Bash

očima

Bohdana Milara

Obsah

1.díl

Proberme si úlohu **Bashe** v operačním systému podrobněji a zkusme k ní najít ekvivalent v oblasti grafického ovládání. Takže když startuje takový systém, spouští se jako první **jádro**, které načte potřebné **moduly** (ovladače apod.). Následuje spuštění programů na pozadí (např. různé servery) – viz známé výpisy OK během startu. Ve stavu, kdy systém běží, disponuje potřebnými ovladači a podpůrnými službami, může předat řízení uživateli. Jaký způsob řízení to bude, záleží na potřebách a možnostech majitele, správce a samotných uživatelů. Můžeme si představit ovládání textové (klávesnice + monitor), grafické (klávesnice + myš + monitor) nebo třeba hlasové (mikrofon + reproduktor). Hlas je asi poněkud futuristická vize, grafika je v současné době standard a na text se již pohlíží nejčastěji jako na přežitek. Textové uživatelské prostředí je z nich skutečně nejstarší. Pochází z dob velkých sálových počítačů. Ty měly jednu hlavní klávesnici a monitor (tzv. konzole). Protože byl počítač drahý (a velký), byla snaha umožnit co největšímu počtu pracovníků používat jej současně. Aby se pracovníci nemuseli těsnat v jedné místnosti u klávesnic napojených přímo na počítač, byly vynalezeny jednoduché přístroje s obrazovkou a klávesnicí, které se připojovaly např. k sériovému rozhraní počítače. Říkalo se jim terminály. Konzole a terminály vytvářely nezbytný hardwarový základ textového prostředí. Pro práci s počítačem potřebujeme stanovit **syntaxi** vzájemné komunikace (něco jako pravidla příkazového pravopisu) a soubor základních příkazů. To představuje jakousi ulitu, skořápku či plášť (anglicky **shell**), v němž se může uživatel bezpečně po systému pohybovat. Dnes má téměř každý počítač jen jeden monitor a klávesnici a k tomu ještě myš. V linuxových distribucích se při startu systému většinou na hlavní **konzoli** automaticky spouští grafické prostředí (nejčastěji některá z variant X Window System). Při přihlašování v grafickém prostředí si pak vyberete svůj oblíbený správce oken (Window Manager), třeba GNOME nebo KDE. Tento správce vám definuje pravidla práce s myší (tedy něco jako myšoidní příkazy) a nabídne soubor základních nástrojů (správce souborů, lištu, hlavní nabídku, ...). Správce oken je tedy takový grafický shell. V textovém prostředí zastávají funkci shellu takzvané vykonavače (interprety) příkazových řádků (Command Line Interpreter – CLI), zkráceně příkazový interpret. Jsou to programy, které zobrazí na začátku řádku pár znaků a (obvykle blikající) kurzor. Čekají na vstup z klávesnice, vykonají zadaný příkaz, jeho výsledky zobrazí na obrazovce a zase zobrazí výzvu s kurzorem. Tento princip používal Unix již v sedmdesátých letech 20. století. Napodobovaly jej později i disketové operační systémy (DOS) vznikající na přelomu 70. a 80. let. Unixové shelly však (i vzhledem ke své delší historii) nabízejí mnohem větší komfort ovládání, jak si postupně v tomto seriálu ukážeme. Příkazových interpretů pro unixové systémy existuje povícero. Dělí se do dvou skupin – Bourne shelly a C shelly. Každá skupina má své výhody a nevýhody. My se zaměříme na **Bash** (i Again Shell), který vznikl jako součást projektu GNU a snaží se kombinovat výhody zástupců obou skupin příkazových interpretů.

Kde najít příkazový interpret?

Dříve, než si ukážeme, co všechno **Bash** dokáže, musíme zodpovědět jednu základní otázku. Kde vlastně v prostředí X takový **Bash** najdeme? Jak spustit textové prostředí, když systém nabootuje do grafiky? Musím si koupit ke svému PC terminál? Jen klid. Žádný terminál si pořizovat nemusíme, i když, pokud bychom jej měli, také by to fungovalo. Podpora klasických terminálů (připojených přes sériové rozhraní) je v unixových systémech přítomna dodnes. Jsou však jednodušší způsoby, jak se k Bashi můžeme dostat. Jedním z nich jsou virtuální konzole. Ty běží tiše paralelně s naší grafickou obrazovkou. Stačí stisknout kouzelnou kombinaci kláves [**Ctrl+Alt+F1**] a máme textovou konzoli. Celkem jich paralelně běží obvykle šest, takže můžete postupně vyzkoušet zaměnit klávesu [**F1**] za [**F2**], [**F3**] až [**F6**]. Při přechodu mezi textovými obrazovkami nemusíme držet [**Control**]. V textovém režimu se (stejně jako v grafice) musíme před začátkem práce nejprve přihlásit. K tomu slouží výzva **Login:**, za niž napíšeme své uživatelské jméno. Po potvrzení klávesou [**Enter**] budeme vyzváni k zadání hesla (**Password:**), které se nebude vypisovat na obrazovku. Po úspěšném zadání by se měl spustit **Bash**. Do grafiky se vrátíte pomocí [**Alt+F7**]. Další možností je spustit si okno s emulátorem terminálu přímo v grafickém prostředí. K tomu slouží např. program **xterm**, **Konsole** (KDE) nebo **gnome-terminal** (GNOME). V něm se hned spustí **Bash** a zobrazí výzvu s kurzorem. Zde se přihlašovat nemusíme, neboť jsme se přihlásili již do správce oken. V neposlední řadě je možné se k našemu počítači přihlásit po síti. Lze připojit terminál (třeba počítač s terminálovým programem) na sériový port. Jednodušší ale možná dnes bude toto přihlášení provést z jiného počítače po síti (např. ethernet). I při vzdáleném přihlášení se po nás bude chtít ověření totožnosti v podobě jména a hesla.

GNU Bourne-Again Shell

Čím začít? Je toho prostě tolik. Nejprve si musíme vzájemně rozumět. Proto začnu vysvětlením základních pojmů používaných v prostředí příkazových interpretů. První, čím se nám **Bash** ohlásí, je několik znaků na začátku řádku, za kterými stojí kurzor. Toto uskupení se nazývá výzva (anglicky **prompt**). Před kurzorem se mohou zobrazovat různé informace od jména přihlášeného uživatele, přes jméno počítače, aktuální adresář, čas až po verzi Bashe. Struktura promptu se dá měnit podle přání uživatele. Jak toho lze přesně dosáhnout, si předvedeme v některém z dalších dílů tohoto seriálu. Původní hodnota je nastavena při instalaci nebo ji určuje správce systému.
Vždy, když se v Bashi zobrazí prompt, je možné zadávat příkazy. Pokud kurzor jen stojí na začátku prázdného řádku, interpret zřejmě vykonává nějaký příkaz. V takovém případě je nutné vyčkat, až práci ukončí, nebo jej násilně ukončit. Násilné ukončení provedeme stiskem kombinace kláves [**Ctrl+c**].
Ukažme si tuto funkci na příkladu. K příkladu využijeme příkaz **sleep** (spát), za který můžeme napsat číslo. Toto číslo udává počet sekund, po které bude **Bash** spát, tj. nevykonávat další příkazy, napište třeba **sleep 5** a stiskněte klávesu

[**Enter**] (jednou z těchto kláves je nutné ukončit zadávání každého příkazu, resp. řádku s více příkazy). Nyní bude Bash pět sekund spát a pak zase ukáže **prompt**. Dále zadejte **sleep 55** a Bash by spal 55 sekund. My ale po několika sekundách budeme chtít spánek přerušit a zase pracovat, a proto sleep ukončíme pomocí [**Ctrl+c**]. Ukáže se nám prompt. Součástí promptu bývá jméno aktuálního (pracovního) adresáře, ale často jen jeho konec. Např. místo **/home/jana** jenom **jana**. Příkaz **pwd** nám zobrazí celou cestu k aktuálnímu adresáři. Nástroj **pwd** patří k tzv. interním příkazům Bashe. Interní příkazy jsou takové, které jsou přímo součástí **shellu**, a ten je před jejich vykonáním nemusí načítat z disku. Obecně to, co zadáme na řádku jako první, je chápáno jako příkaz. Pokud neodpovídá žádnému internímu příkazu, začne Bash hledat spustitelný soubor (program) stejného jména na disku. Příkladem takového externího příkazu je **ls** vypisující obsah aktuálního (nebo zadaného) adresáře. Některé příkazy můžeme použít jen takto samostatně, většina jich ale vyžaduje parametry. To je prostředek k tomu, aby uživatel svůj požadavek upřesnil (např. s čím a jak má příkaz pracovat). Dejme tomu takové **ls /home** vypíše obsah adresáře **/home**. Nemusíme zadávat jen jeden argument. Více argumentů od sebe oddělujeme mezerou. Takže **ls /home /var/spool** vypíše obsahy adresářů **/home** a **/var/spool** (nikoli **/var** a **/spool**, který navíc ani neexistuje). Před argumenty lze většinou uvést ještě různé volby, které mění nebo rozšiřují funkcionalitu příkazu. Třeba **ls -l /home** vypíše opět obsah **/home**, tentokrát ovšem v dlouhém formátu (více informací na jednom řádku, včetně vlastníka a přístupových práv). Také voleb můžeme uvést vícero současně. Kdybychom chtěli zobrazit i skryté soubory (tj. ty, jejichž jméno začíná znakem .), použili bychom volbu **-a**. Nemusíme však zadávat **ls -l -a** ale můžeme si zápis zkrátit na **ls -la** kdy v obou případech získáme podrobný seznam všech souborů v aktuálním adresáři. To jsou volby v krátké (jednoznakové) podobě. K dispozici však bývají (často i pro stejnou funkci) také volby dlouhé. Tak třeba pro zmíněné **-l -a** by bylo ekvivalentem **ls --format=long –all** Je zřejmé, proč mají dlouhé volby před sebou dvě pomlčky. Kdyby tomu tak nebylo, nedalo by se zadávat více krátkých voleb za jednou společnou pomlčkou, neboť by z nich mohlo vzniknout slovo (např. **-a -l** dá dohromady **-al**, což je velmi podobné **–all**).

