až 50 hodin celkového času přihlášení (a pak že den má jen 24 hodin, ještě by to chtělo umět pracovat ve více terminálech současně).

ÚKOLY

1. Vyzkoušejte si zde uvedené příklady (!).
2. Vyzkoušejte si do systému přidat dalšího uživatele příkazy *adduser, useradd* a modifikovat jej *usermod.*
3. Vyzkoušejte si změnu hesla pomocí *passwd.*
4. Vyzkoušejte si přidání skupiny pomocí *groupadd.*
5. Co dělá příkaz *groups*?
6. Změťe vlastníka, případně skupinu vybraného souboru (adresáře) pomocí *chown* a *chgrp.*
7. Odstraňte nově vytvořené uživatele a skupiny- *deluser, delgroup.*

5 Procesy a jejich řízení

Každý program se po spuštění stává procesem. Procesu je v krátkých časových kvantech přidělován procesor a tak postupně vykonává instrukce programu. Nakonec je proces dobrovolně či nedobrovolně ukončen, čímž končí i vykonávání dané instance programu. Linux je preemptivní multitáskový systém, v němž může najednou existovat více procesů, které běží nazávisle na sobě (a pokud má systém i více procesorů může být dokonce vykonávány paralelně).

Každý proces v systému je identifikován jedinečným číslem, jež je označováno jako PID. Navíc každý proces vlastní soukromé prostředky (resp. sdílí prostředky s jinými procesy) minimálně je to virtuální adresový prostor.

5.1 příkaz *ps*

Základní pohled na aktuální stav procesů v systému poskytuje program ***ps***(1). Pokud však aktivujete tento program bez voleb a parametrů, zobrazí jen minimum informací. Zobrazuje totiž jen údaje o procesech svázaných s terminálem daného shellu (typicky jsou to jen dva procesy — vlastní shell a příkaz *ps*), a pro každý proces jen ty nejdůležitější informace (zleva v pořadí PID, připojený terminál, čas běhu a program, podle něhož běží).

Podrobnější informace o všech procesech v systému lze získat použitím voleb *-e* (všechny procesy) a *-l* (dlouhý formát výpisu). Výstup programu `ps -el` na mém počítači v okamžiku psaní tohoto textu obsahoval údaje o téměř osmdesáti procesech (u počítačů připojených na Internet není výjimkou ani počet přesahující stovku). Z tohoto důvodu uvádím pro ukázku zkrácený výpis.

```
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0     1     0  0  76   0 -   817 -      ?        00:00:01 init
1 S     0     2     1  0  94  19 -     0 -      ?        00:00:00 ksoftirqd/0
1 S     0     3     1  0  65 -10 -     0 -      ?        00:00:00 events/0
1 S     0     5     3  0  75 -10 -     0 -      ?        00:00:00 kacpid
1 S     0    18     3  0  65 -10 -     0 -      ?        00:00:00 kblockd/0
1 S     0    28     3  0  80   0 -     0 -      ?        00:00:00 pdflush
1 S     0    31     3  0  69 -10 -     0 -      ?        00:00:00 aio/0
1 S     0    30     1  0  75   0 -     0 -      ?        00:00:00 kswapd0
1 S     0   104     1  0  85   0 -     0 -      ?        00:00:00 kseriod
0 S     0  1204     1  0  66 -10 -   772 -      ?        00:00:00 udevd
1 S     0  1506     1  0  75   0 -     0 -      ?        00:00:00 usb-storage
1 S     0  1762     1  0  79   0 -     0 -      ?        00:00:00 kjournald
5 S     0  1989     1  0  76   0 -   690 -      ?        00:00:00 syslogd
5 S     0  1993     1  0  76   0 -   748 -      ?        00:00:00 klogd
```

```
16 1 S     0  2063     1  0  81   0 -   781 -      ?        00:00:00 acpid
17 5 S     0  2079     1  0  83   0 -  1181 -      ?        00:00:00 sshd
18 1 S     0  2090     1  0  77   0 -   637 -      ?        00:00:00 xinetd
19 5 R     0  2121     1  0  76   0 -   574 -      ?        00:00:00 gpm
20 1 S     0  2162     1  0  76   0 -  1337 -      ?        00:00:00 crond
21 5 S    43  2203     1  0  76   0 -  1697 -      ?        00:00:00 xfs
22 1 S     2  2222     1  0  76   0 -   838 -      ?        00:00:00 atd
23 4 S     0  2260     1  0  78   0 -   601 -      tty1     00:00:00 mingetty
24 4 S     0  2326     1  0  76   0 -  2980 -      ?        00:00:00 gdm-binary
25 4 R     0  2694  2687  2  76   0 - 34601 -      ?        00:00:49 X
26 4 S   500  2926  2687  0  75   0 -  5668 -      ?        00:00:01 gnome-session
27 1 S   500  2958     1  0  78   0 -  1280 -      ?        00:00:00 ssh-agent
28 1 Z   500  2983  2926  0  78   0 -     0 exit   ?        00:00:00 bash <defunct>
29 0 S   500  3004     1  0  76   0 -  3043 -      ?        00:00:02 gconfd-2
30 1 S   500  3018     1  0  79   0 -   777 -      ?        00:00:00 gnome-keyring-d
31 0 S   500  3020     1  0  76   0 -  1968 -      ?        00:00:00 bonobo-activati
32 0 S   500  3039     1  0  75   0 -  1444 -      ?        00:00:00 xscreensaver
33 0 S   500  3064     1  0  75   0 -  3544 -      ?        00:00:02 metacity
34 0 S   500  3069     1  0  75   0 -  6234 -      ?        00:00:02 gnome-panel
35 0 S   500  3071     1  0  75   0 - 10157 -      ?        00:00:02 nautilus
36 0 S   500  3085     1  0  76   0 -  5436 -      ?        00:00:00 gnome-vfs-daemo
37 0 S   500  3096     1  0  76   0 -  5440 -      ?        00:00:00 drivemount_appl
38 0 S   500  3098     1  0  76   0 -  5667 -      ?        00:00:00 clock-applet
39 0 S   500  3108     1  0  75   0 - 12755 -      ?        00:00:06 gnome-terminal
40 0 S   500  3110  3108  0  76   0 -  1249 wait   pts/0    00:00:00 bash
41 0 S   500  3256     1  1  75   0 -  7168 -      ?        00:00:20 lyx
42 0 R   500  3317  3110  0  77   0 -   954 -      pts/0    00:00:00 ps
```

Nejdříve si ozřejmíme význam jednotlivých sloupců výpisu. Formát a detailní význam výpisu se občas mírně mění (především při přechodu na novou minor verzi jádra). Navíc se může lišit u různých implementací programu *ps*.

F	flag	kumulativní příznaky procesu (4 = běží se superuživatelskými právy)
S	state	stav procesu (uvedeny jen běžné): S – spící (čeká na událost) R – běžící [RUNNING] nebo čekající na procesor [WAITING] Z – zombie (mátoha), ukončený avšak ještě neodstraněný
UID	user ID	UID vlastníka procesu (0=root)
PID	process ID	číselný identifikátor procesu
PPID	parent PID	číselný identifikátor rodičovského procesu
C	process utilization	procento využití procesoru
PRI	priority	aktuální priorita procesu (protože se stále mění, nepříliš užitečný údaj)
NI	nice	statická pririta procesu (určuje podíl kvant procesoru). Vysoké priority (vysoký podíl) jsou označeny záporným číslem, standardní nulou, nízké (při zatížení systému běží pomaleji) kladným číslem (absolutně nejnižší je priorita 19)

ADDR		prázdné pole, jehož funkci neznám
SZ	size	velikost procesu. Je to velikost dat, jež by bylo nutno uložit do odkládacího prostoru, při úplném odswapování procesu. De facto jsou to uživatelská data a zásobník (nikoliv například kód a sdílené knihovny).
WCHAN		jméno rutiny, v níž je proces zablokován (jen u spících procesů). Typicky *wait* pokud proces čeká na potomka nebo thread.
TTY	terminal	označení terminálu (pseudoterminálu) s nímž je proces spojen. Některé procesy s treminálem nepracují (démoni a některé GUI aplikace), u nich je uveden otazník.
TIME		skutečný čas běhu procesu (je jen zlomkem času uplynulého od spuštění). Velké časy (desítky minut nebo hodiny) signalizují nekonečný cyklus v programu.
CMD		příkaz, jímž byl proces spuštěn (nultý argument příkazového řádku)

Další informace o procesech lze získat použitím uživatelských formátovacích voleb. Uživatelské formátovací volby se píší jako parametr volby *-o* a skládají se z výčtu jmen sloupců oddělených čárkami.

Jako jména sloupců lze kromě jmen uvedených v předchozí tabulce (první sloupec) uvést i velký počet dalších (viz manuálové stránky *ps*(1)). V následující tabulce uvádím jen ty nejpraktičtější:

%cpu		procentuální využití procesoru. Údaj je přibližný a je vhodný především pro relativní srovnání.
%mem		procentuální použití rezidentní (fyzické) paměti. I tento údaj je přibližný a tudíž je vhodný především pro relativní srovnání.
etime	elapsed time	čas uplynulý od spuštění procesu
start	start time	čas (datum) spuštění procesu
ruser	real user	skutečný uživatel-vlastník souboru
euser	efective user	efektivní uživatel souboru (viz propůjčení identity)
nlwp		počet threadů (vláken) v procesu
psr		číslo procesoru na němž aktuálně proces běží (použitelné u víceprocesorových systémů)
vsize	virtual memory size	velikost virtuální paměti procesu
rss		velikost rezidentní paměti procesu (část procesu umístěná ve fyzické paměti)

cmd	command	celý příkazový řádek s nímž byl proces spuštěn (tj. včetně cesty, voleb a parametrů). Může být zkráceno při zarovnání na šířku terminálu.
comm	command	jméno pod nímž byl proces spuštěn (bez voleb a parametrů)

Všechny výše uvedené údaje o procesech lze zjistit jediným příkazem:
`ps -eo %cpu,%mem,etime,ruser,euser,nlwp,psr,start,vsize,rss,cmd`.

Nyní se pokusíme poněkud zorientovat v jednotlivých procesech podle zkráceného výpisu uvedeného na straně 47 (váš výpis bude samozřejmě poněkud jiný, základní organizace však bude zachována).

Základním východiskem při analýze výpisu je jeho setřídění podle PID. Identifikační čísla se procesům přidělují postupně vždy o jedničku vyšší[1]. Výpis je tak seřazen i podle stáří jednotlivých procesů, přičemž ty spuštěné ihned po zavedení systému jsou na začátku výpisu.

Prvním procesem je proces *init*. Jeho základní funkce byla popsána v kapitole 4.4 (strana 43). Tento proces nejdříve vytváří tzv. jaderné démony (*kernel daemons*), což jsou procesy zajišťující základní funkce systému (jejich běh je nezbytný). Ve výpisu jsou na řádcích 2-16 (i samotný *init* je jaderný démon). Pozornost se zaslouží především zloděj stránek (*kswapd*) na řádku 9 nebo démon pro správu USB zařízení třídy „storage medium“ na řádku 12 nebo logovací démon jádra *klogd* (řádek 15).

Poté systém přechází do jednouživatelského režimu a *init* aktivuje běžné démony (řádky 17-22). Nejčastěji jsou to démoni síťoví (zde jen *sshd* a *xinetd*, neboť nejsem připojen do internetu). S démony se detailněji seznámíme v kapitole8.

Poté jsou spuštěni správci přihlášení (řádky 23,24). Kromě klasického textového *getty* je to grafický správce sezení poskytovaný prostředím *Gnome* (*gdm*). Bezprostředně poté je spuštěn X-server (řádek 25) a po přihlášení i správce (konkrétního) gnome-sezení (spravuje *Gnome* destop jako celek).

Následuje spuštění pomocných aplikací užívaných gnome-desktopem, jež všechny běží na pozadí a nemají ani přidělena GUI okna nebo jiný vizuální prvek (to bohužel z výpisu zjistit nejde). Tato sekce počíná na řádku 27 spuštěním *ssh-agenta* (správce soukromých a veřejných klíčů, poskytuje je ostatním aplikacím) a končí aktivací serveru *bonobo* (obdoba COM technologie Microsoftu) na řádku 31. Zajímavý je výskyt zombie shellu, který se po provedení své činnosti (tu nelze detailněji zjistit) ukončil, nebyl však svým rodičem (*gnome-session*) odstraněn z tabulky procesů. To je chyba, která se však naštěstí neprojevuje negativně na běhu systému (proces jen zbytečně zabírá místo v tabulce procesů, jejíž velikost je omezená).

Poté jsou spuštěny pomocné procesy gnome desktopu, které se již vizuálně projevují. Počínají spuštěním šetřiče obrazovky (řádek 32) a končí spuštěním appletu, jenž zobrazuje čas na některém z panelů (ř. 38). Nejdůležitějším procesem této sekce je tzv. *window manager* (správce oken, zobrazuje dekoraci kolem oken, včetně titulního pruhu). Zde je to program aplikace metacity (ř. 33). Program „Nautilus“ je správcem souborů a kořenového okna (plochy), tj. odpovídá aplikaci Explorer v MS Windows. Je zajímavé, že většina aplikací v této sekci (a sekci předchozí) je potomkem procesu *init*. Tento proces

[1] to však nemusí platit u dlouho běžících systémů (zde se mohou PID překlopit)

však tyto procesy ve skutečnosti nevytvořil, neboť je pouze rodičem adoptivním. Převzal rodičovské povinnosti od původních rodičovských procesů, které již byly ukončeny.

Všechny výše uvedené procesy byly spuštěny automaticky při startu systému. Já osobně jsem spustil jen čtyři procesy na konci seznamu. Nejdříve to byl emulátor terminálu (*gnome-terminal*), ve kterém se spustil shell (*bash*). Pak jsem spustil editor v němž pořizuji tento text (*lyx*). Lze poznat, že jsem jej nespustil příkazem ze shellu (jinak by byl shell jeho rodičem a využíval by stejný terminál)[2]. Nakonec jsem pomocí shellu aktivoval příkaz *ps*.

5.2 Dynamické sledování procesů

Příkaz *ps*(1) není příliš vhodný pro sledování dynamických změn v systému procesů, především proměnlivé míry využití systémových prostředků jednotlivými procesy.

Klasickou aplikací pro sledování procesů už z dob klasického Unixu je program ***top***(1). Tento program se nicméně dále vyvíjí a v nejnovějších reinkarnacích má několik desítek voleb a interaktivních příkazů a je do značné míry konfigurovatelný.

Po spuštění zaujme *top* celou plochu terminálu. V horní části zobrazuje globální infomace o systému. Většina informací z této části je dostupná pomocí jednoúčelových příkazů avšak jen ve statické podobě.

První řádek kromě aktuálního času a času od posledního zavedení systému zobrazuje jeho průměrné zatížení (load average) v poslední minutě, 5 a 15 minutách (statický příkaz ***uptime***(1), nebo první řádek *w*(1)).

Průměrné zatížení odpovídá průměrnému počtu procesů, jež čekají ve stavu WAITING (resp. READY) na procesor[3]. V tomto stavu jsou procesy, které jsou připraveny k běhu (tj. nečekají na jinou událost), ale nemohou protože procesor je využíván jiným procesem. Hodnota blízká nule tudíž signalizuje nízké zatížení procesoru (typické pro desktopové počítače užívané k pořizování textů apod.). Hodnota blízká jedné odpovídá plnému zatížení procesoru (průměrně jeden proces musí stále čekat na procesor). Hodnoty mohou být ještě vyšší, lze je však jen obtížně kvalifikovat. Lze však říci, že řádově odpovídají optimistickému odhadu faktoru zpomalení běhu aplikací (tj. hodnota 2 odpovídá dvojnásobnému zpomalení spíše však mírně vyššímu).

Další dva řádky sumarizují procesy podle jejich základních stavů a využití procesoru. Procerový čas je rozdělen mezi běžící procesy a tzv. *idle* proces. Idle proces udržuje procesor v běhu i při nižších zatíženích (jen malá část procesorů může přecházet do úsporných režimů). Běžící procesy pak využívají procesor ve dvou základních režimech, uživatelském a systémovém (privilegovaném, jaderném). V systémovém režimu přistupují k zařízením a využívají prostředků systému (typicky především souborový systém) včetně režie při přístupu k paměti a při přepínání procesů, v uživatelském vykonávají aplikační programy a provádějí grafické operace v X-Window.

Zkratka *us* označuje procentuální podíl strávený běžnými procesy v uživatelském režimu, *sy* v režimu systémovém. Zvláště je počítán *idle* proces (zkratka *id*) a procesy s vyšší statickou prioritou (zkratka *ni*).

[2] nezbývá než přiznat, že jsem použil ikonu na panelu. Ano, nejsem linuxovský guru, jež vše spouští a ovládá se shellu.

[3] u víceprocesorových systémů je situace poněkud složitější, ale základní charakteristiky odpovídají.

Poslední část hlavičky zaujímají souhrnné informace o paměti, jež jsou ve statické podobě dostupné pomocí příkazu ***free***(1). Jejich struktura je zřejmá, první řádek popisuje obsazení rezidentní (operační paměti spravované správcem paměti), druhá odkládacího prostoru. Údaje v posledním sloupci ukazují velikost dvou speciálních oblastí — systémových bufferů a vyrovnávacích pamětí blokových zařízení (obě oblasti jsou v operační paměti).

Druhá část terminálu zobrazuje základní informace o procesech. Bohužel počet procesů dnes již přesahuje dostupný počet řádků na valné většině terminálů a tak jsou zobrazeny jen některé. Naštěstí jsou implicitně řazeny podle použití procesoru a tak jsou ty nejaktivnější vždy na vrcholu výpisu. Význam (a částečně i popisky) odpovídají programu *ps*.

Kromě pasivního sledování tepu systému lze s programem i komunikovat pomocí stisků některých písmenných kláves. Souhrn těch nejpraktičtějších přináší následující přehled (hvězdičkou jsou označeny přepínače, které po druhém stisku ruší funkci prvního).

q	ukončení aplikace
i*	zobrazí jen běžící procesy
b*	zvýrazní důležité údaje (např. běžící procesy)
S*	zapíná kumulativní mód (viz dále)
f	umožní měnit zobrazené sloupce
M	seřadí procesy podle obsazení rezidentní paměti
T	seřadí procesy podle času běhu
P	seřadí procesy podle času běhu (implicit. nastavení)
k	umožní zabíjet neposlušné procesy

Kumulativní mód zobrazuje u procesů čas běhu se započtením mrtvých potomků. To je vhodné u programů, které postupně vytvářejí podřízené procesy, které za ně vykonávají dílčí úkony (a poté zanikají). Typickým zástupcem těchto programů jsou kompilátory jazyků.

Kromě programu top, lze použít i jeho GUI následovníků. Ty mají většinou lepší rozhraní a umožňují zobrazit více údajů (především globálních). Prostředí KDE nabízí program *ksysguard*, GNOME pak *gnome-system-monitor*. Pro ty, kteří jsou zvyklí stále sledovat parametry systému neustále, lze doporučit aplikaci *gkrellm*.

5.3 Aktivace procesů

Základní mechanismus spouštění procesů prostřednictvím shellu již znáte. Stačí uvést jméno programu, případné parametry a volby a stisknout klávesu *Enter*. Nyní se však na tento proces podíváme detailněji a hlouběji.

První fází spuštění je nalezení spustitelného procesu. Pokud uvedeme program pomocí relativní nebo absolutní cesty je tato fáze triviální. Jestliže je však uvedeno pouze bázové jméno procesu, což je typický příklad, je spustitelný soubor hledán v adresářích uvedených v proměnné shellu PATH (nikoliv v pracovním adresáři !).

Tuto proměnnou si můžete vypsat pomocí vestavěného příkazu *echo*(int). Tento příkaz je velmi jednoduchý, neboť jedinou jeho funkcí je to, že vypíše všechny své parametry na standardní výstup. Navíc však shell provede nad parametry všechna svá nahrazení včetně rozvinutí jmen svých proměnných:

Pokud tedy provedete příkaz ve tvaru `echo $PATH` získáte výstup podobný následujícímu:

```
/usr/kerberos/bin:/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/home/fiser/bin
```

Jednotlivé prohledávané cesty jsou v proměnné PATH odděleny dvojtečkami.

Při hledání spustitelného souboru se cesty prohlížejí zleva doprava a spustí se první nalezený spustitelný soubor. Proto je pořadí v němž jsou cesty uvedeny významné, neboť některé aplikace mohou tímto způsobem překrýt jisté systémové soubory. Většinou to bývají různé bezpečnostní nadstavby (v ukázce je to například bezpečnostní systém Kerberos).

