Začínáme s Linuxem

Jakub Šerých

Digitálně
podepsal
Jakub Šerých
Datum:
2008.11.03
00:06:55
+01'00'

Obsah

Poděkování

Děkuji své ženě Markétě a dětem Jonášovi, Klárce a Hedvice za trpělivost, kterou se mnou mají, když neustále sedím u klávesnice a věnuji se práci, která mě nesmírně baví. Bez jejich podpory bych tuto knížku mohl napsat jen velmi těžko. Také děkuji průmyslovce, na které učím za to, že mi dává dostatečnou svobodu k vymýšlení a rozvíjení mých předmětů, které tak snad studentům mohou přinést nějaký užitek.

Úvod

Tuto knížečku jsem se rozhodl napsat zejména pro studenty volitelného předmětu „Správa internetového serveru“ na Střední průmyslové škole sdělovací techniky v Praze, www.panska.cz, ale pokud pomůže i dalším lidem ke startu s báječným operačním systémem, jakým Linux bezpochyby je, bude to jen k mojí radosti. Určitě je v tuto chvíli i přes moji snahu o pečlivou korekci velmi syrová a jistě plná různých překlepů. Proto budu velmi rád, když mě upozorníte na jakékoliv chyby nebo nejasnosti, které v ní najdete, nejlépe mailem na adresu serych@panska.cz.
Knížku můžete volně šířit.

Používané konvence

Příkazy a názvy souborů o kterých je řeč jsou psány neproporcionálním písmem přesně ve tvaru, ve kterém se s nimi pracuje, například **`cat /etc/hostname`**. Přesný tvar zahrnuje i malá a velká písmena, která Unix rozlišuje a příkazy se zásadně píší malými písmeny.
V knížce to dodržuji například i tehdy, kdy je příkaz na začátku věty a podle české gramatiky by měl začínat velkým písmenem.
Ukázky konkrétních příkazů a jejich výsledků z obrazovky nebo delší sekvence příkazů jsou pro zvýraznění a lepší grafické oddělení psány stylem se šedivým pozadím, například:

```
TuX:~$ ls
pokusy  public_html  skripty
TuX:~$
```

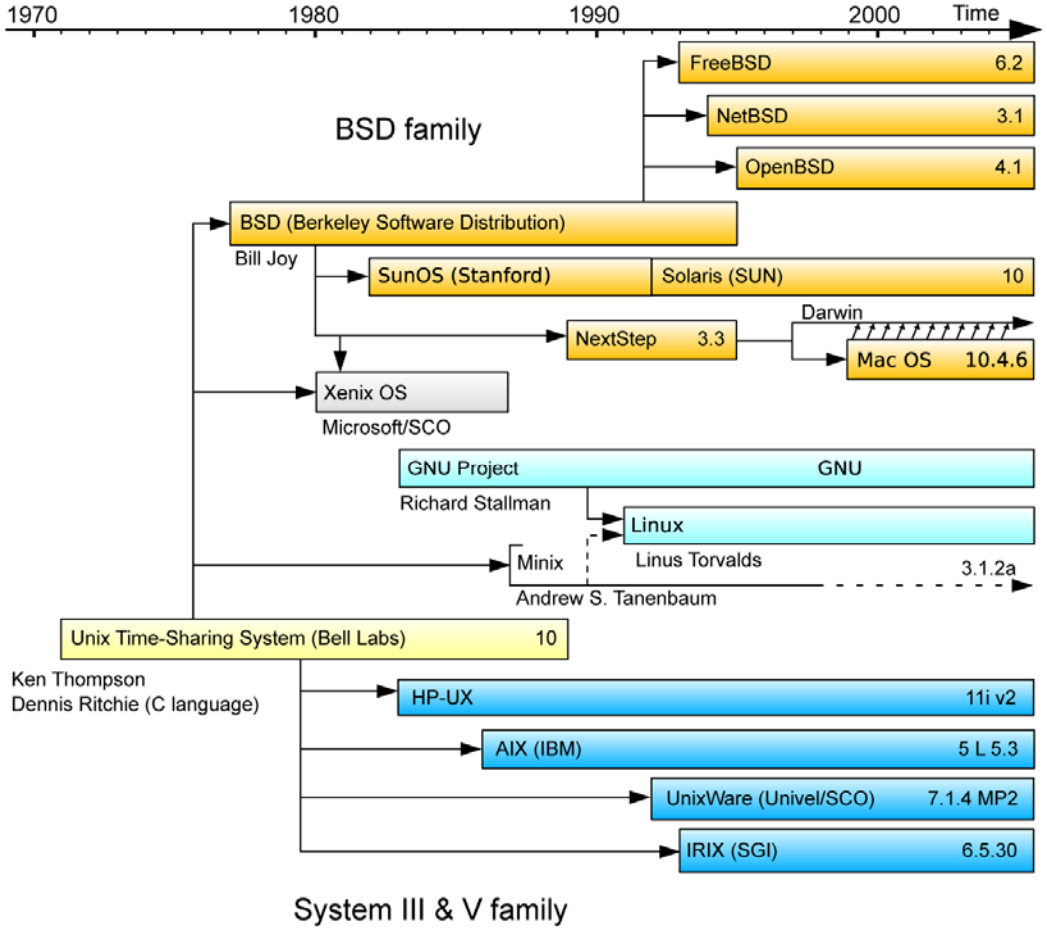
Tento styl je používán ve velikosti písma 12, 11 nebo 10 tak, aby se konkrétní výpisy vešly na šířku stránky v knížce tak, jak jsou vidět na obrazovce.
Důležité pojmy jsou psány tučným písmem.

1 Historie

Historie vývoje Linuxu od úplných začátků je poměrně klikatá a těžko pochopitelná. I přesto je dobré se jí alespoň povrchně zabývat, abychom pochopili, z jakých kořenů se Linux odvíjí.

1.1 Vývoj systému UNIX

V 60. letech probíhaly společné práce MIT[1], AT&T Bellových laboratoří a firmy General Electric na operačním systému Multics pro sálový počítač GE-645. Systém Multics neslavil na trhu příliš velký úspěch. Proto Bellovy laboratoře jeho vývoj zastavily a daly k dispozici zdrojové kódy. Jeden z jejich vývojářů Ken Thompson pokračoval ve vývoji a napsal pro GE-645 hru nazvanou Space Travel. Nicméně zjistil, že hra je na počítači GE příliš pomalá a drahá. Při tehdejších cenách strojového času stál její jeden běh $75.
Proto ji později s Dennisem Ritchie[2] přepsal do assembleru počítače PDP-7. Tato zkušenost je a několik dalších spolupracovníků vedla ke snaze o napsání operačního systému pro PDP-7. Brian Kerninghan mu dal v roce 1970 název Unics, který měl kontrastovat s Multicsem a vyjadřovat, že systém je jednodušší a lehčí než jeho předchůdce. Později byl název upraven na dnes používaný Unix.



Obrázek 1 - vývoj Unixových systémů

Nic z vývoje Unixu v této době nebylo Bellovými laboratořemi finančně podporováno. Teprve v roce 1971 přišly požadavky na využití Unixu na větších počítačích, a to přineslo první finanční podporu ze strany Bellových laboratoří. Do systému byly vpracovány nástroje

[1] Massachusetts Institute of Technology – jedna z nejprestižnějších amerických univerzit
[2] Spolu s Brianem Kerninghanem vytvořili programovací jazyk C

na zpracování textu, a v roce 1973 byl Unix přepsán do programovacího jazyka C, což vedlo k daleko větší přenositelnosti tohoto operačního systému. Bellovy laboratoře předaly licence systému univerzitám, vládě Spojených Států i komerčním firmám. Licence zahrnovaly i zdrojové kódy a vyšly také v knize, která se stala pro mnohé univerzity podkladem pro výuku operačních systémů.
V 80. letech AT&T licencovala Unix III pro komerční užití, což vedlo univerzitu v Berkeley k pokračování vývoje starší verze Unixu, která měla pro akademické instituce finančně výhodnější licence. Tak vznikly dvě výrazné vývojové větve Unixu, AT&T (System III and V family) a BSD[3] family, z nichž se později odvinulo mnoho dalších typů Unixových systémů včetně Linuxu.
I přes toto prvotní oddělení mají obě větve mnoho společného, a jejich prvky a programy se často vzájemně mísí.

1.2 Vývoj Linuxu

V dubnu roku 1991 se rozhodl tehdy 21letý Linus Torvalds, student Univerzity v Helsinkách, že si v assembleru procesorů Intel 80386 a jazyce C naprogramuje několik základních prvků primitivního operačního systému. Inspiroval se přitom systémem Minix, na jehož vývoji se tehdy pracovalo. Uplynulo několik měsíců a 25. srpna 1991 napsal do diskusní skupiny comp.os.minix následující mail:
„*Vyrábím (free) operační systém (jen jako hoby, nebude to nic velkého a profesionálního jako GNU) pro 386(486)ky, klony ATéčka. Dělám na tom od dubna a zdá se, že už to začíná být použitelné. Rád bych od vás slyšel vaše názory na minix – co se vám na něm líbí/nelíbí – protože můj OS se mu podobá (z praktických důvodů jsem například zvolil stejný návrh filesystému).*
V současné chvíli jsem do něj portoval bash(1.08) a gcc(1.40), a zdá se, že to funguje. Z toho plyne, že bych z toho během pár měsíců mohl vyrobit něco prakticky použitelného. [...] Ano – kód je zcela vlastní, není opsaný z minixu a má to multithreadový fs. NENÍ to přenositelné (používá to 386kový task switching atd.), a zřejmě to nikdy nebude podporovat nic jiného než AT – harddisk, protože ten jediný vlastním :-(
Psané je to většinou v Céčku, nicméně mnozí to co píšu asi ani Céčkem nenazvou. Používá to každičkou vlastnost, kterou 386ka nabízí, protože cílem tohoto mého projektu bylo, abych se naučil 386ku. Jak už bylo řečeno, používá to MMU[4] jak na paging (zatím ne diskový), tak na segmentaci. Právě segmentace nejvíc způsobuje závislost na 386ce (každá úloha má k dispozici 64Mb kódový a datový segment a může pracovat maximálně 64 tasků ve 4 Gb paměti. [...] Některé z mých Céčkových souborů (zejména mm.c) jsou ve skutečnosti napůl Céčko a napůl assembler. [...] Oproti minixu to MÁ rádo interupty, takže jsou normálně obsluhovány bez jakýchkoliv snah o jejich omezování a skrývání.“

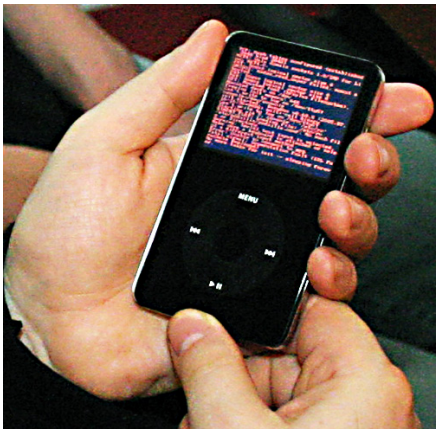
Na tento příspěvek reagovalo mnoho lidí z diskusní skupiny, tím, že přispělo ke vznikajícímu kódu, takže už v září 1991 vznikla verze Linuxu 0.01 čítající 10 239 řádek zdrojového kódu. Ta byla hned v říjnu následována verzí 0.02. V prosinci následovala verze 0.11, která byla jako první self-hostingová, to znamená, že počítač s Linuxem 0.11 mohl sám zkompilovat celý Linux 0.11. U verze 0.12 z února 1992 Linus Torvalds nahradil svoji původní vlastní licenci, která neumožňovala komerční použití Linuxu licencí GNU General Public License[5]. Tato licence umožňuje programy prodávat za jakoukoliv částku, ale kupující nesmí být jakýmkoliv způsobem omezován v tom, aby dělal totéž. Navíc programy musí být dostupné

[3] Berkeley Software Distribution
[4] Memory Management Unit
[5] Viz www.gnu.org

včetně veškerých zdrojových kódů. Kdokoliv je tak může upravovat a upravené distribuovat dále pod jedinou podmínkou, a to že opět splní podmínky dané GNU GPL.
V roce 1992 vznikla verze 0.95 do které byl poprvé implementován systém X Window.

Obrázek 2 - tučňák Tux

Verze 1.0.0 byla uvolněna v březnu 1994 a obsahovala už přes 175 tisíc řádek kódu.
V květnu 1996 se Linus Torvalds rozhodl dát linuxu jako maskota postavičku tučňáka nazvaného Tux.
Verze 2.0.0 mající už přes jeden a půl miliónu řádek kódu přišla na svět v roce 1996 a vývoj stále pokračuje, takže jádra mají v době psaní tohoto dokumentu verze 2.6.x.
Ačkoliv Linus na začátku tvrdil, že jeho systém zřejmě nikdy nepřekročí hranice procesoru Intel 386, dnes je Linux portován na velké množství procesorů od sálových počítačů IBM až po jednočipové procesory. Dokonce jsou dnes některé procesory navrhovány speciálně tak, aby svou architekturou vyhovovaly běhu Linuxu.



Obrázek 3 - Linux běžící na iPodu

2 Distribuce

Filozofie systému Unix (a tedy i Linux) je založená na tom, že jádro je obklopeno velkým množstvím jednoduchých malých prográmků, které se při práci dají používat v obrovském množství různých kombinací. Můžeme si to představit jako jakousi stavebnici složenou z mnoha malých jednoduchých kostiček. Většina těchto prográmků pochází z nadace Free Software Foundation a jsou distribuovány opět pod GPL licencí.
Málokterý uživatel potřebuje ovšem ke své práci všechny existující programy. Navíc instalace systému skládajícího se z jádra a spousty malých programů je sice možná úplně z „čisté louky“, ale je k tomu potřeba velmi dokonalá znalost celého operačního systému a programování.
Proto začaly vznikat tzv. distribuce systému, což jsou veškeré potřebné složky systému předpřipravené ve formě jakýchsi balíčků navíc doplněné o sofistikovaný instalační program a řadu skriptů, které provedou instalaci systému pomocí průvodce přívětivého i pro neznalé uživatele. Jednou z prvních byla dodnes se rozvíjející distribuce Slackware. Ta vznikla už v roce 1993.
Vzhledem k GPL charakteru veškerého softwaru existuje dnes obrovské množství různých více či méně oblíbených distribucí. Každé z nich dali její autoři do vínku nějaký speciální záměr, a tomu pak skladbu celé distribuce přizpůsobují. Existují například distribuce, které můžete spustit rovnou z CDčka, aniž by operační systém musel být nainstalován na daném počítači (tzv. Live distribuce), jsou minidistribuce schopné běžet z USB flash disku, kde zabírají pouze několik MB místa, jiné mají za hlavní cíl z počítače na kterém běží vytvořit router, další jsou specializované třeba pouze pro záchranu dat z poškozeného harddisku atd.
Protože Linux byl portován na velké množství jiných procesorů než byla původní 386ka, existují samozřejmě i distribuce specializované na běh na nějakém nestandardním hardwaru (například na některém z levných routerů Edimax, PDAčkách, atd.).
Některé z distribucí jsou vyvíjeny a prodávány komerčními firmami (například Red Hat nebo SUSE). Zachovávají si přitom GPL charakter, ale komerční firmy do nich vkládají přidanou

hodnotu, kterou prodávají společně s distribucemi. Například Red Hat si můžete za pár korun nakoupit na úhledných CDčkách v každém lepším knihkupectví a nemusíte ho stahovat z Internetu, můžete si k němu dokoupit technickou podporu od firmy Red Hat, což vám například přinese možnost konzultací problémů, výrazně rychlejší přístup ke stahování opravných balíčků než mají uživatelé, kteří si ji nekoupili a další podobné výhody. Stejně tak ale můžete tuto distribuci stáhnout a používat zcela zdarma, čímž ovšem o uvedené výhody přijdete.

2.1 Debian

Distribuce, kterou budeme používat v našem kurzu se jmenuje Debian[6]. Založil jí v roce 1993 tehdejší student Univerzity Purdue (ve státě Indiana) Ian Murdock. Základní myšlenka této distribuce je zachycena v dokumentu Debian Manifesto. Distribuce by podle něj měla být vytvářena čistě otevřenou komunitou na principech GNU GPL. Je tedy distribucí, která nikdy nechce být zkomercionalizovaná. Právě pro svoji úplnou otevřenost a širokou uživatelskou základnu je velmi oblíbená zejména pro instalace Linuxových serverů od malých až po ty velmi výkonné.
Název distribuce složil Ian Murdock ze jména své tehdejší dívky (dnes ženy) Debra a ze svého jména Ian.
Distribuce měla od založení do dnešních dnů čtrnáct významných releasů. Od releasu 1.1 z roku 1996 mají releasy navíc pracovní jména postaviček z filmu Toy Story (Příběh hraček). Hlavní releasy Debianu:

- 0.01 až 0.93 – roky 1993 - 1995
- 1.0 – nikdy nevyšel
- 1.1 – Buzz 1996
- 1.2 – Rex 1996
- 1.3 – Bo 1997
- 2:0 – Hamm 1998
- 2.1 – Slink 1999
- 2.2 – Potato 2000
- 3.0 – Woody 2002
- 3.1 – Sarge 2005
- **4.0 – Etch 2007**

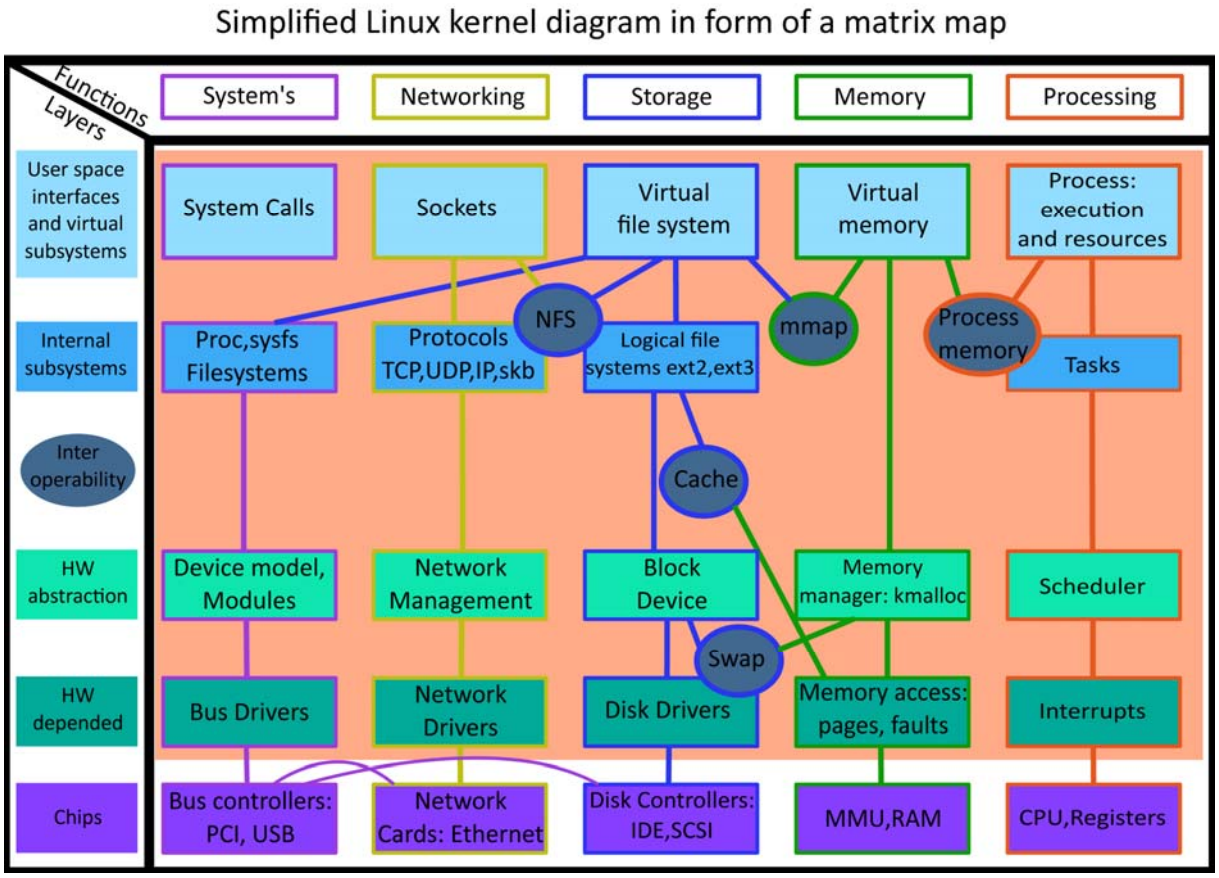
Pro naši práci budeme používat právě poslední release Debian Etch. Distribuce odvozená z Debianu, ale určená především pro pracovní stanice se jmenuje Ubuntu[7] a je také velmi oblíbená, a tedy i rozšířená. Používá stejný balíčkovací a instalační systém, takže je s Debianem v mnohém opravdu velmi kompatibilní.

[6] Viz www.debian.org
[7] Viz www.ubuntu.com

3 Architektura systému

3.1 Jádro

Než začneme se systémem prakticky pracovat, je dobré se alespoň zhruba orientovat v jeho architektuře. Jak již bylo řečeno, základem Unixu je takzvaný **kernel** (česky jádro). Od primitivní Linusovy počáteční verze se kernel výrazně vyvinul, takže dnes je oproti jeho původním předpokladům přenositelný na mnoho různých typů systémů. To zákonitě předpokládá, že musí být postaven vrstvovitě, jak je vidět na jeho zjednodušeném schématu.



Obrázek 4 - zjednodušené blokové schéma kernelu

Úplně dole se nachází vlastní hardware počítače na kterém má Linux běžet. Další vrstvu tvoří drivery specifické pro dané typy hardwaru. Ale už vrstva nad nimi má za úkol abstrahovat konkrétní použitý hardware do jednotné podoby, se kterou počítají všechny vyšší vrstvy systému.

Pokud se podíváme na horizontální členění obrázku, zjistíme jaké základní funkce jádro systému plní. Povšimněte si jednak toho, že síťová komunikace je zabudována přímo v jádru (to byl mimochodem jeden z důvodů prudkého rozvoje Internetu, protože když univerzity od AT&T získaly Unix, základ síťařiny v něm už byl zabudován). Druhá věc, která stojí za zmínku je to, že i nad samotným procesorem je systém budován ve vrstvách, což právě umožňuje již zmíněné snadné portování Unixu na jiné typy procesorů.

Jádro Linuxu je uloženo v souboru s názvem **vmlinuz**.

Ačkoliv o jádru Linuxu by se daly napsat (a byly napsány) celé tlusté knihy, zde se jím více nebudeme zabývat a zaměříme svoji pozornost spíše na uživateli bližší části systému.

3.2 Souborový systém

Z hlediska uživatelského pohledu na systém je důležitý zejména souborový systém. Filozofie Unixu je báječná v tom, že disponuje jediným souborovým systémem rozvíjejícím se z jediného tzv. **kořenového adresáře**. Uživateli tedy například vůbec nezáleží na tom kolik a jakých fyzických úložišť souborů konkrétní počítač má, jestli jsou to skutečné harddisky a na jaké oddíly jsou nebo nejsou rozděleny, zda jsou to třeba CDčkové mechaniky, USB flashdisky nebo třeba pouze síťové disky kdesi hluboko v Internetu. Kterékoliv z takových úložišť se dá **namontovat**[8] do kteréhokoliv místa typického Unixového filesystému a jeho prostřednictvím se k němu poté dá přistupovat. Dokonce i jakékoliv zařízení počítače (jako příklad uveďme třeba sériový port) vystupuje jako speciální soubor ve filesystému a dá se s ním takto komunikovat. Důležitá informace je, že **názvy souborů a adresářů v Unixu rozlišují malá a velká písmena**. **Běžné je používat výhradně písmena malá**, jsou ale zažité případy, kdy se naopak používají písmena velká (například soubory **README**, adresář X windows s názvem **X11** a několik dalších podobných případů)[9].
Souborový systém má v Unixu více nebo méně **pevnou základní strukturu**[10] s jejím typickým využitím. Ta se trochu liší distribuce od distribuce. Proto zde při popisu základní hierarchie adresářů v Unixu budu zdůrazňovat typické využití popisovaných adresářů v Debianu (pokud se liší od jiných distribucí).

3.2.1 / - kořenový adresář

Jak již bylo řečeno, vše se v souborovém systému Unixu odvíjí od takzvaného kořenového adresáře. Ten se značí lomítkem /[11], které slouží jako oddělovač adresářů v cestě k souboru. V kořenovém adresáři se typicky vyskytuje jen naprosté minimum skutečných souborů obsahujících data. Typicky je tam jen link[12] na **vmlinuz** (kernel) a **initrd.img**, což je image ramdisku použitého v počátečních fázích procesu bootování. Mimo těchto dvou linků jsou tam jen adresáře, jejichž základní význam bude popsán níže.

```
bin    dev    initrd      lost+found  opt   sbin     sys  var
boot   etc    initrd.img  media       proc  selinux  tmp  vmlinuz
cdrom  home   lib         mnt         root  srv      usr
```

Obrázek 5 - výpis kořenového adresáře na Debianu

3.2.2 /bin

Adresář s důležitými spustitelnými programy, které zajišťují základní funkčnost systému. Tyto programy jsou používány už během procesu bootování a slouží i při opravách instalace Unixu v případě závažných poruch.