Nápověda

Velmi užitečnou volbou, kterou nabízí většina příkazů, je **--help**. Zadáme-li **touch --help**, vypíše se nápověda k programu **touch**. Na prvním řádku (Usage – použití) vidíme správný postup zápisu příkazu (tzv. **syntaxi**). Je tam samozřejmě vlastní jméno programu (tj. touch) následované textem **[OPTIONS]... FILE...** Volby (anglicky Options) jsou v hranatých závorkách, což znamená, že není povinnost je udávat. Tři tečky zase říkají, že voleb může být více. **File** (tedy soubor) znamená soubor, kterého se chceme příkazem **touch** "dotknout". Souborů může být zadáno více (tři tečky), přičemž jejich jména musí být oddělena mezerou. Na druhém řádku nápovědy je stručný popis příkazu. Dočteme se v něm, že **touch** nezmění obsah souboru, ale nastaví čas posledního přístupu na zadanou hodnotu. Pokud soubor neexistuje, bude jako prázdný vytvořen. Poslední část nápovědy představuje seznam voleb ve zkrácené a nebo plné podobě se stručným popisem.

Manuálové stránky

Komu nestačí nápověda získaná pomocí **--help**, může pro většinu příkazů využít manuálové stránky. Jejich vyvolání je snadné. Stará se o ně program **man**, jehož parametrem je jméno manuálové stránky, což je zpravidla zároveň jméno programu. Po zadání **man ls** tak dostaneme manuálovou stránku programu **ls**. V záhlaví vidíme stručnou definici programu, následuje podrobná syntaxe. I zde platí, že parametry uvedené v hranatých závorkách nejsou povinné. I v případě **ls** se můžeme setkat s vnořenými hranatými závorkami. Ty představují to, že pokud se rozhodneme jejich přímý obsah použít, můžeme si dále volit, zda budou přítomny i určité rozšiřující části volby. Je-li v jedněch závorkách uvedeno několik možností oddělených svislou čárou (znak **|**), znamená to, že si můžeme vybrat jen jednu z nabízených možností. Obdobný význam mají slova oddělená čárkou a uzavřená do složených závorek. Zde však musí být alespoň jedna hodnota použita. Např. **[--color[={yes,no,tty}]]** znamená, že můžeme použít parametr **--color**, a to buď samotný, nebo se znakem **=**. Pokud už ale přidáme rovnítko, musíme pokračovat jedním ze slov **yes,no,tty**.
V další části manuálové stránky bývá podrobný popis daného programu. Tuto část doporučuji si pročíst, neboť se tak můžete snadno dovědět, že určitý příkaz má mnohem více funkcí, než byste čekali. V některých případech zde najdete odkazy na jinou dokumentaci ke zkoumanému problému.
Asi nejčastěji vyhledávanou částí manuálové stránky je přehled voleb. Každý příkaz má vlastní množinu voleb. Je sice pravda, že některé volby se vyskytují téměř u všech příkazů (např. **--help** nebo **--version**), ale záleží vždy jen na programátorovi, zda a jak je do programu zahrne. Delší manuálové stránky mohou obsahovat ještě vysvětlivky zkratek použitých v definici voleb. V případě **ls** je to třeba definice barevné palety pro výpis adresářů. Ke konci stránky se dovídáme něco o chybách, ale bývají zde i odkazy na ostatní manuálové stránky vztahující se k tématu.
Po manuálové stránce se pohybujeme šipkami, klávesami [**PgUp**] a [**PgDn**], ale i dalšími způsoby. (Funguje vlastně celá sada klávesových příkazů jako v editoru **Vim**. Například **gg** pro skok na začátek stránky, **lomítko** pro vyhledání textu. Pozn. redakce.) K vlastnímu zobrazení manuálových stránek je používán program **less**. Ten můžete opustit klávesou [**q**], nápovědu získáte klávesou [**h**].

Závěr

To byl úvod do textového prostředí v Linuxu. Ještě než se pro tentokrát rozloučíme, měl bych zmínit, jak se z Bashe správně odhlásit. Můžete použít jeden z příkazů **logout** a **exit**. Většinou je lze nahradit kombinací kláves [**Ctrl+d**].
A příště se podíváme na další užitečné klávesové zkratky, řízení úloh a jiné zajímavosti.

2.díl

Náplní dnešního dílu budou některé interní příkazy a klávesové zkratky související s řízením procesů – tedy programů spuštěných z Bashe. Ukážeme si také výhody kooperace textového a grafického prostředí.

Úlohy v pozadí a na popředí

V dobách, kdy byly procesory pomalé, trvalo vykonání některých příkazů dlouho. I paměti bývalo tehdy v počítačích mnohem méně, a proto nebylo žádoucí spouštět hodně interpretrů paralelně. Aby i tehdy bylo možné co nejlépe využít víceúlohového prostředí (multitaskingu), zavedly interpretry možnost přepnout program spuštěný v textovém prostředí na pozadí. Spustit nebo přepnout program na pozadí znamená, že v průběhu jeho běhu máme k dispozici **prompt** a můžeme zadávat další příkazy. Ptáte-li se, proč tento historický nástroj zmiňuji v době, kdy máme dost paměti k pouštění velkého množství Bashů vedle sebe, vězte, že i dnes má toto řízení procesů své uplatnění.
Ačkoli je to paradox, tak učit se řídit procesy v Bashi má dnes největší význam díky prostředí X Window System. Jsou uživatelé, kteří příkazovou řádkou opovrhují a pracují pouze graficky. Jsou tací, kteří opovrhují grafikou. Také jsou uživatelé, kteří by grafiku využívat chtěli, ale něco jim v to brání – nedisponují dost silným hardwarem k provozu X, mají ke stroji přístup jen přes **ssh**, jsou nevidomí (pro nevidomé je zatím uspokojivá podpora jen v textovém prostředí) apod.
Mnoho uživatelů ovšem dospělo k názoru, že obě prostředí mají své výhody a nevýhody. Díky systémové integritě Linuxu je naštěstí lze velmi komfortně používat společně a kombinovat tak výhody obou. Docílíme toho spuštěním virtuálního terminálu v prostředí X (**Konsole**, **gnome-terminal**, **xterm**, ...), čímž získáme Bash v okně. To nám umožní nejen rychle přecházet mezi oběma prostředími, ale používat i různé systémové nástroje pro jejich spolupráci. Jednou z forem spolupráce je kopírování textu. Např. výstup nějakého textového příkazu můžeme v terminálovém okně označit myší (táhnutím při držení levého tlačítka), čímž jej automaticky uložíme do schránky. Obsah schránky pak vysypeme (třeba prostředním tlačítkem myši) do jakéhokoli okna, kupříkladu textového editoru **kedit** nebo textového procesoru **OpenOffice.org Writer**. Kopírování samozřejmě funguje i obráceně. Bezproblémová spolupráce grafických a textových aplikací je možná díky tomu, že v nich jádro systému (**kernel**) nedělá rozdíly. Problémem tak není ani nastartovat grafickou aplikaci z terminálového okna. Stačí napsat jméno spustitelného souboru jako příkaz, odenterovat a program se spustí. Do terminálu pak někdy dokonce vrací informace o průběhu své činnosti. A tím se dostáváme k důvodu, proč nás zajímá řízení grafických procesů z Bashe. Spustíme-li prostřednictvím Bashe nějakou aplikaci v grafickém prostředí, neukáže se nám **prompt** dříve, než tato aplikace skončí. Pouštět přitom aplikace z terminálu bývá výhodné. Známe-li jméno programu, je jeho napsání rychlejší než hledání v často rozsáhlé spouštěcí nabídce (viz ikona na panelu, nejčastěji vlevo dole). Některé aplikace dokonce ani položku v hlavní nabídce mít nemusejí. Dalším důvodem jejich spouštění z příkazové řádky je možnost zadat různé parametry.
Jak to tedy udělat, abych měl k dispozici po spuštění Xové aplikace opět **prompt**, mohl dále pracovat nebo spustit aplikaci další? O kombinaci [**Ctrl+c**] jsem mluvil minule. Ta ovšem vede k ukončení aplikace, a to my zde nechceme. Řekněme, že jsem z terminálu spustil **gkrellm** – panel pro monitorování systému, který mi teď blokuje **prompt**. Aplikace běží a kurzor v Bashi stojí (nebo bliká) bez promptu na začátku nového řádku. Takto běžící program můžeme z terminálu zastavit kombinací kláves [**Ctrl+z**]. Objeví se hláška **Stopped**. Od tohoto okamžiku sice máme zase **prompt**, ale program je zamrzlý. Aby mohl ve své činnosti pokračovat, musíme mu povolit činnost na pozadí příkazem **bg**. Po odentrování se ihned rozběhne.
Existuje ale i způsob, jak spustit grafickou aplikaci na pozadí přímo. Dělá se to přidáním znaku ampersand (**&**) za příkaz spouštějící aplikaci. Tak např. XMMS spouštíme z příkazové řádky zadáním **xmms &**. Po odentrování dostaneme zpátky prompt a (někdy s drobnou prodlevou) také okno spouštěné aplikace.