Uspořádání vlastních systémových adresářů již tak významné není, ale v ukázce je zřetelná preference specializovaných adresářů před obecnými. Na posledním místě pak bývá uživatelský adresář spustitelných souborů (u každého uživatele je přirozeně jiný). Do tohoto adresáře by běžní uživatelé měly ukládat své spustitelné soubory nebo alespoň symbolické odkazy na ně.

Typickým rysem standardního nastavení proměnné PATH je absence pracovního adresáře (jenž je označen symbolickým jménem „.“ (tečka)). Důsledkem je skutečnost, že neleží-li soubor v alespoň jedné z cest uvedených v proměnné PATH , lze jej vyvolat jen s uvedením explicitní cesty.

Pokud tedy například programujete a vytváříte skripty mimo PATH adresáře (což je doporučeníhodné) a testujete jejich funkčnost, maje pracovní adresář nastaven na jejich adresář (což je nejpohodlnější), musíte použít cestu „./“ před jejich jménem (například *./mujskript*).

Pořadí PATH adresářů a uvedení či neuvedení pracovního adresáře má relativně velké bezpečnostní dopady (nejvíce pak přirozeně pro superuživatele), neboť se může stát, že po zadání příkazu bude aktivován jiný program než uživatel předpokládá (například místo některého ze standardních programů může být aktivován program s destrukční funkcí nebo podstrčený program – trojský kůň). Proto se nedoporučuje uvádět potenciálně nebezpečné adresáře v PATH cestách (například adresáře, kde má právo zápisu více uživatelů, pro superuživatele jsou to všechny adresáře do nichž mohou zapisovat běžní uživatelé) a to především na začátku seznamu. Zvláště nebezpečné je uvádění pracovního adresáře, neboť nelze žádným zajistit, aby se pracovním adresářem nestal některý z potenciálně nebezpečných adresářů. Uvedení pracovního adresáře v PATH lze tolerovat jen běžným uživatelům a i ti by jej měli mít vždy až na samém konci seznamu.

Pokud si chcete ověřit, odkud bude program spuštěn, použijte příkaz ***whereis***(1). Ten najde odpovídající spustitelný soubor (jen je-li v PATH cestě!), resp. jeho dokumentaci nebo zdrojové texty (jsou-li nainstalovány).

Nalezením spustitelného souboru však startovací anabáze ještě nekončí, neboť většina aplikací v Linuxu používá ke svému běhu sdílené dynamicky linkované knihovny (ty jsou v Linux nazývány *shared objects* a mají příponu *so*). Ty jsou primárně vyhledávány v adresářích */lib* a */usr/lib*. Další vyhledávací cesty jsou uloženy v */etc/ld.so.conf* resp. v souborech konfiguračního adresáře */etc/ld.so.conf.d*. Posledním zdrojem informací o umístění dynamických knihoven je proměnná shellu LD_LIBRARY_PATH (pokud

je definována a exportována). Stejně jako PATH by měla obsahovat seznam adresářů oddělených dvojtečkou.

Závislost spustitelného souboru na dynamických knihovnách lze prozkoumat pomocí programu ***ldd***(1).

Pokud se spuštění procesu podaří, čeká shell na jeho dokončení a až poté zobrazí nový prompt. To programu umožňuje vypisovat na terminál a číst z něj bez kolize se shellem (oba výstupu resp. vstupy by se prokládaly v téměř náhodném pořadí). Navíc shell tak může převzít tzv. návratovou hodnotu procesu. Vracená hodnota je až na jedinou výjimku v kompetenci aplikačních programátorů. Program může vracet hodnoty od 0 do 128, kde nula signalizuje úspěšné dokončení, hodnoty nenulové ukončení chybové (je to však jen úzus). Pokud je návratová hodnota větší než 128, byl proces přerušen externě prostřednictvím (neošetřeného) signálu.

Návratovou hodnotu posledního dokončeného příkazu lze v shellu získat pomocí speciální proměnné shellu s identifikátorem $? (dolar je prefix proměnné, otazník je skutečný identifikátor).

Spuštění na pozadí

Jestliže však program terminálu pro vstup nepoužívá, a pro výstup jej využívá jen velmi omezeně (typicky nejvýše jen pro chybový výstup) může být spuštěn tzv. na pozadí. V tomto případě je proces sice stále propojen s terminálem (pokud se sám od terminálu neodpojí, aby se stal démonem), ale shell na jeho dokončení nečeká a bezprostředně po jeho spuštění vypíše prompt.

Pro spuštění procesu na pozadí stačí za příkaz (včetně případných parametrů, voleb a přesměrování) uvést znak „&“ (ampersand).

Tento způsob spuštění se používá u dvou základních typů procesů:

1. u textově orientovaných procesů a skriptů, jejichž vstup a výstup je přesměrován (zapisují jen do souboru resp. ze souborů čtou). Toto využití programů, které mohou být jinak interaktivní, se označuje jako dávkové zpravování resp. režim (*batch mode*)
2. u X-Window klientů (GUI aplikací), pokud jsou spouštěny ze shellu. Pokud by nebyly spuštěny na pozadí, blokovaly by zbytečně terminál. GUI aplikace však zůstávají spojeny s terminálem do něhož mohou stále vypisovat chybové zprávy. Navíc pokud uzavřete terminál je i jim poslán signál SIGHUP (viz následující podkapitola) na což aplikace reagují bezprostředním ukončením!

Spuštění několika příkazů

Shell však nabízí i spuštění několika procesů na jediné příkazové řádce a to ne méně než pěti různými způsoby.

Nejjednodušší je ***sériové spouštění*** tj. procesy jsou spouštěny postupně jeden za druhým. Zde stačí oddělit jednotlivé příkazy středníky.

Při ***paralelním spouštění*** jsou procesy spuštěny najednou a běží nezávisle na sobě (a shell na žádný z nich nečeká). Zde je oddělovačem příkazů *ampersand* (jenž tak má dvojí roli, signalizuje provádění na pozadí a slouží i jako oddělovač příkazů).

Speciální případem paralelního spuštění jsou tzv. *kolony*. Procesy při jejím použití běží paralelně, ale navíc mezi sebou komunikují pomocí rour. Oddělovačem je v tomto případě svislítko „|“. S kolonami se blížeji seznámíme v kapitole **??** na straně ??.

Pro všechny výše uvedené způsoby vícenásobného spouštění procesů je typické, že spuštěny jsou dříve či později všechny programy na příkazové řádce bez ohledu na výsledek ostatních procesů.

Při *podmíněném spuštění* jsou programy spouštěny postupně, ale jen tehdy pokud předchozí skončí s úspěchem (tj. vrátí nulu). Pro oddělení programů se v tomto případě použije dvojznak „“. Tento zápis je převzat z jazyka C, kde je tento dvojznak logickým operátorem „a zároveň“ s postupným vyhodnocováním (první nepravdivá hodnota ukončí vyhodnocování v posloupnosti logických hodnot spojených logickou spojkou $\wedge$). Podobně i v shellu jsou vyvolávány jednotlivé programy a pokud stále končí s úspěchem (tj. jsou z hlediska logiky pravdivé) vykonávání pokračuje, pokud však jeden z procesů skončí neúspěšně, řetězec se ukončí (výsledek by byl jistě nepravdivý).

Příklad: Pro rekurzivní přesun adresářů nelze použít příkaz *mv*(1), neboť tento příkaz nepodporuje přepínač *-r*. Proto musíme nejdříve provést rekursivní přesun a následně (rekurzivní) výmaz. Pokud tak chceme učinit na jediném řádku (resp. v neinteraktivním skriptu) nelze přirozeně použít paralelní spuštění (nebylo by lze zajistit pořadí vyhodnocování) avšak ani běžné sériové:

```
cp -r sourcedir targetdir; rm -r sourcedir
```

Důvod je zřejmý, neboť zdrojový adresář je smazán i v případě, že se jeho kopírování nepodařilo (např. z důvodu nedostatečných práv v cílovém adresáři). Po provedení tak adresář neexistuje ani v jedné kopii.

Řešením je použití podmíněného spouštění (pomocí, něhož můžeme navíc uživateli signalizovat úspěšné provedení přesunu:

```
cp -r sourcedir targetdir && rm -r sourcedir && echo "OK. Adresář byl úspěšně zkopírován."
```

Stejně jako v jazyce C lze i v shellu použít pro podmíněné vykonávání i dvojznak „||“ (v jazyce C je to operátor logického součtu – nebo).

Nejčastěji se používá pro dva příkazy ve významu, který odpovídá například české větě: „*Ruce vzhůru nebo zemřeš*“.

Na první pohled je zřejmé, že zde nemá smysl uvažovat o logické hodnotě složeného výroku. Věta však vyjadřuje jistotu, že pokud nebude splněna první činnost bude provedena druhá. Bohužel v reálném světě však není zcela jisté co nastane v případě provedení první činnosti. Ve světě logiky a Unixu je to jednoznačné, druhá činnost provedena nebude (již po vyhodnocení prvního výrazu je zřejmé že výsledkem bude pravdivá hodnota). Jinak řečeno druhý příkaz bude proveden tehdy a jen tehdy pokud první skončí neúspěšně.

Příklad: Podmíněné konstrukce s logickým *nebo* se používá ve třech základních konstrukcích:

akce nebo ukončení— pokud se něco nepodaří je příkaz (skript) předčasně ukončen. Konstrukce tohoto typu se v praxi používají pouze ve skriptech.

akce nebo náhradní-činnost— v případě neúspěchu je spuštěn náhradní příkaz (jehož funkce je obdobná neúspěšnému). Náhradní akce lze řetězit pomocí dalších logických nebo. Speciálním případem neúspěchu je neexistence příkazu (programu)

```
firefox URL || mozilla URL || lynx URL
```

akce nebo výpis-chybového-hlášení– neúspěch je signalizován výpisem chybového hlášení

```
mkdir dir || echo "Adresar nebyl vytvoren"
```

5.4 Signály a zabíjení procesů

V počátcích Unixu byly tzv. signály navrženy jako prostředek pro externí ukončování procesů. Aby však mohly procesy na různé důvody svého ukončení ze vnějšku selektivně reagovat, bylo zavedeno několik různých signálů a procesům bylo dovoleno signály ignorovat nebo spustit rutiny, které před ukončením provedly úklid dat.

Dnes se signály používají i jako komunikační či synchronizační prostředky a pro distribuci informací o změnách okolního prostředí procesu (například při tzv. správě úloh). Původní podstata signálu však přežívá v názvu služby resp. programu pro jejich vyslání – *kill*(8) resp. *kill*(int). Stále také platí, že u většiny signálů je implicitní reakcí programu bezprostřední ukončení.

Původně existovalo jen několik (cca deset) signálů. Tyto signály jsou užívány ve všech Unixech a většina z nich užívá stejný číselný identifikátor. Novější Unixy však postupně zavedly i další signály (standard POSIX jich popisuje 19, v Linuxu ne méně než 30[4]). Navíc existuje 32 signálů nové generace (tzv. *real-time signals*).

Uživatelé Linuxu se však v praxi setkávají jen s několika signály a to ve třech základních rolích:

1. aplikace, které provozují, obdrží tyto signály při chybě nebo externí činnosti uživatele (např. stisku klávesy, apod).
2. pomocí signálu chtějí ukončit činnost aplikace, která vůbec nereaguje nebo se chová podivně
3. pomocí signálu komunikují s programem (typicky procesem běžícím na pozadí tzv. démonem)

Prvé dva aspekty signálů z hlediska uživatele jsou shrnuty v následující tabulce:

signál	číslo	vyvolání	KP
SIGHUP	1	uzavření řídícího terminálu	-
SIGINT	2	stisk Ctrl-C	+
SIGQUIT	3	stisk klávesy Ctrl+\ (často však jiné)	+
SIGILL	4	chybná instrukce (řídké – chybná platforma, viry)	++
SIGABRT	6	neošetřená výjimka	+++
SIGKILL	9	neignorovatelné ukončení (explicitní)	!
SIGSEGV	11	porušení ochrany paměti	+++
SIGPIPE	13	přerušení kolony (předčasné ukončení procesu v koloně)	++
SIGTERM	15	měkké ukončení programu (explicitní)	+++

[4] záleží na architektuře

Poslední sloupec naznačuje vhodnost signálu pro ukončování GUI aplikací (a v zásadě i aplikací ostatních včetně démonů). Ty nejvhodnější jsou označeny třemi plus. Signál SIGINT sice jakýkoliv proces ukončí s absolutní jistotou, proces však nemůže nijak reagovat tj. nemůže ani uložit rozpracovaná data či svou konfiguraci. Tímto způsobem však můžete ukončovat jen procesy, jejichž jste reálným vlastníkem (výjimkou je nikoliv překvapivě superuživatel).

Pro komunikaci s démony se nejčastěji používá signál SIGHUP. Většina démonů na tento signál reaguje znovunačtením konfiguračních souborů. Pokud tedy tyto soubory změníte a nechcete resp. nesmíte program restartovat, pošlete démonu tento signál (musíte však být přirozeně superuživatel). Někteří démoni používají i další signály z tabulky (ty označené jedním plus) resp. signály SIGUSR1 a SIGUSR2 (ty nemají žádnou univerzální sémantiku).

Pro posílání signálů lze použít několika programů. Hlavním z nich je program ***kill***(1)[5]. Jeho syntaxe je jednoduchá:

kill [*-signál*] *pid...*

kde *signál* je jméno signálu (viz tabulka), zkrácené jméno (bez SIG) nebo číslo. Pokud signál neuvedete, bude použit signál SIGTERM. Adresát (resp. adresáti) signálu je specifikován svým PID (to je nutné zjistit např. pomocí *ps*).

Pokud však chcete poslat signál více procesům nebo nemůžete rychle najít příslušný PID, existují i inteligentnější (avšak o něco méně bezpečné) aplikace.

Nejjednodušeji se používá program ***killall***(1), jenž se od klasického *kill*(1) liší v zásadě jen možností uvádět jména příkazů na místě PID (ty však na druhou stranu použít nelze). Praktickým rozšířením je i volba *-i*, která vynutí interaktivní potvrzení každého zaslání. Alternativně lze použít příkazy *pkill*(1) nebo *skill*(1), které umožňují sofistikovanější výběr dotčených procesů.

Pro ukončování aplikací v X-Window je vhodné použít programu ***xkill***(1). Po jeho vyvolání se změní tvar kursoru, kterým pak lze označit okno odsouzené aplikace. Program *xkill* však aplikace neukončuje zasláním signálu, ale odpojením od X serveru (tj. otrlá aplikace může i přežít). V prostředí Gnome je od novějších verzí k dispozici tlačítko lišty (*Force Button*), který nedobrovolné ukončování aplikací usnadňuje (a je o trochu bezpečnější).

Poslední možností zasílání signálů a ukončování aplikací je použití interaktivních zobrazovačů procesů počínaje programem *top*(1).

ÚKOLY

1. Vyzkoušejte si zde uvedené příklady (!).
2. Co dělá příkaz pstree?
3. Co provede spuštění procesu: *nohup yes &*. Co dělá *nohup*?
4. Co dělá příkaz *screen?*

[5] existuje i externí program *kill*(1) s podobným rozhraním, pokud však neuvedete cestu bude vyvolán vestavěný příkaz

6 Zpracování textů

6.1 Přesměrování

Většina textově orientovaných programů v Unixu komunikuje s okolím pomocí tří datových kanálů. Vstupní data jsou u většiny programů implicitně čtena z tzv. standardního vstupu (zkratka *stdin*), výstup se děje přes standardní výstup (*stdout*) a chybové zprávy a varování jsou posílána na standardní chybový výstup (*stderr*).

Každý z kanálů je uvnitř procesu přístupný přes pevně daný číselný deskriptor, jenž je již v okamžiku spuštění procesu spojen s některým ze vstupně-výstupních zařízení tj. je již otevřen (ve skutečnosti je zděden od rodičovského procesu). Standardní vstup má deskriptor 0, standardní výstup deskriptor 1, standardní chybový pak deskriptor 2.

Proces sice může ovlivnit propojení těchto deskriptorů s fyzickými vstupními zařízeními, běžné procesy však nic nemění a používají zařízení, která jim nastavil jejich proces-předek. Procesy spouštěné z terminálu dědí (a nadále využívají) nastavení, které jim poskytl shell. Implicitně jsou všechny standardní vstupně-výstupný kanály propojeny s terminálem, tj. proces čte z klávesnice terminálu a výstup směřuje na jeho obrazovku (ať již normální nebo chybový).

Tato propojení lze však na úrovni shellu měnit (nikoliv na úrovni aplikace ta pouze zdědí změněný stav). Mechanismus změny standardní vstupně-výstupních kanálů se označuje jako *přesměrování*.

Základní přesměrování

Nejjednodušším typem přesměrování je přesměrování do souboru resp. ze souboru.

`program < soubor`

přesměrování standardního vstupu, vstup je čten ze souboru

`program > soubor`

přesměrování standardního výstupu, výstup je zapisován do souboru, původní obsah souboru je smazán

`program >> soubor`

přesměrování standardního vstupu s připojováním (výstup je připojen za původní obsah souboru)

`program 2> soubor`

přesměrování standardního chybového výstupu (přepsání)

`program 2>> soubor`

přesměrování standardního chybového výstupu (připsání)

Složitější přesměrování

Tato přesměrování nemusí být dostupná ve všech shellech resp. se může lišit jejich zápis. Zde uvedenou syntaxi používá *bash*(1).

`program n>&m`

přesměrování deskriptoru *n* do deskriptoru *m*, nejčastěji se přesměrovává standardní chybový výstup do standardního výstupu zápisem `2>&1`. Přesněji se jedná o duplikaci deskriptoru (odkaz v tabulce otevřených souborů na pozici *m* je zkopírován i na pozici *n*).

Proto pokud chcete oba výstupu přesměrovat do jediného souboru je nutno použít zápis:

`program >soubor 2>&1`

nikoliv `program 2>&1 >soubor` (duplikován je původní odkaz na terminál, jež se při druhém přesměrování již nemění).

Přesměrování obou výstupů do jediného souboru lze navíc dosáhnout i zkráceným výrazem:

`program &> soubor`

Pokud chcete, aby zdrojem vstupu byl řetězec, je možno použít konstrukci označovanou jako *here string*.

`program <<< řetězec`

Řetězec podléhá téměř všem nahrazováním (substitucím) a po jejich provedení nesmí obsahovat bílý znak. Tento relativně nový zápis vychází z klasického přesměrování *here document*, jenž je používán ve skriptech a lze jej s výhodou použít místo kolony `echo řetězec | program` (ne však ve všech případech).

6.2 Kolony

Mechanismus standardních vstupně výstupních kanálů lze v Unixu použít i pro komunikaci mezi paralelně běžími procesy pomocí tzv. roury (*pipe*). Roury jsou jednosměrné datovody, kterými mohou data proudit z jednoho procesu do druhého.

Na úrovni shellu lze standardní výstup jednoho procesu přesměrovat na vstupní konec roury a její výstup na standardní vstup jiného procesu pomocí tzv. kolony. Výstup jednoho procesu (tzv. producenta) se tak postupně objevuje na vstupu druhého procesu (přenos se děje po jednotlivých bytech nebo menších blocích[1] nikoliv najednou). Oba procesy (běžící paralelně vedle sebe) tak mohou spolupracovat (i když ani jeden z nich netuší nic o existenci druhého).

Roura mezi oběma procesy neslouží jen jako komunikační kanál, ale i jako vyrovnávací paměť (standardně velikost 4KiB). Pokud se rychlost produkce dat (producentem) liší od rychlosti jejich konzumace (konzumentem) jsou data dočasně ukládána v rouře. Pokud se však roura naplní, je proces producenta pozastaven (zablokuje se v operaci zápisu na standardní výstup); pokud se vyprázdní (resp. je prázdná po vytvoření) je pozastaven naopak konzument (ve čtení ze standardního vstupu).

Zápis jednoduché kolony je snadný, stačí napsat dva příkazy oddělené znakem svislítka „|“.

[1] u textových dat po řádcích

```
příkaz-1 | příkaz-2
```

Kolona zajistí spuštění obou procesů (jsou spuštěny najednou a běží paralelně), vytvoření roury a potřebné přesměrování vstupů a výstupů. Příkaz-1 produkuje data (v koloně má funkci producenta) data proudí do druhého procesu, jež je zpracovává (má funkci komzumenta).