3.2.3 /sbin

Adresář svojí funkcí velmi podobný adresáři bin. Liší se pouze tím, že v něm obsažené programy jsou určeny pro systém a administrátora systému, ale nejsou dostupné běžným uživatelům tak jako ty v adresáři /bin.

[8] příkazy mount a umount o kterých bude zmiňka později
[9] V této příručce budu ctít názvy souborů malými písmeny i v případech na začátcích vět, kdy by první písmeno mělo být velké.
[10] Blíže viz: http://www.pathname.com/fhs/
[11] nikoliv zpětné lomítko \ jako v systémech MS DOS/Windows
[12] o lincích v Linuxu bude řeč později, zatím si je v jednoduchosti můžete představit jako „zástupce“ z Windows

3.2.4 /boot

Adresář v němž jsou uloženy všechny soubory nutné pro proces bootování. Typicky je právě zde uložen kernel a bývá zde také vnořený adresář s použitým bootloaderem a všemi jeho potřebnými soubory.

3.2.5 /dev

Toto je adresář, ve kterém se nacházejí speciální soubory určené k přímé komunikaci s hardwarem počítače. Unix rozděluje hardwarová zařízení do dvou velkých kategorií. Jsou to znaková zařízení a bloková zařízení. Mezi **znaková zařízení** patří takové hardwarové komponenty počítače, se kterými se komunikuje prostřednictvím toku jednotlivých znaků. Typicky sem patří například klávesnice, ale také třeba myš, sériové porty, ale i mnoho dalších typů zařízení. S **blokovými zařízeními** se oproti tomu komunikuje po větších blocích dat. Mezi typickými představiteli jsou například disková zařízení, která jsou čtena/zapisována po celých sektorech.

3.2.6 /etc

Tento adresář je místem, ve kterém pracuje běžný administrátor Unixového stroje téměř neustále. Nachází se zde totiž veškeré **konfigurační soubory** všech programů nainstalovaných na daném počítači. Některé základní jsou typicky uloženy přímo v adresáři /etc, ale většina programů si zde vytváří svoje struktury dalších adresářů, kam svoji konfiguraci ukládají v přehledné podobě.
Tak například název jakéhokoliv Unixového počítače je uložen v textové podobě v souboru /etc/hostname, seznam uživatelů najdeme v /etc/passwd, zatímco například základní nastavení sítě je u Debiana v souboru /etc/network/interfaces. Pro příklad odlišností jednotlivých distribucí si uveďme ukázku téhož nastavení u distribuce Red Hat. To tam najdeme v samostatných souborech odpovídajících jednotlivým instalovaným síťovým kartám například tedy /etc/sysconfig/network-scripts/ifcfg-eth0 pro síťovou kartu Ethernet0.
Jiným příkladem konfiguračního souboru může být třeba nastavení virtuálních webů (v případě, že je nainstalován webserver Apache 2), které u Debiana najdete v adresáři /etc/apache2/sites-enabled.

3.2.7 /home

Adresář /home obsahuje podadresáře, které jsou domovskými adresáři jednotlivých uživatelů majících na daném stroji zřízen účet. Vytvoříme-li tedy na daném počítači například uživatele pepa, bude zároveň automaticky vytvořen i adresář /home/pepa, kam si daný uživatel bude moci ukládat všechny svoje soubory, a kde má také osobní nastavení ovlivňující chování systému po jeho přihlášení. Domovský adresář se každému uživateli zobrazuje pomocí symbolu vlnovky ~. Takže pokud si uživatel pepa ve svém domovském adresáři vytvoří adresář data a v něm adresář pracovni, vidí ho jako ~/data/pracovni ačkoliv v adresářové struktuře Unixu je fyzicky umístěn v /home/pepa/data/pracovni

3.2.8 /root

Systémový administrátor v Unixech se typicky jmenuje **root** a má svůj domovský adresář oddělen od domovských adresářů běžných uživatelů (zejména z bezpečnostních důvodů). Účel adresáře /root je tedy pro roota systému stejný jako účel adresáře /home/pepa pro uživatele pepa z výše uvedeného příkladu. Adresář **/root** je ovšem vytvářen přímo při instalaci systému, kdežto adresáře jednotlivých uživatelů až teprve při vytváření jejich accountů na daném systému. Proto bývá také obsah a využití domovských adresářů běžných

uživatelů trochu jiné než je tomu u adresáře /root. Podobně jako běžný uživatel vidí i root svůj domovský adresář jako ~.

3.2.9 /lib

Adresář /lib obsahuje sdílené objektové knihovny[13] potřebné jednak při bootování systému, ale využívané i mnoha různými programy při jeho normálním běhu. V tomto adresáři se také nachází důležitý podadresář **/lib/modules** obsahující další strukturu podadresářů s takzvanými **moduly jádra**[14].

3.2.10 /mnt

Adresář /mnt je určen k montování dalších souborových systémů. Obsahuje obvykle podadresáře s intuitivními názvy, kam montujeme další filesystémy. Například typicky v něm může být adresář /mnt/cdrom, kam se montuje systém CDčka po jeho zasunutí do počítače. Root si zde může vytvořit libovolné další adresáře podle svého uvážení (například /mnt/usb-flash). Samozřejmě nový filesystém lze namontovat kamkoliv bez omezení na adresář /mnt, ale pro přehlednost bývá zvykem se při montování držet právě tohoto adresáře (pochopitelně pokud neexistují nějaké speciální důvody, proč to tak v dané situaci nedělat).

3.2.11 /media

U Debianu je pro montování typických zařízení určen namísto adresáře /mnt adresář /media. Zde najdeme adresáře **/media/cdrom** a **/media/floppy** předpřipravené pro namontování CDčka a diskety.

3.2.12 /lost+found

Pokud se musíme zajímat o tento adresář, nevěští to nic dobrého. Znamená to zpravidla, že se souborovým systémem se děje něco špatného (například když začíná odcházet harddisk). Sem se totiž dostávají soubory, které jsou nalezeny při kontrole filesystému jako nějakým způsobem narušené. Pokud se tedy začnou z nějakých neznámých důvodů ztrácet soubory, bývá toto místo místem poslední záchrany.

3.2.13 /opt

Tento adresář je místem, určeným pro ukládání souborů, které nemají žádné jiné vhodnější logické umístění. Obvykle na všech Unixech zeje prázdnotou.

3.2.14 /proc

Adresář /proc je speciální adresář, ve kterém si vytváří svůj podadresář každý běžící proces a i kernel zde má uložené zajímavé informace. Například pokud chceme zjistit, na jakém procesoru daný Unix běží, není nic jednoduššího než si nechat vypsat soubor /proc/cpuinfo. Dozvíme se tak o procesoru velmi jednoduchým způsobem daleko více informací než jsme schopni například zjistit v celém systému MS Windows. Podobně vypsáním /proc/meminfo získáme dokonalý přehled o současném stavu paměti počítače.

---

[13] Bylo by zbytečné veškerý kód různých programů psát vždy úplně od začátku, když se mnohé jejich funkce velmi podobají. Proto se používají knihovny předem naprogramovaných funkcí, které se už z programů pouze volají.

[14] Moderní jádra jsou modulární, takže například pokud máte v počítači nějakou netypickou síťovou kartu, stačí sehnat a zavést do jádra její modul. U historických verzí kernelu bylo v takovém případě potřeba vždy celý kernel včetně začleněné podpory daného hardwaru překompilovat, což bylo velmi komplikované.

3.2.15 /tmp

Adresář pro dočasné soubory se v Unixech jmenuje /tmp. Adresář je určen skutečně pro uchování dočasných souborů a je systémem pravidelně nemilosrdně mazán. Debian maže tento adresář například při každém bootování.

3.2.16 /usr

Adresář /usr je na Unixovém systému typickým místem pro instalaci jednotlivých aplikací. Mívá obvykle bohatou strukturu podadresářů, jejíž stavba se v různých distribucích poměrně výrazně liší. V Debianu se v ní například opakuje část struktury z kořenového adresáře. Takže spustitelné soubory aplikací najdeme typicky v **/usr/bin** a **/usr/sbin**, které mají obdobný význam jako **/bin** a **/sbin** jen s tím rozdílem, že zde již nejsou aplikace potřebné při procesu bootování systému. Ale jsou tu i další adresáře, například v **/usr/share** se nachází mnoho adresářů se sdílenými součástmi různých aplikací. Třeba v **/usr/share/doc** najdeme soustředěnou většinu dokumentace k jednotlivým programům.

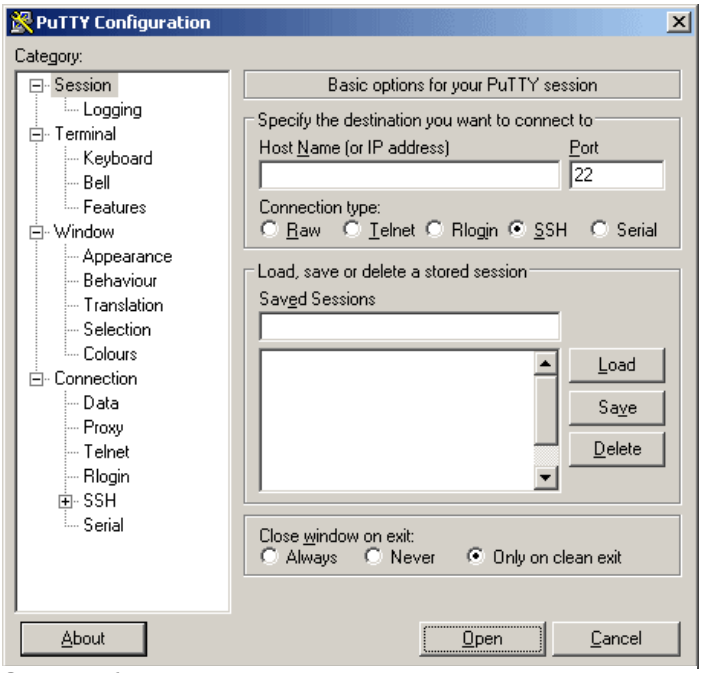
3.2.17 /var

Adresář /var je místo, kam se na Unixech ukládají veškerá data, která se v průběhu činnosti počítače mění. I tento adresář mívá bohatou strukturu podadresářů jejíž stavba je hodně závislá na konkrétní distribuci. U Debianu se například do adresáře **/var/www/** typicky ukládají webové stránky, které má server nabízet, do adresáře /var/backups je dobré dělat zálohy dat, ve **/var/log** najdeme logy systému i jednotlivých programů atd. Toto je po /etc druhý adresář, se kterým typický administrátor Unixu nejvíce pracuje.

4 Základy práce se systémem

4.1 Přihlášení k systému

Po dlouhém teoretickém úvodu je čas se konečně přihlásit k počítači s Linuxem a začít se učit se systémem pracovat. Co k tomu budeme potřebovat? Tak pochopitelně především počítač s nainstalovaným Linuxem. Časem si ukážeme, jak si Linux nainstalovat na jakémkoliv počítači, třeba i tak, abychom se nikterak nedotkli jeho stávající instalace jiného operačního systému. Ale teď na začátku budeme pro jednoduchost pracovat s cvičným serverem, na kterém už Linux je nainstalovaný. Jmenuje se TuX a sídlí ve školní serverovně[15]. Mohli bychom tedy jít do serverovny, pustit si příslušný monitor a z patřičné klávesnice se k serveru přihlásit. Nicméně serverovna je malá a monitor a klávesnici má TuX jen jednu, takže by nemohl pracovat více jak jeden člověk. Daleko účelnější proto bude, když se k serveru přihlásíme vzdáleným terminálem. Uvidíme přesně to, co bychom viděli na jeho vlastním monitoru a naše klávesnice se bude chovat tak, jako by to byla jeho vlastní klávesnice. Navíc se nás takto v jednu chvíli bude moci k serveru připojit více. Dříve se pro to používala aplikace Telnet. Ta si ale vše včetně hesel se serverem vyměňovala v otevřené textové podobě, a proto je dnes považována za velmi nebezpečnou a na Unixových strojích se standardně tento přístup zakazuje. Namísto toho se tatáž činnost realizuje prostřednictvím zašifrované komunikace ssh[16]. Pro umožnění této komunikace musí být na straně serveru nainstalován ssh daemon a na straně klienta musíme použít některý z programů, které umí ssh komunikaci. Nejznámějším a jedním z nejlepších takových programů pro operační systém Windows je program PuTTY[17]. Program PuTTY je malý program mající necelého půl MB a spustitelný bez jakékoliv instalace. Můžeme ho tedy mít uložený třeba na USB flashdisku tak, abychom ho měli kdykoliv a kdekoliv k dispozici. Po spuštění PuTTY stačí do řádky Host Name napsat název serveru nebo jeho IP adresu a po kliknutí na tlačítko Open se objeví přihlašovací obrazovka serveru tak, jako bychom byli přímo fyzicky u jeho monitoru. Bude to vypadat nějak takhle:

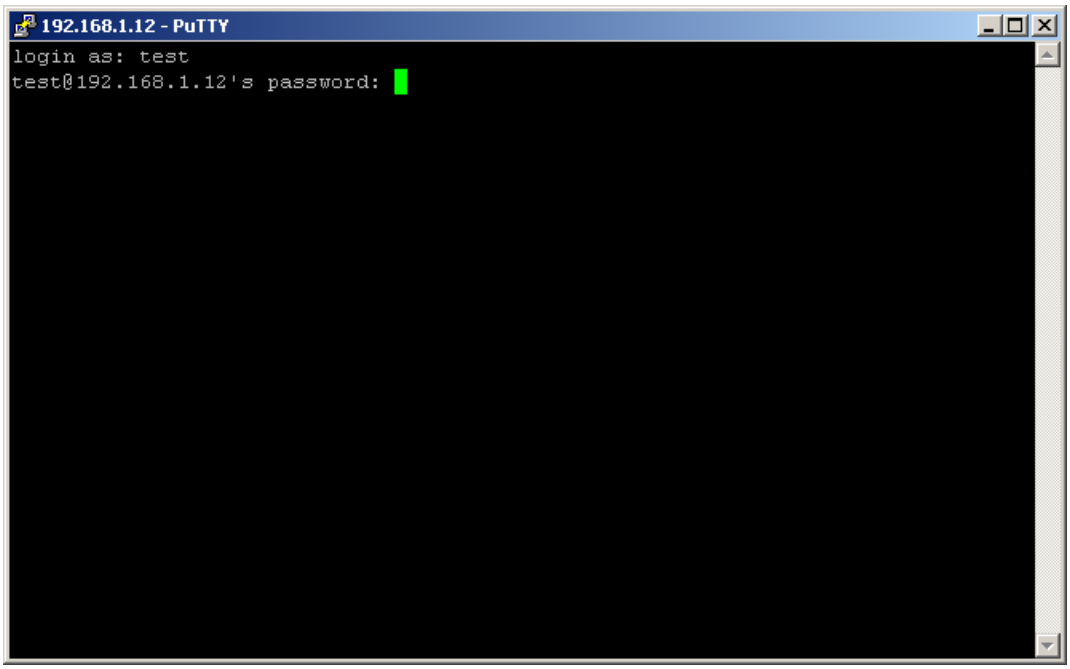


Obrázek 6 - program PuTTY

[15] Čtenáři, kteří nemají přístup k tomuto školnímu serveru mohou přeskočit na kapitolu Instalace a nainstalovat si podle ní svůj vlastní Linux.

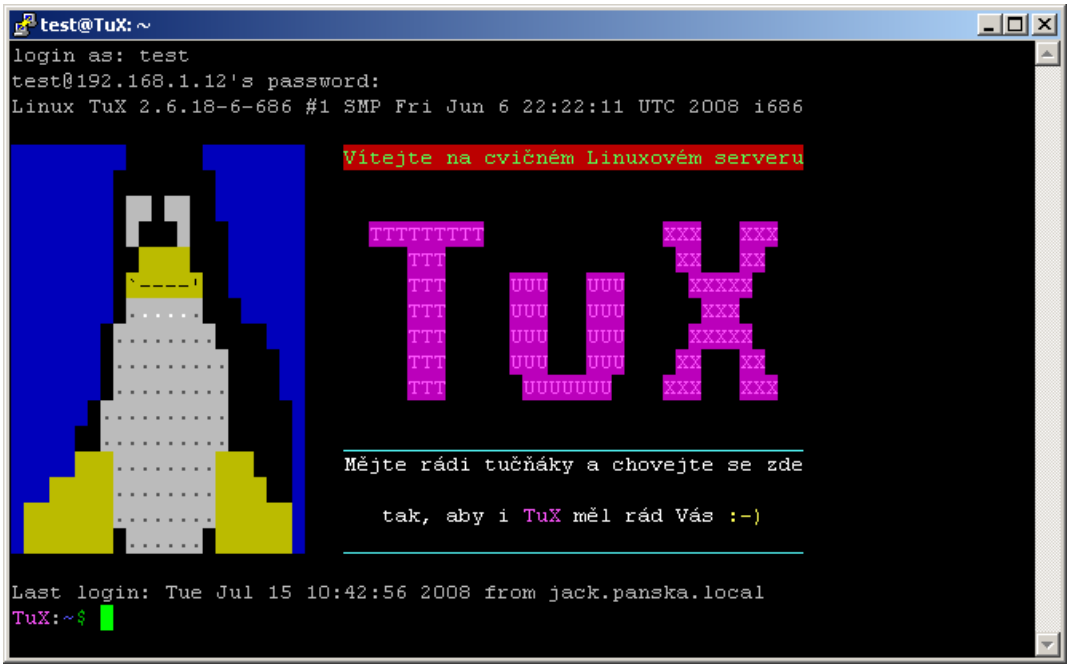
[16] zkratka Secure Shell neboli bezpečná příkazová řádka

[17] Program je open source freeware a snadno ho najdete, když do Googlu zadáte putty download. Na stránce je několik různých programů, vy potřebujete jen putty.exe



Obrázek 7 - vložení uživatelského jména a hesla

Systém nás požádá nejprve o naše uživatelské jméno a poté o vložení hesla.
Ze začátku bude pro Vás nezvyklé, že **Linux při vkládání hesla nic nepíše** (žádné hvězdičky nebo kolečka, jako ve Windows) a kurzor se nehýbe z místa, takže se zdá, že klávesnice nefunguje. Funguje ale normálně a vkládání hesla je tak daleko bezpečnější, protože nikdo Vám nemůže okoukat počet znaků v hesle. Po zadání správného hesla se objeví uvítací obrazovka.



Obrázek 8 - uvítací obrazovka

Z té je pro nás nejzajímavější tzv. **příkazová řádka**:

Obrázek 9 - příkazová řádka

Příkazová řádka se skládá z takzvané výzvy (anglicky prompt) a místa, kam se z klávesnice zapisují vlastní příkazy. Výzva se dá na Unixech nastavit do téměř jakékoliv podoby tak, aby uživateli ukazovala informace, které potřebuje znát. Ta naše obsahuje název serveru (to je velmi užitečné pokud máme otevřených několik oken PuTTY, kterými jsme zároveň připojeni

k několika různým serverům). Za dvojtečkou je pak vidět pracovní adresář[18]. Za ním následuje symbol $, který znamená, že přihlášený uživatel je běžný uživatel systému. Zelený obdélníček za symbolem dolaru je způsob, kterým PuTTY zobrazuje kurzor. Pokud je přihlášen root, uvidíme namísto znaku $ znak # (který jsem navíc nastavil tak, aby se zobrazoval na červeném pozadí). To je užitečné protože uživatel root má všude plná práva a neopatrným chováním může systém velmi snadno poničit.

4.2 exit nebo logout

Práci v shellu ukončíme příkazem **`exit`** nebo **`logout`**. Tím také dojde k odhlášení od počítače a uzavření okna programu PuTTY.

4.3 První jednoduché příkazy

Pojďme si vyzkoušet první jednoduché příkazy. Zapište na příkazovou řádku **`ls`** a stiskněte klávesu Enter.

```
TuX:~$ ls
TuX:~$
```

Na obrazovce by to mělo vypadat jako ve výše uvedeném příkladu. Je zřejmé, že příkaz **`ls`** v tomto momentě nevrátil žádný výstup a Linux jen zobrazil další výzvu, abychom mohli pracovat dál.
Unix rozlišuje malá a velká písmena, takže pokud se například pokusíme příkaz napsat jako **`LS`**, oznámí nám pouze, že takový příkaz nezná.

```
TuX:~$ LS
bash: LS: command not found
TuX:~$
```

4.4 ls

Příkaz **`ls`** vypisuje obsah adresáře (zkratka z anglického **l**i**s**t) a z výstupu se tedy zdá, že adresář, ve kterém se v tuto chvíli nacházíme je prázdný. Pojďme ale příkaz vyzkoušet ještě jednou s tím, že mu zadáme parametr.

```
TuX:~$ ls /etc
...
exim4       login.defs          protocols       wgetrc
fonts       logrotate.conf      python          X11
TuX:~$
```

Příkaz **`ls /etc`** už poskytne poměrně dlouhý a dokonce i barevný výpis, z něhož jsou výše pro ukázku uvedeny pouze poslední dva řádky. Tím, že jsme příkazu **`ls`** zadali jako parametr **`/etc`** jsme mu řekli, že má vypsat soubory v adresáři /etc a to také udělal. Pojďme udělat ještě jeden pokus. Napišme **`ls -al`** a uvidíme zhruba níže uvedený výpis. Co to všechno znamená? Příkazu **`ls`** jsme nyní zadali dva parametry, ale trochu jinak než v minulém pokusu. napsali jsme je totiž za pomlčku. Úplně stejného výsledku bychom dosáhli, kdybychom zapsali **`ls -a -l`**, ale pro zkrácení je možno napsat pomlčku jen jednou, a pak umístit parametry bez mezer těsně za sebe. Tím, že jsme **`ls`** nedali žádný „bezpomlčkový“ parametr, vypisuje adresář, ve kterém se momentálně nacházíme. Vzpomeňme si, že vlnovka

[18] Vlnovka na obrázku znamená, že se nacházíme ve svém domovském adresáři.

~ v promptu znamená, že se nacházíme ve svém domovském adresáři a **ls** v tomto příkladu tedy vypisuje právě obsah našeho domovského adresáře. Ačkoliv se nám to ještě před malou před chvílí nezdálo, nějaké soubory se v něm přece jenom nacházejí.
Znáte z Windows skryté soubory? V Unixu se také dají použít (a samozřejmě se daly použít dávno před tím než MS Windows vůbec byly na světě). V Unixu se za **skrytý soubor** (adresář) považuje cokoliv, co má název začínající znakem tečka.
Právě parametrem **-a** řekneme příkazu **ls**, že má zobrazovat vše (z anglického all), proto vypíše soubory včetně těch skrytých. Parametr **-l** mu zase říká, že chceme dlouhý výpis (zřejmě z anglického long), to znamená, že nechceme vidět jen pouhé názvy souborů, ale i další zajímavé údaje o nich.

```
TuX:~$ ls -al
total 27
drwxr-xr-x 2 test test 4096 2008-07-15 11:11 .
drwxr-xr-x 5 root root 4096 2008-07-15 09:49 ..
-rw------- 1 test test   25 2008-07-15 09:51 .bash_history
-rw-r--r-- 1 test test  220 2008-07-15 09:49 .bash_logout
-rw-r--r-- 1 test test  414 2008-07-15 09:49 .bash_profile
-rw-r--r-- 1 test test 2304 2008-07-15 09:49 .bashrc
TuX:~$
```

Popišme si jen zjednodušeně, co teď ve výpisu na jednotlivých řádkách vidíme. Divná „změť znaků“ v prvním sloupci, například **-rw-r--r--** vyjadřuje přístupová práva k danému souboru. Další dva sloupce říkají, jaký účet a jaká skupina jsou vlastníky daného souboru (zde u většiny souborů **test** a **test**), pak vidíme velikost souboru v Bytech, datum a čas jeho vytvoření a nakonec název souboru. První řádka **total 27** nám říká, kolik bloků na disku všechny soubory zabírají.
Možná si řeknete: „No dobrá, jako ukázka to bylo pěkné, ale jak zjistíme jaké všechny parametry můžeme příkazu ls zadat?“ Odpověď je velmi rychlá: **RTFM!** Některé jemnější povahy si myslí, že ta zkratka znamená Read The Fine Manual, ale všichni ostatní s jistotou vědí, že znamená Read The Fucking Manual :-).