Řízení více souběžných procesů

Z výše zmíněného vyplývá, že programů na pozadí můžeme z jednoho shellu spustit více souběžně. Jak se v nich potom ale vyznat, když jim chceme posílat různé signály? Jednoduše, neboť Bash si tyto úlohy (anglicky **jobs**) čísluje. Seznam běžících (**Running**) nebo stojících (**Stopped**) úloh získáme příkazem **jobs**. V seznamu vidíme na každém řádku jednu úlohu. V hranaté závorce je uvedeno číslo úlohy, následuje status (**Running**/**Stopped**) a zakončuje jméno, resp. příkaz, kterým jsme úlohu spustili. Velmi důležité je zde číslo úlohy, pomocí něhož můžeme jednotlivé z nich oslovovat.
Chceme-li např. vrátit z pozadí do popředí úlohu číslo 2, zadáme **fg %2**. Bash vypíše jméno příslušné úlohy a začne ji provádět na popředí, tj. nevrátí nám prompt. Ten bychom získali známým postupem [**Ctrl+z**] a **bg %2**.
Každý Bash si v sobě spuštěné úlohy čísluje vzestupně od čísla 1. V každém Bashi tak můžeme komandovat pouze ty úlohy, které byly spuštěny právě v něm. Při spuštění každé úlohy se nám ale za jejím číslem (v hranatých závorkách) ukáže vždy ještě jedno, zpravidla čtyř- až pětimístné číslo. To značí číslo procesu v rámci celého systému (tzv. **PID**).
Nyní se podíváme ještě na pár parametrů příkazu **jobs**. Zadáme-li Bashi požadavek ve tvaru **jobs -l**, zobrazí se nám mezi číslem úlohy a jeho stavem ještě **PID**. **jobs -r** vypíše pouze běžící úlohy, **jobs -s** pak pouze ty zastavené. Přepínače lze samozřejmě kombinovat, takže po **jobs -lr** dostaneme seznam běžících úloh s **PID**.
PID je unikátním celým číslem (každé číslo náleží právě jednomu procesu v běžícím systému), které jádro přiděluje vzestupně každému spuštěnému procesu. Číslo 1 má vždy proces **init**, který je v systému spuštěn jako první.
Pro zjišťování **PID** kteréhokoli z běžících procesů (tedy nejen úloh spuštěných v rámci příslušné instance interpretru Bash) slouží program **ps**. Když jej spustíme bez přepínačů (kterých má vskutku požehnaně), uvidíme jen aktuální Bash,

seznam spuštěných úloh a vlastní běžící **ps**. U každého vypsaného procesu můžeme zjistit jeho **PID**, na kterém běží terminálu (**tty2** je druhá virtuální konzole – [**Ctrl+Alt+F2**], **pts/1** je první virtuální terminál – např. okno s Bashem v rámci X), využitý strojový čas a příkaz, kterým byl proces spuštěn.

Z dalších přepínačů vyberme:
• **ps x** – vypíše všechny procesy daného uživatele bez ohledu na terminál (Bash) a s celou syntaxí příkazu;
• **ps a** – vypíše procesy všech uživatelů na daném terminálu;
• **ps u** – vypíše procesy s uvedením uživatele, zátěže CPU a RAM;
• **ps j** – vypíše procesy s **PID** rodičovského procesu;
Jednotlivé přepínače **ps** lze kombinovat. **ps aux** tak bude následek seznam všech procesů všech terminálech s celou syntaxí příkazu, kterým byly spuštěny.

Kill aneb Bash zabiják

Již jsme si řekli, jak se na spuštěné úlohy podívat a jak je přepínat do popředí a do pozadí. Teď si ukážeme, jak ukončit (též sestřelit nebo zabít) nějakou úlohu na pozadí. Lze to samozřejmě udělat tak, že ji nejprve přepneme do popředí a potom ukončíme klávesovou kombinací [**Ctrl+c**]. Existuje ale i jiný způsob. Seznamme se s legendárním příkazem **kill** (zabít). Dle názvu bychom mohli mylně usuzovat, že je práce tohoto nájemného vraha jednotvárná. On je to ale ve skutečnosti nevinný pošťák, který doručuje jednotlivým procesům balíčky s příkazy. Naneštěstí pro jeho pověst v těch balíčcích tu a tam bývá bomba. Seznam typů těchto balíčků, nebo chcete-li nařízení (v unixovém jazyce hovoříme o **signálech**) najdeme mj. v manuálové stránce (**man 7 signal**). Nelekejte se široké nabídky, budeme z nich potřebovat jen pár. Vlastně nás bude zajímat jen číslo 9, což je již zmiňovaná dopisní bomba. Abychom určili adresáta, použijeme číslo úlohy (třeba **kill %3**) nebo číslo procesu (**PID** – např. **kill 7190**). Pokud neuvedeme, jaký signál procesu posíláme, bude doručen **SIGTERM**, tedy číslo **15**. Stiskneme-li potom na prázdném promptu [**Enter**], uvidíme hlášku ve stylu **[3]- Ukončen (SIGTERM) kedit**. Pro doručení jiného signálu uvedeme jeho číslo (nebo jméno) s pomlčkou. Pokud třeba program nereaguje na **SIGTERM** (třeba protože je zaseknutý), můžeme jej násilně ukončit (zabít, sestřelit) signálem **9** (**SIGKILL**) takto: **kill -9 %2**. Po odentrování prázdného řádku uvidíme kupř. **[2]+ Zabit (SIGKILL) ksnake**.

3.díl

Tento seriál se snaží seznámit především začínající uživatele s textovým prostředím operačního systému GNU (Linux), jehož předním představitelem je příkazový interpretr **Bash**. Dnešní díl se věnuje problematice řízení procesů, což znamená spouštění, přerušování a ukončování programů (textových i grafických) v prostředí Bash. Mezi jinými bude řeč i o populárním programu **kill**. Našemu seriálu o práci s textovým interpretrem Bash se dostává prostoru ke třetímu pokračování. Minule jsme se seznámili s řízením procesů a úloh. Jen připomínám, že proces je jakákoli aplikace běžící právě v systému, kdežto úloha je proces spuštěný v konkrétním interpretru Bash. V tomto dílu si předvedeme některé způsoby spouštění programů v Bashi. Je jich totiž více, než jen napsat jméno spustitelného souboru do příkazové řádky. Ukážeme si ale též možnosti, jak běžící procesy sledovat.

Jméno aplikace a cesta

Nejjednodušší způsob spouštění aplikací v Bashi již znáte. Je to prosté zapsání jména spouštěného programu a jeho odentrování. Pokud máme program schován v nějakém nesystémovém adresáři (co to znamená nesystémový, si popíšeme později), musíme zadat celou cestu. Celá cesta se skládá ze jmen všech adresářů od začátku adresářového stromu, tj. od adresáře **/**. Např. program **mojehra** uložený v adresáři **hry**, který je podadresářem adresáře **/opt**, bychom spustili zadáním příkazu **/opt/hry/mojehra**. Tomuto zápisu se říká absolutní cesta. My však můžeme využít i cesty relativní, tj. popsat cestu k souboru nikoli od kořenového adresáře **/**, ale od místa, kde se právě nacházíme. Naše momentální umístění v adresářovém stromu můžeme tušit ze samotného promptu, ale nejspolehlivěji je zjistíme příkazem **pwd**. Dejme tomu, že stojíme v adresáři **/opt** a chceme spustit stejný program jako v předchozím odstavci. Potom nejkratší cestou je **hry/mojehra**. Pro relativní cesty můžeme využívat také speciální názvy adresářů. Jedním z nich jsou dvě tečky (**..**), které značí nadřazený adresář. Dejme tomu, že stojíme právě v adresáři **/home/kuba**, ale chceme spustit soubor **/home/public/editor**. Potom stačí zadat **../public/editor**. Všimněte si, že při zadávání relativní cesty nezadáváme na začátku znak **/**. Druhým pseudoadresářem pro zadávání relativní cesty je tečka (**.**), která označuje aktuální adresář. Ptáte se, proč pojmenovávat aktuální adresář? Je to proto, že GNU (a Unix) pracuje trochu jinak než DOS. Když zadáme jméno spustitelného souboru v DOSu, ten se dívá, zda toto jméno existuje v aktuálním adresáři, a pokud ano, tak jej spustí. Bash prohledává systémové adresáře, protože jsou to většinou ty, kam může instalovat programy jen správce a sníží se tak nebezpečí různých podvržených aplikací, trojských koní a podobných věcí. Popišme si takové ohrožení bezpečnosti na jednoduchém příkladu. Pokud třeba v domovském adresáři nějakého uživatele napíšeme **ls**, spustí se **ls** nainstalovaný v systémovém adresáři **/bin**, a nikoli skript **ls**, který si do svého adresáře umístil škodolibý uživatel, aby pomocí něj mazal soubory jiných uživatelů, kteří jeho adresář navštívili a chtěli se v něm porozhlédnout. Pokud ale přece jen z nějakého důvodu chceme úmyslně spustit program nebo skript ležící v aktuálním adresáři (např. když jsme před tím do příslušného adresáře přešli kupř. pomocí příkazu **cd /opt/hry**), stačí zadat **./mojehra** a zmáčknout [**Enter**] nebo [**Return**].