Pokud vše běží v pořádku ukončí producent svou činnost a uzavře svůj standardní vstup. Po vyprázdnění roury se o tom dozví proces konzument (při vstupu je signalizováno dosažení konce souboru – EOF) a také ukončí svou činnost.

Oba procesy se z hlediska shellu chovají jako jediná úloha. Lze ji tedy spustit jak na popředí (shell čeká na posledního procesu v koloně) tak na pozadí (za celou kolonou je uveden znak „&“) a lze ji dokonce ovládat jako celek (viz kapitola o úlohách na straně 97).

Pokud je nejdříve ukončen druhý proces (např. signálem nebo předčasným ukončením programu), je prvnímu procesu zaslán signál SIGPIPE a shell může vypsat hlášení typu „*broken pipe*“.

Kolony však nemusí být omezeny jen na dva procesy. Pomocí rour lze vytvářet celé (lineárně) sekvence procesů, z nichž každý (kromě prvního) dostává data od předchozího procesu pomocí roury, zpracuje je a následně posílá dalšímu procesu v řadě (samořejmě kromě posledního). Procesy uvnitř kolony tak hrají roli konzumenta i producenta zároveň a jsou označovány joko filtry.

V moderních shellech lze navíc lineární kolony organizovat do složitějších struktur v nichž data proudí v několika vzájemně provázaných datových tocích (obecně však musí tvořit acyklický graf). V praxi jsou však používány jen zřídka.

Z hlediska použití (možných rolí) v kolonách lze všechny linuxovské programy rozdělit do čtyř skupin.

producenti – tyto programy vypisují na standardní výstup textové informace (standardní vstup nepoužívají). Mohou tedy zaujímat jen roli producentů. Mezi producenty patří valná většina programů popsaných v předchozích kapitolách (např. *ls*(1), *ps*(1)).

filtry – programy, které čtou data ze standardního vstupu, transformují je a vypisují na standardní výstup. Kromě své základní pozice filtrů mohou být použity i na místě konečných konzumentů (jejich standardní výstup je směrován na terminál nebo do souboru). Většina z nich je schopna číst svá vstupní data i ze souborů (jejjch jména jsou parametry programu-filtru) a mohou tedy sloužit i jako počáteční konzumenti. Tyto programy hrají v kolonách klíčovou roli, a proto je jim věnována následující podkapitola.

konzumenti – programy, které čtou ze standardního výstupu, ale na standardní výstup nic nevypisují. V zásadě buď data vizualizují bez použití standardního výstupu (stránkovače, vizualizace grafických dat) nebo provádějí specializovaný zápis do souborů (komprese apod.). Některé z nich mohou za použití speciálních parametrů data vypisovat i na standardní výstup a lze je tudíž použít i jako filtry (je to však až sekundární použití).

ostatní – nevyužívají ani standardní vstup ani standardní výstup. Jsou to některé interaktivní programy *top*(1) a především většina X-Window aplikací (klientů). Většinou však využívají standardní chybový výstup a lze je tedy po duplikování standardního chybového výstupu používat jako producentů.

Příkladem budíž například překladače, které na chybový výstup směřují výpis syntaktických chyby a varování. Tento výpis lze pro přesměrování poslat pomocí kolony jinému programu na zpracování (ten jej upraví do přehlednější podoby).

6.3 Filtry

Tato podkapitola je věnována popisu těch nejzákladnějších unixovských filtrů. Výjimkou jsou filtry *grep* a *sed*, jež užívají regulární výrazy a proto je jejich popis uveden až v podkapitolách 6.7.4 na straně 80 (*grep*) resp. 6.7.5 na straně 81 (*sed*).

cat – kopírování a spojování proudů dat

Nejjednodušším filtrem je program ***cat***(1). Jeho základní funkce je jednoduchá, neboť vše co obdrží na standardním vstupu pošle na výstup. Může se tedy jevit zhola zbytečným, opak je však pravdou, neboť je mnohdy zcela nezastupitelný (není však přirozeně ve své základní podobě).

Základní tvar příkazu *cat* (a obecně všech filtrů) je následující:

```
cat soubory...
```

Pokud je uveden bez parametrů tj. beze jmen souborů čte data ze standardního vstupu, pokud je však uveden alespoň jeden soubor, čte data z něj (resp. ze všech uvedených souborů postupně).

Použití příkazu *cat* lze z hlediska syntaxe rozdělit do základních tří oblastí.

I.

```
cat soubor (resp cat < soubor)
```

předání dat ze souboru na standardní vstup

využití:

1. předání dat ze souboru čístému konzumentovi (očekává data jen na standardním vstupu). U většiny standardních příkazů (konzumentů) je to zbytečné, neboť přejímají alternativní vstupní kanál jako parametr. U některých uživatelských programů však tomu tak být nemusí. Příklad: *cat soubor | c2html*
2. vložení obsahu souboru na příkazový řádek. Příklad: *kill ‘cat /var/run/httpd.pid‘*

II.

```
cat > soubor
```

uložení standardního vstupu (z terminálu) do souboru

využití:

1. vložení krátkého textu do souboru. Zadávání textu se ukončí odřádkováním s stiskem klávesy Ctrl+D. Pro delší texty je přirozeně vhodnější použít textový editor.

III.

```
cat soubor-1 soubor-2 ... soubor-n
```

spojení dat z více zdrojů (souborů) do jediného proudu (na standardním výstupu). Místo jména jednoho ze souborů může být uvedena pomlčka. *Cat* čte postupně data ze jednotlivých souborů, pokud narazí na pomlčku použije standardní vstup (typicky přesměrován z kolony).

využití:

1. spojení více souborů do jednoho (soubory nemusí být textové). Příklad: *cat p1 p2 p3 > p*
2. připojení (statického) textu před resp. za data zpracovávaná v koloně. Příklad: *... | cat hlavicka - paticka | ...*
3. sériové spojení více proudů dat do jednoho (např. od různých producentů, z různých kolon, apod.)
   Příklad: *cat <(ps -eo user,pid | grep "root" | cut ...) <(/sbin/fuser -v /etc | ...) | xargs | kill ...*

Filtr *cat* má i několik málo voleb, z nichž nejužitečnější je volba *-n*, pomocí níž lze snadno očíslovat výstupní řádky.

wc — počítání řádků a znaků

Další velmi jednoduchý, ale užitečný filtr. Program ***wc***(1) počítá počet řádku, slov a znaků (resp. bytů[2]) na svém vstupu a vypisuje stručnou statistiku. Pokud není uveden žádný přepínač vypisuje tři čísla v pořadí počet řádku (přesněji počet odřádkování[3]), počet slov (slova jsou oddělena bílými znaky), a počet bytů (nikoliv znaků).

Konkrétní údaj si lze vynutit přepínačem: *-l* (řádky resp. odřádkování), *-w* (slova), *-c* (byty), *-m* (znaky). Zajímavý je i přepínač -L poskytující délku nejdelšího řádku (ve znacích).

Příklad: `ps -e | wc -l`

počet běžících procesů

`find ~ -type l | wc -l`

počet symbolických odkazů v domovském podstromu

`find / | wc -L`

délka nejdelšího absolutního jména v souborovém systému (ve znacích). Správné výsledky však poskytuje, jen je-li spuštěn s právy superuživatele.

cut — rozdělování (rozřezávání) řádků

Filtr ***cut***(1) umožňuje výběr částí řádků (resp. řádku) na základě jejich pozice. Pozici lze adresovat na úrovni jednotlivých znaků[4] (vyber znaky od pátého do desátého) nebo na úrovni polí oddělených libovolným oddělovačem.

Volba úrovně adresování závisí na charakteru vstupních dat. Pokus jsou jednotlivé relevatní údaje zarovnány pod sebou (tj. vstupem je vizuálně viditelná tabulka) je nutno adresovat na úrovni znaků. Pokud jsou odděleny oddělovačem (jednoznakovým a všude stejným) je nutno používat pole.

Speciální pozornost je potřeba věnovat polím, jež jsou oddělena tabulátory. Ty mohou sice vizuálně působit jako zarovnané, ale pro správné rozdělení je nutno použít polí. Nejhorší situace nastává pokud je pro vizuální zarovnání použita směs tabulátorů a mezer. V tomto případě je nutné předzpracování vstupu (například filtrem *tr*(1)).

[2] u systému s vícebytovým resp. různobytovým kódováním (typicky UTF-8) se počet znaků a bytů může lišit.

[3] pokud není poslední řádek odřádkován, není počítán!

[4] existuje i možnost adresování na úrovni bytů, ta je však pro textová data téměř nepoužitelná

Základní syntaxe pro rozřezávání na úrovni znaků má tento tvar:

```
cut -c rozsah [soubory]...
```

kde rozsah je je bud jediné číslo (např. `2`), interval (např. `5-10`), otevřený interval (`5-` je pátý až poslední), resp. několik rozsahů oddělených čárkami (např. `2,5-8,10-`).

Syntaxe při rozřezávání na úrovní polí (*fields*) je obdobná:

```
cut [-d znak] -f rozsah [soubory]...
```

kde syntaxe rozsahu je stejná (nyní se však vztahuje na celá pole). Pomocí přepínače `-d` lze specifikovat oddělovač polí (vždy jen jeden znak, u některých např. mezery je nutná ochrana před shellem), implicitním oddělovačem je tabulátor.

head a tail

Program ***head**(1)* standardně zobrazuje prvních deset řádků standardního vstupu[5] (ostatní de facto zahazuje), program ***tail***(1) deset posledních (ostatní opět zahazuje). Oba programy podporují volbu *-n*, pomocí níž lze určit jiný počet vypisovaných řádků.

Program *head* se používá z několika příčin:

- relevantní informace obsahuje jen prvních n-řádků textového vstupu (typicky jen první řádek)
- vstup je příliš dlouhý a uživateli postačí jen (náhodný) výběr (tj. n-výskytů)
- vstup je setříděn a uživateli stačí znát prvních n-výskytů (*top ten*)

Příklad: `last $USERNAME | head -n 1`

poslední přihlášení uživatele (výstup příkazu *last* je setříděn podle času sestupně). Příkaz lze zjednodušit na `last -n 1 $USERNAME`, neboť last přímo podporuje volbu *-n* (to je však spíše výjimka[6]).

Program *tail* se používá především pro výpis logovacích souborů, jejichž obsah pomalu narůstá přidáváním řádků na konec souboru. Nejnovější (a tím často i nejzajímavější) informace jsou proto na konci souboru. Navíc výpis celého souboru nemusí být vůbec možný (některé logy mají velikost v řádu stovek MiB).

Program *tail* lze používat i ve speciálním režimu, kdy po výpisu *n* posledních řádků se proces zablokuje a čeká na přidání řádku do souboru, poté přidaný řádek vypíše a znovu čeká (zablokování lze přerušit jen zasláním signálu typicky SIGINT stiskem Ctrl+C). Tento režim se zapíná volbou *-f* (folow).

tr

Filtr ***tr***(1) slouží provádění jednoduchých textových transformací na úrovni jednotlivých znaků. V základním režimu slouží k náhradě znaků z jisté množiny (určené řetězcem) za znaky z množiny druhé (určené opět řetězcem) a to v celém zpracovávaném textovém proudu. Transformace je zobrazením z množiny znaků do množiny znak a je jednoznačně určena pozicí znaků v obou řetězcích tj. první znak z prvého řetězce bude nahrazen prvním znakem řetězce druhého, atd. V obou řetězcích lze navíc použít i některé speciální zápisy jako je interval (např. *a-z*), posixovské třídy znaků (např. *[:alpha:]*) nebo escape sekvence (v rozsahu jazyka C).

[5] pokud soubor obsahuje menší počet řádků, vypisuje samozřejmě jen ty existující

[6] v Unixu platí základní filozofie, že programy by se měly soustředit jen na podporu své speciální funkčnosti a vše ostatní přenechat jiným (specializovaným) programům.

Program *tr* se od ostatních filtrů liší, tím že nepodporuje předávání jmen vstupních souborů na příkazovém řádku (nebylo by lze rozlišit transformační řetezce od jmen souborů). Pokud má filtr *tr* číst ze souboru je nutno použít přesměrování standardního vstupu.

Příklad: `tr a-z A-Z <text >upper_text`

do souboru *upper_text* zkopíruje obsah souboru *text*, avšak nahradí všechny malá písmena velkými (pouze latinková bez diakritiky).

`ls | tr " \t" -`

vypíše obsah odresáře, přičemž mezery a tabelátory ve jménech souborů nahradí za pomlčky

Kromě základního režimu transformace poskytuje *tr* i další funkce. Pro ně je typické že specifikován jen jeden řetězec.

`tr -d` *řetězec*

výmaz všech znaků, jež se vyskytují v řetězci, ze vstupního proudu

`tr -s` *řetězec*

nahrazení vícenásobných skupin jednotlivých znaků z řetězce jediným znakem (skupiny jsou tvořeny opakováním jediného znaku).

Příklad: `tr -d "\r" <dos.txt >unix.txt`

odstranění znaků CR ze souboru *dos.txt* (soubor sám se nezmění, výsledek je uložen do souboru *unix.txt*). Tímto způsobem lze převést soubory s odřádkováním CR+LF (MS DOS a Windows) na soubory s odřádkováním unixovským (pouze LF).

Pokud by měl být změněn přímo soubor *dos.text* nelze to provést zápisem `tr -d "\r" <dos.txt >dos.txt`, neboť v okamžiku otevření souboru *dos.text* je tento zkrácen na nulovou délku, a následné čtení (soubor se čte postupně po řádcích) by již žádná data nenašlo. Jediným řešením je použití pomocného souboru a následného přesunu.

`ls -l --full-time | tr -s [:space:] | cut -f 6,9 -d" "`

zobrazí datum poslední modifikace souborů v aktuálním adresáři. Pomocí *tr* je zde vyřešen problém s výstupem (příkazu ls -l), jenž je zarovnán do sloupců pomocí mezer, přičemž šířka polí nemusí být konstantní (mění se šířka pole velikosti souboru podle maximální zobrazované velikosti). Zjednodušením výplní mezi sloupci na jedinou mezeru lze pole snadno rozdělit příkazem *cut* (oddělovačem je právě ta jediná přeživší mezera).

sort

Program *sort* patří svým rozhraním na příkazové řádce k nejsložitějším unixovským programům. Rozhraní, jeho různé kombinace, vedlejší efekty různých voleb jsou tak složité, že by si tento příkaz zasloužil vlastní skripta. Naštěstí jen menší část funkčnosti programu je nezbytně nutná pro většinu praktických úloh.

Volby příkazu *sort* lze rozdělit do dvou skupin. V první jsou volby modifikující řazení (typ třídění), ve druhé parametry určující pozice klíčů podle nichž je tříděno.

Typy třídění (prvá podskupina jsou hlavní typy, druhá možné modifikace).

volba	význam
(bez volby)	řazení lexikografické (podle všech znaků)
-n	řazení číselné (celá čísla)
-g	řazení číselné (libovolná čísla)
-d	řazení alfanumerické (jen podle znaků, čísel a mezer)
-b	ignoruje počáteční bílé znaky
-r	reverzní třídění (sestupné)
-f	ignoruje velikost písmen
-s	stabilní třídění (řádky se shodnými klíči nemění pozici)

Standardním třídícím klíčem filtru *sort* je celý řádek. Detailnější třídící klíč lze specifikovat pomocí volby *-k* a to na úrovni polí (oddělených oddělovačem) resp. dokonce znaků. Volba *-k* se může na příkazovém řádku vyskytovat i vícekrát, v tomto případě první určuje klíč primární, druhá sekundární (použije se jen při shodě v prvním klíči), atd.

Obecný (mírně zjednodušený) tvar volby *-k* je následující:

`-k p-pozice[.z-pozice][b] p-pozice[.z-pozice][bdfnr]`,

kde *p-pozice* a *z-pozice* jsou čísla (pozice počítané od jedné).

První pozice určuje počátek klíče, druhá pozice jeho konec. Obě pozice jsou primárně určeny pomocí polí (standardním oddělovačem polí na úrovni řádku jsou skupiny bílých znaků), indexem počínaje od jedné (p-pozice). Oddělovač polí lze změnit pomocí volby *-t znak*.

Počátek nebo konec lze zpřesnit znakovou pozicí (indexem vztaženým k poli, počítaným od jedné!). Navíc lze pro každý klíč specifikovat typ třídění (pokud se liší od globálního, jež je určen příslušným parametrem). Specifikátory se píší za druhou pozici (lze je psát i za první resp. za obě, ale není to příliš přehledné) a odpovídají znakům globálních parametrů. Specialitou je specifikátor *b*, jehož použití se u jednotlivých pozic liší. U počáteční specifikuje ignorování počátečních bílých znaků (což je praktické), u koncové pak ignorování koncových (což není příliš často používané).

Příklad: `ls -l --full-time | sort -k 5,5nr -k 6,7r`

setřídí soubory v aktuálním adresáři primárně podle velikosti (numericky, sestupně), sekundárně podle času poslední modifikace (sestupně, tj. novější dříve). Časové údaje lze třídit lexikograficky, neboť díky volbě *–full-time* jsou uvedeny v jednotném a pro třídění vhodném ISO formátu. Bohužel při třídění nejsou zohledněny časové zóny, tj. především časy uvedené v zimním (posun +1000) a letním (posun +2000) čase. Soubory vzniklé v hodinách s posunem času (dvakrát jedna hodina ročně) nemusí být setříděny správně. Zcela správné setřídění lze dosíci vynucením pevné časové zóny např. prefixací příkazu přiřazením proměnné shellu `TZ="GMT-1" ls ...` (všechny časy budou uvedeny v SEČ, přiřazení platí jen pro příkazovou řádku).

`find ~ -size +1024k -printf "%10s %p\n" | sort -rnb -k 1,1 | head | cat -n`

vypíše deset největších souborů v domovské adresáři (sestupně, očíslované). Akce *printf* příkazu *find*(1) vypíše o každém souboru větším než 1MiB (omezení lze zvolit různě, jeho funkcí je jen zmenšit velikost dále zpracovávaných dat)

informace o jeho velikosti (popisovač „%s“) a celé jeho jméno (popisovač „%p“). Vyprodukovaný seznam je setříděn (*sort*), omezen na prvních 10 řádků (*head*) a nakonec očíslován (*cat*).

uniq

Filtr ***uniq***(1) odstraňuje ze vstupního proudu duplicitní řádky tj. řádky ze souvislé posloupnosti shodných řádků ponechává pouze jeden. Duplicitní řádky musí tvořit souvislou posloupnost, jinak nejsou všechny duplicity odstraněny. Sovislých poslupností a tím dokonalé jedinečnosti (unikátnosti) řádků lze nejsnadněji dosíci setříděním vstupní postoupnosti (to však může být velmi pomalé!).

Pro testování shody jsou porovnávány celé řádky, parametrem *-f n*, lze přeskočit prvních n polí (pole jsou odděleny skupinami bílých znaků a nelze to změnit).

Příklad: `who | cut -f 1 -d " " | sort | uniq`

seznam aktuálně přihlášených uživatelů. Každý uživatel je uveden nejvýše jednou (i když má více přihlašovacích shellů) a seznam je setříděn.

```
ps -eo rss,comm | sort -k 2,2 -k 1,1rn | uniq -f 1 | sort -rn -k 1,1 | head
-n 12 | cat -n
```

seznam dvanácti programů s největším využitím rezidentní (operační) paměti. Pokud program běží ve více instancích je zobrazena pouze jedna (ta s největším využitím rezidentní paměti). Producentem je program *ps*, jenž vypisuje jen relevantní informace (rezidentní velikost, jméno programu). Aby bylo lze odstranit duplicity je nutno vyprodukovaný seznam setřídit podle jména spustitelného souboru (druhé pole). To však nestačí, neboť *uniq* odstraní všechny duplicitní řádky kromě prvního, tj. na prvním řádku musí být vždy instance s největšími paměťovými nároky. tento požadavek zajistí sekundární třídící klíč (velikost rezidentní paměti = první sloupec). Nyní již lze aplikovat *uniq*, jež však musí přeskočit první sloupec. Následuje opět třídění tentokrát podle velikosti rezidentní paměti a závěrečné odseknutí zbytečných řádků a očíslování.