4.5 man

V Unixech od nepaměti existuje příkaz man (z anglického manual). Používá se zpravidla s jediným parametrem, kterým je příkaz, jehož nápovědu[19] si chceme přečíst. Napišme tedy:

```
TuX:~$ man ls
```

Objeví se strukturovaný text nápovědy. V tomto případě je struktura následující:

- NAME
- SYNOPSIS
- DESCRIPTION
- AUTHOR
- REPORTING BUGS
- COPYRIGHT
- SEE ALSO

Struktura manpage ale není striktně daná a u různých stránek se může lišit.
V textu se můžeme pohybovat kurzorovými klávesami a klávesami PgUp a PgDn. Dokonce v něm můžeme jednoduchým způsobem vyhledávat tak, že zapíšeme lomítko **/** následované

[19] správně se tomu říká manpage neboli manuálová stránka

hledaným řetězcem a stiskem klávesy Enter. **man** na základě toho najde nejbližší výskyt daného řetězce a všechny výskyty navíc zvýrazní podbarvením. Zkuste třeba napsat **/literal**. Další výskyty téhož řetězce pak vyhledáváme zadáním pouhého **/** následovaného stiskem Enter. Pokud se pokusíte nápovědu intuitivně ukončit klávesou Esc, mocv nepochodíte. **man** jen v pravém spodním rohu napíše ESC, ale nic nedělá. ukončit ho totiž musíme klávesou **q** (z anglického quit).
Chtěli byste se naučit další vymoženosti příkazu man? No ale teď už přece víte jak na to! Stačí zapsat:

```
TuX:~$ man man
```

... a můžete se pustit do čtení. Také můžete zkusit příkaz **info**, který se ovládá podobně, ale dokonce disponuje jednoduchými hypertextovými odkazy, na které je možno přecházet podobně, jako když klikáte na odkazy na webovské stránce. Vyzkoušejte třeba **info bash**.

5 Základní příkazy pro práci se soubory a adresáři

Než začneme se samotnými příkazy, je dobré si alespoň velmi povrchně popsat, jak je v Unixu realizován souborový systém. Ne nadarmo jedna známá poučka říká: „**Vše v Unixu jsou soubory. A co náhodou nejsou soubory, jsou procesy**“. Soubory jsou opravdu velmi důležitým prvkem systému. Zpravidla jsou uloženy na blokových zařízeních, přičemž každý soubor má svůj název a ten odkazuje na takzvaný i-node[20], ve kterém se nacházejí „technické“ údaje o souboru, včetně jeho fyzického umístění na blokovém zařízení. Většinu oněch technických údajů jsme už viděli, protože je to přesně to, co vypisuje příkaz **ls -l**. Co jsou to ale adresáře? Vzpomeňte: „Vše v Unixu jsou soubory...“. Ano adresáře jsou jen speciální soubory, ve kterých jsou zapsané názvy souborů v nich uložených.

5.1 ls

Příkaz **ls** jsme již probrali v předchozí kapitole. Zde tedy už jen stručné shrnutí. Příkaz vypisuje obsah adresáře. Bez zadání cesty vypisuje aktuální adresář, jinak ten, který je zadán. Mezi důležité parametry patří:

- **-a** - výpis všech názvů včetně skrytých
- **-l** - výpis v dlouhém formátu
- **-R** - rekurzivní výpis adresáře včetně všech jeho podadresářů

Příkaz má ovšem ještě mnoho jiných parametrů, které umožňují například řazení výpisu podle různých kritérií, výpis velikosti souborů v různých jednotkách a formátech, nebo třeba výpis i-nodu, kde se soubor nachází (parametr **-i**).

5.2 cd

Příkaz **cd** provádí změnu pracovního adresáře (název je z anglického change directory). Je to příkaz, který je zabudovaný přímo v shellu, takže nemá svojí manuálovou stránku a je popsán v manuálové stránce daného shellu. Jeho použití je ale tak jednoduché, že snad ani žádnou manuálovou stránku nepotřebuje. Jediný parametr[21], který se u něj používá je totiž název adresáře, který je navíc dokonce nepovinný. Jeho Již jsme si řekli, že kořenový adresář v Unixu označujeme lomítkem, takže je jasné, že **cd /** udělá náš pracovní adresář z kořenového. Pokud teď použijete příkaz **ls**, uvidíte základní strukturu adresářů v Unixu, o

[20] Používá se jak zápis ve tvaru i-node, tak ve tvaru inode
[21] Ve skutečnosti má ještě nepovinné volby -P a -L, ale těmi se zde nebudeme zabývat

které byla řeč v úvodních kapitolách. Pojďme ale zpátky do našeho domovského adresáře. Víme, že se nachází v adresáři **/home**, napišme tedy příkaz **cd /home** a hned si zase vypišme jeho obsah pomocí **ls**. Uvidíme adresáře všech uživatelů a pomocí **cd** se můžeme snadno přesunout do svého domovského adresáře (např. **cd test** pro uživatele test).
Pojďme ale zkusit totéž udělat ještě jednou trochu odlišným způsobem. Nejprve tedy cd / pro skok do kořenového adresáře, ale pak namísto dvou příkazů **cd /home** a **cd test** můžeme použít příkaz jediný, a sice **cd /home/test**.
A do třetice ještě jinak. Stiskněte kurzorovou klávesu **šipka nahoru**. Zjistíte, že na příkazové řádce se objeví naposledy použitý příkaz. Když budete šipku mačkat dále, budete postupně zpětně procházet historií zadaných příkazů. Šipkou dolů naopak seznamem procházíte dopředným směrem. Můžete tak snadno najít příkaz **cd /**, který jsme už zadávali a stisknutím klávesy Enter ho spustit znovu. Samozřejmě u takto jednoduchého příkazu to práci příliš neulehčí, ale až budete psát složité příkazy přes celý řádek, určitě si na historii rádi vzpomenete. Historie zůstane zachována dokonce i když se odhlásíte a znova přihlásíte a dokonce i po restartu počítače. Řádky vyvolané z historie samozřejmě také můžeme před odesláním klávesou Enter libovolně upravit.
Když jsme se tedy opět dostali do kořenového adresáře, ukážeme si nejrychlejší způsob jak se dostat do svého domovského. Domů se totiž odkudkoliv vrátíme pomocí **cd** zadaného bez dalších parametrů.
K adresářům si musíme říct ještě jednu věc. Když si vypíšeme jakýkoliv adresář včetně skrytých souborů, uvidíme v něm dva zvláštní soubory označené tečkou a dvěma tečkami:

```
TuX:~$ cd
TuX:~$ ls -al
total 18
drwxr-xr-x  4 root root 1024 2008-07-15 10:26 .
drwxr-xr-x 22 root root 1024 2008-07-10 11:32 ..
...
TuX:~$
```

Písmenko **d** v prvním sloupci u obou těchto souborů říká, že se jedná o adresáře[22].
Adresář s názvem jedna tečka **.** symbolizuje aktuální adresář.
Adresář s názvem dvě tečky **..** symbolizuje jeho nadřazený adresář.
Zkusme tedy například přejít do adresáře /etc/apache2/conf.d (příkazem **cd /etc/apache2/conf.d**). Pokuď teď chceme přejít o úroveň výše, provedeme **cd ..** Vraťme se pomocí historie (nebo pomocí **cd conf.d**) zpět a vyzkoušejme **cd ../..**, čímž se dostaneme tentokrát o dvě úrovně výše.

5.3 pwd

Všimli jste si, jak užitečná je výzva shellu? Díky ní (pokud ji máme rozumně nastavenou) neustále vidíme, v jakém adresáři se právě nacházíme. Viz například:

```
TuX:~$ cd /etc/apache2/conf.d
TuX:/etc/apache2/conf.d$ pwd
/etc/apache2/conf.d
TuX:/etc/apache2/conf.d#
```

[22] Pokud máme nastavený barevný výpis, vidíme to také podle modré barvy, kterou jsou adresáře standardně zobrazovány

Pokud bychom výzvu neměli takto nastavenou nebo pokud bychom třeba potřebovali v nějakém skriptu[23] zjistit v jakém adresáři se nacházíme, můžeme použít příkaz **pwd** (z anglického print working directory). Příkaz **pwd** nemá žádné parametry.

5.4 mkdir

Příkaz mkdir slouží k vytváření nových adresářů (název je ze slov make directory). Povinným parametrem příkazu je název nově vytvářeného adresáře. Pro jeho vyzkoušení si v domovském adresáři vytvořte adresář pokusy, který budeme používat při dalších činnostech. Postup bude tedy následující:

```
TuX:~$ cd
TuX:~$ mkdir pokusy
TuX:~$ ls
pokusy
TuX:~$
```

Pokud bychom chtěli v domovském adresáři vytvořit adresář pokusy z nějakého úplně jiného místa, mohli bychom to s našimi dosavadními znalostmi samozřejmě udělat třeba i takhle:

```
TuX:/var/www$
TuX:/var/www$ mkdir ~/pokusy
TuX:/var/www$ cd
TuX:~$ ls
pokusy
TuX:~$
```

Pokud bychom teď chtěli v adresáři pokusy vytvořit dva vnořené adresáře s názvy prvni a v něm vnořený druhy, můžeme to zkusit pomocí **mkdir ~/pokusy/prvni/druhy**. Jak je ovšem vidět:

```
TuX:~$ mkdir ~/pokusy/prvni/druhy
mkdir: cannot create directory `pokusy/prvni/druhy': No
such file or directory
TuX:~$
```

mkdir adresář nevytvoří a místo toho jen vypíše chybovou hlášku. To je proto, že adresář s názvem **prvni** doposud neexistuje a proto v něm příkaz **mkdir** nemůže vytvořit vnořený adresář **druhy**. Musíme tedy pracovat pomaleji a adresáře vytvářet postupně. V našem případě tedy:

```
TuX:~$ mkdir ~/pokusy/prvni
TuX:~$ mkdir ~/pokusy/prvni/druhy

TuX:~$
```

Toto by ovšem při vytváření složitějších struktur bylo velmi zdlouhavé, proto příkaz disponuje volitelným přepínačem -p, který způsobí, že si mkdir vytvoří všechny potřebné adresáře nižších úrovní automaticky. Takže správný příkaz zní:
mkdir -p ~/pokusy/prvni/druhy
Přepínač **-p** můžeme přitom umístit kamkoliv, takže i
mkdir ~/pokusy/prvni/druhy -p je platné zadání. Toto pravidlo ostatně platí obecně pro většinu příkazů a jejich parametrů v Unixu.

[23] Skripty jsou jednoduché programy psané v nějakém tzv. skriptovacím jazyce. Dostaneme se k nim později.

Druhým přepínačem příkazu **mkdir** je přepínač **-m**, který umožňuje přímo při vytváření nového adresáře nastavit uživatelská práva, ale tato možnost se příliš nevyužívá a práva se standardně nastavují příkazem **chmod**.
Pomocí příkazu **mkdir** můžeme také vytvořit několik adresářů najednou tím, že mu v parametrech předáme několik jejich názvů oddělených mezerami. Například **mkdir adr1 adr2 adr3** vytvoří v pracovním adresáři najednou tři nové adresáře s názvy adr1, adr2 a adr3.

5.5 rmdir

Příkaz **rmdir** (zkratka z anglického remove directory), jak už název napovídá slouží naopak k odstraňování prázdných adresářů. Podobně jako **mkdir** umí i on pracovat s několika adresáři zároveň, takže to, co jsme posledním příkladem k **mkdir** vytvořili, můžeme smazat příkazem: **rmdir adr1 adr2 adr3**. Podstatnou podmínkou ale je, že adresáře musí být naprosto prázdné. Pokud by v nich byl byť třeba i jen prázdný podadresář, příkaz **rmdir** nic nesmaže a jen vypíše chybovou hlášku. To je velmi rozumné chování, protože jinak bychom omylem mohli velmi jednoduše poničit celou adresářovou strukturu systému.
Pokud ovšem chceme smazat několikaúrovňovou strukturu pouze prázdných adresářů. můžeme to udělat podobně jako když jsme ji příkazem mkdir vytvářeli pomocí stejného přepínače -p. Takže například **rmdir pokusy/prvni/druhy -p** ve skutečnosti provede **rmdir pokusy/prvni/druhy/**, **rmdir pokusy/prvni/**, **rmdir pokusy/**, nebo chcete-li:
rmdir pokusy/prvni/druhy/ pokusy/prvni/ pokusy/

5.6 cp

Příkaz **cp** (z anglického copy) slouží ke kopírování souborů a adresářů. U všech příkazů, které berou nějaký zdrojový soubor a vytvářejí z něj nějaký jiný cílový se vždy užívá pořadí parametrů tak, jako by operace probíhala ve směru zleva doprava, tedy

příkaz zdroj ⟶ cíl

tedy první se uvádí zdrojový název a jako druhý cílový. Pojďme si příkaz cp vyzkoušet na několika příkladech. Soubory budeme kopírovat do adresáře pokusy, který jsme si vytvořili v našem domovském adresáři a protože zatím nevíme, jak se dají soubory vytvářet, budeme zatím kopírovat existující soubory z různých adresářů Linuxu. Proveďme tedy následující příkazy:

```
TuX:~$ cd
TuX:~$ cp /etc/hostname ~/pokusy/
TuX:~$ ls -l ~/pokusy/
total 4
-rw-r--r-- 1 test test 4 2008-07-19 18:56 hostname
TuX:~$
```

Tím jsme zkopírovali soubor s názvem **hostname**[24] z adresáře **/etc/** do našeho adresáře **pokusy**. Příkazem **ls** jsme se pak přesvědčili, že tam skutečně je.
Nyní se pojďme pokusit zkopírovat celý adresář. V adresáři **/etc/** se nachází podadresář **cron.daily/**[25], který budeme ze cvičných důvodů opět chtít zkopírovat do našeho

[24] Soubor obsahuje název počítače, tedy v našem případě text TuX
[25] Adresář obsahuje skripty, které se na počítači během každého dne automaticky jednou spustí

adresáře pokusy. Tentokrát se ale nejprve pomocí příkazu **cd pokusy** přesuneme přímo do tohoto adresáře. Pokud napíšeme:

```
TuX:~/pokusy$ cp /etc/cron.daily/
```

systém vypíše chybovou hlášku o tom, že jsme nezadali cíl kopírování a radí nám, abychom použili přepínač **--help**:

```
cp: missing destination file operand after
`/etc/cron.daily/'
Try `cp --help' for more information.
```

Jak ale máme zadat cíl kopírování, když chceme něco zkopírovat do adresáře ve kterém se právě nacházíme? Je to naštěstí jednoduché. Vzpomeňte na speciální skryté soubory, o kterých jsme již hovořili. Soubor s názvem tečka . přece označuje v jakémkoliv adresáři právě onen adresář. Zkusme tedy:

```
TuX:~/pokusy$ cp /etc/cron.daily/ ./[26]
```

I tentokrát ale příkaz vypíše jen chybovou hlášku:

```
cp: omitting directory `/etc/cron.daily/'
```

Ta pro změnu znamená, že příkaz vynechal adresář **/etc/cron.daily/**, ale to je právě ten adresář, který jsme chtěli kopírovat. Důvodem je to, že **cp** neumí sám o sobě adresáře kopírovat. Dá se mu ovšem snadno pomoci. Stačí přidat přepínač **-r**[27] a příkaz zkopíruje adresáře klidně včetně jakkoliv hluboké struktury podadresářů. Do třetice tedy zadejme správný příkaz:

```
TuX:~/pokusy$ cp -r /etc/cron.daily/ ./
TuX:~/pokusy$
```

Všimněte si, že pokud je příkaz úspěšný, nedává zpravidla žádný výstup. To je v Unixu obvyklé chování. Pokud systému nedůvěřujeme, můžeme se tedy pomocí **ls** přesvědčit, že adresář byl skutečně zkopírován:

```
TuX:~/pokusy$ ls
cron.daily  hostname
TuX:~/pokusy$
```

Pokud chceme vidět i celý obsah zkopírovaného adresáře, použijme u **ls** parametr **-R**.

```
TuX:~/pokusy$ ls -R
.:
cron.daily  hostname

./cron.daily:
apt       bsdmainutils  find       man-db  standard
aptitude  exim4-base    logrotate  ntp     sysklogd
TuX:~/pokusy$
```

[26] Možná je i varianta se samotnou tečkou bez lomítka

[27] **-r** je z anglického recursive a znamená tedy rekurzivní činnost příkazu do hloubky adresářů. Tento parametr je možné zadat i velkým písmenem **-R** a podobně funguje i u mnoha dalších příkazů. Pokud tedy někdy narazíte na nějaký příkaz, který budete chtít donutit aby pracoval s celými adresáři, hledejte pomocí man nejprve přepínač **-r** nebo **-R**

V Unixu platí (podobně jako i v jiných operačních systémech) „hvězdičková konvence“.
V Unixu má tento nástroj velmi široké možnosti, správně se nazývá **regulární výrazy**, a oproti jednodušším systémům jakými je třeba MS-DOS/Windows jeho činnost vykonává sám systém, nikoliv jednotlivé příkazy. To nám dává jistotu, že bude fungovat u všech příkazů, a navíc u všech naprosto shodně. Na tomto místě se nebudeme regulárními výrazy zabývat dopodrobna a jen si řekneme, že znak * zastupuje libovolný počet libovolných znaků (včetně nulového) a znak ? zastupuje právě jeden libovolný znak.
Pokud tedy provedeme například z adresáře **`pokusy/`** příkaz **`cp cron.daily/a* ../`** dosáhneme toho, že soubory **`apt`** a **`aptitude`** z tohoto adresáře budou zkopírovány do našeho domovského adresáře (**`../`** znamená o jeden adresář výše, než se právě nacházíme).
Příkaz **`cp`** má celou řadu dalších přepínačů. Pokud se s nimi chcete seznámit, RTFM neboli napište příkaz **`man cp`** a začtěte se do manuálu.

5.7 mv

Příkaz **`mv`** je příkazu **`cp`** velmi podobný. Jen namísto kopírování soubory přesouvá/přejmenovává[28]. Vytvořme si tedy v adresáři pokusy podadresář hokusy a přesuňme do něj z tamního adresáře **`cron.daily/`** všechny soubory začínající na písmeno s. Příkazy mohou tedy vypadat následovně:

```
TuX:~/pokusy$ mkdir hokusy
TuX:~/pokusy$ mv cron.daily/s* hokusy/
TuX:~/pokusy$ ls cron.daily/
apt  aptitude  bsdmainutils  exim4-base  find  logrotate
man-db  ntp
TuX:~/pokusy$ ls hokusy/
standard  sysklogd
TuX:~/pokusy$
```

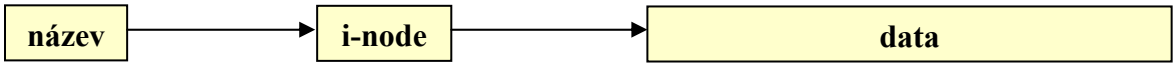
Pomocí **`ls`** jsme si ověřili, že příkaz provedl to, co jsme požadovali.
Teď můžeme ještě přejmenovat adresář cron.daily/ třeba na název skripty/:

```
TuX:~/pokusy$ mv cron.daily skripty
TuX:~/pokusy$ ls
hokusy  hostname  skripty
TuX:~/pokusy$
```

Další možnosti příkazu **`mv`** můžete jako obvykle najít v manuálových stránkách.

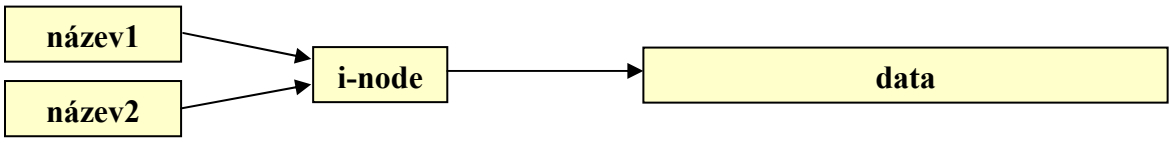
5.8 ln

Dostáváme se k příkazu **`ln`**[29], jehož plné pochopení vyžaduje trochu teorie o architektuře souborového systému. Už jsme si řekli, že **název souboru odkazuje na určitý i-node**, ve kterém jsou další technické údaje o souboru včetně **odkazu na místo, kde na disku leží skutečná data**.
Takže je možné si to představit nějak takhle:

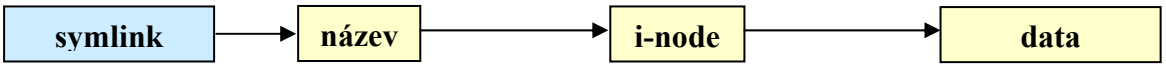


[28] Přejmenování souboru je vlastně totožné s přesunem souboru v rámci stejného adresáře pod nový název.
[29] Funkce příkazu **`ln`** ve skutečnosti umí realizovat pomocí svých přepínačů i další příkazy, například příkaz **`cp`**.

Příkaz **ln** slouží k navázání dalších názvů na existující i-node, což si můžeme představit nějak takhle:



Tomuhle linku se říká **hardlink**. ln ale umí navázet názvy ještě druhým způsobem. Jde o vytvoření takzvaného **symbolického linku**. Tomu je podobný „zástupce“, kterého známe například ze systému MS Windows. Znázornit to můžeme následovně:



Symbolický link je ale ve skutečnosti zcela nový soubor (se svým názvem, i-nodem a datovou částí obsahující cestu k nalinkovanému souboru), takže výše uvedený obrázek je značně zjednodušen. Jedinou nevýhodou hardlinku je, že díky plné závislosti na i-nodech nemůže odkazovat mezi dvěma různými disky. Symbolický link naproti tomu ano.
Dost bylo teorie, pojďme nyní udělat s příkazem **ln** několik pokusů. V adresáři **pokusy** bychom měli mít z minulých pokusů soubor **hostname**. Vytvořme na něj dva hardlinky s názvy **noname** a **nemame** a jeden symbolický link s názvem **symname**:

```
TuX:~/pokusy$ ln hostname noname
TuX:~/pokusy$ ln hostname nemame
TuX:~/pokusy$ ln -s hostname symname
```

Jak je vidět, vytvoření symbolického linku se od hardlinku liší jen tím, že v něm použijeme přepínač **-s**. Všimněte si, že i u příkazu **ln** platí pořadí parametrů zdroj -> cíl zleva doprava.
Nyní se pojďme podívat, co jsme vytvořili. použijeme k tomu samozřejmě náš oblíbený příkaz **ls** a trošku si pohrajeme s jeho parametry. Obyčejný **ls** vypíše jen názvy souborů:

```
TuX:~/pokusy$ ls
hokusy  hostname  nemame  noname  skripty  symname
TuX:~/pokusy$
```

Z výpisu není zřejmé, že **hostname**, **nemame** a **noname** představují identický soubor a **symname** je symbolický link na tentýž soubor. Jediné čeho si můžeme všimnout u barevného výpisu je azurová barva symbolického linku.

Daleko zajímavější je výpis příkazu **ls** s přepínačem **-l**, tedy dlouhý výpis s podrobnostmi:

```
TuX:~/pokusy$ ls -l
total 20
drwxr-xr-x 2 test test 4096 2008-07-19 21:19 hokusy
-rw-r--r-- 3 test test    4 2008-07-19 18:56 hostname
-rw-r--r-- 3 test test    4 2008-07-19 18:56 nemame
-rw-r--r-- 3 test test    4 2008-07-19 18:56 noname
drwxr-xr-x 2 test test 4096 2008-07-19 21:19 skripty
lrwxrwxrwx 1 test test    8 2008-07-19 22:07 symname ->
hostname
TuX:~/pokusy$
```

Všimněme si, že datum a čas vytvoření hardlinků je stejný a odpovídá času vytvoření původního souboru i když jsme linky vytvářeli kdykoliv později. Čas vytvoření je totiž podobně jako většina dalších údajů zapsán v i-nodu. Dále si u souborů **hostname**, **nemame** a **noname** všimněme čísla **3** před slovem **test**[30]. Právě toto číslo říká, kolik linků ukazuje na daný i-node. U symlinku je číslo **1**, protože je to zcela nový soubor. Také čas jeho vytvoření je skutečný a neváže se na čas vytvoření původního souboru. Za šipkou navíc vidíme cestu, na kterou symlink směřuje. Poslední zajímavý údaj je pro nás první písmenko v prvním sloupci. Vidíme, že u adresářů je tam písmeno **d** (directory), u obyčejných souborů (tedy i hardlinků) je tam pomlčka **-** a u symlinků je tam písmeno **l** (link).
Vůbec nejzajímavější bude ale výpis u kterého navíc použijeme přepínače **-a** a **-i**:

```
TuX:~/pokusy$ ls -lai
total 28
32648 drwxr-xr-x 4 test test 4096 2008-07-19 22:07 .
32641 drwxr-xr-x 3 test test 4096 2008-07-19 18:56 ..
32661 drwxr-xr-x 2 test test 4096 2008-07-19 21:19 hokusy
32647 -rw-r--r-- 3 test test    4 2008-07-19 18:56 hostname
32647 -rw-r--r-- 3 test test    4 2008-07-19 18:56 nemame
32647 -rw-r--r-- 3 test test    4 2008-07-19 18:56 noname
32649 drwxr-xr-x 2 test test 4096 2008-07-19 21:19 skripty
32662 lrwxrwxrwx 1 test test    8 2008-07-19 22:07 symname
-> hostname
TuX:~/pokusy$
```

V prvním sloupečku vidíme nyní jedinečné číslo i-nodu. Teď konečně můžeme bez jakýchkoliv pochyb říci, že názvy **hostname**, **nemame** a **noname** ukazují na jediný i-node (zde konkrétně 32647). Oproti tomu **symname** je úplně nový soubor s novým i-nodem. Podívejte se teď ještě na soubory **.** a **..**[31] Proč mají tolik hardlinků (soubor **. 4** a soubor **.. 3**)? Zamysleme se například nad adresářem **.** . Adresář tečka představuje sám sebe. Jsme v adresáři pokusy, tedy **.** = **pokusy**. Tenhle adresář má ale navíc dva podadresáře **hokusy** a **skripty**. A v každém z nich je přeci adresář **..**, který odkazuje na nadřazený adresář, tedy opět na adresář **pokusy**. Když to tedy podtrhneme a sečteme, dostaneme: **pokusy**, jednou **.** a dvakrát **..**, tedy přesně čtyři hardlinky na i-node adresáře pokusy. Kdo nevěří, ať si zapamatuje i-node tečky, a pak si podobně vypíše oba podadresáře a porovná si s tímto číslem číslo i-node tamních dvou teček.