Složené závorky

V praxi někdy nastávají situace, kdy potřebujeme, aby bylo více příkazů provedeno pospolu. Ukážeme si tedy, jak se v Bashi provádějí groupové orgie a že jejich výsledkem jsou děti. Další zářný příklad toho, že Bash a GNU jsou zcela v souladu s přírodou. Když jsme chtěli na chvíli uspat **shell**, použili jsme k tomu v prvním díle příkaz **sleep**. Pokud bychom chtěli, aby se po skončení mikrospánku něco přihodilo, můžeme postavit na jeden řádek dva či více příkazů vedle sebe a oddělit je středníkem. Takže zadáním **sleep 10; echo Budíček!** získáme "Budíček", ale až po deseti sekundách. A nyní si představte, že chceme tuto konstrukci spustit na pozadí. Řeknete si fajn, to je úkol pro **ampersand**. Správně, ale kam ho dát? Žeby **sleep 10; echo Budíček! &**? Chyba – tím pošleme na pozadí pouze echo, ale celý spánek (**sleep**) bude probíhat na popředí, tedy bránit nám využívat **shell**. Pak tedy zůstává jen možnost **sleep 10 & ; echo Budíček!**. A zase chyba – **ampersand** může stát jen na konci příkazového řádku. Co teď? Je to jednoduché. Skupinu příkazů, které potřebujeme provést současně, uzavřeme do složených závorek. Ty pak můžeme klidně poslat na pozadí. Takže vítězem soutěže je:

```
{ sleep 10; echo Budíček! ; } &
```

Provede přesně to, co jsme chtěli. Bude spát na pozadí a po probuzení vypíše text Budíček! Během spánku na pozadí můžeme s Bashem normálně pracovat. Důležitý v této syntaxi je středník před uzavírací závorkou. Doporučuji také kolem obou závorek nechávat mezery. Ještě jsem dlužen vysvětlit to, jak se sdružováním příkazů souvisejí děti. Nechám si to, ale na kapitolu Sledování procesů. Zatím se spokojme s tím, že příkazy uvedené ve složených závorkách se spouštějí v aktuálním shellu.

Exec

Příkaz **exec** (zkratka z anglického execute – provést, vykonat) prostě provede za ním následující příkaz. Říkáte si, že je to zbytečnost? Byla by – nebýt toho, že se takto spuštěný příkaz chová trochu jinak, než příkaz zadaný bez **exec**. Exec je zřejmě nástroj z dob nedostatku operační paměti. Příkaz jím spuštěný totiž v paměti nahradí rodičovský shell. A už tu máme rodinnou ságu. Skutečně – v GNU se vztahy mezi procesy přiměřují ke vztahům rodovým. Setkáváme se tak s rodiči. Jimi spuštěné procesy jsou děti. Ty mohou mít další potomky, atd. Proces spuštěný pomocí **exec** bychom ve světle zmíněného mohli označit za pohrobka. Rodič totiž během porodu (resp. těsně před ním) umírá – obětuje své místo v paměti pro své dítě. Tato situace může budit dojem dobrého skutku, ale má i svá úskalí. Pokud shell ukončí před spuštěním procesu svou činnost, nemá se systém kam vrátit, až nový proces dokončí svůj úkol. Ve virtuální konzoli to vede k odhlášení, v emulátoru terminálu k zavření okna nebo záložky. Na to je třeba pamatovat.

Sledování procesů

Už minule jsme se dívali pod pokličku našeho systému, když jsme si vypisovali různé seznamy běžících procesů. Používali jsme k tomu ovšem jen jednoduché nástroje **ps** a **jobs**. Dnes si v této oblasti ukážeme sofistikovanější programy, které dovedou např. řadit procesy dle různých veličin nebo zobrazit příbuzenské vazby mezi procesy. Pokud v systému nastanou problémy, je nutné najít viníka – většinou nějakou spuštěnou aplikaci. K tomu nám dopomůže nástroj **top**.
Je to interaktivní textový program, tzn. že po spuštění reaguje na určité klávesy. Jeho úlohou je zobrazovat seznam běžících procesů seřazených dle určité charakteristiky. Zkusme si prostě spustit **top**. Uvidíme záhlaví s aktuálním časem a údaji o systému. Pod ním je tabulka procesů uvozená inverzním záhlavím. Tabulka se každých pět sekund aktualizuje. Chod programu můžeme ovlivnit mj. těmito klávesami:

- [**Shift+n**] – třídění procesů podle PID;
- [**Shift+a**] – třídění procesů podle PID od konce;
- [**Shift+p**] – třídění procesů podle zatížení CPU (odhalení zaseknutých procesů);
- [**Shift+m**] – třídění procesů podle objemu zabrané paměti (odhalení viníků swapování);
- [**Shift+t**] – třídění procesů podle spotřebovaného strojového času (odhalení procesů nejvíce zatěžujících systém);
- [**Shift+a**] – třídění procesů podle PID od konce;
- [**m**] – zapnutí nebo vypnutí informací o paměti;
- [**t**] – zapnutí nebo vypnutí souhrnných informací o systému;
- [**h**] – nápověda;
- [**q**] – ukončení programu.

Po zjištění viníka svých problémů nemusíme top opouštět. Po stisknutí [**k**] budeme vyzváni k zadání **PID**, kterému chceme poslat signál (v rétorice minulého dílu by to byl onen pověstný balíček). Po odentrování správného **PID** nám bude nabídnuto zadání signálu. Nezadáme-li žádný, bude odeslán **SIGTERM** (15).

pstree

Minule jsme si předvedli prográmek **ps**, který vypisuje seznam běžících procesů. Umí to na mnoho různých způsobů. Nás dnes bude zajímat ten, kde se vyskytuje sloupeček **PPID**, tedy např. **ps ajx**. Takto získaná tabulka nám ve druhém sloupci ukazuje **PID** – identifikační číslo procesu. První sloupeček představuje **PPID** – **PID** rodičovského procesu.

Kupř. na posledním řádku bychom měli vidět námi spuštěný **ps ajx**. Má své **PID** a podle **PPID** najdeme proces, kterým jsme jej spustili, tedy aktuální shell. Podle **PID** a **PPID** můžeme sestavovat celé rodokmeny, jež by nás zavedly až k jednomu jedinému místu – číslu 1 (kdyby to byla nula, nabízelo by se "One ring to rule them all"). Tento praotec proces je **init** – první program spouštěný kernelem, který má na starosti spuštění systému. Automaticky také adoptuje sirotky – procesy, jejichž rodiče "zemřeli". Láká vás představa znázornit si tuto strukturu graficky? Pokud se spokojíte s pseudografikou terminálu, je tu pro vás **pstree**, tedy něco jako stromečkový **ps**. Lze jej spustit bez parametrů, aby vykreslil standardní podobu stromu. Pomocí přepínačů můžeme tuto podobu změnit. Výběr některých zajímavých je k dispozici v následujícím seznamu:

- **pstree -a** – zobrazení argumentů, s nimiž byly procesy spuštěny;
- **pstree -c** – zabrání shlukování stejných procesů, tj. vypíšou se i ty, které se několikrát za sebou opakují;
- **pstree -G** – na terminálech VT100 bude pro vykreslování čar používat hezčí znaky;
- **pstree -n** – seřadí procesy podle PID (číselně), a ne podle jména;
- **pstree -p** – za každým procesem zobrazí v závorce jeho PID.

kpm

Pokud vám seskupení znaků v textovém terminálu pro znázornění procesového stromečku nestačí, můžete v prostředí X Window využít služeb programu **kpm**. Spustit jej lze např. z emulátoru terminálu nebo pomocí funkce Spustit příkaz (tu vyvoláme též klávesovou zkratkou [**Alt+F2**]) prostým zadáním **kpm** a odentrováním. Objeví se typické KDE okno nahoře s nabídkou. Největší část okna zabírá oblast se seznamem procesů. Dole pak najdeme stavovou lištu a několik ovládacích prvků. V seznamu vidíme mj. název procesu, jeho PID, kolik procent času uživatele a systému spotřebovává, kdo jej spustil a celou syntaxi spuštění. Podle kteréhokoli pole můžeme seznam třídit tak, že na jeho název v záhlaví klikneme. Druhým kliknutím na nadpis provedeme seřazení v obráceném pořadí. Ptáte se, kde je slibovaný stromeček? Odkáži vás na zaškrtávací políčko vlevo dole s příznačným názvem Strom. Pokud jej zaškrtneme, seznam se přeskupí dle rodičovských vazeb procesů. Dále mezi ovládacími prvky v dolní části okna najdeme roletku s filtrem pro vymezení skupiny zobrazených procesů. Tlačítko Obnovit, jak název napovídá, aktualizuje údaje v seznamu. Další tlačítko (Zabít) slouží k ukončování vybraných procesů. Jednotlivé položky seznamu totiž můžeme označovat levým tlačítkem myši. Ve stavové liště vidíme počet spuštěným procesů, volnou paměť a zaplněnost swapu (odkládacího oddílu).