Výstup pro můj počítač v okamžiku psaní těchto řádků, měl tento tvar:

```
1  50904 soffice.bin
2  26372 X
3  18060 gnome-panel
4  15132 lyx
5  14996 gnome-terminal
6  12984 nautilus
7  11740 ggv-postscript-
8  11548 ggv
9  10944 kdeinit
10  10884 gnome-keyboard-
11  10620 konqueror
12  10140 wnck-applet
```

Lze tak snadno vidět jaké aplikace jsou největšími „žrouty“ paměti (OpenOffice, X-Window, Gnome desktop, KDE desktop, přestože jej nepoužívám).

tee – rozdělení (duplikace) proudu

Filtr ***tee***(1) je jakýmsi opakem filtru *cat*(1). Podobně jako *cat* žádná data nemodifikuje, ale na rozdíl od programu *cat*, který umožňuje datové proudy spojovat, slouží *tee* k

rozdělení proudu na (alespoň) dva identické proudy, z nichž každý pak může být zpracován jiným způsobem. Jedna kopie dat je kopírována na standardní výstup (kde může být přímo předána dalšímu programu v koloně), druhá je ukládána do souboru, jehož jméno je parametrem filtru. Pokud je uvedeno více souborů bude do každého z nich uložena vlastní kopie dat.

Při běžném použití jsou duplikáty ukládány do běžných souborů, jež lze chápat jako úložiště dat v různých fázích zpracování (tj. zachovány jsou i pak mezistavy). Namísto souborů lze však uvádět i tzv. procesové substituce, které umožní i další zpracovávání duplikovanách dat (v tzv. pobočných kolonách). Procesorovými substitucemi se zabývá podkapitola 6.5 na následující straně, v niž budou uvedeny i příklady na použití filtru *tee*.

6.4 Konzumenti

Typických konzumentů (tj. programů přijímajících standardní vstup a nic nevypisujících na standardní výstup) je v Unixu málo. Pokud je již tato pozice obsazena (tj. neproudí-li data přímo na terminál), slouží jako konzumenti filtry, jejichž standardní výstup je přesměrován do souboru resp. na jiné textově orientované zařízení.

Klasickými konzumenty jsou především tzv. stránkovače tj. programy, jež zobrazují postupně jednotlivé stránky výstupu na terminálu. Stránkovače se tedy používají pro zobrazení textu (ze souboru nebo z kolony), jenž se nevejde na terminál.

Původním stránkovačem terminálového výstupu byl program ***more***(1). V Linuxu se však používá jeho zpětně kompatibilní nástupce označovaný jako ***less***(1). Tento stránkovač má oproti původnímu mnohé výhody, především umožňuje obousměrný pohyb v dokumentu a celý dokument načítá jen v případě potřeby (lze jej tedy použít i pro čtení souborů o velikosti desítek MiB).

Po spuštění je stránkovač less ovládán pomocí běžných kláves. Ty základní jsou obsaženy v následující tabulce (příkazy dostupné i v *more*(1) jsou opatřeny znaménkem plus před popisem):

příkaz	popis
mezerník, PgDown	+zobrazí další obrazovku (posun vpřed o obrazovku)
Return, ↓	+posun vpřed o jeden řádek
b , PgUp	posun na předchozí obrazovku
k, ↑	posun vzad o jeden řádek
→	posun o půl obrazovky vpravo
←	posun o půl obrazovky vlevo
*n*g	posun na řádek *n*
/*re*	+najde první řetězec odpovídající reg. výrazu (BRE+, viz kap. 6.7)
q	+ukončí program (zobrazení)

6.5 Procesové substituce

Běžné datové proudy jsou lineární a pokud mají pobočné větve, jsou tyto omezeny pouze na čtení souborů resp. jejich ukládání. Pobočné větve se mohou buď spojovat (sbíhat dohromady) nebo rozdělovat. Pro spojování se nejčastěji používá filtr *cat*(1)(řadí data z jednotlivých větví za sebe) pro rozdělování buď *tee*(1) (duplikace, do každé větve tečou stejná data) resp. *csplit*(1) (data jsou rozdělena podle kontextu).

Novější shelly (včetně bash) však umožňují vytvářet i komplexnější pobočné větve, v nichž jsou data vytvářena resp. upravována pomocí dalších příkazů nebo dokonce kolon (tzv. pobočné kolony). Prostředkem, který pobočné zpravování dat umožňuje, jsou na vyšší (shelové) úrovni tzv. procesové substituce a na nižší (systémové) tzv. pojmenované roury.

Pojmenované roury (označované také jako FIFO, neboť se jedná *de facto* o fronty) jsou velmi podobné anonymním rourám používaným u běžných kolon. Stejně jako ony umožňují komunikaci (nejméně) dvou procesů, tím že poskytují jednosměrný datovod (tj. jeden proces může do pojmenované roury zapisovat a druhý z ní může předaná data postupně číst). Stejně jako u klasických rour je zajištěna i základní synchronizace. V souladu se svým názvem však mají svá jména, pod nimiž se objevují v adresářové hierarchii (tj. navenek se tváří jako běžné soubory). Do pojmenované roury může zapisovat libovolný proces, který má souborové právo zápisu (je však zablokován v otevření dokud si stejnou rouru neotevře jiný proces pro četní). Obdobná omezení platí i pro čtení z roury (zde se ovšem čeká na proces-zapisovatel).

Pokmenované roury lze používat téměř stejně jako běžné soubory. Jedinou výjimkou je jejich vytváření, které je možné pouze za pomoci příkazu *mkfifo* (parametrem jsou jména vytvářených rour).

6.6 Vkládání výstupu

Vkládání výstupu je také speciálním případem nahrazování (textové substituce). Umožňuje vkládat textový výstup programu (provedený přes standardní výstup) na příkazový řádek, kde se stává parametrem příkazového řádku dalšího příkazu (resp. dokonce celým příkazem). Stručně řečeno příkaz je substituován za svůj výstup.

Pro vkládání výstupu stačí uzavření příkazu do zpětných apostrofů (rozlišovat od běžných apostrofů, na PC americké klávesnici jsou umístěny pod *Esc*).

Příklad: `head -n 1 ‘find ~ -name "*.c"‘`

zobrazení prvního řádku všech zdrojových souborů jazyka C v domovském podstromu. U každého souboru je nejdříve uvedeno jméno souboru (ve tvaru ==> *jméno* <==), což

Použití zpětných apostrofů však má i jisté nevýhody, zpětné apostrofy nejsou v mnoha fontech příliš zřetelné (resp. jsou zaměnitelné s běžnými[7]) a především nefungují jako závorky, což znemožňuje jejich vzájemné vkládání.

Proto lze v některých shellech (včetně bashe) používat modernější způsob zápisu, v němž vkládání začíná dvojznakem *$(* a končí uzavírající oblou závorkou.

[7] příkladem budiž font použitý pro ukázky kódu v těchto skriptech

Příklad: `head -n 1 $(find $(cat sourcedirs) -name "*.c")`

zobrazení prvního řádku všech zdrojových souborů jazyka v postromech, jejichž kořenové adresáře jsou uvedeny v souboru *sourcedirs* (oddělené bílými znaky typicky odřádkováním).

Mechanismus vkládání standardního výstupu je v Unixu široce používán, neboť umožňuje dynamické generování parametrů, voleb nebo celých příkazů. Použití tedy není omezeno jen na generování seznamů souborů (adresářů, atd.)

Příklad: `mail (who | cut -f 1 -d " " | sort | uniq) <zprava`

pošle všem aktuálně přihlášeným uživatelům e-mail (text je uložen v souboru *zprava*). Pro přenos je použit lokální protokol a zpráva je předána do lokální schránky.

`kill -SIGHUP $(cat /var/run/xinetd.pid)`

posílá démonu xinetd signál SIGHUP, po němž démon znovu načte konfigurační soubor (rekonfugurace). PID démona je uložen v souboru `/var/run/xinetd.pid`. Obdobný soubor vytváří v adresáři */var/run* většina démonů.

Mechanismus vkládání výstupu má však jedno zásadní omezení, jímž je maximální délka příkazového řádku shellu. Tato velikost je relativně velká (řádově desítky KiB), ale přesto ji lze v některých případech snadno překročit. Příkladem budiž například příkaz `file $(find /usr)`, nebude pravděpodobně proveden, neboť výstup programu *find* má velikost několika set KiB (u mne řádově 350 KiB).

Pokud už tato situace nastane (resp. lze ji očekávat), je nutno použít buď vestavěných mechanismů dotčených příkazů (zde např. akce -exec programu *find*(1)) nebo příkazu ***xargs***(1).

Program *xargs* ve své základní podobě očekává jediný parametr, jímž je jméno jiného programu (dále externí program), a na svém standardním parametry tohoto externího programu. Pokud je vstup (=předané parametry) menší než velikost vstupního bufferu shellu, je externí příkaz spuštěn, a na příkazový řádek jsou mu předány parametry ze standardního vstupu. Až potud *xargs* pracuje jako mírně omezená varianta vkládání vstupu (dodat lze jen parametry nikoliv volby a to pouze v jedné úrovni). Pokud je však vstup větší než velikost vstupního bufferu shellu, je vstup rozdělen na bloky dostatečně malé velikosti, a pro každý blok je spuštěna nová instance externího příkazu (implicitně sériově). Každá instance tak získává a následně zpracovává jen určitou část parametrů. To je ve většině případů ekvivalentní jednorázovému zpracování, nikoliv však ve všech (např. pokud zpracování závisí na kontextu předchozích parametrů, např. souvislé číslování řádků, počítání znaků pomocí *wc*(1)).

Kromě externího příkazu lze na příkazové řádce programu *xargs* uvádět i volby a počáteční parametry externího programu (avšak až za parametrem se jménem externího programu).

Příklad: `find ~ -name "*.c" | xargs head -n 1`

obdoba volání z prvého příkladu této kapitoly, jež však bude funkcí i při enormním počtu zdrojových souborů jazyka C v domovském adresáři (resp. enormní délce jejich jmen).

V tomto konkrétním případě nelze použít akci příkazu *find* (tj. zápis `find ~ -name "*.c" -exec head -n 1 '{}' \;`), neboť příkaz *head* v tomto případě nevypisuje záhlaví jména souborů (jeho parametrem je vždy jen jediný soubor). Pokud však existují dva ekvivalentní příkazy, jeden užívající akcí uvnitř *find*, druhý programu *xargs*, není vždy snadné rozhodnout jaký z nich zvolit. Řešení pomocí

xargs je ve většině případů výrazně rychlejší (externí program se volá jen jednou nebo v malém počtu instancí), ale akce poskytují první výsledky ihned po nalezení prvního souboru (*xarg* až po nalezení všech nebo po naplnění bufferu).

6.7 Regulární výrazy

6.7.1 Základní syntaxe

Regulární výrazy umožňující formální popis jistých podtříd množiny řetězců, patří už od počátků Unixu ke standardním prostředkům nabízeným na uživatelské úrovni. Původní jazyk regulárních výrazů byl však v průběhu více než třiceti let postupně rozšiřován a to ne vždy jednotným způsobem. V současnosti existují tři základní verze tohoto jazyka:

základní regulární výrazy (BRE): původní a velmi omezená syntaxe, později byla rozšířena takřka na úroveň rozšířené (avšak se zachováním zpětné kompatibility, nestandardizováno, označení BRE+). Tuto úroveň poskytuje většina nespecializovaných unixovských programů (*find*, *Emacs*, *sed*).

rozšířené regulární výrazy (ERE): rozšíření základní úrovně (bez zachování zpětné kompatibility). Užívána především specializovanými nástroji (grep, jež však podporuje i BRE). Standardizováno na úrovni standardu POSIX (včetně znakových tříd). Následující text primárně popisuje tuto syntaxi regulárních výrazů.

perlovské regulární výrazy (PRE): regulární výrazy používané v jazyce Perl (verze 5). Jsou nejobecnější a nejsilnější a mají vestavěnu podporu Unicode. Popis této úrovně leží mimo zaměření tohoto skripta. Perlovské regulární výrazy jsou však zpětně kompatibilní s rozšířenými regulárními výrazy (tj. vše uvedené zde pro ERE lze použít i v PRE).

Regulární výraz vždy popisuje jistou množinu řetězců. Množina může být tvořena jediným řetězcem, konečnou množinou řetězců, resp. potenciálně nekonečnou podmnožinou univerzální množiny řetězců (ti tvoří libovolná posloupnost znaků z abecedy daného počítače, v Linuxu je to to dnes znaková sada Unicode). Tuto množinu budeme v souhlase s terminologií gramatik (jejíž součástí regulární výrazy) označovat jako jazyk (popsaný daným regulárním výrazem).

Regulární výraz může obsahovat dva typy znaků:

terminály (znaky): popisují přímo znaky řetězců daného jazyka (na příslušných pozicích). Např. znak „a“ si vynucuje použití stejného znaku ve všech slovech jazyka, jenž je popsán daným regulárním výrazem.

neterminály (metaznaků): popisují jistou znakovou množinu (vynucují znak z dané množiny na dané pozici), pozici (vynucují danou ralativní nebo absolutní pozici) [tzv. kotvy] resp. operace nad regulárními výrazy [operátory].

Protože oba typy znaků musí být zvoleny ze standardní znakové sady systému (neexistuje žádná specializovaná abeceda neterminálů, kterou nelze použít v řetězcích) musí být používány různé escape sekvence. Bohužel jejich používání není příliš jednotné (výjimkou je pouze PRE). V zásadě však platí že číslice a písmenné znaky jsou vždy terminály, ostatní znaky jsou povětšinou také terminály, až na omezenou skupinu metaznaků (viz následující přehled). Pokud jsou tyto metaznaky použity ve funkci běž-

ných znaků, musí být uvozeny zpětným lomítkem. Na druhé straně existují metaznaky, tvořené písmennými escape sekvencemi (tj. zpětným lomítkem a písmenem).

Nejjednodušší jsou tzv. elementární regulární výrazy, jež jsou tvořeny jedním teminálem nebo neterminálem. Elementární regulární výrazy, vždy popisují jazyky tvořené pouze jednoznakovými řetězci.

znak	**význam**
terminál	řetězec tvořený daným znakem-terminálem
.	řetězec tvořený libovolným znakem (jediným)
\.	řetězec tvořený tečkou (escape sekvence)
[znaková-množina]	řetězec tvořený právě jedním znakem ze znakové množiny
[^znaková-množina]	řetězec tvořený libovolným znakem, který však nesmí být ze znak. množiny
*, \\, \^, \$	terminály (escape sekvence speciálních znaků) [v BRE i ERE]
\+, \?, \\|,	terminály [jen v ERE]
\[, \],	terminály (escape sekvence závorek) [v BRE i ERE]
\(, \), \{, \}	terminály (escape sekvence závorek) [jen v ERE]

Znaková množina je buď výčet znaků, znakový interval (oddělovačem dolní a horní meze je pomlčka) nebo libovolná kombinace znaků a intervalů. Znakové intervaly jsou zkratkou za výčet, jež počíná znakem dolní meze, a končí znakem horní meze (včetně) a obsahuje i znaky, jejichž kód leží mezi kódy obou mezních znaků (kódy podle aktuální znakové sady). Nejčastěji se používají intervaly z ASCII znakové sady (jež je podmnožinou většiny znakových sad používaných systémů) jako např. číslice 0-9, velká písmena latinské abecedy A-Z, apod. Při použití úplné sady Unicode jsou intervaly víceméně nepoužitelné, neboť pro většinu znaků nenabízí Unicode žádná rozumná uspořádání. Uvnitř znakové množiny není nutno používat escape-skevence (ani pro metaznaky), pouze znak „]“ musí být uveden na počátku znakové sady, znak „-“ na konci a znak „^“ kdekoliv kromě počátku.

Některé znakové množiny lze jen obtížně popsat kombinací výčtu a intervalů (především při použití znakové sady Unicode). Je to například množina všech číslených znaků (včetně arabsko-indických, čínských, atd. číslic), bílých znaků (kromě mezery, tabelátoru a odřádkovače je v Unicodu několik desítek dalších znaků s podobnou funkcí). Pro ty nejčastěji existují v základních a rozšířených regulárních výrazech speciální znakové množiny (označované jako posixovské znakové množiny) s jejichž zástupcem jsme se již setkali u filtru *cut*(1). Následující tabulky je shrnuje:

znak. množina	význam
[:alnum:]	alfanumerické znaky (*alpha*+*digit*)
[:alpha:]	písmenné znaky

znak. množina	význam
[:cntrl:]	řídící znaky (většina znaků s kódem 1-31)
[:digit:]	číslice
[:graph:]	tisknutelné znaky bez mezer (*print−space*)
[:lower:]	malé písmena (minusky)
[:print:]	tisknutelné znaky
[:punct:]	tisknutelné znaky, jež nejsou mezerou ani alfanumerickým znakem (*graph−alnum*)
[:space:]	mezerové (bílé) znaky
[:upper:]	velká písmena (verzálky)
[:xdigit:]	šestnáctkové (hexadecimální) číslice

Základní operací nad regulárními výrazy je **řetězení**. Zřetězením dvou regulárních výrazů vzniká nový regulární výraz, který popisuje jazyk, jehož slova vznikají zřetězením slov jazyka prvého výrazu se slovy druhého výrazu. Operace řetězení není specifikována žádným operátorem, pouze stačí zapsat dva regulární výrazy bezprostředně za sebe (na pořadí přirozené závisí, řetězení není komutativní operací). Operace i její zápis je tak intuitivní, že si většina uživatelů tuto operaci ani neuvědomuje (někdy se to však hodí, viz níže).

Příklad: ...

regulární výraz popisující jazyk všech tříznakových řetězců. Je to zřetězení tří elementárních výrazů tvořených metaznakem tečka (na pořadí nezáleží, řetězení je asociativní)

```
[A-Z]..[^0-9]
```

regulární výraz popisující jazyk čtyřznakových řetězců, jež začínají malým latinským písmenem a nekončí číslicí.

```
[[:lower:][:digit:]][[:space:]][^[:alnum:]]
```

regulární výraz popisující tříznakové řetězce, jejichž prvním znakem je malé písmeno nebo číslice (v rámci jedněch složených závorek lze použít i více posixovských množin, výsledkem je jejich sjednocení). Druhým znakem řetězce musí být mezerový (bílý znak), zde pozor na dvojité složené závorky, vnější jsou součástí metaznaku, vnitřní posixovské znakové množiny. Posledním znakem může být libovolný znak, vyjma alfanumerického (mezera, speciální nebo kontrolní znak).

Pomocí řetězení lze popisovat jen jazyky, jejichž řetězce mají pevný počet znaků (a které jsou vždy konečné). Při popisu jazyků s řetězci různých délek a jazyků potenciálně nekonečných je nutno používat tzv. opakovače (opakovací operátory, *repetitors*). Opakovače se aplikují na regulární výrazy (podvýrazy) a určují kolikrát se v celkovém řetězci mohou opakovat podřetězce popsané daným regulárním podvýrazem (jednotlivá opakování nemusí být stejná, stačí pokud každé z nich odpovídá regulárnímu podvýrazu). Aplikace opakovačů má vyšší prioritu než řetězení.

opakovač	počet opakování
*	žádné až nekonečně mnoho
?	žádné nebo jedno (volitelný výskyt)
+	jedno až nekonečně mnoho
{n,m}	*n* až *m* opakování (včetně)
{n,}	*n* až nekonečně mnoho opakování
{n}	právě *n* opakování

Základní regulární výrazy původně podporovaly pouze první opakovač (*). Novější verze poskytují i další opakovače, ale z důvodu zachování kompatibility musí být předcházeny zpětným lomítkem (u složených závorek obě závorky). Je to bohužel poněkud matoucí: u BRE (základní) je „+“ terminálem a „\+“ neterminálem (opakovačem), naopak u ERE ERE (rozšířené) je „+“ neterminálem (opakovačem) a „\+“ terminálem (znakem plus).