[30] Při psaní této příručky používám účet test, jinak tam každý bude mít své vlastní uživatelské jméno.
[31] Kvůli nim jsme u příkazu **ls** použili přepínač **-a**

I u příkazu **ln** samozřejmě najdete v manuálových stránkách daleko více přepínačů, ale základ, který jsme si zde vysvětlili pro běžnou práci bohatě stačí.

5.9 rm

Už máme pár zkušeností s některými příkazy a s chováním operačního systému, takže se můžeme vrhnout na poněkud nebezpečnější příkazy[32]. Příkaz **rm** (z anglického remove) je totiž určen k odstraňování souborů.
Pomocí parametrů se mu předávají názvy souborů ke smazání, kterých může být i více než jeden. Samozřejmě lze při zadání využít i regulární výrazy. To z příkazu **rm** činí nástroj, který dokáže mazat velmi rychle. **rm *** smaže všechny soubory v aktuálním adresáři. Pokud si nejste jisti a chcete, aby se u regulárních výrazů o mazání jednotlivých konkrétních souborů ještě ujišťovat, použijte u **rm** přepínač **-i**. Ten bude u každého souboru klást otázku, zda se má opravdu mazat a odpovíte-li na ni čímkoliv jiným než písmenem **y** (z anglického yes), soubor mazat nebude:

```
TuX:~/pokusy/skripty$ rm * -i
rm: remove regular file `man-db'? y
rm: remove regular file `sysklogd'? y
```

Naopak pokud není žádoucí, aby **rm** kladl jakékoliv otázky[33], je vhodné použít přepínač **-f** (z anglického force).
Schválně jestli odhadnete, co by mohlo znamenat **rm -rf ***. Ano jedná se o nekompromisní smazání všeho co se nachází v aktuálním adresáři, a to včetně rekurzivního mazání všech jeho podadresářů. Bez přepínače **-r** příkaz **rm** adresáře mazat nedokáže, k tomu je přece již zmiňovaný příkaz **rmdir**, ale s přepínačem **-r** funguje i na ně, a to dokonce i když jsou neprázdné. Proto se k mazání adresářů používá možná i častěji než samotný **rmdir**. Při jeho použití totiž nemusíme přemýšlet, zda je mazaný adresář prázdný nebo jestli dokonce neobsahuje nějakou strukturu podadresářů.
Více o **rm** jako obvykle najdete pomocí **man rm**.

6 Standardní vstup, výstup, přesměrování a roura

Většina příkazů v Unixu pracuje tak, že očekává vstup dat ke zpracování z klávesnice a svůj výstup naopak odesílá na obrazovku. Těmto dvěma zařízením se říká **stdin** a **stdout**.

6.1 cat

Příkaz **cat** je na první pohled strašně nezajímavý. Bez parametrů nedělá totiž nic jiného, než přepisování standardního vstupu na standardní výstup. Než si to vyzkoušíme, musíme si ale říci, jak mu dáme najevo, že má se svojí prací skončit. Provádí se to kombinací kláves **Ctrl-D** na samostatné řádce. Spusťme tedy **cat** a začněme mu zadávat data:

```
TuX:~$ cat
ahoj
ahoj
nazdar
nazdar
TuX:~$
```

[32] Ono například i při nesprávném použití příkazů **cp** a **mv** lze v systému nadělat pěknou paseku, ale nemalujme čerta na zeď.
[33] Hodí se to zejména při psaní skriptů, jejichž běh by otázka přerušila.

Ve výše uvedeném příkladu jsem zadal slovo ahoj, stisknul **Enter**, zadal nazdar, stisknul **Enter** a nakonec stisknul **Ctrl-D**. Je vidět, že po každém stisku klávesy **Enter** příkaz ze stdin zadaná data prostě jen opíše na stdout, takže každý řádek je pak na obrazovce dvakrát.
Zdá se to na první pohled nesmyslné a nudné, ale přesto je právě **`cat`** v Unixu jedním z nejpoužívanějších příkazů[34].
Pokud příkazu zadáme jako parametr název souboru, pak nebere data ze standardního vstupu, ale právě s tohoto souboru. cat tedy umí vypsat obsah souboru na obrazovku. Konečně si můžeme například prohlédnout obsahy souborů, se kterými jsme v předešlých cvičeních pracovali:

```
TuX:~$ cat /home/test/pokusy/hostname
TuX
TuX:~$
```

Z tohoto krátkého výstupu je vidět, že soubor hostname[35] obsahuje opravdu jen název počítače (v našem případě 3 znaky a neviditelný znak konce souboru, proto je v našem případě jeho velikost 4 Byty, jak jsme si mohli všimnout například při experimentech s příkazem **`ls`**). Pokud se pokusíme vypsat nějaký delší soubor, zkuste například **`cat /etc/services`**, výpis projede velkou rychlostí obrazovkou a zastaví se na ní teprve až poslední stránka. Tak se zdá, že si takto moc nepočteme :-(
Nevěšte hlavu, hned si ukážeme jeden z nejmocnějších nástrojů Unixu, který nám pomůže podobné problémy snadno řešit.

6.2 Přesměrování

Co kdybychom potřebovali výstup z nějakého příkazu dostat někam jinam než na obrazovku? Pomůže nám takzvané přesměrování, které se v Unixu řeší znaky **>** a **<**.
Znak > znamená **přesměrování standardního výstupu** někam jinam. Vyzkoušejme tedy následující:

```
TuX:~/pokusy/hokusy$ cd ~/pokusy
TuX:~/pokusy$ cat > novytext
Toto je novy textovy soubor.
Ma dve radky.
TuX:~/pokusy$
```

Normálně **`cat`** opisuje standardní vstup na standardní výstup. Teď jsem ho ovšem přesměrovali do souboru **`novytext`**, takže jsme schopni jednoduchým způsobem vytvářet soubory[36]. K textovému editoru to má samozřejmě daleko. Když například uděláme chybu, nelze ji takto opravit, ale nový soubor jsme takto nepochybně vytvořili. Nakonec přesvědčit se o tom můžeme příkazem **`ls`** a obsah souboru si můžeme nechat vypsat opět pomocí **`cat`**. Jestliže nyní stejný příkaz vyzkoušíme ještě jednou[37], ale zapíšeme jiný text zjistíme, že přesměrování původní soubor přepíše, aniž by se nás na cokoliv ptalo.
Pokud nám to vadí a raději bychom dopisovali nová data na konec původního souboru, stačí namísto jednoho **>** použít **dvojici těchto znaků** >>. Jestliže požadovaný soubor zatím

[34] Příkaz **`cat`** umí popravdě i spojovat soubory, odtud (z anglického concatenate) je i jeho název, ale paradoxně se k této činnosti používá velmi zřídka.
[35] Standardně je tento soubor v adresáři **`/etc`**, takže pokud jste si ho třeba při experimentech ze svého pokusného adresáře smazali, můžete použít **`cat /etc/hostname`**
[36] Práci opět ukončujeme pomocí **Ctrl-D** na samostatné řádce.
[37] Vzpomeňte na historii shellu. stačí opakovaně stisknout klávesu **šipka nahoru** a k původnímu příkazu se snadno vrátíme.

neexistoval, Unix ho vytvoří, ale pokud již existuje, nová data budou pouze připsána na jeho konec.
Znak < naopak znamená **přesměrování standardního vstupu**. Ukážeme si to pro jednoduchost zatím zase jen pomocí **cat**. Přesměrujeme-li do příkazu **cat** náš soubor **novytext**, **cat** ho prostě vypíše na obrazovku:

```
TuX:~/pokusy$ cat <novytext
Toto je novy textovy soubor.
Ma dve radky.
TuX:~/pokusy$
```

Tento příklad je poněkud nesmyslný, protože víme, že pro tento případ můžeme příkazu **cat** zadat název souboru k vypsání jako parametr **cat novytext**, nicméně funguje tak, jak bychom očekávali.
cat dokonce takto může v jistých případech nahradit příkaz **cp**. Vyzkoušejte:

```
TuX:~/pokusy$ cat >dalsitext <novytext
```

Nezáleží přitom, zda napíšeme přesměrování v tomto pořadí nebo použijeme-li naopak:

```
TuX:~/pokusy$ cat <novytext >dalsitext
```

Také si uvědomte, že **<novytext** a **>dalsitext** nejsou parametry příkazu **cat**, ale od tohoto příkazu naprosto oddělené přesměrování **stdin** a **stdout**.
Pomocí několikanásobného zopakování příkazu:

```
TuX:~/pokusy$ cat <novytext >>dalsitext
```

můžeme text ze souboru **novytext** do souboru **dalsitext** několikanásobně namnožit. To už bychom příkazem **cp** dokázali těžko.

6.3 Roura

Už umíme přesměrovat výstup programu do souboru nebo naopak data ze souboru do vstupu programu. Jak je vidět, u přesměrování je vždy jedním ze členů operace nějaký soubor. Často je ale potřeba přesměrovat výstup jednoho programu do jiného programu k dalšímu zpracování. Právě k tomuto účelu slouží znak pipe (česky roura) **|**. Vraťme se teď k našemu problému s výpisem dlouhého souboru pomocí cat. Text souboru rychle projel monitorem a na obrazovce nám zbyla jen jeho poslední stránka. V Unixu ale existuje program, který vypíše na obrazovku pouze jednu stránku dat ze standardního vstupu. Na jejím konci napíše **more** (česky dále) a očekává stisk mezerníku[38]. Po něm zobrazí další stránku a tak dále. Program se podle svého charakteristického nápisu jmenuje právě **more**. Zkusme teď činnost programů cat a more propojit právě prostřednictvím roury (níže uvedený výpis je zkrácen):

```
TuX:~$ cat /etc/services | more
# Network services, Internet style

...
msp    18/tcp             # message send protocol
--More--
```

[38] Mimo mezerníku můžeme stisknout i některé jiné klávesy. Například **Enter** způsobí výpis pouze příští řádky, klávesa **q** znamená okamžité ukončení výpisu.

I tento příklad je zbytečně složitý, protože i programu **more** můžeme rovnou parametrem říci, který soubor má vypsat, zde tedy **more /etc/services**, ale už třeba několikastránkový výpis programu **ls** jinak než pomocí roury příkazem **more** zpracovat nelze. Vyzkoušejte například **ls –la /etc | more**.
Programy, které berou data ze standardního vstupu, nějak je zpracují a výsledek pošlou na standardní výstup někdy také nazýváme filtry. V Unixu je jich veliké množství, zde si ovšem stručně představíme pouze několik takových, které se při práci s tímto operačním systémem používají téměř notoricky.

7 Filtry

Jednotlivé příkazy jsou zde popsány jen velmi stručně. Většina z nich nabízí širokou škálu různých přepínačů takže pro hlubší informaci o nich je vhodné navštívit patřičné manuálové stránky pomocí známého **man *prikaz***.

7.1 cat

Již uvedený program **cat** je vlastně jakýmsi prázdným filtrem, který nic nedělá, jen předává data ze vstupu na výstup.

7.2 more

Filtr **more** slouží k prohlížení dlouhých výstupů po jednotlivých stránkách. často se používá také přímo, tak, že mu parametrem předáme název souboru, kterým chceme listovat.

7.3 tac

Název příkazu **tac** je opačně napsaný název **cat**. A podle toho se příkaz i chová. Přehodí totiž při výpisu **pořadí řádků**. První bude posledním a naopak poslední bude na jeho výstupu prvním.

7.4 rev

Podobně se chová i příkaz **rev**, který ale nepřehazuje pořadí řádkků, ale pořadí písmen na každém řádku. Jednotlivé řádky tedy **píše pozpátku**. Pro vyzkoušení posledních dvou příkazů a ověření toho, že roury můžeme řetězit napište příkaz:

```
ls -la /etc | rev | tac
```

:-)))

7.5 wc

Ne, tento příkaz nemá se záchodem nic společného. Jeho zkratka totiž pochází z anglického word count (neboli počet slov). Pomocí tohoto filtru můžeme snadno udělat takové věci, jako například spočítat soubory v adresáři:

```
TuX:~# ls /etc/ | wc –w
148
TuX:~# ls -d /etc/*/ | wc –w
59
TuX:~#
```

První příkaz spočte všechny soubory a adresáře z adresáře **/etc**, druhý spočte pouze podadresáře tohoto adresáře.

7.6 sort

Filtr **sort** jak i jeho název prozrazuje dokáže seřadit řádky vstupu podle různých kritérií, které mu lze zadat pomocí jeho přepínačů.

7.7 tee

Zajímavým i když ne tak často používaným filtrem je program **tee** (anglická výslovnost písmena T). Chová se totiž jako jakýsi T-článek vřazený v potrubí tvořeném Unixovými rourami. Data, která tímto potrubím tečou zároveň kopíruje do souboru.
Pokud tedy například zapíšeme následující už docela dlouhý příkaz:

```
TuX:~/pokusy# ls /etc | tee popredu | rev | cat > pozpatku
```

vytvoříme jím zároveň dva soubory ve kterých je výpis příkazu **ls** popředu a pozpátku, jak i jejich názvy napovídají.

7.8 grep

Filtr **grep** je jedním ze skutečně nejužitečnějších. Jeho název je zkratkou anglických slov global regular expression print a jak z něj vyplývá, filtruje vstupní data pomocí regulárních výrazů. Pomocí regulárních výrazů se dají vytvářet i velmi složité konstrukce a jejich výklad by vyžadoval několik samostatných kapitol, proto zde uvedu jen několik ukázkových příkladů (pro jednoduchost s pomocí výpisu adresáře **ls /etc**).
ls /etc | grep se – vypíše pouze řádky, které obsahují řetězec „se“
ls /etc | grep ^se – vypíše jen ty, které řetězcem „se“ začínají
ls /etc | grep [a,u]se – vypíše řádky obsahující znaky se před kterými je buďto písmeno a nebo u
ls /etc | grep ^a.*se – vypíše řádky začínající znakem „a“ a obsahující řetězec „se“
Zájemci najdou k regulárním výrazům a příkazu **grep** velké množství informací na Internetu i v manuálových stránkách samotného příkazu.

7.9 tail

Slovo **tail** znamená konec nebo ocas a vypisuje tedy ze vstupu pouze poslední řádky. standardně jich je 10, ale pomocí přepínače **-n** jich můžeme stanovit libovolný počet. Tento příkaz se mimo použití v roli filtru velmi často používá také přímo s parametrem, který udává název souboru, ze kterého se má vypisovat. Nejčastěji se tímto způsobem díváme do systémových logů, což jsou obvykle velmi dlouhé soubory ze kterých nás nejvíce zajímá právě poslední vývoj událostí. Typické je:

```
TuX:~/pokusy# tail /var/log/syslog
```

Možná se vám zdá, že ve výpisu je více než 10 řádků. Ale to je jen proto, že řádky v logu jsou tak dlouhé, že se nevejdou na jediný řádek displeje. Dokázat to můžeme například pomocí **tail /var/log/syslog | wc -l**.
Příkaz umí navíc i živě vypisovat změny na konci souboru. Provede se to přepínačem **-f**. Můžeme se tak například podívat jak přibývá log web serveru apache, když se podíváme na stránky, které server poskytuje: **tail -f /var/log/apache2/access.log**
Příkaz pak ukončíme pomocí klávesové kombinace **Ctrl-C**.

7.10 head

Příkaz head dělá opak toho co tail. Vypisuje jen prvních 10 řádek ze vstupu. Také u něj můžeme pomocí přepínačů ovlivnit chování. Kombinací obou pak můžeme například vypsat jen jeden určitý řádek ze vstupu:

```
TuX:~/pokusy# ls –l  /etc | head -n 6 | tail -n 1
drwxr-xr-x 7 root   root     1024 2008-07-10 19:32 apache2
TuX:~/pokusy#
```

Touto kombinací filtrů jsme nejprve vyfiltrovali prvních 6 řádků z výstupu příkazu **ls**, ale vzápětí jsme z této šestice pomocí **tail** vypsali jen poslední jeden řádek. Výsledkem je tedy pouze šestý řádek výstupu příkazu.

7.11 awk a sed

Do kategorie filtrů bychom v podstatě mohli zařadit i příkazy **awk** a **sed**. Jsou to příkazy určené pro složitou manipulaci s textovými řetězci. Zavádějí vlastní programovací jazyky, pomocí kterých se s řetězci manipuluje, ale jejich popis je samozřejmě nad rámec této stručné publikace. Proto pro určitou představu jen po jednom příkladu od každého:
ls /etc | sed -e 's/o/0/g' – zamění ve výpisu písmeno „o“ za nulu
ls -il /etc | awk '{print $1, $3, $6}' – vybere z výpisu **ls** jen i-node, počet linků a velikost souboru (první, třetí a šestý údaj na každém řádku).
awk a **sed** jsou opravdu velmi mocné univerzální nástroje a zájemcům o další informace doporučuji četné tutoriály (některé jistě i v češtině), které lze najít na Internetu.

8 Textové editory

V minulé kapitole jsme si ukázali některé vybrané filtry, pomocí kterých s našimi současnými znalostmi chování systému už dokážeme zvládat mnohé náročné úkoly. Přesto plnohodnotná editace textu pouze pomocí těchto příkazů a jejich kombinací by byla velmi náročná a pomalá (i když určitě ne nemožná). Proto se v Unixu velmi brzy po jeho prvním uvedení objevují první jednoduché řádkové editory (nejprve **ed** později rozšířený na **ex**), které se později vyvíjejí na takzvané vizuální editory. Řádkové editory uměly editovat vždy pouze jediný řádek textu samostatně, takže pro editaci se vždy musel nejprve zvolit řádek, zeditovat a následně zase zvolit další, kdežto „vizuální“ editory už editují volně celý text.

8.1 vi

Zcela tradičním editorem Unixu je program s názvem **vi**[39]. Editor vi je velmi mocný nástroj a kdysi obrovsky předběhl svoji dobu. V dnešní době, kdy jsme si zvykli vše ovládat zcela intuitivně myší už vypadá poněkud staromódně, ale na druhou stranu se v něm dají i dnes dělat kouzla, která „myší“editory zvládají jen s velkými obtížemi. Proto má editor **vi** řadu skalních uživatelů, ale i lidí kteří pokud mohou, se mu vyhnou velkým obloukem. Musím přiznat, že sám patřím do té druhé skupiny. I přesto je dobré alespoň základní filozofii tohoto editoru znát, protože mnohé další programy jsou na ní založeny a používají podobné ovládání. Editor **vi** má dva režimy, textový a příkazový. Do příkazového režimu se dostaneme stiskem klávesy **Esc**. Do textového zadáním příkazu **i** nebo **a**. Právě zvládnutí této modality editoru je pro začátečníky velmi náročné, ale po čase zjistíte, že se s tím dá žít a pracovat i poměrně rychle.

[39] V roce 1976 ho napsal Bill Joy pro jednu z prvních BSD verzí Unixu. Název **vi** je zkratkou slova visual.

Ukažme si malý příklad editace. V adresáři **~/pokusy/** napište **vi vitext**. Tím vytvoříme nový soubor s názvem **vitext** a zahájíme jeho editaci[40]. Po otevření se nacházíme v příkazovém režimu. Stiskneme proto **i** (přepnutí do režimu insert) a vložíme třeba následující text:

```
Tohle je prvni
text napsany
v editoru vi.
```

Stiskneme **Esc** čímž se přepneme do příkazového režimu a napíšeme **ZZ** (velkými písmeny). Tím dojde k uložení textu a ukončení editoru.
Pojďme nyní text upravit na:

```
Tohle je prvni
textik napsany
v podivnem editoru vi.
```

Příkazem **vi vitext** se vrátíme k editaci našeho souboru. Budeme chtít vložit slovo „podivném“. Jsme v příkazovém režimu a když napíšeme **/edi**, najde nám **vi** místo s řetězcem „edi“. Kurzor tedy bude stát na e v tomto slově. Dostaneme se do vkládacího režimu písmenem **i** a dopíšeme „podivnem “. Teď se potřebujeme dostat o řádek výše. Stiskneme **Esc** a v příkazovém režimu stiskneme klávesu **k** a pak opakovaně klávesu **h** tak, abychom se dostali těsně za slovo text. Stiskneme **i** a dopíšeme „ik “. A opět soubor uložíme sekvencí **Esc ZZ**.

Alespoň pár základních příkazů:

- **h**, **j**, **k**, **l** – kurzor doleva, dolů, nahoru, doprava
- **w**, **b**, **0**, **$** - kurzor o slovo doprava, doleva, na začátek řádky, na konec řádky
- **x**, **dd**, **u**, **.** – smazat znak, smazat řádku, zpět poslední příkaz, znova poslední příkaz
- **:w**, **:q!**, **ZZ** – uložení souboru, vyskočení z **vi** bez ukládání, uložení a vyskočení

Zájemci o další informace najdou na Internetu mnoho různých tutoriálů a kurzů práce s **vi**.

8.2 nano

Příjemnější textový editor, který je navíc zahrnut už v základní instalaci Debiana se jmenuje **nano**. Název připomíná starší oblíbený editor **pico**, který jím byl nahrazen a se kterým je kompatibilní. **nano** spustíme podobně jako **vi**, tedy příkazem **nano** následovaným názvem souboru.
I tento editor má své mouchy (například při práci prostřednictvím PuTTY v něm nefunguje numerická klávesnice a čísla se musí zadávat z normální velké klávesnice), ale jeho ovládání je velice intuitivní a blízké editorům, na které jsme zvyklí z Windows. Na spodních dvou řádkách zobrazuje jednoduché menu, takže uživatel si ovládací klávesy nemusí pamatovat, ale má je neustále před sebou.

8.3 Další editory

Debian nabízí celou řadu balíčků s dalšími editory a jejich různými rozšířeními. Na našem cvičném serveru je nainstalován ještě editor **joe** a **ne**. Každý z nich má své klady a zápory a je asi nejvíce otázkou zvyku, ve kterém z nich se vám bude nejlépe pracovat. Tak si je vyzkoušejte a rozhodněte se pro ten, který vám bude sedět nejlépe[41].

---

[40] Pokud by soubor už v té době existoval, otevře se ve **vi** a můžeme ho upravovat.
[41] Velmi užitečným programem pro práci s Linuxovými servery z operačního systému MS Windows je mimo programu PuTTY také freewarový program WinSCP. Umožňuje přenos souborů mezi MS Windows a počítačem s Linuxem, takže je můžete editovat libovolným editorem přímo ve Windows, což je zejména u rozsáhlejších souborů daleko příjemnější.

Kdykoliv bude v této příručce řeč o editaci, budu jako editor používat **nano**, takže pokud si oblíbíte některý jiný, nahrazujte si v dalším textu výrazy typu **nano nazev_souboru** tím svým oblíbeným, například **ne nazev_souboru**.
Nebo snad **vi nazev_souboru** ? ☺

Obrázek 10 - editor nano

9 Procesy

Až doposud jsme se de facto zabývali pouze první složkou systému Unix, kterou je souborový systém. Ale jak praví ona okřídlená věta: „... co v Unixu nejsou soubory, mohou být už jen procesy.“ Pojďme se tedy alespoň na chvilku zastavit u toho, co to procesy jsou, a jak se s nimi dá pracovat.

9.1 Rozdělení procesů

Unix je mnohouživatelský víceúlohový systém. To znamená, jak už víme, že na jednom počítači může být přihlášeno zároveň více uživatelů a dokonce každý uživatel může pracovat na více úlohách zároveň[42]. Systém tedy musí mít v sobě zákonitě zabudovanou velmi propracovanou správu všech těchto úloh[43]. Každá taková spuštěná úloha se nazývá v Unixu procesem.
Některé procesy jsou spouštěny samotnými uživateli po přihlášení k terminálu. Ty se nazývají **interaktivní procesy** a jsou závislé právě na tom, že běží terminál (mimochodem samozřejmě také proces).
Ale mnoho dalších procesů musí být zahájeno už při bootování systému a spouštění různých služeb zcela nezávisle na činnosti uživatelů. Takovým procesům říkáme **démoni**. Většinou se chovají tak, že jsou při bootování systému spuštěny a na pozadí pouze čekají, až pro ně přijde nějaký požadavek na činnost. Typickým příkladem jsou například procesy zajišťující službu webserveru. Jsou spuštěny a nedělají nic do chvíle, než přijde ze sítě požadavek na nějakou

[42] Aniž bychom si to uvědomovali, tak i veškerá naše dosavadní činnost probíhala ve více úlohách najednou.
[43] Mimochodem právě tato správa úloh byla základem prvních pokusů Linuse Torvaldse

webovou stránku. V tom okamžiku se postarají o její odeslání, a poté opět ustoupí na pozadí a vyčkávají na další požadavek.
Ještě existuje jeden druh procesů. Říkáme jim **automatické procesy**. Tyto procesy také nepotřebují k činnosti terminál. Jsou spouštěny buďto v určitých předem nastavených časových intervalech, nebo sledují okamžitý výkon systému a pokud klesne pod určitou prahovou hodnotu, samy se rozbíhají.