Závěr

Tak jsme zvládli probrat spouštění procesů a dokončit výklad o jejich kontrole a řízení. Problematice proměnných jsem se tentokrát úspěšně vyhnul, ale nebojte. Příště si to bohatě vynahradíme. Dozvíte se nejen, co to je, a jak se to používá. Ukážeme si i některé z nich, např. tu se jménem **PATH** (cesta).
Z dalších klávesových zkratek KDE bychom upozornili na [**Alt+Ctrl+D**], jež zobrazí plochu, [**Alt+Ctrl+L**] uzamkne stanici nebo [**Alt+Ctrl+K**] přepínající rozložení klávesnice. Za povšimnutí stojí také magické čtyřkombinace pro rychlé odhlašování, restartování a vypínání počítače nebo pouhý stisk klávesy **Print Screen** (snímek obrazovky).

4.díl

Tak vás hezky vítám u čtvrtého setkání s Bashem. Umíme toho pořád víc, ale k napsání nějakého užitečného skriptu to zatím nestačí. Posuneme se proto dnes zase kousek kupředu úvodem do problematiky proměnných prostředí. Abychom si je dostatečně vychutnali, naučíme se nejprve ohánět příkazem **echo**.

echo

Echo znamená v angličtině ozvěnu. Tento příkaz vypíše na standardní výstup zadané argumenty, oddělí je mezerou a zakončí znakem nový řádek. Můžeme si tak zakřičet např. **echo Halooo**. Počítač by měl prostě vypsat Halooo. Tak jsme prokázali, že **echo**, tedy ozvěna, funguje. Zkusíme-li zadat více argumentů (slov) současně, vypíše **echo** všechny argumenty – také současně. Rozšiřme naše volání na **echo Volám do lesa.** Ozve se Volám do lesa. Vidíme tedy, že Bash je kompatibilní s příslovím, jak se do lesa volá, tak se z lesa ozývá. Pokud bychom chtěli, aby **echo** chápalo více slov jako jeden argument, můžeme je uzavřít do uvozovek (") nebo apostrofů (,). Ty pak nejsou ve výsledku zobrazeny. Z **echo "Volám do lesa" 'a na pole'** tak vznikne Volám do lesa a na pole. Rozdíl mezi uvozovkami a apostrofy je v tom, že uvozovky dovolují Bashi provádět speciální znaky. Naopak to, co je v apostrofech, bude vypsáno přesně tak, jak to zadáme. Prakticky si to ukážeme dále v souvislosti s proměnnými. Příkaz **echo** je sice velmi jednoduchý, ale i tak má k dispozici několik přepínačů. Jedním z nich je **-n**, který zamezí přidání znaku nový řádek na konec výstupu **echo**. Toho využijeme, chceme-li z více příkazů **echo** poskládat jeden řádek výstupu. Dalším přepínačem je **-e**. Ten způsobí, že **echo** bude rozumět speciálním skupinám znaků, které jsou uvozeny zpětným lomítkem. Některé z nich najdete v poznámce na okraji. Můžeme vyzkoušet malou tabulku zadanou v jednom řádku:

```
echo -e 'Kernel\tVerze\nLinux\t2.6.8'
```

Tabulátory (**\t**) zajistí stejné odsazení druhého sloupce v obou řádcích. Znak nový řádek (**\n**) snad nepotřebuje vysvětlení. Za povšimnutí však stojí, že se speciální znaky zadávají těsně k vypisovanému textu. Poslední přepínač příkazu **echo** je **-E**, který je opakem **-e**. Konkrétně přinutí **echo**, aby nerozlišovalo speciální znaky začínající zpětným lomítkem. Zkusme zadat:

```
echo -E 'Kernel\tVerze\nLinux\t2.6.8'
```

Výsledkem nebude tabulka. Ještě pár otázek k obráceným lomítkům. Pokud bychom v posledním příkladu (ten s **-E**) neuvedli uvozovky, vypadal by výpis ještě malinko jinak. Obrácená lomítka by se v něm neobjevila. Znaky za nimi (**t** a **n**) však ano. To vyplývá z funkce obráceného lomítka mimo uvozovky. V takových situacích zajistí zobrazení bezprostředně následujícího znaku bez jeho speciálního významu. Tak můžeme pomocí příkazu **echo** vypsat uvozovky, apostrof nebo dolar. Zkusme si:

```
echo uvozovky \", apostrof \‘, dolar \$
```

Další přepínače:
\a - výstraha (zvonek)
\b - zpětné mazání
\c - potlačit přebývající konec řádku
\e - znak escape
\f - nová stránka (FF)
\n - nový řádek
\r - návrat vozíku (CR)
\t - horizontální tab
\v - vertikální tab
**** - obrácené lomítko
\0nnn - znak zadaný osmibitovou hodnotou nnn v osmičkové soustavě (nula až tři osmičkové číslice)
\nnn - znak zadaný osmibitovou hodnotou nnn v osmičkové soustavě (jedna až tři osmičkové číslice)
\xHH - znak zadaný osmibitovou hodnotou HH v šestnáctkové soustavě (jedna nebo dvě šestnáctkové číslice)

Proměnné

Operační systémy se již dlouhou dobu neobejdou bez možnosti nastavit určité parametry za chodu. Využívá se k tomu nástroj zvaný proměnná prostředí. Je to něco jako proměnná v matematických rovnicích, ale nemusí mít jen číselnou hodnotu. Hodnoty proměnných prostředí lze snadno zjistit, což z nich dělá dobrý prostředek pro předávání informací mezi běžícími programy. Snadno se dají měnit, čímž lze ovlivňovat chování programů nebo i celého systému za běhu. Proměnné jsou definovány svými jmény, která zpravidla tvoříme velkými písmeny bez diakritiky. Nejprve se podíváme, jak lze proměnné přiřadit hodnotu. Řekněme, že chceme do proměnné **CISLO** uložit hodnotu **5**.
Provedeme to znaménkem "rovná se", takže **CISLO=5**. Abychom předešli nežádoucí interpretaci přiřazované hodnoty ze strany Bashe, můžeme ji obalit uvozovkami, tedy **CISLO="5"**. Nyní by měla naše proměnná obsahovat hodnotu **5**.
Jak si ale hodnotu proměnné ověřit? Použijeme k tomu nám již známý příkaz **echo**. Zápisem **echo CISLO** bychom ale dospěli pouze k výpisu textu **CISLO**. Příkaz **echo** by prostě vypsal to, co jsme mu zadali.
Chceme-li, aby Bash pracoval s hodnotou proměnné, musíme před její jméno zadat znak $. Výpisu hodnoty naší proměnné tak dosáhneme příkazem **echo $CISLO**.
Podobným způsobem můžeme ukládat do proměnných textové řetězce. Např. **MUJTEXT="Nějaký text"**, a potom **echo $MUJTEXT**. Pokud ve vaší konzoli není správně nastavena čeština, používejte klidně texty bez diakritiky.
Nyní můžeme pomocí příkazu **echo** kombinovat textové řetězce s hodnotami proměnných.
Třeba **echo "Moje číslo je $CISLO"**. Nebo **echo "Můj tajný text zní: $MUJTEXT"**.
Spojování řetězců lze využít i při deklaraci proměnných. Mějme **JMENO="Bart"** a **PRIJMENI="Simpson"**.
Potom **CELE="Celé jméno zní $JMENO $PRIJMENI."** a z **echo $CELE** vypadne celá věta s doplněným jménem a zakončená tečkou. Hodnotu jednou naplněné proměnné můžeme změnit jejím doplněním nebo přepsáním.
Přepsání se provede prostě tak, že dané proměnné přiřadíme novou hodnotu. Tak třeba **MUJTEXT="Volám do lesa"**.
Doplnění textu do proměnné uděláme složením s použitím původní hodnoty proměnné a nového obsahu.
Např. **MUJTEXT="$MUJTEXT a na pole"**. Nyní můžeme zkusit **echo $MUJTEXT**. K původnímu obsahu byl doplněn zadaný text.