Příklad: `[a-z].[0-9]*`

jazyk který obsahuje 1) dvouznakové řetězce začínající malým písmenem, 2) alespoň tříznakové řetězce, jež začínají výše uvedeným dvojznakovým řetězcem a pokračují (dekadickou) číslicí s libovolným počtem číslic.

příklady řetězců jazyka: ax, a2, a23, a235, a#5669, aa256654, a 2636 (a-mezera-...)

protipříklady (řetězců, jež nejsou v jazyce): a, a2a, #a2, #235, a53a25

`([a-z].[0-9])*`

jazyk obsahující řetězce, jejichž délka je dělitelná třemi. Každý řetězec se skládá z posloupnosti trojznaků, v nichž je první znak minuskou, druhý může být libovolný a třetí je číslicí. Jednotlivé trojice nemusí být shodné. Do jazyka patří navíc i prázdný řetězec.

příklady řetězců: aa3, aa3b#5m22, aA2aA2

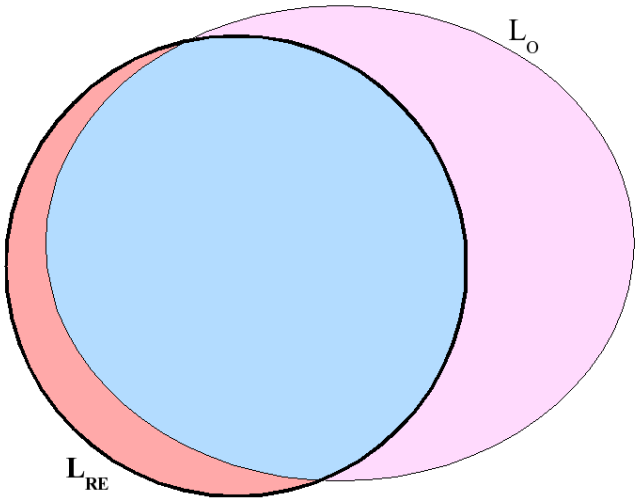
protipříklady: a, ax, axm, ax5bm, ax5666

Regulární výrazy se používají pro testování správného formátu vstupních informací, vyhledávání podřetězců s relavantní strukturou, resp. pro popis podřetězců, které budou substituovány. Aby tuto svou funkci plnily musí být zcela přesné tj. musí přesně specifikovat množinu přípustných řetězců (resp. vyhledávaných resp. substituovaných podřetězců).

Chybou je tedy jak regulární výraz, který nepopisuje všechny neočekávané řetězce (i když to může být jen zanedbatelné množství řetězců), tak i regulární výraz, který naopak popisuje i řetězce, které by do jazyka patřit neměly. V nejhorším případě může nastat i souběh obou chyb.

Problematiku vztahu očekávaného jazyka (L_O) a jazyka popsaného skutečným regulárním výrazem (L_{RE}) shrnuje obrázek 6.1 na následující straně (Vennův diagram).

Největší část zaujímá oblast průniku obou jazyků tj. řetězce, které jsme chtěli regulárním výrazem popsat a jež skuptečně do jazyka patří. Tmavá oblast vlevo obsahuje řetězce, které neměly být rozpoznávány (nehodí se do očekávaného jazyka), ale jsou regulárním výrazem přijaty (tj. např. při vstupu nebudou odmítnuty, resp. budou vyhledány resp. substituovány i když bychom nechtěli, apod.). Světlá oblast vpravo obsahuje naopak řetězce, které v jazyce očekáváme, ale nebudou regulárním výrazem



Obrázek 6.1: Očekávaný a skutečný jazyk popsaný regulárním jazykem

odmítnuty (tj. při vstupu budou odmítnuty i když jsou správné; nebudou vyhledány resp. substituovány, i když bychom chtěli; apod.).

Dosažení dokonalého stavu ($(L_O = L_{RE})$ není jednoduché a mnohdy spíš nemožné, neboť:

1. obě množiny bývají v praxi velmi velké (formálně často nekonečné, prakticky v řádech triliónů resp. vyšších). Otestovat všechny dotčené řetězce je proto zcela nemožné.
2. množinu $L_O$lze často jen s obtížemi popsat (tj. nelze ji vyjádřit několika větami v přirozeném jazyce nebo matemetickým výrazem rozumné délky). Někdy může být snadnější specifikace doplňkové množiny $\Sigma^* - L_O$, kde Σ^*je množina všech řetězců.
3. obtížné (i když o něco snadnější) je i zjištění jazyka, jež odpovídá danému regulárnímu výrazu na dostatečné úrovni abstrakce. Problémem zde není vlastní formální popis jazyka (tím je samotný regulární výraz), ale o jeho popis na úrovni popisu jazyka očekávaného (ten se jen výjimečně popisuje pomocí opakovačů, řetězení apod.)
4. každý jednotlivý jazyk (množina jazyků) může být obecně popsána větším počtem regulárních výrazů (ve skutečnosti nekonečným). Většina těchto jazyků jsou naštěstí umělé konstrukce (např. netriviální regulární výrazy popisující jazyk obsahující jediný řetězec), ale u složitějších jazyků může existovat více rozumných regulárních výrazů s obdobnou složitostí[8].

Jak tedy postupovat? Bohužel, jednoduché řešení neexistuje. Pomoci může především seznámení s teoriií gramatik a především konečných automatů. Nejužitečnější jsou však praktické zkušenosti, tj. vytváření a testování většího počtu regulárních výrazů.

Než však budeme moci diskutovat několik konkrétnějších příkladů je nutno uvést další důležitý operátor regulárních výrazů — operátor volby „|“. Tento operátor je binární a zapisuje se tudíž mezi dva regulární výrazy. Jazyk definovaný výsledným regulárním

[8]nedefinuji zde pojem rozumné reg. výrazy ani složitost reg. výrazů, snad postačí příklad: rozumný (méně složitý) = `[a-z]+`, nerozumný (zbytečně složitý) = `([abcd]|[zf]|[f-y]{1}){1,}`

výrazem je sjednocením jazyků, jež jsou popsány oběma operandy. Jinak řečeno řetězec musí odpovídat buď prvnímu operandu (regulárnímu podvýrazu) **nebo** druhému (neměl by však odpovídat oběma, viz níže). Operátor volby má nejnižší prioritu (menší než řetězení a samozřejmě i opakování). V základních regulárních výrazech nebyl původně k dispozici, většina programů využívajících původní syntaxi však nabízí zpětně kompatibilní tvar „`\|`“ (vztah terminál a neterminál je pak opět opačný než u ERE).

Operátor volby je velmi užitečný (a v mnoha případech zcela nezbytný), někdy však jeho použití vede k zbytečně složitým regulárním výrazům. Snažte se proto jeho použití omezit na nejmenší možnou míru a aplikovat je na co nejmenší podvýrazy.

S tímto požadavkem úzce souvisí i **základní omezení** související s operátory volby: při testování shody řetězce s regulárním výrazem (což je základní operace při libovolném použití regulárních výrazů) se v případě shody s operátorem volby musí rozhodnout podle prvního písmene, zda bude použit první operand nebo druhý. Jinak řečeno, první znak všech řetězců jazyku popsaného prvním operandem se musí lišit od všech prvních znaků řetězců jazyka duhého operandu.

Regulární výraz splňující tuto podmínku se nazývá deteministický, pokud ji neplňuje nedeterministický (úzce to souvisí s deterministickými a nedeterministickými automaty). I když existují implementace připouštějící i použití nedeterministických regulárních výrazů (nikoliv však všechny[9]), je jejich použití velmi neefektivní, neboť čas zpracování může v nejhorším případě růst až exponenciálně. Navíc každý nedeterministický regulární výraz lze přepsat na deterministický (důkaz, viz ekvivalence deterministického a nedeterministického konečného automatu). Jednoduše řečeno, **používejte jen deterministické regulární výrazy**.

Příklad: `koč(k([yu]|ou?|á(m|ch)|a(mi)?)|ek|ce)`

jazykem tohoto regulárního výrazu jsou všechny přípustné pádové tvary slova kočka. Výraz obsahuje minimum operátorů volby a ty jsou aplikovány na minimální podvýrazy. Navíc rozhodnutí pomocí jaké větve operátoru volby je popsán daný řetězec, lze učinit na základě prvého písmene (např. pokud po řetězci *kočk* příjde písmeno *a*, je jasné že zbytek řetězce musí být popsán regulárním podvýrazem „`a(mi)?`“), to jest je deterministický. Navíc je to jak nejkratší tak nejefektivnější regulární výraz pro daný jazyk, a měla by mu být dána přednost.
Regulární výraz tvaru `kočka|kočky|kočce|...|kočkami` není deterministický (podle prvního písmene nelze rozhodnout jakou větví jít)!

6.7.2 Praktické příklady regulárních výrazů

Nyní již můžeme přistoupit k rozsáhlejšímu praktickému příkladu. Naším úkolem bude vytvořit regulární výraz popisující všechny řetězce, které jsou dekadickými číselnými literály (obecně v Pascalských a C-jazycích bez různých typových prefixů a sufixů a bez zohlednění dosažitelného rozsahu nebo přesnosti).

Typové příklady řetězců jazyka: 129, -129, +129, 129., 129.5, .25, 2e3, 2E3, -2e3, 2e-3, 2.e+2

Typové příklady řetězců mimo jazyk[10]: . (tečka), -., e2, -e2, .e2, 2e, 2e+, 2-, 2.-, 2..3, prázdný řetězec

[9] např. jejich použití v programu *grep*, vede k podivnému chování

[10] výběr protipříkladů je omezen pouze řetězce obsahující znaky, jež je se mohou nacházet v řetězcových konstantách

Z příkladů lze vypozorovat že číslo se skládá z několika částí:

- znaménko (minus, resp. plus)
- celá část
- desetinná tečka
- desetinná část
- exponent
- znak e nebo E (příznak exponentu)
- znaménko exponentu (minus nebo plus)
- hodnota exponentu (celé číslo)

Při prvotním rozboru se může jevit, že všechny hlavní části čísla jsou nepovinné (opravdu každé z nich může chybět). To by však v důsledku znamenalo, že je lze vynechat všechny najednou tj. správným číslem by byl i prázdný řetězec. Dokonce i v případě, že nevynecháme vše, můžeme získat neplatné zápisy čísel (viz většina protipříkladů). Tudy cesta nevede.

Jedním z možných řešení složitějších problémů je jejich rozdělení na nezávislé části a následné řešení dílčích částí. V případě regulárních výrazů lze využít rozdělení regulárního výrazu na dílčí části a ty řešit jednotlivě. Bohužel dělení je nutno provést v okamžiku, kdy regulární výraz ještě neznáte. Naštěstí postačí, pokud dokážete v řetězcích daného jazyka najít oddělené části, které jsou na sobě zcela nezávislé, tj. změny ve tvaru jedné části se nikdy neprojevují na části druhé (speciálním případem změny je i vymizení dané části!). V našem konkrétním případě lze číselný literál rozdělit na mantisu a exponent.

Regulární výraz pro exponent je jednodušší, neboť jeho tvar není příliš variabilní (jedinou variabilitou je znaménko a velikost písmena „e“). Navíc lze exponent dále rozdělit na ještě jednodušší výrazy:

znak exponentu (malé nebo velké E): `[eE]` (akceptovatelný je i výraz `e|E`, je však méně efektivní)

znaménko (+,- nebo chybí): `[+-]?`

hodnota exponentu (neprázdná posloupnost dekadických čísel): `[0-9]+`

regulární výraz pro celý exponent: `[eE][+-]?[0-9]+`

U regulárního výrazu pro mantisu vyjděme z regulárního výrazu `[+-]?[0-9]+\.?[0-9]*` (neprázdná posloupnost číslic + tečka + posloupnost číslic). Tento regulární výraz je mírnou modifikací původní představy, že všechny části čísla jsou nepovinné (zde je nepovinná pouze tečka a desetinná část). Tento výraz však popisuje omezený jazyk neboť nepřipouští zápisy bez celé části (jako např. `.5`, `+.25` apod[11].). To však není jediný problém tohoto regulárního výrazu. Zamysleme se jak bude interpretováno např. číslo 999. Jednoznačná odpověď není bez znalosti implementace programu, jež bude regulární výraz vyhodnocovat[12], možná. Tři devítky mohou být interpertovány jako trojice číslic před (nepoužitou) desetinnou čárkou, nebo jako jedna číslice před (nepoužitou) desetinnou čárkou následovaná dvěma číslicemi v desetinné části (tečka je nepovinná), resp. jako dvě číslice v celé a jedna v desetinné části. U většiny programů pracujících s regulárními výrazy není jednoznačná interpretace nezbytná, pro přijetí řetězce postačuje pouze existence alespoň jedné interpretace. Dokonce ani v případě, že chceme

[11] mnoho progrmátorů tento zápis nepoužívá (včetně mě), ale programovací jazyky jej připouští

[12] tj. bude po předběžném zpracování regulárního výrazu rozhodovat, zda vstupní řetězec danému regulárnímu výrazu odpovídá tj. zda patří do jím posaného jazyka.

regulární výraz použít pro rozklad řetězce (tj. v našem případě na celou a desetinnou část) nedojde u většiny programů k chybě neboť opakovače jsou tzv. žravé tj. pojmou maximální počet znaků z řetězce (zde tedy první podvýraz [0-9]+ pojme všechny číslice celého čísla, což je správné).

Navzdory tomu nelze tyto nejednoznačné výrazy doporučit, neboť jednak vnáší do systému nedeterminovanost (nelze například obecně zaručit, že všechny programy budou používat jen a pouze žravé opakovače) a také mohou výrazně prodloužit čas nutný pro ověření shody řetězce s regulárním výrazem (program musí v jistých případech ověřit všechny možné interpretace).

Východiskem ke lepšímu řešení je skutečnost, že desetinná část a desetinná tečka spolu úzce souvisí, tj. není desetinné části bez desetinné tečky. Správnější zápis (i když o něco delší) má proto tvar:

```
[+-]?[0-9]+(\.[0-9]*)?          struktura RE: znaménko celá-část (desetinná-část)?
```

Tento zápis nicméně stále neřeší problém s čísly bez celé části (tj. začínajícími desetinnou tečkou). Náhrada opakovače + za * u první číslicové posloupnosti stále není možná (dovolovala by prázdné řetězce nebo číslo tvořené jen znaménkem).

Nezbývá tedy nic jiného než použít operátor volby a řešit čísla bez celé části jako speciální případ (který přímo nesouvisí s ostatními tvary mantisy):

```
[+-]?([0-9]+(\.[0-9]*)?|\.[0-9]+)struktura RE: znaménko (první-větev|druhá-větev)
```

Znaménko musí být vytknuto před obě variantní větve, neboť může být před oběma typy čísel (pokud by bylo zapsáno v obou větvích, nezískaly bychom deterministický regulární výraz). Závorky kolem celého variantního podvýrazu jsou nezbytné, neboť skládání má vyšší prioritu a znaménko by se tak počítalo jen první větvi. Naopak závorky kolem každé z větví nutné nejsou (můžete je však pro přehlednost použít, nesmíte však přitom zapomenout na povinné vnější). První větev popisuje čísla s celou částí (a nepovinnou destinnou) a je shodná s výrazem uvedeným výše (bez specifikace znaménka). Druhá větev popisuje čísla bez celé části, která musí začínat tečkou a musí obsahovat ve své desetinné části alespoň jednu číslici (samotná tečka není přípustná). Volba je deterministická, neboť první větev musí začínat libovolnou číslici a druhá tečkou (a tečka není číslicí).

Nyní už máme regulární výraz jak pro (povinnou) mantisu tak (nepovinný) exponent a výsledný regulární výraz vytvoříme jejich složením (exponent je navíc uzavřen celý do závorek a označen jako nepovinný):

```
[+-]?([0-9]+(\.[0-9]*)?|\.[0-9]+)([eE][+-]?[0-9]+)?
```

Výsledek není příliš přehledný, neboť v regulárním výrazu nelze použít mezer[13]. Důvodem je přirozeně skutečnost, že mezera by byla intrepretována jako terminál (a byla by na daném místě vyžadována i v číselných literálech).

V příkladu jsme použili přehlednější rozšířené syntaxe (ERE). Pomocí původní základní syntaxe (BRE) není možno regulární výraz zapsat (chyběl opakovač ? a operátor volby). V kompatibilním rozšíření základní syntaxe (BRE+) má výraz podobný tvar jako u ERE přibývá však escape sekvencí (opakovače +,? oblé závorky a operátor | musí být zapsány pomocí escape sekvencí aby byla zachována zpětná kompatibilita)

```
[+-]\?\([0-9]\+\(\.[0-9]*\)\?\|\.[0-9]\+\)\([eE][+-]\?[0-9]\+\)\?
```

[13] některé nástroje jako např. Perl umožňují použít tzv. volné syntaxe, u níž lze použít bílé znaky pro formátování (včetně odřádkování). Bohužel je to však jen výjimka potvrzující pravidlo.

Nyní je vám asi zcela jasné proč používáme syntaxi ERE. Pokud jste již nuceni použít syntaxi BRE+, navrhněte a otestuje regulární výraz pomocí rozšířené syntaxe a teprve poté přidejte nezbytná zpětná lomítka.

6.7.3 Kotvy

Nyní když již jste schopni tvorby a interpretace středně složitých regulárních výrazů, můžeme přistoupit k poslední syntaktické kategorii regulárních výrazů tzv. kotvám (*anchors*). Kotvy jsou metaznaky, jež se neshodují se znaky, ale s význačnými mezizna-kovými pozicemi.

Kotvy mají význam především u nástrojů, jež nejsou orientovány na kontrolu shody celého řetězce s regulárním výrazem (na což byly zaměřeny předchozí podkapitoly), ale na vyhledávání vzorů uvnitř rozsáhlejších textů (typicky řádků). Například program *grep*(1) funguje jako filtr, která čte řádky ze standardního vstupu a na standardní vstup vypisuje jen ty řádky, které obsahují podřetězec, jenž se shoduje s regulárním výrazem (regulární výraz je předán jako parametr příkazového řádku). Rozhodující tedy není shoda celého řádku, ale postačuje jen shoda libovolného podřetězce.

Například zavoláme-li program *grep* s parametrem (regulárním výrazem) `[0-9]`, nebu-dou vypsány jen řádky obsahující jeden číslicový znak (odřádkování se do řádku u *grepu* nikdy nepočítá), ale všechny řádky obsahující alespoň jednu číslici (na libovolné pozici).

Na množinou všech možných řádků (=řetězců bez odřádkování), které budou vypsány programem *grep*, se můžeme dívat jako na jazyk (jazyk = podmnožina množiny všech řetězec). Tento jazyk však není shodný s jazykem popsaným příslušným regulárním výrazem, ale je jeho vlastní nadmnožinou. Kromě řetězců původního jazyka obsahuje i řetězce, u nichž je před řetězcem původního jazyka libovolná posloupnost znaků resp. za tímto řetězcem libovolná posloupnost následuje.

Příklad: `grep '.'`

regulární výraz popisuje všechny jednoznakové řetězce. *Grep* však vypíše všechny řetězce, jež obsahují alespoň jeden znak (shodovat se bude vždy první znak v ře-tězci). Za znak se počítají i tabulátory a mezery (i když při výstupu nemusí být vidět).

`grep 'a*'`

regulární výraz popisuje jazyk, jenž obsahuje řetězce obsahující pouze písmena „a“ a prázdný řetězec. Grep však vypíše všechny vstupní řetězce (žádné neodfil-truje), neboť libovolný řetězec obsahuje jako podřetězec prázdný řetezec (nebo jinak každý řetězec obsahuje alespoň nula znaků „a“). Obecně je proto chybou používat u grepu regulární výrazy, které připouštějí i prázdné řetězce (a které zároveň nepoužívají kotvy).

`grep -E 'a+'` resp. `grep 'a\+'`

grep ponechá jen řádky obsahující alespoň jedno písmeno „a“. Grep s přepína-čem -E umožňuje používat rozšíření regulární výrazy (ERE). Původní verze (bez přepínač -E) podporovala jen základní syntaxi, dnes proto vyžaduje její kompa-tibilní rozšíření (BRE+). Pokud se však trochu zamyslíte, zjistíte že použití opa-kovače není nezbytné, neboť pro vynucení existence alespoň jediného „a“ stačí na místě regulárního výrazu použít jen terminál „a“.

`grep 'pes'`

vypíše všechny řádky obsahující podřetězec „pes“. Vypíše proto nejen řádky obsahující slovo „pes“, ale například i řádky se slovem *peskovat*, *pestík*, *vulpes* apod.

Kotvy umožňují přesněji specifikovat pozici vyhledávaného podřetězce uvnitř řádku rep. uvnitř prohledávaného řetězce a tak umožňují přesněji určit kontext podřetězce.

Dvě základní kotvy jsou podporovány ve všech typech regulárních výrazů:

^ – počátek řádku (prohledávaného řetězce). Pozice **před** prvním znakem.

$ – konec řádku (prohledávaného řetězce). Pozice **za** posledním znakem.

Příklad: `^root`

tento regulární výraz splňují všechny řádka (řetězce) začínající podřetězcem *root.* Za řetězcem root může být libovolný text (i prázdný).