9.2 Informace o procesech

Správce procesů si musí o běžících procesech udržovat tabulku informací. Nejdůležitějším údajem o každém procesu je jeho PID (Process IDentificator). Dalšími důležitými parametry jsou například stav procesu, identifikátor rodičovského procesu, hodnota udávající jak moc proces zatěžuje procesor atd.

9.3 ps

Základní příkaz, kterým vypisujeme informace o procesech z této tabulky se jmenuje **ps**. Napíšeme-li tento příkaz, uvidíme zhruba tyto údaje:

```
TuX:~$ ps
  PID TTY          TIME CMD
 8973 pts/0    00:00:00 bash
 8993 pts/0    00:00:00 ps
TuX:~$
```

Na první pohled se zdá, že procesů běží nějak podezřele málo. Ale to je jen proto, že vidíme jen své vlastní interaktivní procesy. Stav přesně odpovídá tomu, co děláme. V momentě, kdy jsme příkaz spustili, běžel určitě shell (příkazový interpretr) s názvem **bash** a pod ním jsme pustili proces **ps**.
Spustíme-li **ps** například s parametry **xjf**, uvidíme o procesech daleko podrobnější informace:

```
TuX:~$ ps xjf
 PPID  PID PGID  SID TTY  TPGID STAT UID TIME COMMAND
 9010 9012 9010 9010 ?       -1 R   1001 0:00 sshd:test@pts/0
 9012 9013 9013 9013 pts/0 9034 Ss  1001 0:00  \_ -bash
 9013 9034 9034 9013 pts/0 9034 R+  1001 0:00      \_ ps xjf
TuX:~$
```

Například údaj **PPID** udává číslo rodičovského procesu, takže z prvních dvou sloupců můžeme krásně odvodit hierarchii procesů, kterou nám **ps** dokonce ve sloupci **COMMAND** vyjádří názorně graficky.
Příkaz **ps** má parametrů hodně a navíc se bohužel výrazně liší v BSD a AT&T verzích Unixu. V Linuxu je **ps** sice kompatibilní s oběma typy parametrů, ale chaos vzniklý touto nejednotností zbytečně uživatele mate. Nicméně pro běžnou práci uživatel potřebuje vpodstatě jen dvě formy spuštění **ps**. Samotné **ps**, které jak jsme si již ukázali, vypíše jen procesy patřící danému uživateli. Druhá zajímavá možnost je **ps ax**. Parametry **ax** (píše se bez pomlčky) způsobí, že **ps** zobrazí úplně všechny procesy běžící v danou chvíli na daném systému. Jen pro zajímavost, když ho provedu v tomto momentě na našem cvičném serveru, napočítám 51 běžících procesů. Samozřejmě detaily o **ps** se zájemci jako obvykle dozví v obsáhlých manuálových stránkách k tomuto příkazu.

9.4 Život a komunikace procesu

Každý proces musí být nějakým způsoben zahájen, ukončen a musí být schopen komunikovat s okolím. Zde si v naprosté stručnosti i za cenu některých zjednodušení a nepřesností uděláme představu o průběhu života procesu.

Zahájení každého procesu se děje metodou zvanou **fork** (vidlička). Spočívá to v tom, že rodičovský proces nejprve okopíruje sám sebe se všemi svými atributy, a teprve poté do této své kopie vloží skutečný kód nově vznikajícího procesu.

Při **řádném ukončení** naopak proces předá svému rodiči návratový kód, ze kterého rodič pozná, jakým způsobem proces skončil (například zda nedošlo k nějaké závažné chybě atd.).

Komunikace procesu s okolím se děje pomocí takzvaných **signálů**. signálů existuje celá řada, každý má své číslo, zkratku a pro proces určitý význam. Výpis všech signálů můžeme provést příkazem **`kill -l`**.

9.5 Stavy procesu, fg a bg

Stavy procesu si nejlépe vysvětlíme na několika následujících příkladech. Doposud jsme používali jen programy, které vykonaly nějakou činnost zpravidla v neznatelně krátkém čase, skončily a předaly řízení zpět shellu (například náš oblíbený **`ls`** a i většina dalších). Existují ale i programy, jejichž činnost trvá dlouhou nebo dokonce i nekonečnou dobu. Takové programy je potřeba umět nejen spustit, ale i ukončit. **Ukončení procesu** běžícího v interaktivním režimu se provádí klávesou **Ctrl-C**. Vyzkoušejme to na jednoduchém programu s názvem **`yes`**, který nedělá nic jiného, než že opakovaně na obrazovku zobrazuje písmenko y, a to do té doby, než ho ukončíme[44]. Napište příkaz **`yes`** a až se nabažíte opakujícího se **`y`**, ukončete ho klávesou **Ctrl-C**.

Dovedete si jistě představit, co by udělal příkaz **`yes > soubor-yyyy`**. Ano, posílal by opakující se **`y`** do souboru s názvem **`soubor-yyyy`**. Pokud tento pokus chcete udělat, velmi rychle příkaz ukončete pomocí Ctrl-C, protože soubor se zvětšuje neuvěřitelnou rychlostí. Zkuste si poté pomocí **`ls -l`** vypsat, jak veliký soubor **`yes`** stihnul vytvořit. Naštěstí Unix má pro podobné případy jakýsi bezedný odpadkový koš. Je to zařízení null, do kterého můžeme posílat jakákoliv data horem dolem a jediným výsledkem je, že data nenávratně zmizí. Pokud tedy provedeme **`yes >/dev/null`**, máme krásný příklad nekonečného zcela neškodného procesu. Pojďme to vyzkoušet:

```
TuX:~$ yes > /dev/null
```

Po odmáčknutí klávesy Enter kurzor skočí na další řádku a proces běží a běží a běží... Hloupé na tom je to, že dokud neskončí, nemůžeme v našem terminálu nic dělat (ani se například pomocí **`ps`** přesvědčit, že proces skutečně běží). Už víme, že pomocí **Ctrl-C** proces můžeme ukončit. Je tu ale ještě jedna možnost. Pomocí **Ctrl-Z** ho můžeme dočasně pozastavit.

[44] Smyslem programu je automaticky odpovídat na otázky (yes/no) u některých příkazů. Jiný řetězec bude opakovat, pokud mu ho zadáme jako parametr (např. **`yes no`** bude donekonečna opakovat **`no`**).

Udělejme to a pak si pomocí **ps w** vypišme stav procesů:

```
TuX:~$ yes > /dev/null

[1]+  Stopped                 yes >/dev/null
TuX:~$ ps w
  PID TTY      STAT   TIME COMMAND
 9524 pts/0    S      0:00 bash
 9566 pts/0    T      0:01 yes
 9567 pts/0    R+     0:00 ps w
TuX:~$
```

Po stisku klávesy **Ctrl-Z** nám systém vypíše, že máme zastavený jeden proces, a následný výpis **ps w** ukáže, že **yes** je ve stavu **T** (pozastaveno) . Proces zastavil svoji činnost a **na pozadí** čeká na další signály, které mu řeknou, co dál. Zpět do popředí ho můžeme dostat příkazem **fg** (z anglického foreground). To se hodí například, když během práce potřebujeme na chvíli odskočit do shellu, něco tam provést a zase se vrátit k původní činnosti.
Někdy je ale žádoucí, aby proces na pozadí pokračoval ve své činnosti zatímco my budeme pracovat **na popředí**. Toho můžeme dosáhnout tak, že proces pozastavíme pomocí **Ctrl-Z** a poté ho příkazem **bg** (z anglického background) rozeběhneme na pozadí. Výpis **ps w** v tom případě ukáže stav procesu **R** (z anglického running). i z tohoto stavu můžeme proces dostat do popředí příkazem **fg**.
Pokud od začátku víme, že nějaký proces chceme spustit na pozadí, uvedeme na konec příkazové řádky znak **&**. Ten způsobí, že proces poběží na pozadí, aniž bychom ho tam museli interaktivně umisťovat klávesou **Ctrl-Z** a ze stavu **T** poté rozbíhat příkazem **bg**.
Vyzkoušejte si všechny možnosti na našem cvičném procesu **yes > /dev/null** a vždy si pomocí **ps** vypište jeho stav.

9.6 kill

Příkaz **kill** slouží k odesílání libovolného signálu vybranému procesu. Sice můžeme poslat jakýkoliv signál, ale při běžné činnosti budeme nejvíce posílat právě signál **SIGKILL**, který příkazu dal název a způsobí ukončení procesu, kterému byl odeslán. **SIGKILL** má číslo **9**, takže v praxi příkaz bude ve tvaru **kill -9 12345**, kde 12345 v našem příkladu představuje číslo procesu, který chceme ukončit.

9.7 top

Posledním příkazem, který si ve vztahu k procesům ukážeme je příkaz **top**. Ukazuje interaktivní tabulku běžících procesů v reálném čase se spoustou zajímavých údajů, ze kterých je jasně vidět, jak jednotlivé procesy zatěžují procesor a paměť počítače. Příkaz má přepínače, kterými se dá jeho chování ovlivnit při spuštění a klávesy, kterými se ovlivňuje přímo při běhu (ukončujeme ho samozřejmě pomocí **Ctrl-C**). Jeho činnost je podobná Správci úloh ze systému MS Windows.
Bližší informace najdete opět v manuálových stránkách.

10 Další zajímavé příkazy

Až doposud byly příkazy v této příručce řazeny do tematických celků podle účelu svého využití. V této kapitole si řekneme o některých dalších příkazech, které se jinam nevešly. Samozřejmě příkazů a programků jsou v Linuxu stovky, a zde probereme jen několik z nich, ale snažil jsem se vybírat právě ty, které se běžnému uživateli při práci hodí nejvíce.

10.1 date

Příkaz **`date`** zobrazuje nebo nastavuje systémové datum a čas. Má spoustu přepínačů, kterými se dá ovlivnit tvar výstupu nebo naopak vstupu příkazu, ale nejběžněji se používá ve tvarech:

- **`date`** zobrazí aktuální datum
- **`date –s 19:45`** nastaví čas na 19:45
- **`date –s 2008-07-29`** nastaví datum na 29.7.2008

Většina Unixových počítačů připojených na Internet (včetně našeho cvičného serveru) má ale čas řízen pomocí **ntp**[45] serverů a nastavováním data a času se u nich není třeba zabývat.

10.2 cal

Když nevíme, kolikátého zrovna je, může nám pomoci příkaz **`cal`**, který zobrazí jednoduchý a přehledný kalendářík:

```
TuX:~$ cal
     July 2008
Su Mo Tu We Th Fr Sa
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31

TuX:~$
```

Tvar jeho výstupu se dá ovlivnit několika přepínači.

10.3 uptime

Třetí příkaz ze skupinky zabývající se časovými údaji je příkaz **`uptime`**. Vypisuje dobu, po kterou počítač běží, což se možná zdá na první pohled zbytečné, ale když například potřebujete zjistit, jak server přežil noční bouřku, může se hodit:

```
TuX:~$ uptime
 18:58:09 up 12 days, 8:26, 1 user, load average: 0.00,0.00,0.00
TuX:~$
```

Navíc příkaz ukazuje průměrné zatížení procesoru za poslední jednu, pět a patnáct minut.

[45] ntp – network time protocol – je protokol pro předávání informací o přesném času na Internetu

10.4 chmod a chown

chmod spolu s příkazem **chown** patří k jedněm z nejdůležitějších příkazů. Nastavují se jimi totiž práva k souborům a adresářům. Než si popíšeme jak zacházet s těmito příkazy, vzpomeňme nejdříve na podrobný výpis souborů pomocí příkazu **ls** a všimněme si podrobněji informací, které obsahuje (výpis v následujícím příkladu je zkrácen pouze na tři řádky).

```
TuX:~$ ls -l /etc
...
-rwxr-xr-x 1 root root   306 2008-07-10 11:17 rc.local
drwxr-xr-x 2 root root  1024 2008-07-10 18:46 rcS.d
-rw-r--r-- 1 root root  2555 2004-12-06 14:59 reportbug.conf
...
```

Nejprve si všimněme první skupiny znaků na každém řádku. V prostředním řádku je v prvním sloupci písmeno **d**, což znamená, že se jedná o adresář, u jiných pomlčkou **-**, která označuje obyčejný soubor. Můžeme ještě narazit na písmeno **l** pro symbolický link a **c** nebo **b** pro soubory znakových respektive blokových zařízení. **První písmeno** skupiny znaků tedy určuje o jaký typ souboru se jedná.
Za písmenem pro typ souboru se nacházejí tři skupiny písmen **rwx** z nichž některá jsou nahrazena pomlčkami. Písmena značí **r** – právo na čtení (**r**ead), **w** – právo na zápis (**w**rite) a **x** – právo na spuštění programu (e**x**ecute). Pokud se na daném místě nachází písmeno, právo je přiděleno, pokud je tam pomlčka, příslušné právo je odňato. Proč jsou skupiny práv tři?
První skupina znaků určuje práva vlastníka daného souboru. To je primárně ten, kdo soubor vytvořil, nicméně vlastníka je možno změnit právě příkazem **chown**, ke kterému se brzy dostaneme.
Druhá skupina znaků se týká skupiny uživatelů. V Unixu je totiž možné vytvářet skupiny uživatelů a jednotlivé uživatele do nich libovolně zařazovat. Za normálních okolností bývá zvykem každému uživateli vytvořit i stejnojmennou skupinu, ve které je zařazen.
Třetí skupina znaků patří libovolnému jinému uživateli, tedy někomu, kdo soubor ani nevlastní, ani nepatří do určené skupiny.
Vlastník a skupina jsou uvedeny v dalších sloupcích výpisu, v našem příkladu je vlastníkem všech uvedených souborů **root** a skupina **root**.
Teď už tedy víme všechno pro to, abychom byli schopni přečíst nastavení práv u souborů z našeho příkladu. rc.local je obyčejný soubor, se který může vlastník dělat všechno (**rwx**), zatímco skupina i ostatní uživatelé jej mohou jen číst nebo spouštět, ale nemohou do něj zapisovat (**r-x r-x**). Stejně je na tom **rcS.d** jen s tím rozdílem, že se jedná o adresář. Právo x zde neznamená právo spuštění programu, ale právo přístupu do adresáře. Soubor **reportbug.conf** má práva nastavena tak, že vlastník z něj může číst i do něj zapisovat, ale skupina i ostatní uživatelé z něj mohou jen číst (**rw-r--r--**) . Každopádně soubor není pro nikoho spustitelný.
Práva se nastavují příkazem **chmod**. Nejlépe bude si to ukázat na praktickém příkladu. Vytvořme si nejprve v našem pokusném adresáři textový soubor třeba s názvem **prava** a zapišme do něj alespoň jeden řádek textu. Pokud používáte editor **nano** provedete to takto:

- **nano ~/pokusy/prava**
- zápis krátkého textu
- **Ctrl-X** a pak klávesa **y** a potvrzení názvu souboru klávesou **Enter**

Pokud si teď pomocí ls –l vypíšeme obsah adresáře pokusy, najdeme ve výpisu řádek:

```
-rw-r--r-- 1 test test   40 2008-07-27 22:16 prava
```

Z toho je vidět, že vlastníkem a skupinou je ten, kdo soubor vytvářel a práva jsou nastavena na read write pro vlastníka a read pro všechny ostatní[46]. Zkusme teď i vlastníkovi odepřít právo zápisu. Provedeme to příkazem **chmod** s následujícími parametry:

```
TuX:~/pokusy$ chmod u-w prava
TuX:~/pokusy$ ls –l
...
-r--r--r-- 1 test test  40 2008-07-27 22:16 prava
...
```

Jak je vidět, příkaz **chmod** se používá tak, že se nejprve uvede požadované nastavení práv a pak soubor (nebo více souborů třeba pomocí regulárních výrazů), u kterého chceme práva upravit.
Práva se dají určovat několika způsoby. Zde jsme si uvedli ten, který je možná pro začátečníka trochu srozumitelnější, ale jeho zápis je pracný a poměrně nepřehledný. Spočívá v tom, že se uvede nastavení práv pomocí kombinace písmena **u**,**g**,**o**,**a**, znaku **+** nebo **–** a písmen příslušných práv. Znaky **u**, **g**, **o** respektive **a** znamenají vlastníka (**u**ser), skupinu (**g**roup), ostatní (**o**ther) respektive všechny (**a**ll). Znak **+** nebo **–** znamená, zda chceme právo nastavit nebo odebrat a o samotných právech **r**, **w**, **x** jsme už mluvili. V příkladu uvedené nastavení tedy znamená „vlastníkovi odeber právo write“ (**u-w**). Pokud je třeba takto nastavit práva více skupinám uživatelů, oddělují se jednotlivé složky čárkami. Například zápis **chmod u+x,g-w,o+rx soubor** znamená „přidej vlastníkovi právo spouštět, skupině odeber právo zápisu a ostatním přidej právo čtení a spouštění“.
Druhý způsob nastavení práv používá pro jejich vyjádření trojici oktalových[47] číslic. Každá vyjadřuje nastavení práv pro jednu ze skupin uživatelů vztahující se k souboru. Výsledná číslice je daná součtem čísel: **4** pro právo **read**, **2** pro právo **write** a **1** pro právo **execute**. **chmod 754 prava** tedy provede toto: „vlastníkovi nastav plná práva (4 + 2 + 1 = 7), uživatelům ze skupiny práva čtení a spuštění (4 + 1 = 5) a ostatním pouze právo čtení“. Nejběžnější nastavení je **chmod 755 soubor** pro spustitelné soubory (**rwxr-xr-x**) a **chmod 644 soubor** pro nespustitelné (**rw-r--r--**).
Použití příkazu chown bude pro nás po pochopení příkazu **chmod** už naprosto triviální. Tímto příkazem se mění vlastník a skupina daného souboru. Tvar příkazu je velmi podobný příkazu **chmod** jen s tím rozdílem, že namísto práv se zapisuje, kdo má být vlastníkem:skupinou.
Pokud by například root chtěl změnit vlastníka a skupinu souboru **prava** na sebe, provedl by to příkazem **chown root:root prava**.
Jak **chmod**, tak **chown** mají další přepínače z nichž asi nejužitečnější je **-r**, které způsobí rekurzivní provedení příslušného příkazu na soubory včetně všech podadresářů.

10.5 su

Příkaz **su** (z anglického **s**uper **u**ser) slouží ke změně uživatele během činnosti. Superuživatel v Unixu má tradičně uživatelské jméno **root** a jeho práva v systému jsou naprosto neomezena. Proto je poměrně nebezpečné běžnou činnost vykonávat pod tímto účtem a je lepší se

[46] Je to dáno nastavením takzvané masky **umask**, ale její popis je nad rámec této publikace.
[47] Oktalová neboli osmičková číselná soustava se u historických počítačů často používala, a toto je jedno z míst, kde zůstala zakonzervovaná až do dnešní doby.

k systému hlásit jako obyčejný uživatel. Teprve v momentu, kdy běžná uživatelská práva pro určitý úkon nestačí, může se uživatel právě tímto příkazem změnit na roota, provést potřebné úkony a zpět se změnit na běžného uživatele se standardními právy. Bez dalších parametrů příkaz změní identitu na roota, pokud zadáme jako parametr uživatelské jméno, změníme identitu právě na toto jméno. Samozřejmě vždy při tom musíme znát heslo patřičného uživatele:

```
TuX:~$ su
Password:
TuX:/home/test# ps
  PID TTY          TIME CMD
12763 pts/0    00:00:00 su
12764 pts/0    00:00:00 bash
12767 pts/0    00:00:00 ps
TuX:/home/test# exit
TuX:~$
```

Ve výše uvedeném příkladu uživatel test změnil příkazem **su** svoji identitu na roota. Příkaz **su** ve skutečnosti změní identitu uživatele a spustí nový shell (**bash**). To je vidět ve výpisu procesů příkazem **ps**. Všimněte si také, že příkaz zůstane ve stejném adresáři, jen pro roota to není domovský adresář, a proto ho v promptu vidí celým názvem oproti znaku **~**. Prompt také končí znakem **#** namísto obvyklého **$**. Pokud nově spuštěný shell ukončíme, ukončí se i činnost příkazu **su**.

10.6 exit a logout

Příkazy **exit** a **logout** ukončují činnost interaktivně spuštěného shellu. **exit** se dá použít kdykoliv, kdežto **logout** jen v případě shellu, který byl spuštěn bezprostředně po přilogování k počítači. Tyto příkazy se tedy používají pokud chceme opustit režim **su** nebo se odhlásit od počítače. Nemají samostatné manuálové stránku, protože jsou součástí shellu, takže pokud se o nich chcete dozvědět více, najděte si je v **man bash**.

10.7 who

Příkaz **who** vypisuje všechny uživatele, kteří jsou momentálně k počítači přihlášení. Jedná se o výpis skutečných přihlášení, takže použití příkazu **su** na něj nemá vliv.

```
TuX:~$ who
root     pts/0        2008-07-28 10:18 (jack.panska.local)
test     pts/1        2008-07-28 10:49 (adsl.nextra.cz)
serych   pts/2        2008-07-28 10:50 (jack.panska.local)
test     pts/3        2008-07-28 10:50 (jack.panska.local)
TuX:~$
```

Z uvedeného výpisu je vidět, že momentálně je k počítači přihlášen uživatel **root**, uživatel **test** dokonce dvakrát ze dvou různých strojů a uživatel **serych**.

10.8 whoami

Příkaz **whoami** (Who am I - kdo jsem já?) vypadá na první pohled nesmyslně. Možná si myslíte, že tak přepracovaní, abyste ani nevěděli „čí jste“ snad nikdy nebudete. Pokud ale použijete příkaz **su** (třeba i několikanásobně), teprve poznáte jeho užitečnost. Vypisuje totiž právě uživatele, za kterého momentálně vystupujete.

10.9 sudo

Příkaz **sudo** je podobný příkazu **su** jen s tím rozdílem, že nemusí spouštět interaktivní shell, ale provede v roli jiného uživatele pouze příkaz specifikovaný svými parametry. Další výhodou oproti **su** je to, že uživatel nemusí znát heslo roota. Povolení spouštět **sudo** a hlavně přesné nastavení toho, co daný uživatel může a nesmí dělat je totiž definované v konfiguračním souboru **/etc/sudoers** .

10.10 mount a umount

Na začátku jsme si říkali, že Unix má filesystem vybudovaný na jediném kořenovém adresáři **/**. Právě příkazy **mount** a **umount** umožňují do tohoto souborového systému přimontovat a nebo z něj naopak odmontovat jiné blokové zařízení. Pokud například do počítače s Unixem strčíme CDčko a chceme s ním začít pracovat, bez jeho namontování do souborového systému se nám nepodaří ho zpřístupnit. Právo montovat zařízení do filesystému má normálně jen root. Bloková zařízení mohou mít filesystémy nejrůznějších typů, proto příkaz mount potřebuje typ filesystému znát. Nejběžnější způsob použití je:
mount - t typ_filesystemu zarizeni adresar_pro_namontovani
Například za předpokladu, že CDčko vystupuje v systému jako zařízení **/dev/hdb**, můžeme ho namontovat příkazem:

```
TuX:/dev# mount -t iso9660 /dev/hdb /media/cdrom/
TuX:/dev#
```

Pokud potom provedeme **ls /media/cdrom0**, uvidíme soubory a adresáře, které se na CDčku nacházejí jako součást našeho souborového systému.
Příkaz **mount** je ale používán už při samotném bootování systému. Pro tyto účely je v adresáři **/etc** soubor **fstab**, ve kterém je nastaveno, která zařízení se mají kam montovat. Prostřednictvím přednastavení v tomto souboru lze potom montovat zjednodušenými příkazy, takže například namontování CDčka by mohlo vypadat jako: **mount /media/cdrom**.
Odmontování filesystému se provádí příkazem **umount**, kterému stačí pomocí parametru už zadat jen, který adresář chceme odmontovat, například **umount /media/cdrom**.
S montováním filesystémů se dá dělat spousta důmyslných triků jako například montování obrazů disků, vytváření ramdisků v paměti počítače, montování vzdálených síťových disků atd., ale to už je nad rámec této publikace.

10.11 df

Příkaz **df** (z anglického **d**isk **f**ilesystem usage) vypisuje momentální stav zaplnění disků[48]:

```
server:~# df
Filesystem   1K-blocks    Used Available Use% Mounted on
/dev/hda1      2885780  558992   2180200  21% /
tmpfs           225924       0    225924   0% /lib/init/rw
udev             10240      32     10208   1% /dev
tmpfs           225924       0    225924   0% /dev/shm
/dev/hda3     72785240 9969752  59118132  15% /var
```

Z výpisu je jednak vidět, jak jsou namontované jednotlivé filesystémy (poslední sloupec výpisu) a jednak, jak jsou jednotlivé disky zaplněné. I tento příkaz má řadu přepínačů,

[48] Tento výpis jsem provedl na jiném Linuxovém serveru, protože náš cvičný server TuX má hardwarové RAID diskové pole, u něhož mají disky poněkud netypické a matoucí názvy.

kterými lze ovlivnit jeho chování (například lze upravit jednotky, ve kterých se budou vypisovat kapacity jednotlivých filesystémů).