PATH

Pravděpodobně nejznámější systémovou proměnnou v unixových operačních systémech (a nejen v nich) je **PATH**. Název proměnné tvoří anglické slovo znamenající cesta. Cesta ve významu posloupnosti adresářů v adresářové struktuře našeho systému souborů. Zkusme se podívat, co **PATH** ve vašem systému obsahuje. Když jsem já zadal **echo $PATH**, vypadlo na mne /usr/local/bin: /usr/bin: /bin: /usr/X11R6/bin: /home/milarb/bin. Je to seznam pěti cest nezakončených lomítkem a oddělených od sebe dvojtečkou. Tento seznam prochází operační systém (resp. Bash), když mu zadáme ke spuštění nějaký program pouze jeho jménem bez uvedení cesty k němu. Můj systém by se nejprve díval do /usr/local/bin, pokud by tam program nenašel, zkusil by /usr/bin, pokud ani tam ne, pak /bin, dále /usr/X11R6/bin a nakonec /home/milarb/bin, tedy podadresář **bin** v mém domovském adresáři. Pokud Bash nenajde spustitelný soubor daného jména v žádném z adresářů uvedených v **PATH**, ohlásí nám dobře známé **command not found**. Všimněte si,

že Bash hledá pouze v zadaných adresářích, a nikoli v aktuálním adresáři, a proto je musíme spouštět konstrukcí **./prikaz**. To je jedna z odlišností unixových a dosových systémů. Bystřejší z vás možná napadlo, jak by se tento "problém" dal obejít. Stačilo by přece na konec proměnné PATH přidat řetězec ":.". Ano, fungovalo by to, ale je to výrazné narušení bezpečnosti systému. V historii unixových útoků se vyskytl nejeden trojský kůň schovaný do spustitelného souboru **ls** umístěného v nějakém adresáři, který poslušně vypsal obsah adresáře (samozřejmě kromě sebe sama), a následně vykonal nějaký záškodnický kód. Mnohem méně ohrozíme bezpečnost a upgradujeme atraktivnost našeho systému, pokud do proměnné **PATH** přidáme /usr/games. S použitím dříve nabytých poznatků můžeme suverénně zadat:

```
PATH=$PATH:/usr/games
```

Závěr

Tak tolik pro dnešek k proměnným. Příště v nich ale budeme pokračovat, takže se můžete těšit na pole, kulaté závorky, příkaz export a příklady některých dalších systémových proměnných.
Útok pomocí trojského koně je snadný. Pokud má uživatel v proměnné prostředí **PATH** aktuální adresář (tedy tečku), nic nebrání útočníkovi umístit trojského koně například do adresáře /tmp. Stačí vytvořit program, který přidělí superuživatelská práva útočníkovi a vypíše obsah adresáře, sestavit jej a pojmenovat jako **ls**.
Pokud správce systému spustí příkaz **ls** v adresáři **/tmp**, trojský kůň provede akci (nastaví potřebná práva útočníkovi) a vypíše obsah adresáře, takže správce nic nebude tušit. Sofistikované trojské koně jsou k nerozeznání od původního příkazu (vznikají obvykle jako modifikace originálního programu **ls**) a jsou schopny z výpisu schovat sebe sama.
Od této chvíle by měl jeden z uživatelů neomezená práva, což by si správce jistě nepřál. Proto není vhodné přidávat aktuální adresář do proměnné PATH.

5.díl

Vítám vás u dalšího dílu našeho bashového receptáře. Ani jsme se nenadáli a máme tu jaro. Jako správní pěstitelé otevřeného kódu proto vyrazíme na pole. Potom se z širých lánů půjdeme schovat do bezpečí závorek, odkud budeme sledovat proměny (resp. proměnné) naší úrody, abychom nakonec mohli sklízet úrodu ostře nabroušeným promptem.

Jaké pole?

Nebojte se, nespletl jsem se, tento článek nebyl určen do zemědělského časopisu. Polem se zde zabývat budeme, nikoli však ve smyslu polnohospodářském, nýbrž systémovém. A nebude to ani diskové pole. Půjde o pole proměnných (anglicky **array** či v množném **arrays**). O běžných proměnných jsme se již bavili dříve. Je to určité označení (jako např. ono známé x v matematických rovnicích), za kterým se skrývá určitá hodnota, jež se může v průběhu práce systému měnit, a tím ovlivní jeho chování. Můžeme mít proměnnou kralici, jejíž hodnota bude představovat počet králíků v naší králíkárně. Vedle toho může mít proměnnou husy, kruty, slepice atd. Jejich hodnoty nastavíme rovnítkem (např. **kralici="5"**). Číst hodnotu pak lze třeba příkazem **echo $kralici**. A takhle můžeme definovat zvířectvo, dokud nám bude paměť (a **swap**) stačit. Po nějaké době ale dospěje většina lidí do situace, kdy potřebuje přistupovat k jednotlivým proměnným na základě nějakého klíče nebo kdy jednotlivé proměnné potřebuje ještě dále členit. Tak vezměme třeba **kralici** (jména proměnných neskloňujeme) a chtějme si zvlášť evidovat počet samců a samic. A právě nyní je čas vyrazit na to pole. Na poli je totiž víc místa, abychom si králíci (tedy vlastně králíky) a králice (nebo jak se to správně říká) mohli postavit pěkně do řad. Řady zvířátek můžeme číslovat a, jak už to tak v Linuxu a podobných systémech bývá, číslujeme od nuly. 0 bude třeba angorský, 1 burgundský a 2 kastorex. Tomuto označení položky pole říkáme index (anglicky subscript). Číslo položky zapisujeme do hranatých závorek bezprostředně za jméno pole.

Práce s polem

Hodnotu určité položce pole přiřadíme s použitím hranatých závorek takto: **kralici[0]="3"**. Pozor ale při práci s hodnotou – tam musíme použít ještě složené závorky, takže **echo ${kralici[0]}**. Podobně můžeme přiřadit třeba **kralici[1]="5"** a **kralici[2]="4"**. Co jsme tím získali?
Na první pohled jen tři proměnné se složitěji tvořeným jménem a nutností dávat si pozor na složené závorky. Možná ale změníte názor, když zadáte **echo ${kralici[*]}**. Jedním příkazem tak můžete získat přehled o hodnotách všech položek pole. Tím ovšem výhody pole nekončí. Podobně jako můžeme naráz číst, můžeme naráz zapisovat. Vytvořme pole drubez (0 bude třeba kur, 1 husy, 2 kachny) a zkusme jej naplnit hodnotami. Stačí zapsat **drubez=(14 8 6)**. A můžete zkusit e**cho ${drubez[1]}** nebo **echo ${drubez[0]}** (pro výpis nulté položky lze použít též **echo ${drubez}**). Pokud se pokusíme nadefinovat stejné pole znovu, zmizí jeho původní hodnoty. Dobře si to ukážeme na příkladu **drubez=(16 9)**. Nyní jsme definovali nové hodnoty **$drubez[0]** a **$drubez[1]**. Pokud zkusíme provést **echo ${drubez[2]},** uvidíme, že nic neuvidíme, protože výsledkem bude prázdný řetězec. Novou definicí se totiž celé pole vyčistilo. Nešlo jen o přepsání hodnot dotčených položek pole. Můžeme definovat hromadně jen některé položky pole jako třeba **drubez=([0]=16 [2]=9)**. Můžeme hromadně doplnit další položky k již existujícím. Jak to udělat, když jsme řekli, že nová definice smaže původní obsah? Obdobně, jako při natahování řetězce v běžné proměnné:

drubez=(${drubez[*]} [3]=6 [4]=4). V případě, že nadefinujeme moc velké pole, které už nepotřebujeme, nebo se chceme pole zbavit z jiného důvodu, máme tady **unset**. Syntaxe je jednoduchá: **unset drubez** (nepoužijeme znak dolar, protože pracujeme s proměnnou jako takovou, nikoli s její hodnotou). Stejný efekt by mělo **unset drubez[*]**.
Lze mazat i jednotlivé položky, a to pomocí třeba **unset drubez[1]**. Velmi důležitou, možná nejdůležitější, vlastností pole je možnost adresovat jednotlivé položky pole pomocí jiné proměnné. Toto tvrzení může znít složitě, ale ve skutečnosti jde o jednoduchou věc. Zkuste si:

```
polozka="1" ; echo ${drubez[$polozka]}
```

To je prostě věc, kterou s normálními proměnnými neuděláte. "Mám pro vás jeden tip, který až tak nesouvisí s interpretrem Bash, ačkoli bez něj by byl život v Bashi těžší. Někteří uživatelé totiž nevědí, že u české klávesnice se speciální znaky (@#$%^&) dají napsat pomocí šedé (resp. pravé) klávesy [**Alt**] v kombinací s číslicemi 1 až 0 (případně jinými písmenky). Například znak zavináč se dá v systému X Window napsat pomocí dvou kombinací: [**AltGr+2**] nebo [**AltGr+v**]. V Bashi se tedy dá dost dobře programovat i na české klávesnici je to jen otázka zvyku."