`^[A-Z][a-z]+[[:space:]]+[0-9]{9}$`

výpis všech řádků, jež začínají slovem s počáteční verzálkou, poté následuje alespoň jeden mezerový znak (typicky mezera nebo tabulátor) a konec tvoří deviticiferná číslice. Podobnou strukturu mohou mít například záznamy v jednoduchém telefonním seznamu.
Regulární výraz je v tomto případě ukotven na obou stranách, a budou jej tudíž splňovat jen řádky, jež odpovídají regulárnímu výrazu jako celek. Tak lze pomocí kotev dosáhnout aplikace regulárního výrazu na celý řádek (nebo obecně vstupní řetězec) tj. nikoliv jen testování existence odpovídajícího podřetězce.

`^[[:space:]]*$`

regulární výraz, jež se shoduje s tzv. prázdnými řádky. Jako prázdné řádky se označují nejen řádky beze znaků (kromě případného koncového odřádkování) ale i řádky obsahující pouze mezerové (bílé znaky), neboť ty nelze od skutečně prázdných řádků vizuálně rozlišit.

Dalšími významnými meziznakovými pozicemi jsou hranice slov, kde slovo je definováno jako řetězec alfanumerických znaků resp. podtržítek[14]. Většinou leží hranice mezi alfanumerickým a nealfanumerickým znakem, výjimkou však je slovo na počátku nebo na konci řádku (řetězce), kde hranice slova splývá s počátkem řádku resp. jeho koncem.

Bohužel pro označení kotev na této pozici nepanuje úplná jednota. V zásadě však existují dva přístupy:

1) různé označení pro začátek (pozice před prvním znakem slova) a konec (pozice za koncem slova) slova:

\< – začátek slova

\> – konec slova

2) jediné označení pro obě hranice (což postačuje, neboť je lze v regulárním výrazu odlišit podle kontextu)

\b – hranice slova

Program *grep* podporuje obě možnosti (v BRE i ERE), Emacs pouze první, Perl a napodobovatelé (PRE) pouze druhou.

Příklad: `\bkoč(k([yu]|ou?|á(m|ch)|a(mi)?)|ek|ce)\b`

resp.

[14] slovo je však mnohdy definováno šířeji jako libovolná posloupnost znaků (zde v tomto významu používám důsledně slovo řetězec, aby nedocházelo k nejasnostem) resp. jako posloupnost nemezerových znaků (jak je tomu např. v shellu)

`\<koč(k([yu]|ou?|á(m|ch)|a(mi)?)|ek|ce)\>`

regulární výraz který při použití v programu *grep* najde všechny řádky pojednávající o kočkách (resp. v jiném režimu zjistí zda soubor kočky zmiňuje). Pokud však řádek (resp. soubor) pojednává jen o kočkodanech nebo kočkování nebude řádek (resp. u souboru žádný řádek) vypsán. V některých případech však nemusí být daný regulární výraz dostatečný, neboť ignoruje zmínky o kocourech a koťatech (i když i zde se jedná o jedince druhu *Felis cattus*).

6.7.4 Filtr grep

Krátký úvod do používání programu ***grep***(1) byl podán již v předchozí podkapitole. Základní funkcí programu *grep* je filtrování řádků podle regulárního výrazu. Ve standardním režimu tohoto programu jsou na standardní výstup vypisovány jen ty řádky, které obsahují podřetězec odpovídající regulárnímu výrazu (používajícím kompatibilní rozšíření základní syntaxe BRE+). Stejně jako u ostatních filtrů mohou být vstupní data přejímána nejen ze standardního vstupu, ale i přímo ze souborů (ty je nutno uvést jako parametry). Obecně má volání *grepu* následující tvar:

`grep [volby] regulární-výraz [soubory...]`

regulární výraz nemusí být uzavřen do uvozovek nebo apostrofů, ale pokud obsahuje znaky interpretované shellem musí být tyto chráněny. Proto je vhodné složitější regulární výrazy (tj. výrazy obsahující metaznaky) chránit pomocí apostrofů (uvozovky nemusí vždy stačit).

Funkci příkazu grep lze modifikovat resp. téměř zcela změnit pomocí relativně velkého počtu voleb. Ty nejdůležitější a nejpraktičtější shrnuje následující tabulka.

přepínač	**význam**
-c	místo výpisu odpovídajících řádku vypisuje jména souborů a počet řádků se shodou (i když je nulový)
-l	vypisuje jen jména souborů, u nichž byla alespoň jedna shoda s reg. výrazem (nepoužitelné u std. vstupu)
-i	nejsou rozlišována velká a malá písmena
-n	před každý vypsaný řádek uvede jeho pozici v souboru
-r	rekurzivně prohledá adresář, na každý soubor pak aplikuje vyhledávání reg. výrazu
-v	inverze výstupu (vypisovány resp. počítány jsou jen řádky, v nichž není podřetězec splňující reg. výraz)
-f *file*	regulární výrazy jsou čteny ze souboru (může jich být uvedeno více, oddělovačem je odřádkování)

Některé přepínače lze i kombinovat např. `grep -ric unix ~` vypíše jména všech souborů v domovském postromu a u každého z nich počet řádků, jež obsahují podřetězec „unix“ (nehledí se na velikost písmen). Podobně `grep -Ervl '.{16}' ~` vypíše všechny soubory domovského podstromu neobsahující řádek delší 16 znaků (regulární výraz používá ERE syntaxi).

Grep v klasickém Unixu používal základní syntaxi regulárních výrazů (BRE). Z důvodů kompatibility je tato syntaxe standardem i v linuxovském *grepu* (i když zde s možností použití kompatibilních rozšíření BRE+). Protože původní syntaxe měla omezenou vyjadřovací sílu, byla brzy vytvořena nová verze programu *grep* užívající rozšířenou syntaxi (ERE), jež byla pojmenována jako *egrep*[15]. Linuxovský *grep* (přesněji *grep* podle normy POSIX) již podporuje obě syntaxe, resp. i další syntaxe přímo, přičemž použitá syntaxe musí být signalizována pomocí volby (kromě BRE, jež je implicitní). Následující tabulky shrnuje přepínače a syntaxe podporované nejnovější verzí grepu:

přepínač	**syntaxe**
-G	BRE (resp. BRE+), není nutno uvádět, jelikož je implicitní
-E	ERE (rozšířená syntaxe)
-F	na místě reg. výrazu lze použít pouze řetězce (řádově až 10× rychlejší)
-P	syntaxe jazyka Perl (PRE), jen u nejnovějších implementací *grepu*)

6.7.5 Filtr sed

Filtr ***sed***(1) je tzv. proudový editor, tj. nástroj na dávkové editování textových proudů (na rozdíl od ručního editování vizuálního). Pomocí *sed* lze provádět velmi složité transformace textu (ovšem jen s omezenou pomocnou pamětí) a lze jím nahradit všechny speciální filtry uvedené v kapitole 6.3 (včetně *grepu*). Pro většinu složitějších trensformací je však vhodnější použít skriptovacích jazyků vyšší úrovně jako je Perl či Python. Zde se omezíme jen na jednoduché (jednořádkové) příkazy.

Jednoduchý příkaz filtru *sed* se skládá ze dvou částí — nepovinné adresy a vlastního příkazu, jehož jméno tvoří jediný znak (může však mít další parametry).

Jednotlivé řádky programu lze adresovat několika různými způsoby:

- absolutní čísla řádků (počítané od jedné)
- poslední řádek (označený symbolem $)
- regulární výraz (uzavřený mezi dvěma lomítky) – všechny řádky jejichž podřetězec odpovídá regulárnímu výrazu

Pro adresování souvislých posloupností řádků lze použít tzv. rozsahy, které jsou tvořeny dvěma adresami, jež jsou odděleny čárkou. První adresa určuje počátek rozsahu (první řádek), druhá pak označuje řádek poslední. Zásadním pravidlem je skutečnost, že rozsah popisuje vždy alespoň dva řádky. Tj. například v případě, že koncová adresa je udána regulárním výrazem, je tento aplikován poprvé na řádek bezprostředně následující za řádkem určeným počáteční adresou.

Sed pracuje v cyklu pro každý řádek vstupního proudu dat. Nejdříve je načten řádek do pracovního bufferu. Potom jsou na pracovní buffer aplikovány všechny příkazy[16] (pokud je před příkazem rozsah, jen tehdy, pokud je aktuální řádek v daném rozsahu). Nakonec je obsah bufferu vypsán (pokud není po provedení příkazů prázdný). Automatický výpis (autovýpis) lze potlačit volnou *-n* na příkazovém řádku (výpis je pak nutno vynutit explicitně například příkazem *p*).

[15] příkaz *egrep* lze použít i v Linuxu, kde se však jedná jen o symbolický odkaz na *grep*, jenž se při spuštění pod tímto jménem chová jako by byl použit s volbou -E

[16] některé se však mohou projevit až po výpisu

Sed poskytuje sadu několika desítek příkazů. V praxi se však používá jen několik (a ještě méně v jednořádkových příkazech). Následující tabulka je shrnuje:

příkaz	**význam**
q	quit – ukončení zpracování (následující řádky nejsou vypsány)
d	delete – vymaže pracovní buffer (nic nevystupuje ani při autovýpisu)
p	print – vypíše obsah pracovní bufferu
w *soubor*	write – zapíše obsah pracovního bufferu do souboru
s/RE/STR/PAR	substitute – nahrazení (substituce textu), viz dále

Příkazy pro sed lze zadávat buď přímo na příkazovém řádku (přepínač *-e příkazy*) nebo je lze připravit předem do textového souboru (tzv. sed-skriptu) a použít volbu *-f soubor-skriptu*.

Příklad: `sed -e '10q'`

vypíše jen prvních 10 řádků (jako *head*(1)). Všechny jsou vypsány v rámci autovýpisu. U desátého řádku je proveden příkaz *q(uit)*, jenž zpracování ukončí (pracovní buffer je však ještě vypsán).

`sed -ne '/^BEGIN/,/^END/w test'`

uloží do souboru test všechny řádky vstupu od řádku začínajím slovem BEGIN až po řádek začínající slovem END (resp. až do konce souboru není-li takový). Na standardní výstup není vypisováno nic. Pokud se po END vyskytne další BEGIN je nový tento nový úsek připojen k původnímu (soubor *test* se znovu neotvírá). Apostrofy jsou použity jako ochrana před shellem (zde pro ochranu nezbytné mezery po příkazu *w*, ale doporučuji je používat vždy).

```
sed -ne '/^BEGIN/,/^END/{
w test
p
}
/^END/q'
```

vylepšená verze výše uvedeného. Kromě zápisu do souboru jsou řádky mezi BEGIN a END vypisovány na standardní výstup (kde je může přejímat další filtr). Navíc výstup je ukončen po prvním řádku s END.
Příkaz již nelze napsat do jediného řádku (odřádkování je syntakticky významné), lze jej však napsat přímo v interaktivním shellu. Po odřádkování prvního řádku se totiž vypíše sekundární výzva (impl. tvar >), neboť řetězce uzavřený v apostrofech není dokončen. To se opakuje až do zadání posledního řádku, jenž očekávaný (ukončovací) apostrof obsahuje. Pohodlnější je však přirozeně editace pomocí textového editoru (jednorázově tak lze učinit pomocí Ctrl+X Ctrl+E).

Hlavním a nejužitečnějším příkazem editoru *sed* je substituční příkaz *s*. Jeho základní funkcí je nahrazení podřetězce určeného regulárním výrazem (syntaxe BRE+) jiným textem (určen běžným řetězcem). Z toho vyplývá základní tvar příkazu:

`s/regulární-výraz/řetězec/`

Nahrazující řetězec může být prázdný, v tomto případě je nalezený (= s regulárním výrazem se shodující) řetězec vyjmut bez náhrady.

Za poslední lomítko lze navíc psát tzv. modifikátory. Nejdůležitější shrnuje následující tabulka:

modifikátor	význam
g	nahradí všechny výskyty podřetězce (nejen ten první)
p	vypsání pracovního bufferu po substituci (zkratka za dva příkazy: *s* a *p*)
i	při aplikaci regulárního výrazu nerozhoduje velikost písmen

Příklad: `sed -e 's/Unix/Linux/g'`

nahrazení všech výskytů řetězce Unix ve vstupním proudu řetězcem Linux (na všech řádcích [chybí rozsah] a v každém řádku ve všech výskytech [modifikátor *g*]).

```
sed -e '/^[[:space:]]*$/s/^/==prazdny radek==/'
```

Všechny prázdné řádky ze standardního vstupu jsou na výstupu nahrazeny řádky s textem „==prazdny radek==“. Ostatní řádky jsou ponechány beze změny. První regulární výraz je adresou označující všechny prázdné řádky, poté následuje znak příkazu (s) a jeho parametry. Prvním parametrem je regulární výraz jež identifikuje nahrazovaný text (zde je to prázdné místo bezprostředně před začátkem řádku), druhý text, jenž bude na tuto pozici vložen.

```
find ~ -type f ! -empty -exec file '{}' \; | sed -ne
's/\(.*\):[^:]*\(relocatable\|executable\|data\|text\|document\|file\)[^:]*$/\1\n\2\n/ip'
```

zobrazení základního typu souboru u (téměř) všech souborů v domovském podstromu. Výstup u každého souboru obsahuje vždy nejdříve jméno souboru, pak na dalším řádku jeho základní typ (jednoslovný) a nakonec prázdnou řádků (pro snadnější vizuální orientaci).

Výpis standardní detailní informace pro každý soubor v domovském podstromě zajišťuje příkaz *find*(1), jehož akcí je příkaz *file*(1). Tento výstup je filtrován a upravován pomocí příkazu substituce. Regulární výraz[17] popisuje strukturu výstupu příkazu *file*: nejdříve je jméno souboru (v chráněných závorkách uvedený předpis .*), poté dvojtečka (bude to vždy ta poslední na řádku) a poté popis formátu souboru (text bez dvojtečky). Někde uvnitř tohoto předpisu je uvedeno jedno z klíčových slov *relocatable*, *executable*, . . . , *file* (pokud jich je v popisu uvedeno více, bude se řetězec shodovat pouze s nejlevější přípustnou alternativou). Všechny alternativy jsou uzavřeny do závorek (jenž zde neslouží pouze ke změně priorit, viz dále). Na konci regulárního výrazu je kotva konce řádku (ta je nutná, neboť jinak nelze zaručit, že dvojtečka je poslední na řádku tj. nepatří do jména souboru).

Po regulárním výrazu následuje (po lomítku) předpis náhrady (co bude místo původního vstupního řádku). Tento předpis obsahuje kromě znaků nového řádku (resp. jejich escape sekvencí) tzv. *zpětné odkazy* (*backreference*), které odkazují na ty části řetězce, jež se shodovali s regulárními podvýrazy v závorkách (u BRE+ musí být závorky opatřeny zpětnými lomítky). Na jednotlivé podřetězce (a tím i s nimi se shodující podvýrazy) se lze odkazovat pozičně, přičemž rozhodující je pozice první otvírací závorky. Tj. např. odkaz \1 se odkazuje na podřetězec určený první dvojicí závorek (zde je to jméno souboru jež se shoduje s podvýrazem .*), odkaz \2 na druhý (zde to jest jedno z alternativních klíčových slov).

Poslední částí příkazu substituce je modifikátor *i* (na velikost písmen není při aplikaci regulárního výrazu brán zřetel) a modifikátor *p* (vypsání řádku na němž byla provedena substituce, autovýpis je vypnut parametrem -n).

Jak bylo naznačeno výše řešení není dokonolé, neboť některé soubory nejsou

[17] užívá syntaxe BRE+

vypsány. Důvodem je především nedodržení pravidel pro zápis informací o formátech souborů – tj. nepoužití klíčových slov (ani klíčová slova *document* nebo *file*[18] nejsou zcela košer, ale vyskytují se bohužel dosti často). Dalším prohřeškem je použití dvojtečky v popisu, což prakticky znemožňuje rozlišení jména souboru.

ÚKOLY

1. Vyzkoušejte si zde uvedené příklady (!).

[18] informace, že soubor je souborem nebo dokumentem není příliš zajímavá (např. místo *MS Word Document* by bylo vhodnější použít např. zápis *MS Word document data*, neboť je tak jasně signalizováno, že soubor není textový)

7 Úvod do skriptů

7.1 Proměnné shellu

V předchozích kapitolách jsme používali shell jako interaktivní rozhraní pro přímé ovládání operačního systému. Shell lze však používat i jako neinteraktivní interpret příkazů uložených v textových souborech tzv. skriptů. Skripty však nejsou pouhými neinteraktivními dávkami příkazů, ale mohou obsahovat i základní řídící konstrukce jako jsou podmínky nebo cykly resp. mohou ukládat textová data do proměnných.

Nejdříve zaměříme pozornost právě na tyto proměnné. Z hlediska syntaxe a základní sémantiky se příliš neliší od proměnných jiných skriptovacích jazyků. V souladu se zaměřením shellu jsou však omezeny pouze na přechovávání řetězů (tj. nikoliv čísel či dokonce složitějších objektů).

Na rozdíl od specializovaných programovacích jazyků však není použití proměnných omezeno na uchovávání mezivýsledků a řízení běhů skriptů (dále je označuji jako lokální proměnné skriptu). Relativně velká skupina proměnných je nastavena již po spuštění shellu a definuje prostředí v němž běží jednotlivé skripty i další aplikační programy (napsané v jiných programovacích jazycích) tzv. proměnné prostředí. Další specifickou skupinou jsou konfigurační proměnné, které ovlivňují chování vlastního shellu (např. co je vypisováno v promptu, kde jsou hledány spustitelné soubory, atd.).

Pro obě poslední skupiny proměnných (nikoliv však pro lokální proměnné) je typické, že jsou tzv. exportovány tj. jsou viditelné i v procesech, jež jsou ze shellu spuštěny (bez ohledu zda se jedná o dceřinné shelly nebo jiné aplikace). Pro předání informací se používá mechanismu tzv. *environmentu*, jenž je spojen s každým procesem v systému a obsahuje textové hodnoty identifikované textovým identifikátorem. Tento *environment* může být na úrovni libovolného procesu nejen čten, ale i měněn či rozšiřován o další dvojice identifikátor–hodnot. Navíc je environment automaticky děděn všemi dceřinými procesy. Přenos informací pomocí environmentu je však vždy jednosměrný ve směru rodičovský proces ←dětský proces. Environment je obecným mechanismem (tj. přímo nesouvisí se shellem), je však téměř výhradně užíván pouze pro distribuci exportovaných proměnných shellu.

Moderní shelly poskytují desítky proměnných prostředí a konfiguračních proměnných. Nyní se seznámíme s těmi nejdůležitějšími:

COLUMNS	počet sloupců v aktuálním textovém terminálu
LINES	počet řádků aktuálního terminálu

DISPLAY	aktuální grafický displej systému X Window. Pokud je tato proměnná definována, lze spouštět X Window aplikace (tj. je k dispozici lokální či vzdálený terminál pro jejich zobrazení).
EDITOR	implicitní systémový editor (spouští jej některé příkazy, jež potřebují editaci např. konfiguračních souborů). Standardně je to *vi*(1)
HOME	domovský adresář
HOSTTYPE	architektura počítače (lze použít při testování podporovaných architektur např. v instalačních skriptech)
LANG	akuální jazykové nastavení (locales). Nejčastěji osbahuje kód jazyka, země a užívané kódování znaků. Pro češtinu a kódování UTF-8 má hodnotu
OSTYPE	typ operačního systému (standardizovaný). Pro Linux je to řetězce *gnu-linux*.
PS1	tvar primární výzvy (promptu). Kromě běžných znaků může obsahovat i escape sekvence, jež jsou při zobrazení substituovány za některé zajímavé informace.
PS2	sekundární výzva (pokračovací řádek). Standardně znak „>“.
SHELL	identifikace shellu (spustitelný soubor shellu)
SHLVL	hloubka zanoření shellu (počet rodičovských procesů, jež jsou také shellem + 1)
USER	přihlašovací jméno uživatele, jež vlastní danou instanci shellu

Mezi nejdůležitější proměnné prostředí patří i proměnné, jenž určují umístění důležitých systémových souborů jako je PATH či LD_LIBRARY_PATH (viz kapitola 5.3) nebo MANPATH (manuálové schránky) či MAILPATH (lokální schránka).

Hodnotu jednotlivých proměnných shellu lze vypsat na standardní výstup pomocí vestavěného příkazu *echo*(int), pro výpis hodnoty všech definovaných proměnných shellu je možno použít vestavěný příkaz *set*(int) (jenž primárně slouží ke konfigurování shellu, je-li však použit bez parametrů vypíše seznam proměnných).