10.12 du

Podobným příkazem je **du** (z anglického disk usage). Namísto zkoumání celkového zaplnění filesystémů se ale zabývá velikostí jednotlivých adresářů. Spustíme-li ho bez jakýchkoliv parametrů, změří velikosti podadresářů v aktuálním adresáři:

```
TuX:~$ du
52      ./pokusy/skripty
12      ./pokusy/hokusy
88      ./pokusy
116     .
TuX:~$
```

Z příkladu vidíme, že v prvním sloupci je velikost, kterou adresáře zabírají. Vyjadřuje počet bloků na disku, přičemž jeden blok je ve většině případů 1024B. Pokud chceme velikost vyjádřit jinak, pomohou nám jako obvykle přepínače. Například pomocí **-h** dosáhneme vyjádření velikosti v „lidsky čitelných" jednotkách (kB, MB, GB – podle potřeby). Pokud nechceme vidět počet zabraných bloků na disku, ale skutečnou velikost dat v adresářích, použijeme přepínač **-b**:

```
TuX:~$ du -b
21214   ./pokusy/skripty
8686    ./pokusy/hokusy
34271   ./pokusy
45544   .
TuX:~$
```

Pochopitelně tato velikost je výrazně menší, než skutečně zabraný prostor na disku. Chceme-li příkaz použít při čištění disku, bude se nám určitě hodit profiltrování jeho výstupu filtrem **sort**, který řádky seřadí podle velikosti, a možná i další parametry. Například **du --max-depth=2 /lib | sort -n** setřídí vzestupně podle velikosti podadresáře adresáře **/lib** do druhé úrovně zanoření. Chceme-li provést totéž, ale třídit sestupně, přidáme příkazu sort ještě přepínač **-r** (reverse).

10.13 find

Jak název napovídá, příkaz **find** se používá k prohledávání souborového systému. Ačkoliv příkaz má mnoho parametrů, kterými se dá ovlivnit co a kde se má hledat a je dobré si prohlédnout jeho manuálovou stránku, protože jeho možnosti jsou skutečně nepřeberné, jeho nejběžnější použití vypadá následovně: **find kde_hledat -name nazev** . Samozřejmě jakožto název hledaného souboru může být použit libovolný regulární výraz. Například **find /usr -name mysql*** najde všechny soubory s názvem začínajícím na **mysql** v adresáři **/usr** a všech jeho podadresářích. Síla příkazu spočívá ale i v tom, že s nalezenými výsledky lze (dokonce několika různými způsoby) provést nějakou činnost. Například **find /tmp -name core* -type f -print | xargs /bin/rm -f** najde v adresáři **/tmp** všechny soubory (nikoliv adresáře **-type f**) s názvem začínajícím na **core** a prostřednictvím filtru **xargs** je rovnou smaže[49].

[49] Samozřejmě je třeba být u takovéhoto použití příkazu maximálně obezřetný a nejdříve si například nechat soubory jen vypsat třeba pomocí **find /tmp -name core* -type f -print | more**.

10.14 locate a whereis

Podobnou službu jako příkaz **find** dělají i příkazy **locate** a **whereis**. Oproti příkazu **find**, který skutečně prohledává reálnou adresářovou strukturu ale tyto příkazy prohledávají jen databázi s touto strukturou. To má své výhody i nevýhody. Hlavní výhodou je výrazně rychlejší hledání a fakt, že v databázi nejsou omezena práva, takže jsou vidět i soubory v adresářích, do kterých by jinak běžný uživatel právo přístupu neměl. Nevýhodou je, že databáze je systémem updatovaná jen v poměrně dlouhých časových intervalech (obvykle jednou denně nebo i jednou týdně)[50]. Takže pokud jsme například vytvořili soubor před 5 minutami, je velmi pravděpodobné, že ho pomocí těchto příkazů nenajdeme.
Jak se s příkazy pracuje a jaký je mezi nimi rozdíl? Představme si, že například potřebujeme najít program **mysql**. Pokud použijeme jednoduchý příkaz **locate mysql**, dostaneme v tomto případě velmi dlouhý výpis[51].
Zpravidla ale hledáme jen spustitelné programy a jejich konfigurační soubory. Právě příkaz whereis je na toto specialista. Hledá ve stejné databázi jako **locate**, ale vybírá jen určité adresáře a jen určité typy souborů v nich. **whereis** vytváří výpis trochu nestandardně oddělený mezerami a ne novými řádkami, ale každopádně hledání **mysql** dá výrazně menší výstup:

```
TuX:~$ whereis mysql
mysql: /usr/bin/mysql /etc/mysql /usr/X11R6/bin/mysql
/usr/bin/X11/mysql /usr/share/mysql
/usr/share/man/man1/mysql.1.gz
```

Souborů našel v tomto případě 6. Pokud mu navíc dáme přepínač **-b**, což znamená, že má hledat jen binární soubory, najde jich jen 5. Pomocí přepínače **-m** hledá jen manuálové stránky, což se může také hodit.

10.15 wget

Dovedete si představit webový browser pracující pouze na příkazovém řádku? Pokud ne, tak vyzkoušejte **wget**. pracuje se s ním velmi jednoduše. Jako parametr se mu dává pouze URL souboru, který má stáhnout a **wget** ho uloží do aktuálního adresáře. Například příkazem:
wget http://www.debian.org/releases/stable/i386/install.pdf.cs
si tedy stáhneme z oficiálního webu Debianu instalační příručku v češtině.
I příkaz **wget** má pochopitelně řadu přepínačů, kterými se dá ovlivnit jeho chování, ale to už je nad rámec této publikace.

10.16 tar

Tento příkaz vychází z historických kořenů, kdy nezbytným hardwarovým zařízením každého počítače byla magnetopásková zálohovací jednotka. Kupodivu je přesto **tar** (z anglického **t**ape **a**rchiver) jedním z nejpoužívanějších příkazů. Při archivaci na pásky bylo totiž nutné celé struktury adresářů a v nich obsažených souborů zabalit do jediného archivního souboru a ten potom nahrát na páskovou jednotku. Naopak při dearchivaci se musí archiv rozbalit do původní struktury. A právě o tyto činnosti se stará **tar**. Má rozsáhlé možnosti popsané v manuálových stránkách, ale pro běžnou činnost v podstatě stačí znát pouze přepínače pro běžné rozbalení a zabalení archivu. Názvy archivních souborů typicky končívají množinou znaků .tar[52].

[50] root může databázi kdykoliv updatovat pomocí příkazu **updatedb**.
[51] Pomocí **wc -l** jsem v něm na našem cvičné serveru napočítal 1759 řádek!
[52] Použití „přípon“ je sice v Unixu obecně málo obvyklé, ale právě u souborů **tar** naopak téměř jisté.

Rozbalení archivu do aktuálního adresáře se provede příkazem: **`tar –xvf archiv.tar`**
Zabalení adresáře do archivu naopak příkazem: **`tar –cvf archiv.tar adresar/`**
Někdy je také dobré jen **vypsat obsah archivu**, což se provede příkazem:
`tar -tvf archiv.tar`
Původní **`tar`** jen balil a rozbaloval strukturu adresářů, ale neprováděl přitom kompresi dat, která je u archivů velmi užitečná. Současné verze už tuto kompresi umí dělat. zpravidla se komprimuje programem gzip nebo bzip2. Vznikají tak soubory, které bývá zvykem pojmenovávat tak, že na konec jejich názvu se dává skupina znaků **`.tar.gz`** respektive **`tar.bz`** . Práce s příkazem **`tar`** se liší jen tím, že se přidává přepínač **`–z`** v případě **`.gz`** komprese a **`–j`** v případě **`.bz`** komprese. Tedy například **`tar –xvzf archiv.tar.gz`**. pro rozbalení archivu komprimovaného pomocí gzipu.

10.17 ping

Ještě několik příkazů týkajících se práce v síti. Prvním z nich je **`ping`**, jehož méně dokonalého bratra jistě znáte i z jiných operačních systémů. Slouží k testování síťového připojení a používá se téměř stejně jako v jiných operačních systémech, například: **`ping 192.168.15.25.`** Rozdílem je, že v Unixu ping běží trvale, dokud ho neukončíme pomocí **Ctrl-C**.

10.18 traceroute

Unixový příkaz **`traceroute`** byl vzorem tracert z MS Windows, a proto i jeho použití je velmi podobné. Vypisuje cestu paketů a hodí se zejména při hledání routovacích problémů v síti. Jen krátký příklad použití z lokální sítě:

```
TuX:~$ traceroute santiago.panska.cz
traceroute to santiago.panska.cz (192.168.10.3), 30 hops
max, 40 byte packets
 1  192.168.1.1 (192.168.1.1) 0.974 ms 0.841 ms 0.825 ms
 2  192.168.10.3 (192.168.10.3) 0.663 ms 0.738 ms 0.608 ms
TuX:~$
```

Samozřejmě jak příkaz **`traceroute`**, tak i předchozí **`ping`** mají v Unixu výrazně větší množství přepínačů než jejich sourozenci v MS Windows, takže i jejich možnosti jsou v Unixu jsou výrazně širší.

10.19 nslookup a dig

Další příkaz, který byl z Unixu přejat do ostatních operačních systémů je nslookup. Podobně jako ve Windows se dá i v Unixu používat jednorázově nebo interaktivně. Zjišťuje údaje ze systému DNS. Například:

```
TuX:~$ nslookup www.debian.cz
Server:         192.168.1.5
Address:        192.168.1.5#53

www.debian.cz   canonical name = debiancz.debian.cz.
Name:   debiancz.debian.cz
Address: 195.113.161.73

TuX:~$
```

Modernější příkaz, který nahrazuje v Unixu **nslookup** a dává úplnější informace se jmenuje **dig** (z anglického **d**omain **i**nformation **g**roper (hmatač)). Na stejný dotaz odpoví takto:

```
TuX:~$ dig www.debian.cz

; <<>> DiG 9.3.4-P1.1 <<>> www.debian.cz
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 34552
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0,
ADDITIONAL: 0

;; QUESTION SECTION:
;www.debian.cz.                 IN      A

;; ANSWER SECTION:
www.debian.cz.          86261   IN      CNAME
debiancz.debian.cz.
debiancz.debian.cz.     86261   IN      A
195.113.161.73

;; Query time: 3 msec
;; SERVER: 192.168.1.5#53(192.168.1.5)
;; WHEN: Wed Jul 30 22:46:43 2008
;; MSG SIZE  rcvd: 70

TuX:~$
```

Pochopitelně oba tyto příkazy mají řadu přepínačů popsaných v příslušných manuálových stránkách.

10.20 ifconfig

Posledním důležitým příkazem spojeným se sítí je **ifconfig**, který se používá pro nastavování a zjišťování konfigurace síťových rozhraní. Pro běžného uživatele[53] je nejdůležitější jeho použití úplně bez parametrů, kdy pouze vypisuje stav všech síťových rozhraní počítače:

```
TuX:~# ifconfig
eth0 Link encap:Ethernet  HWaddr 00:08:C7:4B:90:F1
     inet addr:192.168.1.12  Bcast:192.168.1.255  Mask:255.255.255.0
     inet6 addr: fe80::208:c7ff:fe4b:90f1/64 Scope:Link
     UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
     RX packets:1330155 errors:0 dropped:0 overruns:0 frame:0
     TX packets:53763 errors:0 dropped:0 overruns:0 carrier:0
     collisions:0 txqueuelen:1000
     RX bytes:125914848 (120.0 MiB)  TX bytes:11058979 (10.5 MiB)
```

Zde je výpis zkrácen pouze na rozhraní **Ethernet 0**, normálně následuje velmi podobně vypadající výpis pro další rozhraní včetně rozhraní **Local Loopback**.

[53] příkaz má standardně právo spouštět jen root.

11 Instalace systému

V této kapitole v krátkosti probereme způsob, jak nainstalovat Linux z distribuce Debian[54]. Kompletní cca 120 stránková instalační příručka je v češtině ke stažení ze stránek www.debian.org.

11.1 Hardwarové nároky

aneb na čem můžeme Linux nainstalovat

Teoreticky téměř na čemkoliv. Jak se píše v instalační příručce Debianu, distribuce neklade na hardware žádné jiné nároky než samotné jádro a jednotlivé programy GNU. To znamená, že minimální instalaci systému lze teoreticky provést na dnes už skutečně extrémně slabém hardwaru.

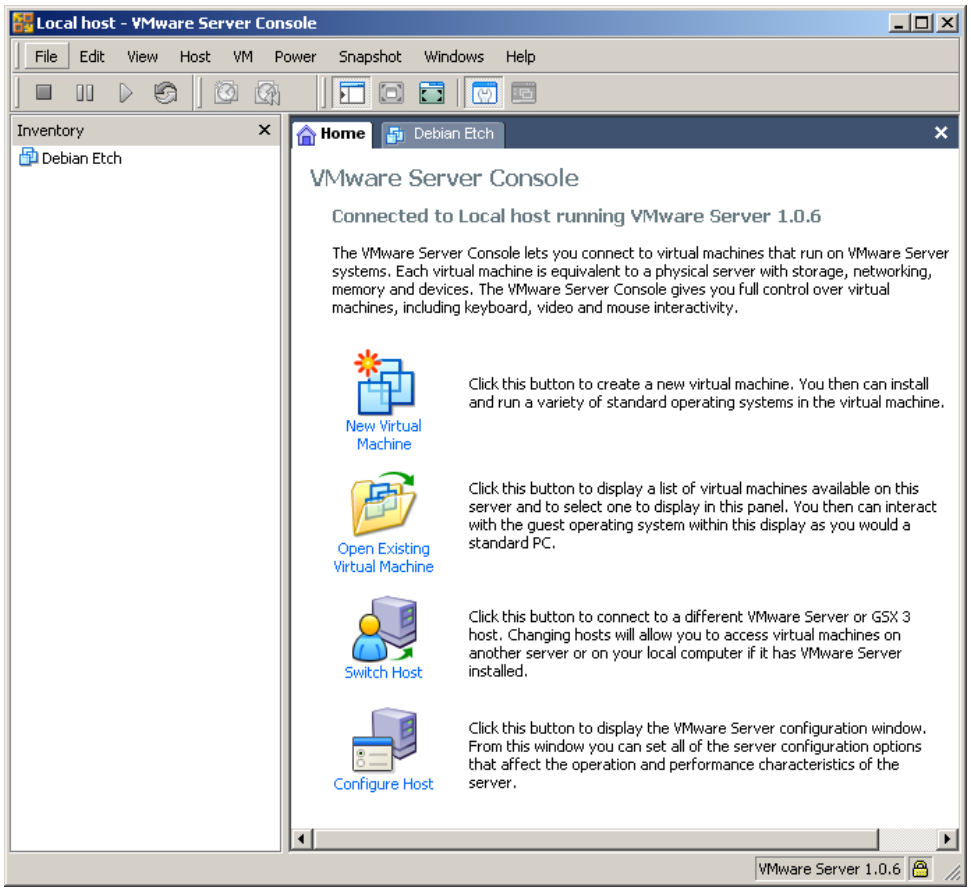
Prakticky pro běžnou bezproblémovou instalaci potřebujete počítač minimálně od procesoru Intel Pentium[55] výše se 48MB RAM a 500MB prostoru na disku. To je naprosté minimum. Doporučení je pro servery 256MB RAM a 1GB na disku, pro pracovní stanice se systémem X Windows a balíkem kancelářských aplikací 512MB RAM a 5GB na disku. Samozřejmě čím lepší a rychlejší procesor budete mít, tím rychleji systém poběží. Pro slušnou pracovní stanici se doporučuje minimálně Pentium IV na 1GHz, pro běžné menší servery jsou nároky na výkon procesoru výrazně nižší.

Co se týká dalšího hardwaru, pro server bez X Windows stačí jakákoliv grafická karta od VGA výše, klávesnice a samozřejmě je dobrá síťová karta a přístup na Internet. U běžných počítačů se zpravidla na problémy s kompatibilitou hardwaru nenaráží, ale občas bývají problémy s některými z notebooků.

Zlatý hřeb ale až na konec. Pro experimentální instalaci Linuxu, ba dokonce bych měl správně napsat Linuxů, totiž žádný samostatný hardware nepotřebujete. Lze je totiž nainstalovat v prostředí MS Windows, aniž byste ho jakkoliv narušili. Kouzlo kterým se vám to povede zrealizovat se jmenuje **virtuální počítač**. Existuje několik programů, které takové virtuální počítače realizují. Ten, se kterým mám osobní zkušenosti (a to jen ty nejlepší) se jmenuje VMWare Server. Dá se zdarma stáhnout ze stránek http://www.vmware.com/products/server/. Pro instalaci potřebujete ještě licenční číslo, které Vám firma VMWare zdarma vygeneruje a pošle mailem na základě vyplnění jednoduchého formuláře. Když VMWare server nainstalujete, stačí si už jen pomocí průvodce vytvořit nový virtuální stroj (New Virtual Machine). Průvodce se Vás zeptá na pár údajů jako například kolik paměti a diskového prostoru stroji přenecháte, jaký systém na něm poběží atd. Pak už stačí jen tlačítkem se šipkou v paletě pod menu programu virtuální stroj spustit a můžeme začít s plnohodnotnou instalací. Z hlediska MS Windows si VMWare vytvoří jen složku s několika soubory pro každý virtuální stroj. Pokud nemáte nouzi o místo na disku, můžete si nechat disk předpřipravit celý. Pokud jste třeba v průvodci novým virtuálním strojem nadefinovali, že bude mít 5GB disk (tuto velikost pro experimenty s Debianem doporučuji), vytvoří se rovnou 5GB soubor, do kterého se bude celý filesystém nového stroje ukládat. Pokud chcete disk šetřit a nevadí Vám, že běh virtuálního stroje bude neznatelně pomalejší, můžete celkovou předpřípravu disku vynechat a soubor se bude pružně zvětšovat (případně zmenšovat) tak, jak budete do Linuxu doinstalovávat (odinstalovávat) další věci.

[54] Konkrétně se budeme zabývat instalací verze 4.x s názvem Etch, ale velmi podobně instalace probíhá i u starších verzí, a je pravděpodobné, že i u novějších verzí nebude příliš odlišná.

[55] Myslí se tím skutečně už první procesor Pentium, nikoliv současná plejáda všech možných vylepšených a zrychlených Pentií II, III, IV, Dual Core atd. Jádro reálně běží dokonce už od procesoru Intel 486.



Obrázek 11 - Základní okno VMWare serveru

Virtuální server skutečně vytvoří plnohodnotný nový počítač v počítači. Počítač, který má svojí „síťovou kartu“ zapojenou v „síti“ se skutečným počítačem, USB porty, které fungují tak, jak mají, CD mechanikou, ze které se dá normálně číst atd. S výkonem procesorů, velikostí pamětí a disků současných počítačů není opravdu problém, když na jednom hardwaru běží více virtuálních strojů. Navíc VMWare odstíní veškeré případné problémy s kompatibilitou skutečného hardwaru.

11.2 Odkud Debian nainstalujeme

Debian je GNU software, takže je k dispozici zcela zdarma. Samozřejmě centrálním místem na Internetu je web www.debian.org, ale existuje i velké množství jeho dalších zrcadel rozmístěných po celém světě. Uvedený server na Vás bude mluvit hezky česky[56], takže se v něm snadno zorientujete. Když v menu na levém okraji stránky kliknete na položku „Obrazy CD“, dostanete se na stránku, která popisuje (už anglicky) několik možností, jak Debian získat. To, který způsob zvolíte bude záležet asi především na Vaší Internetové konektivitě. Pokud máte jakékoliv pevné připojení s rychlostí od zhruba 512kbps výše, je jednoznačně nejrozumnější volba stáhnout si minimální bootovací CDčko pro síťovou instalaci. Pokud máte vytáčené, nějak extrémně pomalé nebo žádné připojení k Internetu, je rozumné si někde v místě s připojením stáhnout CDčka respektive DVDčka s kompletní instalací nebo je případně koupit (české prodejce najdete také na hlavní stránce www.debian.org). Instalace z těchto kompletních CDček je ale rozhodně méně příjemná než ze sítě a navíc balíčky na nich samozřejmě časem zastarávají[57].
Dále budu tedy popisovat síťovou instalaci s minimálním bootovacím CDčkem. Instalace z kompletních CDček/DVDček je téměř totožná, jen nevyžaduje nastavení připojení k síti.

[56] Pokud ho navštívíte českým prohlížečem z českých MS Windows.
[57] Systém se pochopitelně neustále vyvíjí a často vycházejí novější a opravené verze různých programů.

Nejdříve tedy **stáhneme minimální bootovací CDčko**. Osobně používám **netinst image**, která je optimalizovaná tak, že se vejde i na maloformátové CDčko (do 180MB). Klikneme na odkaz s architekturou i386 a vybereme soubor, který se bude jmenovat nějak jako: **`debian-40r4-i386-netinst.iso`**[58]. Soubor otevřeme v softwaru pro vypalování a **CDčko si vypálíme**.
Hotové CDčko vložíme do mechaniky stroje (ať už skutečného nebo virtuálního), na který budeme Debiana instalovat a stroj spustíme[59].

Obrázek 12 - Úvodní obrazovka instalátoru

Objeví se úvodní obrazovka a pro instalaci už jen stačí stisknout **Enter**. Kdo by si chtěl předem prohlédnout co ho přesně čeká krok po kroku, nechť do Googlu zadá (bez uvozovek) výraz: „Debian Etch Perfect Setup“ a najde mnoho míst se screenshoty jednotlivých kroků instalace. Zde budu sice výrazně stručnější, ale instalaci popíšu beze zbytku.

1. Po stisku **Enter** se zavede jednoduché jádro systému do RAMdisku a spustí se instalátor
2. Na první obrazovce průvodce vybereme jazyk. Vyberte určitě **English**!
3. Dále vybereme umístění. Zvolte **other** a po stisku **Enter** najděte **Czech Republic**
4. Dalším krokem je výběr klávesnice, zvolte **American English.**
5. Instalátor bude chvilku prověřovat CDčko, nahrávat nějaké komponenty a dále se pokusí pomocí DHCP **nainstalovat síť**. Pokud se mu to z nějakého důvodu nepodaří, dá nám možnost nastavit síť ručně.
6. Jakmile síť běží, budeme dotázáni na **hostname** a **domain name**. Doporučuji nějaké krátké hostname. Pokud neinstalujete skutečný server, který bude skutečně natvrdo pracovat na Internetu, na doménovém jménu příliš nezáleží.
7. V dalším kroku musíme rozhodnout jak připravíme disk[60]. Začátečníkům doporučuji volbu **Guided – use entire disk**, na další obrazovce **zvolit disk** a na další zvolit **All files in one partition**.

[58] Číslo 40r4 v názvu v tomto případě znamená verzi 4.0 (Etch), release 4. Nových releasů vychází i několik do roka, ale vůbec nevadí pokud instalujeme z nějakého staršího releasu. Síťová instalace si všechny novější balíčky sama najde, takže se vlastně během instalačního procesu automaticky upgraduje na nejnovější release.
[59] V BIOSu musí být nastaveno, aby stroj bootoval z CDčka.
[60] U virtuálního serveru se nebojte, na fyzický hardwarový disk instalátor nesáhne, bude pracovat jen na nastaveném virtuálním disku, tedy fyzicky v MS WIndovs na jednom nebo několika souborech v adresáři s virtuálními stroji.

8. Instalátor připraví **dvě partitiony**, jednu pro kořenový filesystém a jednu pro swap file. Jejich velikosti se budou odvíjet z přiděleného prostoru na disku a velikosti paměti RAM. Přípravu ukončíme výběrem **Finish partitioning and write changes...** a potvrzením **Yes**.
9. Disky se naformátují, což bude podle jejich velikosti chvilku trvat.
10. Budeme vyzváni k zadání **hesla pro uživatele root** (2x). Toto heslo **nezpomeňte**, jinak budete pravděpodobně nuceni systém přeinstalovat[61].
11. Dále musíte vytvořit alespoň **jednoho dalšího běžného uživatele** a vymyslet mu heslo.
12. Proběhne instalace základu systému z CDčka (to bude chvilku trvat) a u netinstal způsobu instalace **budete vyzváni k nakonfigurování package manageru**. Na první otázku odpovězte **Yes**.
13. Na dalších dvou obrazovkách Vám bude nabídnuta země (ta kterou jste zvolili v kroku 3) a zrcadlo pro stahování instalačních balíčků. Já standardně volím http://ftp.cz.debian.org
14. Pokud pro přístup k síti používáte **proxy server**, vyplňte na další obrazovce jeho přesnou adresu (včetně **http://** a nezpomeňte na konci uvést **za dvojtečkou číslo portu**). Pokud k webu přistupujete přímo, nechte řádek nevyplněný.
15. Systém apt pak bude chvíli (v závislosti na rychlosti připojení) **scanovat archivy a updatovat** databázi balíčků.
16. Na další obrazovce týkající se **package usage sur**vey odpovězte **No**.
17. Za chvíli se objeví důležitá obrazovka, kde je možné **zvolit komponenty pro instalaci**. Mezi volbami se pohybujeme kurzorovými klávesami a zaškrtáváme klávesou mezerník. Doporučuji ale v tuto chvíli zvolit **jen Standard system**, který je už předzaškrtnutý.
18. Instalátor bude v závislosti na rychlosti sítě a množství zaškrtnutých voleb více či méně dlouho stahovat balíčky a instalovat je. Na pomalejší síti a při větším počtu zaškrtnutí to může trvat i desítky minut.
19. Nakonec odpovíme na otázku, zda má být instalován **GRUB Yes**, systém ho nainstaluje, vystrčí CDčko z mechaniky, **my ho vyndáme** a pomocí **Continue** počítač zrestartujeme.
20. Po restartu se objeví přihlašovací obrazovka a **náš nový Linuxový stroj je připraven** pro nás začít pracovat[62].