Závorky složené a kulaté

Složené závorky se používají nejen pro oddělení jména proměnné nebo pole od dalších znaků v rámci příkazu. Jak jsme si ukázali ve třetím díle, dokážou také seskupovat několik příkazů do jednoho celku. To se může hodit hned v několika případech - když chceme použít konstrukci, která umí vykonat pouze jeden element, chceme-li najednou přesměrovat výstup z několika příkazů nebo třeba pokud chceme nechat celou skupinu úloh provést na pozadí. Lze tak bez obav zadat:

```
{ sleep 10 ; echo Budíček! ; } &
```

Středník za posledním příkazem je zde nutný, podobně jako mezera za otevírací a před zavírací závorkou. Můžeme použít také závorky kulaté, které mezery ani závěrečný středník nepotřebují. Příkaz by pak mohl vypadat takto:

```
(sleep 10 ; echo Budíček!) &
```

Vedle způsobu zápisu se konstrukce s kulatými a složenými závorkami liší ještě v jednom – předávání hodnot proměnných prostředí. To, co uzavřeme do složených závorek, poběží stále v aktuálním Bashi, takže po skončení těchto operací budou provedené změny stále platit. Zadejme **husy="5"**, potom **{ husy="3" ; echo $husy ; }** a nakonec **echo $husy**.
Na první echo (to v závorkách) by měla být reakce 3, protože jsme v závorkách hodnotu proměnné změnili z 5 na 3. Na druhé echo (za závorkami) bude odpověď zase 3, protože příkazy provedené v závorkách jsou platné pro aktuální shell. Jinak tomu samozřejmě bude v případě závorek kulatých. Zadejme **husy="5"**, potom **(husy="3" ; echo $husy)** a nakonec e**cho $husy**. V okamžiku otevření závorky se spustí nový shell, který provede pouze příkazy v závorce. První **echo** tedy opět vypíše hodnotu 3, kterou nastavujeme uvnitř závorek. Potom se závorka zavře, ukončí se příslušný Bash a s ním zmizí i hodnoty proměnných v něm nastavené. Proto druhý shell vypíše hodnotu 5, kterou jsme v něm nastavili před tím, než byl zavolán druhý Bash.

Systémové proměnné

Hodnoty proměnných, které uvádím kolem článku, získáte příkazem **echo**. Pro výpis všech aktuálních proměnných a jejich hodnot slouží příkaz **env**, který můžeme spustit bez parametrů.
Za zvláštní zmínku stojí, že **BASH_VERSINFO** je pole, které má 6 položek (0 až 5). Obsahuje jednotlivé komponenty označení verze Bashe. Např. pod **BASH_VERSINFO[2]** se skrývá patch level, hodnotu celého pole zobrazíme přirozeně příkazem **echo ${BASH_VERSINFO[*]}**. Takových proměnných je ale více. Jmenujme jenom některé:

BASH_VERSION = verze interpretru Bash
GROUPS = seznam skupin, jichž je současný uživatel členem
HISTSIZE = počet zadaných příkazů, které si Bash pamatuje
HOME = domovský adresář
HOSTNAME = jméno počítače
HOSTTYPE = typ počítače
MAIL = soubor s lokální schránkou
OLDPWD = předchozí pracovní adresář
OSTYPE = typ operačního systému
PWD = aktuální pracovní adresář
RANDOM = náhodné číslo do 0 do 32767
SECONDS = počet sekund od startu shellu
SHELL = určuje výchozí interpretr
TMP = dočasný adresář
USER = jméno uživatele

PS1 – syntaxe promptu

Co to je prompt, jsme si řekli už v prvním díle. Tehdy jsem také slíbil, že vám jednou ukážu, jak jej změnit. Ten čas právě nadešel. Prompt je skupina znaků, kterou v interpretru vídáme asi nejčastěji. Spokojeně si hoví před našim kurzorem a cosi se nám snaží sdělit. Prompt v mém systému mi říká, který uživatel je v daném Bashi přihlášen, na kterém běží Bash počítači (to se hodí, když jsem v různých terminálech přes ssh přihlášen k různým počítačům) a také ve kterém stojím adresáři. Pokud se vám to nelíbí, máte štěstí, protože používáte svobodný software. V případě Bashe je konfigurace promptu hračkou (alespoň pro toho, kdo četl pozorně minulý díl seriálu). Syntaxe promptu je totiž definována v proměnné – konkrétně v proměnné **PS1**. Zkusil jsem proto napsat **echo $PS1** a vypadlo **[\u@\h \W]\$**. To není chyba, tak to má skutečně být. První znak je hranatá závorka, která se mi v promptu objevuje hned na začátku. Následuje **\u**, což je aktuální uživatel. Zavináč se opět přímo vypíše a za ním **\h** – tj. jméno počítače. Následuje mezera a **\W** – aktuální adresář (jen jeho nejnižší úroveň). Pak už jen formální uzavření hranaté závorky a místo ostrého hrotu znak dolaru, za nějž se postaví kurzor. Než začneme zkoušet, uschovejme si původní hodnotu promptu: **PSold=$PS1**. Nejprve malá recese: **PS1="C:/>"**. Protože konfigurujeme Bash změnou hodnoty proměnné, úprava se projeví okamžitě. Tak se nelekněte. Zkusme něco originálnějšího: **PS1="\d, \A \u@\h \w > "**. Některé konstrukce využitelné při stavbě vlastního promptu uvádím na okraji. Jak vidno, dá se s tím vyřádit do sytosti. Komu by to nestačilo, tak podotýkám, že součástí definice může být jiná proměnná! Doporučuji pak dát definici do apostrofů. Např. P**S1='\u@\h $PATH > '**. Zpátky k normálu se vrátíme pomocí **PS1=$PSold**. Je to váš nástroj na orání systému, takže si jej nabruste dle libosti.

Konstrukce promtu:

\d = datum
\h = jméno počítače po první tečku
\H = celé jméno počítače
\t = aktuální čas ve formátu 24, HH:MM:SS
\T = aktuální čas ve formátu 12, HH:MM:SS
\A = aktuální čas ve formátu 24, HH:MM
\u = jméno uživatele
\v = verze Bashe
\V = verze Bashe včetně patch level
\w = pracovní adresář
\W = nejnižší jméno pracovního adresáře

Závěr

Tentokrát to byla poctivá tvrdá polnohospodářská práce. Přeji hodně úspěchů a zábavy s úrodou nových vědomostí. Příště se podíváme na standardní vstupy a výstupy, jejich přesměrovávání a další možnosti komunikace mezi procesy. Již minule jsme se seznámili s proměnnou **PATH**.

6.díl

Minule jsme si ukázali, jak obdělávat v Bashi pole a jak si přizpůsobit "rukojeť" tohoto nástroje (tedy prompt) vlastní potřebě. Nejen na poli je po zimě rušno. Také stavební dělníci mají plné ruce práce. Bydlím na okraji města. Člověk by čekal za oknem romantickou přírodu, šumění větví a zpěv ptáků. Místo toho slyším bagry a vidím samé výkopy. Již druhým rokem se u nás buduje kanalizace. A tak mne napadlo, že se s vámi o tento zážitek podělím a budeme se dnes věnovat rourám. A protože do každé roury něco teče a na konci to teče ven, přidáme k tomu vstupy a výstupy. Jak již jistě víte, v unixových systémech (jakým je i Linux) odvádí většinou práce spousta malých specializovaných programků. Např. při vypalování CD a DVD se používá **mkisofs**, **cdrecord**, **cdrdao**, **cdrwtool**, **cdda2wav** apod. Abychom si nemuseli pamatovat všechny parametry těchto textových příkazů, vznikají nástavby. Nástavbou pro vypalování CD je program, který téměř neví, co to CD je. Jeho úlohou je předložit uživateli hezké prostředí (např. okno s ikonkami, výpisem adresáře, dialog se záložkami pro konfiguraci apod.), pomocí kterého se na pozadí ovládá starý dobrý **cdrecord** a jeho kamarádi. Takovou nástavbou je třeba **k3b** v grafice nebo **Bashburn** v textovém režimu. Programky spouštěné na pozadí se musejí nějakým způsobem domlouvat. To, co vyrobí jeden z nich, musí umět předat tomu dalšímu a tak pořád dál, dokud není úkol splněn. Řízení tohoto procesu je dalším úkolem nástavby nebo prostředí, v němž jsou programy spouštěny. Takovým prostředím je i Bash. Nástrojů pro zprostředkování komunikace mezi procesy má hned několik.