Nejjednodušším způsobem vytvoření proměnné je přiřazení hodnoty pomocí zápisu `proměnná=hodnota`[1] (proměnná nemusí být předem definována). Takto vytvořená proměnná však není exportována, tj. je přístupná pouze z aktuálního shellu. Exportování již existující proměnné lze zajistit pomocí vestavěného příkazu ***export***(int), jehož parametrem je buď jméno jediné proměnné nebo seznam jmen (jednotlivá jména musí být oddělena mezerou). Přípustný je také zápis: `export proměnná=hodnota`, pomocí něhož můžeme proměnnou vytvořit, inicializovat a exportovat pomocí jediného příkazu.

Použití (čtení) proměnné umožňuje zápis `$proměnná`, který je substituován za hodnotu proměnné ve fázi nahrazování (tzv. *variable expansion*, provádí se ještě před nahrazením žolíků, spolu s ostatními substitucemi řízenými znakem $). Se substitucemi (nahrazeními), jež užívají znaku $ (vkládání výstupu[2], aritmetická substituce), sdílí substituce proměnných i další společnou vlastnost, jsouc prováděna i uvnitř uvozo-

[1] kolem znaku „=“ nesmí být mezery!
[2] včetně vkládání pomocí zpětných apostrofů

vek (nikoliv však uvnitř apostrofů). Alternativním zápisem vkládání proměnné je uzavření jména proměnné do složených závorek (tj. `${proměnná}`), jenž je nutno použít u některých víceznačných zápisů (například pokud by bezprostředně za identifikátorem proměnné měl následovat alfanumerický znak).

Příklad: `PS1='\u@\h:\w \$ '`

nastavení primární výzvy na promptu. Výzva bude vypisovat v prvé části uživatelské jméno [\u] a jméno počítače [\h] (bez doménové části) oddělené zavináčem. Ve druhé části výzvy po dvojtečce je vypsán aktuální pracovní adresář [\w] (celý, s využitím zpětné substituce domovského adresáře na vlnku). Výzva končí jak je zvykem znakem „$“ resp. „#“ (v případě superuživatele), což zajišťuje escape sekvence \$. Pokud by proměnná nebyla dříve definována a exportována měla by jen lokální platnost (tj. neprojevovala by se v podřízených shellech). Většinou je však definována (a exportována) již v okamžiku prvého zobrazení výzvy pomocí globálního inicializačního skriptu (viz níže).

`export EDITOR=emacs`

nastavení Emacsu jako systémového textového editoru. Pokud již byla proměnná definována a exportována jedná se pouze o přiřazení nové hodnoty. Pokud tomu tak není (což je běžné), zápis proměnou definuje a zároveň exportuje (což je nutné neboť proměnná je používána i v dceřinných shellech a ostatních aplikacích).

`users=$(grep -E "^[^:]+:[^:]*:[5-9][0-9][0-9]:" /etc/passwd | cut -d : -f 1)`

vložení seznamu běžných uživatelů do proměnné *users*. Předpokládá se, že běžní uživatelé maji UID mezi v rozsahu 500–999 (což u menších systémů při standardní politice přidělování UID platí).

`grep -E $(tr " " "|" <<< $users) /var/log/messages | less`

výpis všech řádků hlavního logovacího souboru, jež obsahují jména běžných uživatelů (tj. s jistou pravděpodobnostní se jich týkají[3]). Logovací soubor může číst pouze *root*.

Při nahrazování proměnných lze kromě jednoduchých zápisů použít i různé typy podmíněných zápisů:

`${jméno-řetězec}`
Pokud je proměnná nastavena[4], je substituována její hodnota, jinak je vložen řetězec. Vhodné v případě, kdy nelze zaručit, že proměnná je v okamžiku použití definována.

`${jméno:-řetězec}`
Pokud je proměnná nastavena a je neprázdná[5], je substituována její hodnota, jinak je vložen řetězec.

`${jméno=řetezec}`

Pokud je proměnná nastavena, je substituována, jinak je do ní vložen řetězec a ten je zároveň substituován.

`${jméno:=řetezec}`

Pokud je proměnná nastavena a je neprázdná, je substituována, jinak je do ní vložen řetězec a ten je zároveň substituován.

[3] relativně vysokou. pokud uživatelé nejsou identifikovány běžnými anglickými nebo unixovskými slovy (*is*,*open*,*daemon*,*root*, apod.) nebo jmény aplikací (démonů)
[4] nastavená = byla do ní alespoň jednou vložena hodnota
[5] do proměnné lze vložit i prázdný řetězec pomocí zápisu „*proměnná*=“

7.2 Skripty

Jako skripty (*scripts*) označujeme v Unixu textové soubory obsahující kód, jež může být interpretován (vykonán) shellem. Protože existuje více druhů shellů existuje i více druhů skriptů. V souladu s předchozími kapitolami se budeme zabývat skripty pro *Bourne Again Shell* (bash). Většina z nich je však kompatibilní i s ostatními shelly typu Bourne shell. V Linuxu se však můžete setkat i se skripty pro C-shell (avšak relativně řídce, zcela výjimečné jsou u systémových skriptů).

7.2.1 Spouštění skriptů

Hlavním (a historicky nejstarším) způsobem spouštění skriptů je aktivace nového dceřinného shellu, jemuž je na příkazovém řádku předáno jméno souboru se skriptem:

`bash soubor-skriptu`
respektive
`sh soubor-skriptu` (skript musí být kompatibilní s klasickým Bourne shellem)

Nevýhodou tohoto zápisu je skutečnost, že není shodný s voláním běžných (binárních, zkompilovaných) programů (ty se volají jen jménem), tj. uživatel je nucen rozlišovat skripty a binární programy.

Novější shelly (včetně všech Linuxovských) proto umožňují volat skripty přímo tj. jen uvedením jejich jména. Skript však musí splňovat dvě dodatečné podmínky:

- musí mít nastaveno právo ke spuštění (pro třídu uživatelů, do níž patří uživatel vlastnící proces shellu)
- prvním řádkem musí být speciální poznámka určující interpret (nemusí to být jen shell, postačí libovolný překladač interpretovaného jazyka např. Perl či Python)

Speciální poznámka na počátku skriptu musí začínat dvojznakem „#!“, jenž je bezprostředně následován jménem spustitelného souboru intrepretu (s absolutní cestou). V případě skriptů Bourne shellu má proto tvar:
`#!/bin/sh`
je-li vyžadován přímou bash (tj. používají-li se některé pro bash specifická rozšíření) pak tvar:
`#!/bin/bash`.

I při přímém spuštění pomocí hash-dong notace[6] je pro vykonání skriptu spuštěn nový shell. Jakékoliv definice či nastavení proměnných se proto po skončení provádění skriptu zapomenou[7] a nikterak neovlivní interaktivní shell (či shell provádějící jiný skript, z něhož byl daný skript zavolán).

Proto existuje ještě třetí způsob vyvolání skriptu, jenž zajistí že skript bude vykonán přímo volajícím shellem. Pro spuštění tímto způsobem lze využít vestavěný příkaz ***source***(int) nebo tzv. *dot*-notaci (před jménem skriptu se uvede tečka následovaná mezerou):

`source soubor-skriptu`

respektive

[6] anglický název podle běžného čtení dvojznaku #!
[7] exportování nepomůže, týká se jen směru rodičovský proces ←proces-potomek, nikoliv směru obráceného!

```
. soubor-skriptu
```

Tento způsob aktivace skriptu se používá pro skripty, jež definují proměnné prostředí resp. fungují jako jakési konfigurační soubory pro různé aplikace (konfigurační volby jsou de facto přiřazením do proměnných prostředí, které jsou následně zděděny vlastní aplikací).

7.2.2 Poziční parametry skriptu

Při vyvolání skriptu (jakýmkoliv výše uvedeným způsobem) lze předat na příkazové řádce neomezený počet parametrů, k nimž lze uvnitř skriptu přistupovat pomocí tzv. pozičních parametrů.

Poziční proměnné mají tvar `$n`, kde *n* je číslo od 1 do 9 (pozice se počínají od jedné) respektive `${n}`, kde *n* není shora omezeno. Poziční parametry lze použít kdekoliv v textu skriptu, nelze však do nich přiřazovat.

Počet parametrů je uložen v proměnné $#.

Seznam všech parametrů oddělených mezerou (či přesněji prvním znakem konfig. proměnné IFS) lze získat prostřednictvím proměnné `$*` resp. `$@`. Obě proměnné se liší jen v případě, jsou-li použity uvnitř uvozovek. Zápis `"$*"` se rozvine na ekvivalent `"$1 $2 ..."` (tj. na jediné slovo) naopak zápis `"$@"` na `"$1" "$2" ...` (tj. na více slov).

Speciální význam má parametr $0, jenž je substituován na jméno pod nímž byl skript spuštěn. Obecně to nemusí být jméno souboru v němž je uložen, neboť může být spuštěn i prostřednictvím symbolického odkazu.

7.2.3 Podmínky

Podmíněné konstrukce (příkaz *if*, cyklus *while*, apod) umožňují řídit běh skriptu na základě návratových hodnot programů. Tyto programy hrají v těchto konstrukcích roli podmínek, jež jsou splněny (pravdivé), pokud příkaz skončí úspěšně (s návratovou hodnotou 0), a nesplněny při neúspěšném ukončení (návratová hodnota větší než nula).

I když lze na místě podmínky použít libovolný příkaz, existuje skupina programů, které jsou právě pro podmínky určeny. Tyto příkazy nic nevypisují ani nečtou (a nemění ani žádné další prostředky systému) a tak jediným projevem jejich činnosti je návratová hodnota.

Velmi jednoduché jsou programy *true*(1) a *false*(1), z nichž první skončí vždy s úspěchem a druhý neúspěšně (nic jiného nedělají). V některých případech jsou však nenahraditelné[8] (např. pro zápis nekonečného cyklu). Bohužel jejich použití není příliš efektivní, neboť jejich velikost je přibližně 14KB (neboť mimo jiné podporují i dvě standardní GNU volby) a dynamicky linkují (mega)knihovnu *libc*. Proto je vhodnější použít vestavěný příkaz : (ano jeho název je dvojtečka), jenž nedělá nic a to úspěšně (jako *true*)

Pro testování řetězcových, číselných a souborových podmínek slouží příkaz ***test***(1). Tento příkaz přijímá na příkazovém řádku řetězcové či číselné parametry, spojené parametry ve funkci relačních a logických operátorů resp. unární či binární predikáty nad soubory, jejichž operandy jsou jména souborů (predikáty i jména souborů jsou

[8] není však přirozeně těžké je napsat např. v jazyce C

opět parametry příkazu *test)*. Jednotlivé parametry (tj. operátory i operandy testovaných podmínek) musí být od sebe odděleny alespoň jednou mezerou.

Nejdříve uvedeme výběr těch nejdůležitějších operací (predikátů) nad soubory (úplný přehled viz manuálové stránky):

-d soubor	pravdivý, pokud soubor existuje a je adresářem
-f soubor	pravdivý, pokud soubor existuje a je běžným (regulárním) souborem
-L soubor	pravdivý, pokud soubor existuje a je symbolickým odkazem
-s soubor	pravdivý, pokud soubor existuje a má nenulovou velikost
-r soubor	pravdivý, pokud soubor existuje a lze z něj číst (tj. skript má právo čtení)
-w soubor	pravdivý, pokud soubor existuje a lze do něj zapisovat (tj. skript má právo zápisu)
-x soubor	pravdivý, pokud soubor existuje a skript má právo *execute* (reg. soubory lze spouštět)
soubor1 -nt soubor2	pravdivý, když je *soubor1* novější než *soubor2*
soubor1 -ef soubor2	pravdivý, když jsou soubory identické (tj. jsou na stejném zařízení a mají stejné i-číslo). Užitečné, je-li alespoň jeden soubor odkazem.

Další skupinu operací tvoří řetězcové relace, rovnosti (==) a nerovnosti (!=). Tyto běžné relace však nelze použít pro tetsování prázdných řetězců (včetně prázdných proměnných), neboť ty se v shellu rozvinou na prázdné místo a v porovnání tak chybí operand. Místo nich je však možno použít predikátu **-z** *řetězec*, jenž je pravdivý pro prázdný řetězec.

Další skupinou jsou aritmetické (číselné) relace, které se vyjadřují dvojznaky ve stylu Fortranu (nezaměňovat s řetězcovými porovnáními).

$n = m$ -eq

$n \neq m$ -ne

$n < m$ -lt

$n > m$ -gt

$n \leq m$ -le

$n \geq m$ -ge

Poslední skupinou operátorů jsou logické spojky, které mají stejný tvar jako u příkazu *find*(1), tj ! pro negaci, *-a* pro logický součin (and) a *-o* pro logický součet (or).

Některé moderní shelly (včetně *bash*) mají speciální zkrácený zápis pro volání příkazu *test*. Místo zápisu `test prametry`, lze v nich použít zápis `[␣parametry␣]` (mezery za otvírací a před zavírací závorkou jsou povinné).

7.2.4 Hlavní programové konstrukce

Pro řízení toku zpracování ve skriptech lze využít běžných programových konstrukcí jako je podmíněný příkaz a různé cykly. Sémanticky jsou shodné s obdobnými konstrukcemi ostatních imperativních jazyků, syntaxe je však poněkud svérazná (alespoň z pohledu programátorů v jazyku C a jeho následovnících).

Podmíněný příkaz

```
if příkaz-podmínka
then příkazy
[ elif příkaz-podmínka
then příkazy]...
[ else příkazy ]
fi
```

Příkaz *if* začíná klíčovým slovem `if` následovaným příkazem ve funci podmínky (viz předchozí podkapitola) a končí klíčovým slovem `fi` (if obráceně). Všechna výše uvedená odřádkování jsou povinná, lze je však nahradit středníkem (tj. celou konstrukci lze uvést i na jediné řádce).

Kromě základní větvě (za prvním `then`), může obsahovat několik větví *elif* (zkratka za *else if*) a nejvýše jednu větev *else*. Sémantika je zřejmá, nejdříve je vykonán příkaz v podmínce if, skončí-li s úspěchem (= je pravdivý) provedou se pouze příkazy ve větvi za prvním `then`. Skončí-li neúspěšně, jsou postupně vyhodnocovány příkazy podmínky za jednotlivými *elif* (samozřejmě jen pokud existují). Jestliže některý z nich skončí s úspěchem, jsou vykonány příkazy v příslušné větvi (za *then*). Skončí-li všechny příkazy na místě podmínek neúspěchem a existuje-li větev *else*, pak jsou provedeny její příkazy.

Shell nevyžaduje žádné speciální vertikální zarovnání jednotlivých větví, vhodné odsazování však může skript učinit čitelnějším.

Místo některých jednoduchých podmíněných konstrukcí s jedinou větví, lze použít podmíněné spuštění aplikací za použití oddělovačů && resp. || (viz kapitola 5.3 na straně 52).

Cykly

```
while příkaz-podmínka
do příkazy
done
```

Syntaxe i sémantika cyklu *while* je obdobná ostatním jazykům. Specialitou je pouze zakončení těla cyklu klíčovým slovem `done`. Mírnou modifikací cyklu while je cyklus until, jenž se liší pouze v interpretaci podmínky (cyklus je ukončen při prvním úspěšném vyvolání příkazu tj. pravdě).

Druhým hlavním cyklem Bourne shellu je cyklus *for* (někdy podle své funkce označovaný jako *foreach*)

```
for název-proměnné in seznam-slov ...
do příkazy
done
```

Seznam je tvořen řetězcem, v němž jsou jednotlivá slova oddělena bílými znaky (přesněji libovolným oddělovačem z proměnné IFS). Cyklus postupně prochází slovo za slovem a v každé iteraci vkládá aktuální slovo do proměnné, jejíž název je uveden za klíčovým slovem `for` (název je bez prefixu $).

Seznamem může být libovolný literál, jenž je však podroben většině běžných nahrazení. Důležitá jsou především souborová nahrazení (žolíky jsou nahrazeny seznamem jmen odpovídajících souborů, přes než pak cyklus iteruje) a vkládání výstupu (cyklus iteruje přes slova nebo přesněji řádky výstupu z nějakého příkazu).

Kromě cyklu *foreach* existuje v shellu bash (nikoliv však ve starších) i aritmetický cyklus *for*, jenž je přímou obdobou stejnojmenné konstrukce jazyka C (samozřejmě až na vymezení těla cyklu).

Oba cykly lze předčasně ukončit za použití příkazu ***break***(int). Tento příkaz může mít navíc nepovinný číslelný parametr, jenž určuje kolik vnořených cyklů bude opuštěno (implicitně je přirozeně opuštěn jen jeden, ten nejvnitřnější).

Vstup a výstup na úrovni shellu

Typickou vlastností programování na úrovni shellu, je skutečnost, že většinu činností provádějí externí programy (nikoliv tedy shell sám). To se týká i vstupně/výstupních operací, jež jsou z valné většiny prováděny prostřednictvím externích programů. Zcela zásadně to platí především pro vstup ze souborů a výstup do nich. U interaktivního standardního výstupu a především vstupu to však platí jen zčásti.

Pro výstup textových informací na úrovni shellu se používá vestavěný příkaz ***echo***(int). Tanto příkaz vypisuje všechny své parametry na standardní vstup (oddělené mezerou). Navíc celý výstup ukončuje odřádkováním. Automatické odřádkování lze zakázat uvedením volby *-n*. Další volba *-e* povoluje interpretaci běžných escape sekvencí (v zásadě stejných jako v jazyce C).

Příkaz *echo* vypisuje na standardní vstup, ale za použití běžných přesměrování lze jeho výstup přesměrovat např. na výstup chybový (`echo >&2`) nebo do souboru.

Pro formátovaný výstup lze použít vestavěný příkaz ***printf***(int). Prvním parametrem tohoto příkazu je formátovací řetězec, který je obdobou formátovacího řetězce knihovní funkce jazyka C *printf*(3). Podporovány jsou všechny hlavní popisovače pro celá čísla, řetězce a znaky včetně specifikací zarovnání a šířky polí. Na rozdíl od jazyka C nehrají příliš velkou roli specifikace datových typů (prakticky vždy lze použít specifikaci „s“ pro řetězce, předpony typů se nepoužívají). Pomocí popisovače %q lze zobrazovat řetězce, u nichž jsou nebezpečné znaky chráněny před shellem (kvotovány).

Základním prostředkem vstupu na úrovni skriptu je příkaz ***read***(int). Tento příkaz pozastaví běh skriptu a po zadání textu a stisku *Return*, uloží tento text do proměnné, jejíž jméno je parametrem příkazu (do proměnné je zapisováno a je tedy uvedena bez prefixu $). Jestliže je však použito na příkazovém řádků vícero proměnných, vloží se postupně do každé z nich jedno slovo ze vstupu (slova jsou oddělena bílými znaky resp. znaky z proměnné IFS). Případný přebývající text je vložen do poslední proměnné.