Abychom mohli k serveru přistupovat příjemněji, je vhodné si ještě nad rámec standardní instalace **nainstalovat ssh a zjistit IP adresu serveru**. Přihlaste se jako `root` a napište příkaz:

`apt-get install ssh openssh-server`.

Budete vyzváni ke vložení CDčka a ssh se nainstaluje. pak si ještě příkazem `ifconfig` zjistěte IP adresu vaší síťové karty.

Tím je instalace hotová a mělo by být možné se buďto ze sítě nebo v případě virtuálního stroje z rodičovského počítače **normálně přihlásit programem PuTTY**. Je to totiž z mnoha důvodů daleko příjemnější než pracovat přímo v okně VMWaru.

12 Balíčkovací systém

Právě poslední příkaz použitý v minulé kapitole provedl nainstalování programů ze dvou balíčků distribuce Debian. Ale než si tento a další příkazy popíšeme, vysvětleme si nejprve, co to je balíčkovací systém a jak funguje.

12.1 K čemu slouží balíčkovací systém

Jak se píše v jedné parafrázi: „Na počátku byl `.tar.gz...`“ Ano, přesně tak nějak vznikaly první balíčkovací systémy a ruku v ruce s nimi i první distribuce Linuxu. Historicky vůbec

[61] Samožřejmě existují postupy, jak se tomu vyhnout, ale nejsou pro začátečníka zrovna triviální.
[62] U VMWare serveru se klávesnice aktivuje kliknutím do okna virtuálního stroje a zpět do MS Windows se dostaneme kombinací kláves Ctrl-Alt.

první distribuce nazvaná Slackware skutečně dodnes používá balíčky s příponou .tgz (což je zkrácené .tar.gz) a jejich obsah je možné (ačkoliv to není příliš účelné) programem **`tar`** opravdu rozbalit.
Před zavedením balíčkovacích systémů se musela instalace jakéhokoliv programu provádět tak, že se na daném počítači přímo zkompiloval ze zdrojových kódů. Není problém tyto zdrojové kódy sehnat, protože licence GNU GPL, pod kterou je většina programů šířena, jejich poskytnutí klade přímo za povinnost. Samotná kompilace ale rozhodně není triviální proces, zabere poměrně dost času, a často se před ní musí shánět větší množství různých knihoven, které při ní ten který program vyžaduje. Tvůrci prvních distribucí tento proces zjednodušovali tím, že jednotlivé programy zkompilovali a spolu se všemi potřebnými knihovnami a dalšími soubory zabalili do archivů. Při instalaci mnoha takových balíčků ovšem často nastávají problémy se vzájemnými závislostmi. Proto novější balíčkovací systémy, mezi které patří například právě Debian/Ubuntu systém s balíčky **`.deb`** nebo například RedHat systém s balíčky **`.rpm`**, obsahují důmyslné nástroje pro ověřování závislostí, mohou spouštět různé preinstalační a postinstalační skripty atd.

12.2 Balíčkovací systém Debianu

Balíčkovací systém Debianu se postupně vyvíjel. Dnes obsahuje šest hlavních a řadu pomocných aplikací, které se vzájemně doplňují a mnohdy se jejich funkce dokonce překrývají. Řešení Debianu je považováno za jeden z nejdokonalejších balíčkovacích systémů vůbec, takže je dokonce přebírán do zcela jiných operačních systémů.
Základní a nejstarší nástroj se jmenuje **`dpkg`**. Je to program, který rozbaluje nebo naopak vytváří balíčky. Tento program je sice srdcem balíčkovacího systému Debianu, ale jeho ovládání není příliš uživatelsky přívětivé. Proto brzy vzniklo několik snah vytvořit mu lepší uživatelské rozhraní. Výsledkem jedné z nich je program **`dselect`**. Jako mezivrstva mezi programy **`dpkg`** a **`dselect`** byl vyvinut program **`apt`**. Paradoxně ačkoliv **`apt`** je jen utilita pro práci z příkazového řádku, ujala se daleko více než program **`dselect`**.
Později vznikla pro program **`apt`** a potažmo **`dpkg`** ještě další uživatelská rozhraní. Jmenují se **`aptitude`**, **`synaptic`** a **`tasksel`**[63]. Pojďme se na jednotlivé programy podívat blíže.

12.2.1 dpkg

Příkaz **`dpkg`** je úplným základem balíčkovacího systému. Debian používá dva odlišné typy balíčků.
Prvním z nich jsou **balíčky obsahující zdrojové kódy**. Ty mají příponu **`.dsc`** a obsahují archiv (**`.orig.tar.gz`**) s originálními zdrojovými kódy a rozdílový soubor změn (**`.diff.gz`**) provedených ve zdrojových kódech kvůli přizpůsobení Debianu[64]. Zdrojové balíčky se ovládají programem **`dpkg-source`**.
Pojďme se ale věnovat spíše druhému typu balíčků, se kterými se běžný uživatel dostane do styku daleko častěji. Jsou jimi **binární balíčky** s příponou **`.deb`**. Tyto balíčky obsahují přeložené spustitelné programy a všechny součásti, které ke své činnosti potřebují. Spustitelné programy jsou ale zpravidla závislé na různých knihovnách. Mnohé z nich se navíc používají opakovaně v mnoha různých programech, to je ostatně také důvod, proč se vůbec knihovny používají. Aby všechny knihovny nemusely být opakovaně zabaleny ve všech balíčcích, které je využívají, mají balíčky Debianu propracovaný systém kontroly závislostí. Ty jsou zapsány

[63] **`tasksel`** má trochu odlišnou funkci. Slouží pro instalaci celých množin balíčků, takzvaných tasků.
[64] Různé distribuce včetně Debianu různým způsobem využívají strukturu Unixových adresářů pro umístění jednotlivých souborů, mají trošku jiné systémové proměnné, mírně odlišným způsobem řeší spouštění programů při bootování atd. Právě těchto odlišností se týkají změny prováděné ve zdrojových kódech v balíčcích **`.dsc`**.

ve speciálním souboru uvnitř balíčků **.deb** a **dpkg** během instalace balíčku zkoumá, zda jsou všechny dodrženy. Pokud ne, upozorní, které balíčky se musí nejprve doinstalovat. Typy závislostí mezi balíčky A a B jsou definovány takto:

- **depends** – balíček A je plně závislý na nainstalování balíčku B
- **recommends** – autor balíčku A předpokládá, že téměř všichni, kdo ho budou používat, budou používat zároveň i balíček B
- **suggests** – balíček B obsahuje součásti, které zpravidla rozšiřují funkcionalitu A, takže je doporučována jeho instalace
- **replaces** – balíček B plně nahrazuje balíček A
- **provides** – balíček B plně zajišťuje funkcionalitu balíčku A
- **conflicts** – balíček B je v konfliktu s balíčkem A

Nejčastěji budete program **dpkg** používat v případě, že získáte nějaký balíček z umístění mimo klasická úložiště Debianu. V tom případě provedete jeho instalaci příkazem **dpkg --install nazev-balicku.deb**. Tento postup ale doporučuji opravdu jen v případě, že nemáte jinou možnost. Je při něm třeba jednak dávat pozor na závislosti a navíc bývají problémy s jejich udržením v případě upgradů systému. Klasický způsob instalace balíčků si ukážeme později, ale teď zpět k použití programu **dpkg**. Ten totiž běžný uživatel využije spíše ke zjišťování nainstalovaných balíčků. Pomocí **dpkg -l** se vypisují všechny balíčky nainstalované v systému. Bývá jich poměrně velké množství, například náš cvičný server TuX jich má něco kolem 300. Ukázka několika řádek výpisu:

```
...
ii  grep       2.5.1.ds2-6   GNU grep, egrep and fgrep
ii  groff-base 1.18.1.1-12   GNU troff text-formatting system
ii  grub       0.97-27etch1  GRand Unified Bootloader
ii  gzip       1.3.5-15      The GNU compression utility
ii  hostname   2.93          utility to set/show the host name or
domain
...
```

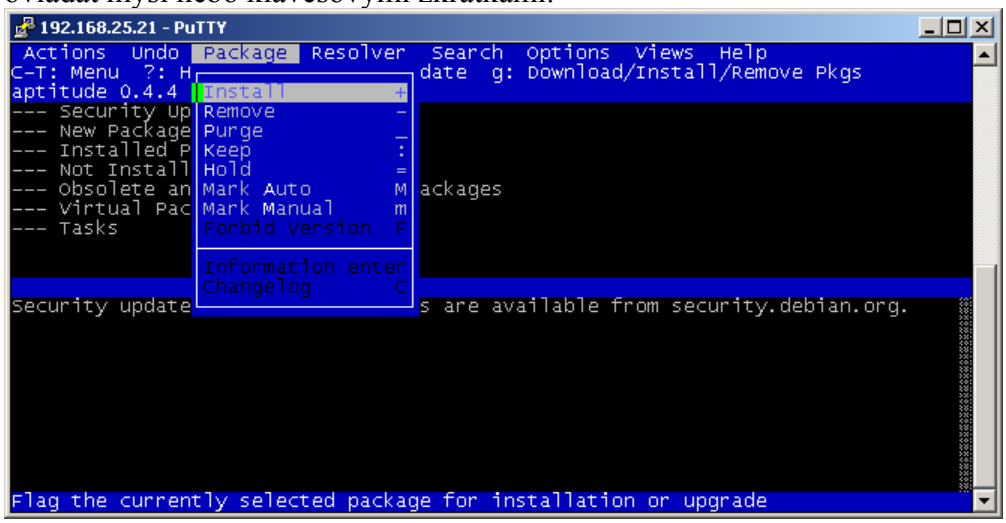
Další užitečný přepínač je **-L**. K němu je třeba programu **dpkg** dodat jako parametr název balíčku. Program pak vypíše všechny soubory, které byly z balíčku nainstalovány. Ukažme si to například na balíčku **hostname**:

```
TuX:~# dpkg -L hostname
/.
/bin
/bin/hostname
/bin/dnsdomainname
/usr
/usr/share
/usr/share/doc
/usr/share/doc/hostname
/usr/share/doc/hostname/copyright
/usr/share/doc/hostname/changelog.gz
/usr/share/man
/usr/share/man/man1
/usr/share/man/man1/hostname.1.gz
/usr/share/man/fr
/usr/share/man/fr/man1
/usr/share/man/man1/dnsdomainname.1.gz
TuX:~#
```

Na tomto příkladu jednak vidíme, kolik souborů obsahuje tak primitivní balíček, jakým je **hostname** starající se pouze o název počítače. Jednak také to, že v něm jsou nadefinovány veškeré potřebné adresáře, takže i kdyby například hypoteticky v době jeho instalace neexistoval tak základní adresář jako **/bin** nebo **/usr**, balíček si ho vytvoří.
Pokud je v balíčku obsažen konfigurační skript, **dpkg** ho po rozbalení souborů spustí, a tak proběhne konfigurace daného programu. Je-li potřeba ji z nějakého důvodu provést někdy později znova, dá se použít příbuzný příkaz **dpkg-reconfigure nazev-balicku**.

12.2.2 aptitude a dselect

dpkg je sice jádrem celého balíčkovacího systému, ale jeho použití není příliš uživatelsky přívětivé. Proto bylo vyvinuto několik příjemnějších uživatelských rozhraní. Prvním z nich je program **aptitude**, který nabízí interaktivní prostředí opatřené nabídkou, která se dá ovládat myší nebo klávesovými zkratkami.

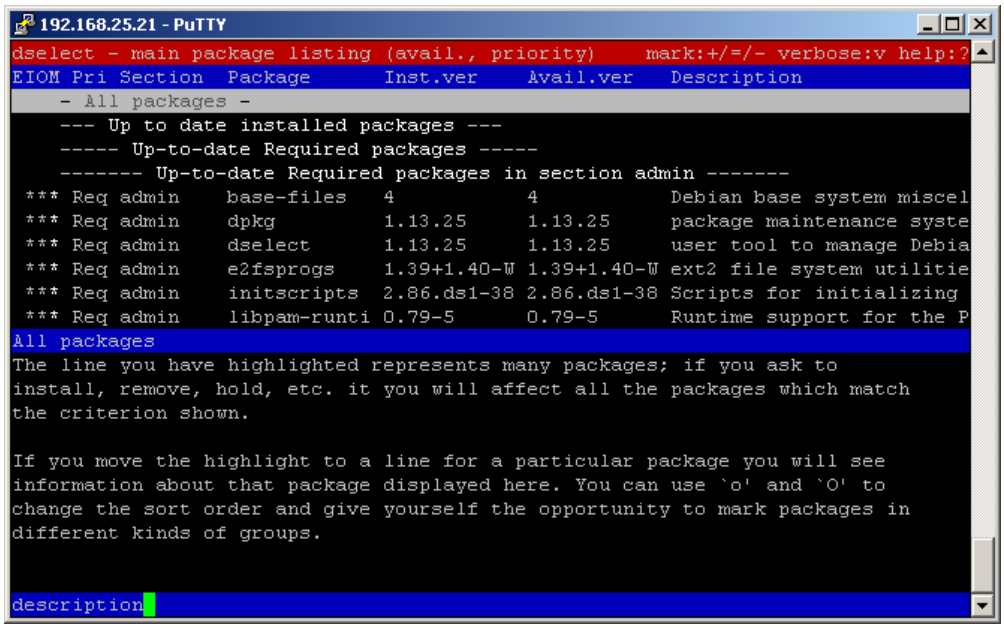


Obrázek 13 - Program aptitude

Uveďme alespoň některé důležité klávesové zkratky programu aptitude:

Klávesa	Akce
F10	Menu
?	Help, kompletní výpis zkratek
u	Update informací o balíčcích z Internetu
+	Označení balíčku k instalaci nebo upgradu
-	Označení balíčku ke smazání se zachováním konfigurace
_	Označení balíčku ke smazání včetně konfigurace
U	Označení všech upgradovatelných balíčků k upgradu
g	Stažení a instalace vybraných balíčků
q	Ukončení s uložením změn
x	Ukončení bez uložení změn
Enter	Prohlédnutí podrobných informací o balíčku
C	Prohlédnutí protokolu změn balíčku
/	Vyhledání textu (první výskyt)
\	Hledání dalších výskytů textu

Druhým programem je starší **dselect**. Jeho ovládání jednoduchými seznamy voleb a klávesovými zkratkami je z dnešního pohledu už trochu zastaralé, ale přesto je jeho použití pro někoho možná příjemnější než použití samotného **dpkg**.



Obrázek 14 - Program dselect

Klávesové zkratky se píší velkými písmeny a jejich filosofie je trochu nezvyklá:

Klávesa	Akce
Q	Quit. – Potvrdit současné volby, ale program ukončit
R	Revert! – Zpátečka, tak jsem to nemyslel.
D	Damn it! – Hrome! Udělej, co ti říkám, nezajímá mě, co si myslíš!
U	sUggested – Nastav vše do obvyklého stavu.

12.2.3 apt

Paradoxně nejpoužívanějším je pro správu balíčků v Debianu program, který byl původně vyvinut pouze jako spojovací článek mezi **dpkg** a uživatelskými rozhraními. Je dobré se ho naučit používat, nejen proto, že práce s ním je velmi rychlá a příjemná, ale i proto, že v různých návodech a tutoriálech na Internetu týkajících se instalací balíčků zásadně nenajdete jiné instrukce než právě pro **apt**.

Co tedy **apt** dělá, jak funguje a jak se nastavuje? Program **apt** stahuje balíčky z přednastavených úložišť a prostřednictvím **dpkg** ihned provádí jejich instalaci. Přitom se automaticky stará o závislosti. Také si udržuje lokální databázi balíčků, kterou umí navíc updatovat, takže ji není nutné pokaždé znovu celou stahovat ze sítě. Než ovšem začneme program používat, je třeba ho nastavit[65]. Zásadní nastavení, které uživatele zajímá je vjednom až dvou souborech. Důležitější z nich je soubor **/etc/apt/sources.list**. V něm se nachází seznam úložišť, ze kterých má **apt** balíčky stahovat. Typicky může tento soubor na českých instalacích obsahovat následující text:

```
#deb cdrom:[Debian GNU/Linux 4.0 r1 _Etch_ - Official i386
NETINST Binary-1 200$

deb http://ftp.cz.debian.org/debian/ etch main
deb-src http://ftp.cz.debian.org/debian/ etch main

deb http://security.debian.org/ etch/updates main contrib
deb-src http://security.debian.org/ etch/updates main contrib
```

[65] Základní nastavení programu apt je provedeno při samotné instalaci Debianu

První řádek je dobré hned po instalaci zakomentovat (znak **#** na začátku), jinak bude **apt** při každé činnosti vyžadovat vložení instalačního CDčka. Další dva řádky z našeho příkladu určují úložiště základních a zdrojových balíčků. Z hlediska rovnoměrnosti zatížení a rychlosti stahování je vhodné nastavit lokální úložiště (u nás například **ftp.cz.debian.org**). Poslední dva řádky určují úložiště pro bezpečnostní updaty, které je naopak z důvodů aktuálnosti dobré nastavit na centrální úložiště updatů. Všimněte si, že na koncích řádků jsou verze systému a typy balíčků, které se mají stahovat (Debian 4.x – **etch**, balíčky hlavní distribuce **main**). S nastavením úložišť lze provádět důmyslná kouzla, přidávat nestandardní úložiště[66] a přesně nastavovat priority stahování a upgradování z nich, ale to jsou již pokročilé a poměrně nebezpečné techniky jejichž popis je nad rámec této publikace.
Druhým souborem, který může ale také nemusí v adresáři **/etc/apt** být je soubor **apt.conf**. V něm je uloženo nastavení web proxy, pokud je potřeba, aby **apt** natahoval balíčky prostřednictvím proxy serveru. V souboru je jediný řádek:

```
Acquire::http::Proxy "http://nazev:port";
```

Řádek v tomto souboru nelze zakomentovat, takže pokud potřebujete proxy přístup vypnout, nezbývá než tento soubor smazat nebo přejmenovat.
apt se skládá z několika dílčích programů. Pojďme se podívat na dva nejdůležitější z nich.

12.2.4 apt-get

Program apt-get je hlavní a nejpoužívanější součástí z této rodiny. Stahuje a updatuje databázi balíčků a provádí jejich instalaci a odinstalaci. Vždy než začneme balíčky instalovat, je dobré provést aktualizaci vnitřní databáze. Provádí se to příkazem **apt-get update**:

```
TuX:~# apt-get update
Get:1 http://ftp.cz.debian.org etch Release.gpg [386B]
Get:2 http://ftp.cz.debian.org etch Release [58.2kB]
Ign http://ftp.cz.debian.org etch/main Packages/DiffIndex
Ign http://ftp.cz.debian.org etch/main Sources/DiffIndex
Get:3 http://ftp.cz.debian.org etch/main Packages [5624kB]
... (zkráceno)
Fetched 7792kB in 13s (575kB/s)
Reading package lists... Done
TuX:~#
```

Pokud chceme upgradovat systém na aktuální verze programů, je dobré hned připojit **apt-get upgrade**:

```
TuX:~# apt-get upgrade
Reading package lists... Done
Building dependency tree... Done
The following packages have been kept back:
  linux-image-2.6-686
The following packages will be upgraded:
  bind9-host debconf debconf-i18n dnsutils grub initramfs-tools
initscripts
  libbind9-0 libc6 libc6-i686 libdns22 libisc11 libisccc0 libisccfg1
  liblwres9 libpcre3 locales selinux-policy-refpolicy-targeted sysv-rc
  sysvinit sysvinit-utils tzdata
23 upgraded, 0 newly installed, 0 to remove and 1 not upgraded.
Need to get 14.5MB of archives.
After unpacking 1765kB disk space will be freed.
Do you want to continue [Y/n]? y
```

[66] Existují například různé sbírky nestanddardních balíčků.

Výše uvedený opgrade systému je velmi rozumné dělat pravidelně, protože i v Linuxu se pochopitelně objevují různé bezpečnostní díry, které mohou znamenat potenciální nebezpečí jeho napadení.
Instalace nového balíčku se provede příkazem **apt-get install nazev-balicku**. Pojďme si to ukázat na testovacím balíčku Debianu **hello**:

```
TuX:~# apt-get install hello
Reading package lists... Done
Building dependency tree... Done
The following NEW packages will be installed:
  hello
0 upgraded, 1 newly installed, 0 to remove and 20 not upgraded.
Need to get 48.5kB of archives.
After unpacking 188kB of additional disk space will be used.
Get:1 http://ftp.cz.debian.org etch/main hello 2.1.1-5 [48.5kB]
Fetched 48.5kB in 0s (137kB/s)
Selecting previously deselected package hello.
(Reading database ... 22444 files and directories currently
installed.)
Unpacking hello (from .../hello_2.1.1-5_i386.deb) ...
Setting up hello (2.1.1-5) ...
TuX:~#
```

Balíček obsahující klasický primitivní program „Hello world!“ se stáhne a nainstaluje. Nyní se můžete pomocí **dpkg -l** a **dpkg -L hello** ujistit, že je skutečně nainstalovaný a zjistit, jaké soubory byly kam přidány. Pokud by balíček obsahoval nějaké závislosti, **apt-get** by je vypsal a zeptal by se, zda je má instalovat. Samozřejmě většinou je jediná správná odpověď na tuto otázku **y** (ze slova yes). Instalace balíčků, zejména pokud obsahují hodně závislostí nemusí být vždy zcela bezproblémová. Proto je dobré si ji zejména na ostrých serverech nejprve vyzkoušet nanečisto. Toho se dosáhne přepínačem **-s**. Pokud si například zkusíme s tímto přepínačem znovu nainstalovat balíček hello, dostaneme tuto informaci:

```
TuX:~# apt-get install hello -s
Reading package lists... Done
Building dependency tree... Done
hello is already the newest version.
0 upgraded, 0 newly installed, 0 to remove and 20 not upgraded.
TuX:~#
```

Odinstalace balíčků se provádí příkazem **apt-get remove nazev-balicku**. Takto příkaz provede odstranění všech souborů mimo konfiguračních. Pokud chceme odstranit i veškerou konfiguraci související s balíčkem, přidáme přepínač **--purge**.

Pojďme si to například ukázat na našem balíčku hello, i když zrovna ten pochopitelně zrovna žádnou konfiguraci nemá:

```
TuX:~# apt-get remove hello --purge
Reading package lists... Done
Building dependency tree... Done
The following packages will be REMOVED:
  hello*
0 upgraded, 0 newly installed, 1 to remove and 20 not upgraded.
Need to get 0B of archives.
After unpacking 188kB disk space will be freed.
Do you want to continue [Y/n]? y
(Reading database ... 22485 files and directories currently
installed.)
Removing hello ...
TuX:~#
```

Příkaz apt-get má mnoho dalších možností a vzhledem k jeho důležitosti doporučuji si pžečíst jeho manuálovou stránku.

12.2.5 apt-cache

Druhým příkazem, o kterém si zde v krátkosti povíme je **`apt-cache`**. Stará se o místní databázi balíčků, kterou, jak už víme natáhneme ze serverů pomocí **`apt-get update`**. Příkaz má širokou paletu možností, ale my si ukážeme pouze dvě nejčastější použití. Prvním z nich je **`apt-cache search regexp`**, který se hodí pro prohledávání databáze balíčků. Pokud neznáme přesný název některého z nich, můžeme pro jeho vyhledání použít regulární výraz. Výraz se navíc hledá nejen v názvu, ale i v popisu balíčku, takže pokud alespoň trochu víme, jak by se balíček měl jmenovat, je velká pravděpodobnost, že ho najdeme. Například příkaz **`apt-cache search mysql*`** najde 238 balíčků souvisejících s databází mysql[67].
Druhým velmi častým využitím **`apt-cache`** je příkaz ve tvaru **`apt-cache showpkg nazev-balicku`**, který ukáže veškeré informace (včetně závislostí, které nás zpravidla zajímají nejvíce) o daném balíčku. I u příkazu **`apt-cache`** doporučuji pečlivé projití manuálových stránek.

[67] Samozřejmě dalším spolehlivým způsobem, jak balíčky hledat je zadat do Googlu výraz *debian package predpokladany-nazev* nebo prohledat oficiální stránky packages.debian.org, které mají také svůj vyhledávač.

13 Jednoduché skripty

V této poslední kapitole si stručně ukážeme, co jsou to skriptu shellu, jak fungují a jak jednoduché skripty napsat. Rozhodně se ale nebude jednat o nějakou učebnici skriptování, protože její rozsah by musel výrazně překračovat možnosti této publikace. O skriptování je na Internetu mnoho různých tutoriálů a kurzů, takže zájemci další potřebné informace snadno dohledají.

13.1 Shell

Všechno, co jsme doposud v Linuxu dělali jsme dělali za pomoci **příkazového interpreteru, takzvaného shellu**. Shell je spuštěn obvykle po přihlášení uživatele[68], poskytuje výzvu, po vložení příkazu zjistí, zda se jedná o jeho vnitřní příkaz, funkci nebo vnější program a podle toho ho adekvátním způsobem vykoná. Unixových shellů existuje několik a jejich kompletní seznam na konkrétní instalaci Linuxu najdeme v souboru /etc/shells. Mezi známé Unixové shelly patří Bourne shell (**sh**), C shell (**csh**), Korn shell (**ksh**) a dalších několik ovšem již daleko méně populárních. Velmi se osvědčil Bourne shell, ale pro Linuxu byl vyvinuta GNU verze ještě dokonalejšího shellu. Jmenuje se Bourne-Again shell (**bash**). **bash** je plně zpětně kompatibilní s Bourne shellem, ale doplňuje ho o řadu dalších užitečných vlastností a funkcí z dalších populárních shellů. Protože bash se v Linuxech používá defaultně a protože má opravdu velmi široké možnosti a je velmi populární, budeme se zabývat právě skripty pro tento shell.

13.2 První jednoduchý skript

Představme si, že nás vždy po přihlášení k systému zajímá datum a čas, jak dlouho již systém běží, kteří uživatelé jsou přihlášeni a kolik místa zbývá na disku. Tyto informace, jak už víme můžeme zjistit tak, že napíšeme příslušné příkazy jeden po druhém na příkazový řádek. To je ale poněkud neefektivní. Pojďme to zkusit jinak.
Přejděte do svého domovského adresáře, vytvořte si tam podadresář skripty a přejděte do něj. Následně svým oblíbeným editorem vytvořte soubor s názvem **prvni.sh**:

```
TuX:/etc$ cd
TuX:~$ mkdir skripty
TuX:~$ cd skripty/
TuX:~/skripty$ nano prvni.sh
```

Do souboru napište pod sebe jednotlivé příkazy:

```
date
uptime
who
df
```

Pak z editoru vyskočte s tím, že soubor uložíte. Pokud teď použijeme příkaz **source**, jehož více používaný zkrácený název je tečka **.**[69] a dáme mu jako parametr název našeho souboru, právě běžící shell soubor vezme a po jednotlivých řádcích přečte a vykoná příkazy:

[68] Shell se také spouští například po příkazu **su**, při vykonávání skriptů, nebo ho můžeme klidně spustit ručně.
[69] Tato tečka nemá nic společného s tečkou označující aktuální adresář. Shell podle kontextu pozná o kterou tečku se jedná.

```
TuX:~/skripty$ . prvni.sh
Tue Aug 12 21:26:11 CEST 2008
 21:26:11 up 1 day, 10:37, 1 user, load average: 0.00,
0.00, 0.00
root pts/0  2008-08-12 20:30 (ip-160-218-129-
131.eurotel.cz)
... (zkráceno)
```

Tento způsob se občas sice může hodit, ale aby vzniknul klasický skript, je dobré soubor ještě trochu upravit. Také spouštět ho budeme trochu jiným způsobem. Otevřete soubor znovu editorem a před příkazy na úplně první řádek napište:

`#!/bin/bash`

Znak **#** označuje komentář a shell bude při interpretaci skriptu vše od tohoto znaku až do konce řádky ignorovat. Speciální řádka, kterou jsme zapsali se ovšem chová malinko jinak. Říká běžícímu shellu, kterým shellem se má skript interpretovat. Při normálním spuštění skriptu, které si ukážeme za chvilku se totiž spouští nový shell, který teprve skript vykoná. Můžeme tedy psát skripty pro různé shelly a řádkou **`#!/bin/nazev-shellu`** určíme, pro který z nich je skript určen.

Pojďme ještě skript malinko okomentovat a pomocí příkazu **`echo`** zpřehlednit jeho výstup:

```
#!/bin/bash
# Nas prvni skript vypisujici uzitecne informace
clear   # vycisti obrazovku
echo ”Dnes je: ‘date‘”
echo    # takhle jen odradkujeme
echo ”System jiz bezi: ”
uptime
echo
echo ”Jsou prihlaseni: ”
who
echo
echo ”Stav disku: ”
df
```

Skript uložíme a pokusíme se ho spustit napsáním jeho názvu na příkazový řádek:

```
TuX:~/skripty$ prvni.sh
bash: prvni.sh: command not found
TuX:~/skripty$
```

Jak vidíte, takhle skript zatím spustit nelze. Je to proto, že shell má určené cesty, ze kterých se mají spouštět programy a skripty, a pokud skript není umístěn na některé z nich, shell ho nedokáže najít. Pomůžeme mu tedy tím, že mu napíšeme přesné umístění skriptu. Mohli bychom tedy psát **`/home/uzivatelske-jmeno/skripty/prvni.sh`**, ale již víme, že tento náš adresář má hardlink **`./`** . Jsme-li tedy v adresáři se skriptem, můžeme namísto dlouhé cesty psát zcela totožný výraz **`./prvni.sh`**:

```
TuX:~/skripty$ ./prvni.sh
bash: ./prvni.sh: Permission denied
TuX:~/skripty$
```

Jak je vidět, tentokrát narazíme na něco jiného. Shell hlásí, že nemáme právo skript spustit. Zkusme si tedy vypsat a upravit oprávnění k našemu souboru:

```
TuX:~/skripty$ ls –l
total 4
-rw-r--r-- 1 test test 210 2008-08-12 22:31 prvni.sh
TuX:~/skripty$ chmod 755 prvni.sh
TuX:~/skripty$ ls –l
total 4
-rwxr-xr-x 1 test test 210 2008-08-12 22:31 prvni.sh
TuX:~/skripty$
```

Pokud máte nastavený barevný příkaz **ls**, všimněte si, že po úpravě oprávnění se bude soubor **prvni.sh** vypisovat zelenou barvou. Právě ta indikuje, že soubor má oprávnění ke spuštění. Pokud teď znovu napíšeme **./prvni.sh**, shell by již neměl hlásit žádnou chybu a měl by náš první zdokonalený skript vykonat[70].
Jak vidíte příkaz **echo** vypisuje to, co je za ním uzavřeno v našem případě v uvozovkách. Také se dají použít apostrofy, ale mají jiný význam. Text v apostrofech **echo** pouze tupě opíše, kdežto text v uvozovkách se snaží nejprve vyhodnotit, a teprve pak provést výstup. Všimněte si, že v prvním výskytu **echo** v našem skriptu je příkaz **date** uzavřený v opačných apostrofech[71]. Právě ty shellu oznamují, že v nich je příkaz, který se má provést. **echo** vyhodnotí výstup tohoto příkazu, a teprve pak provede výstup na obrazovku.
Bude vhodné, si cestu k našim skriptům přidat k cestám, kde shell hledá spustitelné soubory. Cesty jsou uloženy v proměnné **PATH** o čemž se můžeme přesvědčit jejím vypsáním:

```
TuX:~/skripty$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games
TuX:~/skripty$
```

Je vidět, že jednotlivé cesty jsou v této proměnné odděleny dvojtečkami. Přidání naší cesty provedeme příkazem:

```
TuX:~/skripty$ export PATH="$PATH:~/skripty"
TuX:~/skripty$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games:~/skripty
TuX:~/skripty$
```

Jak je vidět ze znova provedeného výpisu, naše cesta se k proměnné **PATH** přidala, takže teď můžeme náš skript spouštět přímo zadáním jeho názvu:

```
TuX:~/skripty$ prvni.sh
```

13.3 Proměnné

Objekt **PATH**, se kterým jsme pracovali na konci minulé kapitoly se nazývá proměnná. Řada názvů takových proměnných je rezervována pro speciální funkce v shellu a zpravidla jsou jejich hodnoty nadefinovány inicializačními skripty. Všechny tyto proměnné si můžeme

[70] Samozřejmě stále také funguje takzvaný sourcing **. prvni.sh**. Ve spuštění **./prvni.sh** a **. prvni.sh** je ovšem značný rozdíl. Sourcing provádí náš aktuální shell a nejsou pro něj k souboru potřeba práva ke spuštění. Při plnohodnotném spuštění skriptu je spuštěn nový shell uvedený v řádku **#!/bin/nazev-shellu** a ten teprve provede skript. Po jeho dokončení spuštěný shell skončí a předá zpět řízení našemu původnímu shellu.
[71] Na anglické klávesnici ho najdeme na klávese těsně pod klávesou **Esc**.

vypsat příkazem **printenv**. Ale libovolné další proměnné si můžeme kdykoliv nadefinovat sami. Bývá zvykem jim pro přehlednost dávat názvy velkými písmeny, každopádně jak víme, Unix rozlišuje vždy velká a malá písmena, takže **POKUS**, **Pokus** a **pokus** jsou tři rozdílné proměnné. Pojďme si tedy proměnnou **POKUS** nadefinovat a vypsat:

```
TuX:~/skripty$ POKUS="Testovic"
TuX:~/skripty$ echo $POKUS
Testovic
```

Vidíme, že obsah proměnné získáme zápisem $NAZEV-PROMENNE. Teď můžeme zkusit obsah naší proměnné změnit. Třeba tak, že k ní něco přidáme.

```
TuX:~/skripty$ POKUS="Testov $POKUS Testenko"
TuX:~/skripty$ echo $POKUS
Testov Testovic Testenko
```

Proměnné mohou obsahovat jakékoliv typy hodnot, text, čísla atd. Shell dokáže vyhodnocovat jednoduchou celočíselnou aritmetiku, takže můžeme zkusit třeba tohle:

```
TuX:~/skripty$ PROM1=100
TuX:~/skripty$ PROM2=5
TuX:~/skripty$ PROM3=10
TuX:~/skripty$ VYSLEDEK=$[$PROM1-($PROM2*$PROM3)]
TuX:~/skripty$ echo $VYSLEDEK

50
```

Aby se aritmetika provedla, musíme výraz uzavřít do konstrukce **$[ ]**. Pokud bychom napsali pouze:

```
TuX:~/skripty$ VYSLEDEK="$PROM1-($PROM2*$PROM3)"
TuX:~/skripty$ echo $VYSLEDEK
100-(5*10)
```

neproběhlo by aritmetické vyhodnocení výrazu a shell by s proměnnými pracoval jen jako s textovými řetězci.
Veškeré proměnné a funkce, které máme nadefinované v shellu si kdykoliv můžeme vypsat pomocí příkazu **set**. Výpis bývá díky funkcím nadefinovaným v inicializačních skriptech dosti dlouhý a ne příliš přehledný. S proměnnými lze dělat spoustu dalších kouzel, ale to už je nad rámec této publikace.

13.4 Parametry

Ve skriptech můžeme zjišťovat hodnoty takzvaných pozičních parametrů. To jsou řetězce, oddělené mezerami zapsané za název skriptu, podobně jako píšeme parametry za normální Unixové příkazy. Díky tomu můžeme vytvářet skripty s parametry, což je velmi užitečné. K parametrům přistupujeme pomocí konstrukce **$cislo**, kde cislo udává pozici parametru na příkazové řádce. Parametry jsou číslovány zleva od jedničky, může jich být libovolný počet a samotný název skriptu je považován za parametr nula. Počet parametrů zjistíme pomocí konstrukce **$#**. Otevřeme tedy editor a napišme nový jednoduchý skript s názvem **paratest**.

```
TuX: :~/skripty$nano paratest
```

Názvu minulého skriptu jsme dali příponu **.sh**, ale to v Unixu vůbec není nutné, protože přípony nerozeznává. To, že soubor obsahuje skript pro daný shell pozná z první řádky:

```
#!/bin/bash
```

Skript bude vypadat následovně:

```
#!/bin/bash
# Jednoduchy skript pro pokusy s parametry
clear
echo "Skript s nazvem: $0 byl spusten s $# parametry"
echo "konkretne jsou to:"
echo -e "\t$1\n\t$2\n\t$3\n\t$4\n\t$5"
```

Skript uložte, nezapomeňte změnit práva pomocí chmod 755 a zkuste ho spustit s několika parametry:

```
TuX:~/skripty$ ./paratest prvni 2. -treti 4
```

Výstup by měl vypadat následovně:

```
Skript s nazvem: ./paratest byl spusten s 4 parametry
konkretne jsou to:
        prvni
        2.
        -treti
        4

TuX:~/skripty$
```

Všimněte si, že parametry jsou brané jako textové řetězce od mezery k mezeře bez ohledu na to, co obsahují.
Jak funguje ta záhadná poslední řádka skriptu s příkazem **echo**? Přepínač **-e** zajistí, že **echo** bude zpětné lomítko s následujícím znakem zpracovávat jako speciální tzv. **escape sekvence**. Těch je celá řada, ale právě sekvence **\n** – odřádkování a **\t** – znak tabulátoru se používají nejčastěji. **echo** tedy vypíše tabulátor, první parametr **$1**, odřádkuje, tabulátor, druhý parametr **$2** atd. Náš skript je připraven na výpis pěti parametrů, proto také při pokusu se čtyřmi parametry vynechal za posledním jednu volnou řádku. Zkuste ho spustit bez parametrů, s jedním parametrem a třeba se šesti parametry, abyste viděli, jak se chová. Jeho chování s různými počty parametrů není příliš pěkné, ale to vyřešíme hned v následující kapitolce.

13.5 Řízení běhu a opakované vykonávání skriptu

Síla skriptování spočívá v tom, že běh skriptů se dá ovládat na základě podmínek a pomocí cyklů. Detailní popis těchto prvků je nad rámec této publikace, proto si zde pouze uvedeme jednoduchý příklad, který nastíní, co zhruba se ve skriptech dá dělat.
Základní podmíněná konstrukce má syntaxi:
if [PODMINKA]; then PRIKAZY fi nebo
if [PODMINKA]; then PRIKAZY-ANO else PRIKAZY-NE fi
Podobně jako v jiných programovacích jazycích lze tyto příkazy vnořovat do sebe. Použitelných podmínek je celá řada a dají se spojovat logickými operátory do složitých konstrukcí. Vedle běžného testování řetězců **if** umí testovat výstupní kódy programů, existenci a různé stavy souborů atd. Jako příklad takové podmínky uvedu

`if [soubor1 -nt soubor2]`, který je pravda pokud **`soubor1`** byl změněn později než **`soubor2`**.
Dalším podmíněným příkazem je příkaz **`case`** se syntaxí:
`case VYRAZ in MOZNOST1)PRIKAZY1;;MOZNOST2)PRIKAZY2;;atd. esac`
Pro opakované provádění příkazů se ve skriptech dají používat příkazy cyklů **`for`**, **`while`** a **`until`** a s tím spojené příkazy **`break`** a **`continue`**.
Nyní jednoduchý příklad na kterém si ukážeme použití některých z těchto výrazů. Pokusíme se napsat zdokonalený výpis pozičních parametrů. Vytvořte soubor s názvem **`parametrovnik`**, který bude obsahovat následující skript:

```
#!/bin/bash
# Ukazkovy skript pro praci s parametry
echo "Jsem skript s nazvem $0 a ukazu vam jake parametry jste mi
zadali"
if [ "$#" == 0 ]; then   #test existence parametru
   echo "Nedelejte si za me legraci a zadejte mi nejake parametry!"
   exit 1;
fi
if [ "$#" -lt 3 ]; then  #test poctu parametru
   echo "No moc jste se s temi parametry nepredali, ale aspon neco!";
else
   echo "No $# je konecne slusny pocet parametru!";
fi
CISLO="1"   #inicializace promenne
for PARAMETR in "$@"; do   #cyklus prochazejici parametry
echo -e "\tParametr $CISLO ma hodnotu: $PARAMETR"
CISLO="$[$CISLO + 1]";   #inkrementace promenne
done
```

Po uložení a obvyklé změně práv zkuste skript spustit s různými počty parametrů. Pokud uděláte při programování někde nějakou chybu, skript doběhne až k ní a tam vypíše chybovou hlášku. Zkuste například namísto prvního **`then`** napsat překlep **`tneh`** a skript znova spustit. Měl by vypsat zhruba následující:

```
Jsem skript s nazvem ./parametrovnik a ukazu vam jake parametry jste
mi zadali
./parametrovnik: line 7: syntax error near unexpected token `fi'
./parametrovnik: line 7: `fi'
```

Jak vidíte, pro psaní skriptů je z důvodů snadného hledání chyb vhodný editor, který ukazuje pozici kurzoru. Zkuste skript znova otevřít v editoru a za první řádku přidat parametr **`-x`**:

```
#!/bin/bash -x
```

To způsobí, že shell bude při běhu skriptu vypisovat veškeré příkazy a chyby tak dohledáte daleko snadněji. Zkuste nyní skript spustit znova. Až bude skript odladěn, pouze parametr **`-x`** odstraníte.

13.6 Aliasy

Aliasy se používají pro zjednodušení často používaných příkazů. Řetězec aliasu je shellem expandován jen pokud se na příkazovém řádku nachází na prvním místě. Aliasy se definují příkazem alias a ruší příkazem unalias.

Příkaz alias bez parametrů vypíše všechny v současné době nadefinované aliasy.

```
TuX:~$ alias
alias ls='ls --color=auto'
alias la='ls --color=auto -a'
TuX:~$
```

V uvedeném příkladu jsou nadefinovány aliasy **ls** a **la**. Všimněte si, že alias **ls** se překrývá s vnitřním příkazem Unixu, zatímco **la** vytváří nový příkaz. Kdykoliv tedy napíšeme příkaz **ls**, shell nejprve zjistí, že se jedná o alias, provede jeho expanzi na **ls --color=auto**, a pak teprve příkaz provede[72]. **la** z našeho příkladu provede výpis souborů včetně skrytých. Napsat například **la /etc** je rozhodně příjemnější a rychlejší než **ls –a /etc**. Samozřejmě pokud chceme výpis v barvách, i u tohoto aliasu musíme rovnou doplnit parametr **–color=auto**, protože aliasy se v shellu rozbalují jen jednou.
Aliasy si můžeme nadefinovat jak se nám zachce, jen musíme dát pozor, abychom jimi omylem nezamaskovali nějaký existující příkaz Unixu. Pokud například chceme ušetřit práci při spouštění našeho oblíbeného editoru, můžeme si vyrobit alias **e**. Nejprve je dobré zkusit, zda **e** není nějakým příkazem.

```
TuX:~$ e
bash: e: command not found
TuX:~$
```

A teď už můžeme vytvořit alias:

```
TuX:~$ alias e='nano'
```

Až příště budeme editovat, stačí napsat **e soubor** namísto **nano soubor**.
Další užitečné aliasy mohou být například:

```
alias ..='cd ..'
alias ...= 'cd ../..'
alias ....= 'cd ../../..'
alias fnd= 'find / -name'
alias untar='tar -xvf'
alias untarz='tar -xvzf'
```

Pochopitelně fantazii se při tvorbě aliasů meze nekladou, jen je potřeba dát pozor na konflikty s příkazy Linuxu.

13.7 Funkce shellu

Představme si, že bychom chtěli nějaký „alias“ nahradit ne jedním, ale několika příkazy. To alias jako takový neumožňuje. Můžeme si ale v shellu nadefinovat svoji vlastní funkci, která slouží přesně k tomuto účelu. Například bychom chtěli namísto aliasu z minulého příkladu vytvořit funkci **untar**, která by archiv rozbalila a rovnou ho jako již nepotřebný smazala.
Definice funkcí má následující syntaxi:

```
nazev()
{
prikazy
}
```

Zapišme tedy normálně na příkazový řádek funkci:

[72] Jak je zřejmé, přidaný parametr zajistí oblíbený barevně odlišený výpis souborů a adresářů.

```
TuX:~$ untar()
> {
> tar -xvf $1
> rm $1
> }
TuX:~$
```

Shell ze syntaxe pochopí, že se jedná o definici funkce a nabízí další řádky uvozené zobáčkem, dokud neuzavřeme složenou závorku. Vidíme, že funkce zpracovávají parametry pomocí konstrukce **$cislo**, tedy stejným způsobem jako skripty.
Teď si můžeme zabalit náš adresář se skripty a pak ho pomocí této nové funkce rozbalit někde jinde.

```
TuX:~/skripty$ tar -cvf skripty.tar ./
TuX:~/skripty$ ls
parametrovnik  paratest  prvni.sh  skripty.tar
TuX:~/skripty$ cd ../pokusy
TuX:~/pokusy$ mkdir pok-skripty
TuX:~/pokusy$ cd pok-skripty/
TuX:~/pokusy/pok-skripty$ mv ~/skripty/skripty.tar .
TuX:~/pokusy/pok-skripty$ ls
skripty.tar
TuX:~/pokusy/pok-skripty$
```

Vytvořili jsme nový adresář a do něj přestěhovali archiv se skripty. pokud bychom ho nyní rozbalili pomocí příkazu **tar –xvf skripty.tar**, rozbalili by se soubory, ale archiv by v adresáři zůstal také. Když ovšem použijeme naší funkci **untar skripty.tar**, dojde k rozbalení a hned také smazání archivu.
Všechny nadefinované funkce můžeme vypsat příkazem **set** a zrušit naší definici můžeme pomocí **unset untar**.

13.8 Uživatelské konfigurační skripty

Už jsme se naučili mnoho užitečností, kterými si můžeme značně zjednodušit práci v Linuxu. Problém ovšem je, že věci jako přidání cesty k našim skriptům nebo definice aliasů a funkcí bude fungovat jen v rámci právě spuštěného shellu, takže po odhlášení a opětovném přihlášení budeme muset všechno definovat znova. Možná vás napadne si napsat skript, který bude všechny potřebné definice obsahovat a ten si po každém přihlášení spustit. Řešení je ovšem ještě jednodušší. Takový skript totiž už existuje[73]. Je to skrytý soubor ve vašem domovském adresáři s názvem **.bashrc**. Tento skript shell spustí vždy ihned po přihlášení, takže jakékoliv definice, které do něj umístíme v našem pracovním prostředí budou platit pokaždé, když se přihlásíme. Když si ho prohlédnete, zjistíte, že je to poměrně dlouhý skript s různými nastaveními.
Všimněte si řádky, která přiřazuje proměnné **PS1** podivný řetězec:

```
PS1='\[\033[1;35m\]\h\[\033[0m\]:\[\033[1;34m\]\w\[\033[0;3
2m\]\$\[\033[0m\] '
```

Proměnná **PS1** určuje v shellu **bash** tvar výzvy na příkazovém řádku. Podivné sekvence znaků jako například **\[\033[1;35m\]** jsou takzvané ANSI sekvence určující barvy

[73] Ve skutečnosti **bash** spouští několik různých takových skriptů v závislosti na typu jeho spuštění, ale pro uživatelská nastavení je nejdůležitější právě **.bashrc**.

zobrazení a sekvence jako **\h** určují, co se má v promptu zobrazit (v tomto případě **hostname**). V našem případě tedy celá sekvence znamená:

- barva purpurová
- hostname (\h)
- barva normální
- :
- barva tmavě modrá
- pracovní adresář (\w)
- barva zelená
- $ nebo # (\$)[74]
- barva normální

Pokud chcete s promptem experimentovat, vyhledejte si na Internetu výborný a rozsáhlý tutoriál „Bash prompt howto“, kde se dozvíte o promptu vše.
Dále si ve skriptu .bashrc všimněte závěrečné sekce, která definuje několik aliasů¨a několik dalších je v ní zakomentovaných. Těsně před touto sekcí je trojice řádek, která stojí za povšimnutí:

```
if [ -f ~/.bash_aliases ]; then
    . ~/.bash_aliases
fi
```

Je to totiž typická konstrukce, která se ve skriptech často používá pro jejich zpřehlednění. Příkaz **if** zjistí, zda ve vašem domovském adresáři existuje soubor s názvem **.bash_aliases** a pokud ano, provede jeho sourcování, to znamená, že řádku po řádce vykoná v něm uvedené příkazy. To má stejný efekt, jako kdyby příkazy byly napsané přímo v našem skriptu, ale přitom je máme uložené v samostatném souboru. Pokud si tedy chcete nadefinovat své vlastní aliasy, doporučuji nedělat to přímo ve skriptu **.bashrc**, ale založit si soubor **.bash_aliases** a teprve v něm uvést všechny jejich definice.
Podobně pokud si chcete vytvořit nějaká svoje další speciální nastavení, doporučuji pro přehlednost na úplný závěr skriptu .bashrc uvést pouze řádky:

```
if [ -f ~/.bash_moje ]; then
    . ~/.bash_moje
fi
```

Pokud soubor .bash_moje nebude existovat, nic se díky konstrukci if nebude dít. Pokud si ho ale v domovském adresáři vytvoříte, můžete si do něj přehledně umístit vše potřebné.

Tato kraťoučká kapitolka o skriptování je opravdu jen úplně stručným nástinem, jak skripty vypadají a co zhruba mohou dělat. Jejich možnosti jsou pochopitelně řádově větší, než jsme si mohli ukázat v rozsahu této útlé knížečky. Například jen v The Linux Documentation Project je úvodu do bashe věnováno zhruba 140 stran a samozřejmě existují další tlusté knihy jen o skriptování v shellech. Nemusíte ovšem sahat jen po papírových knihách, Internet je plný různých howto a tutoriálů na toto téma.

[74] **$** se ukáže u normálních uživatelů, **#** u uživatele root

SPŠ sdělovací techniky
www.panska.cz
2008