Vestavěný příkaz *read* poskytuje několik velmi užitečných voleb, jmenovitě:

-n *n*	read přečte maximálně n znaků (užitečně pro $n = 1$, kdy po zadání znaku nemusí uživatel odřádkovat)
-t *sec*	nezadá-li uživatel vstup do daného počtu sekund, skončí příkaz s neúspěchem
-p *text*	vypsání výzvy (promptu) *text*
-e	při vstupu se použije knihovna readline (tj. je možná stejná řádková editace jako v shellu)
-d znak	vstup je ukončen až znakem *znak* (nikoliv odřádkováním). Ukončovací znak již není uložen do proměnné.
-s	vstupující text není zobrazován (tj. je vypnuto tzv. *echo*)

7.3 Ukázky skriptů

7.3.1 Upovídané kopírování

Standardní příkaz pro kopírování *cp*(1) vypisuje stručný výstup pouze při chybě. Navíc za chybu nepovažuje přepsání cílového souboru (přepíše jej bez jakéhokoliv varování). Toto chování lze sice změnit pomocí několika přepínači (-i, -v), ale ani tak to není dokonalé. Texty jsou stručné (a často jen v angličtině), při přepisování nelze pozbat zda jsou soubory stejné, jaký z nich je mladší apod. Nic vám však nebrání vytvořit si pro kopírování vlastní skript. Následující výpis ukazuje jedno z možných řešení, jenž je však pro zjednodušení omezeno pouze na kopírování jediného souboru do adresáře.

```
1  #!/bin/sh

3  if [ $# -ne 2 ]
   then
5      echo "Pouziti:␣xcp␣soubor␣adresar" >&2
       exit 3
7  fi

9  source=$1
   shortsrc=$(basename $1)
11 dir=$2
   target="$dir/$shortsrc"
13 ok=true

15 if [ ! -f "$source" ]
   then
17     echo "Soubor␣$source␣neexistuje␣(resp.␣neni␣beznym␣souborem)" >&2
       ok=false
19 elif [ ! -r "$source" ]
   then
21     echo "Soubor␣$source␣nemate␣pravo␣cist" >&2
       ok=false
23 elif [ ! -w "$dir" ]
   then
```

```
25      echo "Nemate␣pravo␣zapisovat␣do␣adresare␣$dir" >&2
        ok=false
27  elif [ -d "$target" ]
    then
29      echo "V␣adresari␣$dir␣existuje␣adresar␣stejneho␣jmena␣jako␣kopirovany␣soubor"
        ok=false
31  fi

33  if [ $ok == "false" ]
    then
35      echo "Soubor␣nemohl␣byt␣zkopirovan!" >&2
        exit 1
37  fi

39  if [ -f "$target" ]
    then
41      echo "Cilovy␣soubor␣$target␣jiz␣existuje:"
        echo -n "------------------------------------------------------"
43      echo    "--------------------------------"
        printf "␣␣␣␣␣%35s\t%35s\n"   "ZDROJOVY␣SOUBOR" "CILOVY␣SOUBOR"
45      srcsize=$(stat -c "%s" "$source")
        trgsize=$(stat -c "%s" "$target")
47      printf "vel:␣%35d\t%35d\n"  $srcsize          $trgsize
        srcmt=$(stat -c "%y" "$source" | sed -e 's/:[0-9]\+\.[0-9]\+//')
49      trgmt=$(stat -c "%y" "$target" | sed -e 's/:[0-9]\+\.[0-9]\+//')
        printf "cas:␣%35s\t%35s\n"  "$srcmt"           "$trgmt"
51      srcsum=$(md5sum "$source" | cut -f 1 -d "␣")
        trgsum=$(md5sum "$target" | cut -f 1 -d "␣")
53      printf "md5:␣%35s\t%35s\n"  $srcsum          $trgsum
        echo -n "------------------------------------------------------"
55      echo    "--------------------------------"
        echo -n "Chcete␣cilovy␣soubor␣prepsat␣?␣[a/n]␣"
57      read -sn 1 odpoved
        echo
59      if [ "$odpoved" != "a" -a "$odpoved" != "A" ]
        then
61          echo "Soubor␣nebyl␣prepsan"
            exit 2
63      fi
    fi
65
    errormsg=`cp␣"$source"␣"$target"␣2>&1`
67  if [ $? -eq 0 ]
    then
69     echo "kopirovani:␣$source␣->␣$target␣(OK)"
    else
71     echo "Pri␣kopirovani␣doslo␣k␣chybe␣(systemova␣zprava:␣$errormsg)"
       echo "Soubor␣nemohl␣byt␣zkopirovan!" >&2
73  fi
```

Skript lze rozdělit do několika částí oddělených vždy prázdnou řádkou. První část (nepočítáme-li identifikaci interpretu ve speciální poznámce na první řádce) ošetřuje

počet zadaných parametrů. Je-li různý od dvou, je vypsán krátký informační řetezec[9] se vzorem volání skriptu a skript je ukončen (s návratovou hodnotou rovnou 3).

Ve druhé části jsou inicializovány všeobecně používané proměnné.

7.3.2 Výskyt slov v souborech

Skripty lze použít i pro zobrazení základních statistik o souborech. Následující skript prochází textové soubory v adresáři (první parametr skriptu) a pro každý z nich zjišťuje výskyt jistých slov (zbývající parametry skriptu). Mírou kterou vypisuje, je podíl počtu řádku s daným slovem k počtu neprázdných řádků vyjádřený v promilích (desetinách procenta). Tímto způsobem lze vyhledávat soubory, jež s jistou pravděpodobnostní pojednávají o tématu, pro něž jsou vyhledávaná slova klíčová. Skript však není příliš rychlý a použitelný je pouze pro omezený počet prohledávaných souborů (stovky, tisíce).

```
#!/bin/sh

dir=$1
shift

for file in $(find "$dir" -type f ! -empty -size -265k )
do
  file $file | grep 'text[^:]*$' >/dev/null 2>/dev/null || continue
  lines=$(grep -E -c [^[:space:]] "$file" 2>/dev/null)
  for text in "$@"
  do
    count=$(grep -E -c "$text" "$file" 2>/dev/null)
    if [ $count -gt 0 ]
    then
        quot=$(( 1000 * $count / $lines ))
        printf "%d\t%-10s\t%-15s\n" $quot "$text" "$file"
    fi
  done
done | sort -rn
```

7.3.3 Hlídač procesů

Pro zobrazení dynamiky procesů se standardně používá program *top*(1). Pokud však chcete sledovat jen vznik a zánik procesů, není tento program příliš vhodný (nové resp. zaniklé procesy se nesnadno identifikují). Nic vám však nebrání napsat si skript, který bude informace o nově vzniklých a zaniklých procesech zobrazovat přehledně.

```
#!/bin/sh

trap "rm␣/tmp/{pre,old,new}ps.$$␣2>/dev/null" EXIT

ps -eo comm= > /tmp/preps.$$
sort /tmp/preps.$$ | uniq > /tmp/oldps.$$
```

[9] výstup příkazu echo je přesměrován na standardní chybový výstup.

```
7
while true
9 do
  ps -eo comm= > /tmp/preps.$$
11  sort /tmp/preps.$$ | uniq > /tmp/newps.$$
  closproc=$(diff -d /tmp/oldps.$$ /tmp/newps.$$ | grep '^<' | tr -d "<\n" |
13    sed -e 's/␣/,/g
␣␣␣␣␣␣␣␣␣␣␣␣␣s/^,//')
15  openproc=$(diff -d /tmp/oldps.$$ /tmp/newps.$$ | grep '^>' | tr -d ">\n" |
    sed -e 's/␣/,/g
17 ␣␣␣␣␣␣␣␣␣␣␣␣␣s/^,//')
  [ -z "$openproc" -a -z "$closproc" ] || echo "Change␣at:" $(date +"%H:%M:%S")
19  [ -z $openproc ] || echo "␣New␣opened:␣$openproc"
  [ -z $closproc ] || echo "␣New␣closed:␣$closproc"
21  mv /tmp/newps.$$ /tmp/oldps.$$
  sleep 10
23 done
```

Základní myšlenka tohoto skriptu je jednoduchá. V jistých intervalech (zde 10 vteřin) se do dočasného souboru vloží výpis aktuálních procesů (pro přehlednost je jsou programy s více instancemi počítány jen jednou).

ÚKOLY

1. Vyzkoušejte si zde uvedené příklady (!).
2. Napište skript, který prohledá zadané adresáře a vygeneruje html stránku s náhledy obrázků v daném adresáři. Pro tvorbu náhledů použijte program *imagemagick*.

8 Služby OS

8.1 Úlohy (jobs)

Klasický Unix byl sice multitáskovým systémem, jeho shell však podporoval pouze postupné (sériové nikoliv paralelní) vykonávání interaktivních procesů. I když bylo možno spouštět i příkazy na pozadí, ty nesměly s terminálem komunikovat. Jinak řečeno po spuštění interaktivního procesu byl terminál tímto procesem okupován, a uvolněn byl až po jeho ukončení. Podobně se až dodnes chová například shell u MS Windows.

Po vzniku celoobrazovkových aplikací (např. editorů) však tento přístup omezoval jejich použití, neboť po dobu jejich běhu nešlo spouštět další příkazy (resp. pouze přes jejich vestavěné shelly). Proto byl v C-shellu zaveden mechanismus tzv. úloh, které tento problém řešily a navíc umožňovaly snadnější správu úloh tvořených více procesy tj. kolon. V současnosti, kdy je používáno okenní GUI rozhraní, význam úloh poněkud poklesl, stále jsou však užitečné (a navíc mnohé servery nemají z bezpečnostních důvodů GUI rozhraní aktivované).

Po spuštění kolony je vytvořeny tzv. úloha, která sdružuje všechny procesy, jež jsou v koloně uvedeny. Úloha je však vytvořena i při spuštění jednoduchých příkazů (úloha pak obsahuje jen jediný proces). Každá úloha je v shellu identifikována malým přirozeným číslem (job ID)[1].

Úlohy se vzhledem k terminálu nacházejí ve dvou stavech. Úloha na popředí (vždy jen jediná) je jako jediná oprávněná zapisovat nebo číst z terminálu. Úlohy na pozadí jsou v případě pokusu o zápis nebo čtení terminálu pozastaveny (*stopped*) tj. jsou pozastaveny všechny jejich procesy.

Úloha je implicitně spustěna na popředí (její stav však lze změnit). Pro spuštění úlohy, jež prvotně běží resp. je pozastavena na pozadí, lze použít „&“ [ampersand] (stejně jako u procesů, zde se jen nacházíme na vyšší úrovni abstrakce).

Pro výpis aktuálních úloh daného shellu lze použít interní příkaz ***jobs***(int). Výpis obsahuje jen úlohy na pozadí (žádná úloha na popředí v danou chvíli neexistuje, neboť terminál využívá shell) a u každé uvádí kromě jejího identifikátoru i stav (Running - běží na pozadí, není pozastaven, Stopped - je pozastaven) a příkaz jímž byla vyvolána.

Úlohu, jež běží na pozadí lze do popředí přesunout vestavěným příkazem ***fg***(int) (*foreground*), jehož parametrem je znak procento následované číslem úlohy. Pokud není číslo úlohy uvedeno, je do popředí přesunuta tzv. aktuální úloha, což je úloha, která byla na popředí naposledy resp. byla spuštěna jako poslední (ve výpisu *jobs* je uvedena se znakem +). I když lze na popředí přesunout libovolnou úlohu, má to smysl

[1] úlohy jsou na rozdíl od procesů vázány na konkrétní shell (instanci shellu). Každý shell má vlastní množinu úloh a úloha mimo svůj domovský shell nemá smysl.

jen u úloh pozastavených (běžící vůbec nekomunikují a pravděpodobně vůbec nechtějí komunikovat s terminálem, typicky jsou to GUI aplikace).

Pro přesun úlohy z popředí na pozadí lze použít řídící klávesu Ctrl+Z (závisí na nastavení terminálu, pozor na českou resp. anglickou klávesnici). Pokud úloha komunikuje s terminálem je po přesunu zároveň i pozastavena (tj. např. její výpis se přeruší).

S čísly úloh se můžete setkat i dalších situacích. Například při změně stavu úlohy (včetně ukončení) je vypisováno příslušné hlášení (standardně však až po následujícím promptu). Všechny procesy v úloze lze najednou zabít, pokud ve vestavěném příkazu *kill*(int) použijete místo čísla procesu identifikátor úlohy (začínající znakem procento).

8.1.1 Démoni

Démon je program, který běží v pozadí, čeká na události, které nastanou, reaguje na ně a poskytuje služby. Konfigurace démonů je částečně závislá na Vaší distribuci. Proto si to uvedeme pouze bodově.

Spouštění démoni se obvykle nachází v souboru *rc.conf* na řádku uvedeným *DAEMONS, např. DAEMONS=(syslog-ng network netfs crond).*

V adresáři /etc/rc.d/ se nachází skripty pro spuštění služeb, typické použití je */etc/rc.d/služba {start|stop|restart}.*

Zde si uveďme jen některé démony

- CUPS - tiskový démon
- CRON - démon pro plánované spouštění úloh
- HALL - správa zařízení
- ALSA-podpora zvuku

ÚKOLY

1. Nastavte démon CRON tak, aby každý první den v měsíci prováděl zálohu souborů. Použijte k tomu některý ze skriptů naprogramovaných dříve. Nastavení lze provést pomocí *crontab*.

9 Síťové služby OS

Pro nastavování síťových rozhraní se používá příkaz *ifconfig*, případně je možné použít příkazy *ip* a *route*. První pro nastavení IP adresy na zařízení, druhý pro nastavení směrování. Uveďme si některé ukázky použití.

- *ifconfig* - výpis informací o všech síťových zařízeních
- *ifconfig eth0 10.0.0.1 up* -nastaví zařízení eth0 na IP adresu 10.0.0.1
- *ifconfig eth0 down* - vypnutí zařízení eth0

Pro přidělení IP adresy na straně klienta se používá příkaz *dhclient*. Pro práci se bezdrátovými sítěmi je celá řada nástrojů, uveďme si jen některé z nich

- *iwconfig* -podobně jako ifconfig, ale pro wifi zařízení
- *iwlist*-skenování dostupných sítí, např. *iwlist eth1 scan*

9.1 SSH

Pro bezpečné používání komunikace je nutné používat zabezpečený komunikační kanál, zde se zaměříme na SSH (Secure Shell).

Seznamte se s programem ssh, zaměřte se na:

- Použití programu *ssh* a jeho základní volby
- Tunelování a forwarding portů
- Autentizace pomocí veřejného klíče a jeho generování
- Přenos souborů pomocí příkazu scp
- Vyzkoušejte si instalaci a konfiguraci démona *sshd* na vaše zařízení.

9.2 Web server

Pro provozování webu je třeba mít nainstalovaný webserver. Nejznámější je jistě Apache, ale existují i další např. NGINX, Lighttpd, či Cherokee. V dalším se zaměřme na instalaci Apache. Jednotlivosti se mohou lišit, ale pro smysluplné používání potřebujete mít nainstalovaný Apache s vhodnou databází (MySQL) a např. PHP. Tato kombinace se nazývá LAMP (Linux, Apache,MySQL,PHP). Protože se jedná o častou úlohu, existují přímo nástroje pro její řešení. Ve světě Ubuntu existuje nástroj *tasksel*, který obsahuje konfigurace pro řešení nejčastějších úloh jako jsou instalace LAMP serveru, DNS serveru, OpenSSh (viz výše). V jiných ditribucích je potřeba použít alternativní nástroje, případně použít následující sekvenci instalací.

- Instalace Apache a jeho konfigurace
- Instalace MySQL
- Instalace PHP

ÚKOLY

1. Pokud máte zařízení s podporou WIFI, tak napište skript, který vám vypíše pět WIFI s nejkvalitnějším signálem.
2. Připojte se na Váš systém s nainstalovaným SSH serverem (sshd).
3. Nainstalujte jednoduchý LAMP server pro Vaši distribuci, spusťe jej a ověřte jeho funkčnost.

10 Sdílené souborové systémy

Základní sdílené souborové systémy jsou

- NFS(Network File System)
- SMB (Server Message Block)
- CODA

Seznamte se s jejich základními vlastnostmi a principy (!)

- Mechanismus připojování
- RPC

V dalším se budeme věnovat NFS.

NFS rozlišuje server a klienta. Server distribuuje do sítě své souborové systémy, klient si je připojuje (viz příkaz mount).) Pro instalaci serveru budeme potřebovat (v závislosti na distribuci) následující balíky:

- nfs-kernel-server - podpora NFS do jádra (případně podobný název)
- portmap - démon pro přidělování portů klientům
- nfs-common-potřeba jak pro server, tak i pro klienta

Poté je třeba server nakonfigurovat, tj. určit co se bude distribuovat, obvykle stačí

- uvést sdílený adresář do souboru /etc/exports a
- nastavit způsob sdílení

Nakonec je potřeba server znovu restarovat (/etc/init.d/nfs-kernel-server reload).

Na straně klienta stačí pak svazek jen připojit pomocí *mount*, případně modifikovat soubor */etc/fstab* pro automatické připojování. Poznamenejme, že většina grafických umožňuje též připojování distribuovaných souborových systémů.

ÚKOLY

1. Nainstalujte NFS server a ověřte jeho funkčnost.
2. Nainstalujte FTP server pro vaši distribuci, spusťe jej a ověřte jeho funkčnost.

11 Instalace software

Pro instalaci software lze použít v podstatě dvou základních přístupů:

1. Ze zdrojového kódu, tj. program je nutno přeložit a pak nainstalovat.
2. Pomocí správců balíků.

Instalace ze zdrojových kódů probíhá následovně:

- Získání zdrojových kódů (např. *git, svn*)
- Konfigurace, zjištění systémového prostředí a generování souborů pro překlad (typicky skript *./configure*)
- Vlastní překlad, za použití programu make (typicky *make*)
- Instalace opět za použití programu make (make *install*)

Program make, je nástroj pro automatizaci překladů. Postup překladu a instalace je uveden v souboru makefile, seznamte se s jeho strukturou.

Při instalaci pomocí zdrojových kódů se očasto objeví problém s chybějící knihovnou, v tom případě nezbývá než knihovnu doinstalovat, kde se zase může opět vyskytnout problém ...

V případě použití správce balíků se lze tomuto vyhnout, neboť správci řeší tyto závislosti automaticky. Správci balíků jsou na různých dsitribucích různí, uveďme si nejčastěji používané

- *apt* (Ubuntu, Debian)
- *yast* (Suse)
- *yum*, *dnf* (Fedora)
- *pacman* (ArchLinux)
- *emerge* (Gentoo)

Proto, aby mohli správci potřebné balíky stahovat, musí být známo jejich umístění tzv. repozitáře, nejčastěji jsou uvedeny v */etc/jménosprávcebalíků*

Pro svoji distribuci a pro *apt*(!) si prostudujte:

- instalace a odstranění balíků
- vyhledání balíků a jejich závislostí
- stažení informací o repositářích
- modifikace zdrojů (repositářů), jejich přidávání a odebírání

ÚKOLY

1. Vyzkoušejte si vlastní překlad aplikace, zkuste např. zkompilovat a nainstalovat přehrávač Mplayer (http://www.mplayerhq.hu/design7/dload.html)

2. Napište si vlastní jednoduchý makefile pro instalaci.

12 Zálohování

Pro zálohování se ve světě Linuxu používá program *tar* (tape archive). Základní funkcí tohoto programu je slučování souborů do archivu. Běžně používané přepínače jsou

- c- vytvoření nového archivu
- x - extrakce
- r -přidání souborů do archivu
- u- update archivu
- d-porovnání
- z- touto volbou se navíc použije komprese pomocí programu gzip (.gz)
- j-komprese pomocí programu bzip2 (.bz2)
- f specifikace jména archivu

Ukázka použití je následující:

tar vczf zaloha.tar.gz adresar,

tento příkaz bude provádět zálohování adresáře *adresar* do *zaloha.tar.gz*, bude to zároveň komprimovat (*z*) a výstup bude upovídaný (*v*). Poznamenejme, že při použití komprimace, nemusí vždy fungovat update, porovnání, či přidání do archivu.

Pro vytvoření obrazu (iso souboru) je možné použít program mkisfs, případně genisoimage. Ukázka použití je následující

mkisofs -o mojeiso.iso zaloha.tar.gz,

kde dojde k vytvoření iso souboru ze zálohy.

Dalším užitečným nástrojem je program *dd* (data duplicator). Tento program čte ze vstupu, který je specifikován parametrem *if* a posílá na výstup, parametr *of*. Opět si ukažme použití

dd if=/dev/sda5 of=zaloha.img,

v tomto případě se zařízení, které je mapováno na soubor */dev/sda5* zkopiruje do souboru *zaloha*. Podobně je možné přenést vytvořenou zálohu na další zařízení

dd if=zaloha.img of=/dev/sda6,

Prostudujte si:

- Další užití programu tar (jsou jich spousty)
- Užití programů pro kompresi: gzip, gunzip, zip, bzip
- Užití programu *dd* a *dcfldd* pro kopírování dat na disk, věnujte pozornost volbám programu dd (ovlivňuje to výkon při kopírování).

ÚKOLY

1. Napište za pomocí uvedených příkladů skript, který zazálohuje (tar+iso+zápis na CDROM) vybrané adresáře.

Literatura:

Shah S., Soyinka W. *Administrace systému Linux*. Grada Publishing, Praha, 2007. ISBN 978-80-247-1694-7.

Fišer J. *GNU/Linux pro začátečníky* . skriptum PřF UJEP, 2005

Petrlík L. *Jemný úvod do systému UNIX*. Kopp, České Budějovice, 2000. ISBN 80-85828-28-6.

Elektronické zdroje:

Linux – Dokumentační projekt, kolektiv autorů, Vydavatelství a nakladatelství Computer Press, Brno, 2008 [online], Dostupné z http://linux.webatlas.cz/

Seriály věnované Linuxu na www.root.cz [online], Dostupné z http://www.root.cz/serialy/