Výstupy

Většina textových GNU nástrojů během své práce nebo na jejím konci zobrazí výsledek této činnosti na terminálu, z něhož byl spuštěn. V tomto případě jde o tzv. standardní výstup. Často ani nevnímáme, že se na terminálu mísí dva různé výstupy – standardní a standardní chybový. Vezměme takový program pro výpis obsahu adresáře, tedy **ls**. Zadám-li **ls /home**, uvidím obsah adresáře **/home**, což je standardní výstup **ls**. Pokud ale udělám překlep a zadám

ls /mohe, uvidím nejspíš hlášku "**ls: /mohe: není souborem ani adresářem**". To je standardní chybový výstup. Jeden příkaz může v rámci jedné operace vygenerovat oba typy výstupů. Velkou výhodou unixového (a tedy i GNU) prostředí je, že oba tyto výstupy umí přesměrovat jinam než do terminálu. Bash k tomuto účelu využívá špičatou závorku doprava. Chceme-li např. vytvořit textový soubor s výpisem obsahu adresáře **/home**, využijeme k tomu příkaz **ls**, jehož výstup přesměrujeme do souboru **seznam.txt**. Provedeme to příkazem **ls /home > seznam.txt**. Výsledek můžeme zkontrolovat pomocí **cat seznam.txt**. Analogicky můžeme zkusit **ls /var > seznam.txt**. Zkusíte-li si soubor prohlédnout teď, zjistíte, že jeho obsah byl přemazán výpisem adresáře **/var**. Co si ale počít, nechceme-li původní soubor přepsat, ale nová data připojit na jeho konec? Pro přesměrování do souboru s připojením (append) využívá Bash dvou špičatých závorek. Vyzkoušejte si následující posloupnost příkazů:

```
echo "*** DOMOVSKE" > seznam.txt
ls /home >> seznam.txt
echo "*** SYSTEMOVE" >> seznam.txt
ls /usr >> seznam.txt
cat seznam.txt
```

Tak jsme dosáhli, že místo toho, aby byl výsledek práce programu (nebo dokonce několika programů) zobrazen na obrazovce, bude uložen do souboru na disk. Může ale nastat situace, kdy výsledek práce programu vůbec nepotřebujeme. Ba naopak by nás jeho výstup na obrazovce obtěžoval. Tehdy můžeme využít speciálního souboru, který nám unixové systémy nabízí, a to **/dev/null**. Ten v systému funguje jako černá díra. Co se tam hodí, to zmizí. Např. **ls /home > /dev/null**. Proč je dobré takový zdánlivý nesmysl někdy dělat, si ukážeme v některém z příštích pokračování. Vraťme se k překlepům, třeba **ls /home /bni** (mělo tam být samozřejmě /bin), a proveďme přesměrování standardního výstupu do souboru: **ls /home /bni > seznam.txt**. Do souboru se vypsal obsah **/home** a na obrazovce se objevilo chybové hlášení "**ls: /bni: není souborem ani adresářem**". Je to proto, že standardní chybový výstup jsme ponechali beze změny. Standardní chybový výstup má interní označení 2 (1 je standardní) a toto číslo je nutné uvést (není-li uvedeno, je to bráno jako 1) před znak přesměrování. Nejlépe to ukáže příklad:

```
ls /home /bni 2> chyby.txt
```

Nyní vidíme na obrazovce obsah **/home** a žádnou chybovou hlášku. Tu si naopak můžeme přečíst v souboru **chyby.txt** (třeba zadáním **cat chyby.txt**).
Můžeme být ovšem nároční a chtít přesměrovat oba výstupy současně. Uděláme to třeba následovně:

```
ls /home /bni > seznam.txt 2> chyby.txt
```

Nyní je standardní výstup v **seznam.txt** a chybový v **chyby.txt**. A na závěr případ, kdy chceme přesměrovat oba výstupy do téhož souboru. Abychom jméno (případně i s celou cestou) nemuseli psát dvakrát, stačí přesměrovat jeden výstup a druhému říci "tak, kam míří ten první:

```
ls /home /bni > seznam.txt 2>&1
```

Už i toto je ale zastaralá a příliš dlouhá forma. Dnes stačí: **ls /home /bni &> seznam.txt.**

Vstup

Podobně jako lze přesměrovat výstupy, je to možné udělat i se vstupem. Slouží k tomu špičatá závorka doleva.
S touto funkcí se nesetkáváme tak často jako s přesměrováním výstupu, protože většina příkazů je připravena převzít vstupní soubor jako parametr. Např. si vezměme takový **cat**. Když zadáme prostě **cat** bez parametrů, skočí nám kurzor na nový řádek a čeká na (standardní) vstup – z klávesnice. Zadáme-li určitý text zakončený klávesou Enter, provede s ním **cat** to, k čemu je předurčen – pošle jej na standardní výstup. Tento režim ukončíme kombinací kláves [**Ctrl+d**].
Většinou ale **cat** používáme k výpisu obsahu souboru, tedy nikoli standardního vstupu. Po zadání **cat list.txt** uvidíme na standardním výstupu obsah souboru **list.txt**. Stejný efekt by ovšem mělo **cat < list.txt**
Ještě stále najdeme pár programů, které pro práci se souborem vyžadují přesměrování vstupu. Patří mezi ně i **cpio**. Pokud chceme např. rozbalit archiv se jménem **archiv.cpio**, musíme zadat **cpio -i -d < archiv.cpio** (parametr **-i** značí rozbalit, **-d** vytvořit potřebné adresáře).

Potrubní pošta aneb Roury

Roura je mocný nástroj. Jde o spojení obou typů přesměrování. Přesněji řečeno přesměrovává standardní výstup jednoho programu na standardní vstup druhého programu. Používá se k tomu znak ‚**|**' (známý též jako svislítko). Např. **cat** má přepínač **-n,** s jehož pomocí čísluje řádky svého výstupu. Řekněme, že bychom chtěli ve výpisu adresáře jednotlivé soubory očíslovat. Není proto nic snadnějšího než zadat:

```
ls | cat -n
```

Nebo něco zábavnějšího:

```
ls | rev
```

Problémem bývají pro mnoho uživatelů dlouhé výpisy z některých programů. Již jsme si ukázali, jak je přesměrovat do souboru, který pak můžete otevřít a prozkoumávat třeba ve svém oblíbeném editoru. Proč se ale zdržovat a zaplácávat disk dočasnými soubory, když tuto práci zvládneme v příkazovém řádku? Na prohlížení dlouhých souborů je třeba vhodný less:

`ls -l /etc | less` (program opustíte klávesou **q**)

Programem často využívaným v této potrubní poště je **grep** – nástroj na vyhledávání řetězců. Ten zajistí, že budou vypsány jen ty řádky, které obsahují hledaný řetězec. Hledáme např. všechny spuštěné instance Bashe. K výpisu běžících programů slouží ps:

```
ps aux | grep bash
```

Příznivci DOSu by mohli namítnout, že takovou rouru měl jejich oblíbený systém již dávno. Ano, byla tam, ale velmi omezená. Co je totiž na unixové rouře pozoruhodné, je možnost jejího spojování do dlouhého potrubí. Řekněme, že bychom chtěli výše vypsaný seznam spuštěných Bashů (pokud jste měli jen jeden, tak si jich bokem z testovacích důvodů pár spusťte) abecedně setřídit (v prvním poli je uživatel, který daný proces spustil):

```
ps aux | grep bash | sort
```

To stále není vše. Můžeme pokračovat ve stavbě dále. Dejme tomu, že z takto dosaženého výpisu budu chtít zobrazit jen tu část řádku od výpisu terminálu, na kterém Bash běží (tj. od 37. znaku včetně). Zadám jednoduše:

```
ps aux | grep bash | sort | cut -c 37-
```

A tak bych mohl pokračovat dál a dál. Očíslovat řádky (**cat -n**), spočítat znaky (**wc**), obrátit pořadí znaků v řádku (**rev**), ... Délka potrubí není omezena a fantazii se meze nekladou. Vraťme se ale ještě k tématu prostého přesměrování výstupu. Zkusme:

```
ls /home /sibn | cat -n
```

Řádek s chybou není číslován. Neprošel totiž rourou. Byl vypsán přímo programem **ls**. Co kdybychom ovšem chtěli do roury poslat i chybový výstup? Půjde to takhle:

```
ls /home /sibn 2>&1 | cat -n
```

Ještě se zmiňme o možnosti přesměrovat výstup do souboru a zároveň jej ponechat na obrazovce. Slouží k tomu příkaz **tee**, do nějž výstup předchozího procesu pošleme rourou. Příklad: **ls /boot | tee seznam.txt** a potom **cat seznam.txt**. Uvidíme dvakrát totéž.

Závěr

Doufám, že se v tom potrubním systému neztratíte. Je to vskutku velmi silný nástroj. Znaky ‚**<**' **a** ‚**>**', které Bash používá k přesměrování vstupu a výstupu, nazývám v textu špičatými závorkami tak, jak jsem se to naučil ve škole. Není ale neobvyklé slyšet o nich jako o zobáčcích či šipkách doprava a doleva. Existují i zřejmě nespisovné výrazy většítko (od "větší než") a menšítko ("menší než"). Velmi originální je ovšem "šipka do Ruska" a "šipka do Německa". U přesměrování obou výstupů na stejné místo pomocí **2>&1** je třeba dbát na to, aby tento řetězec stál až za přesměrováním standardního výstupu. Jinak by byl totiž namířen tam, kam v tu chvíli míří standardní výstup, tj. na obrazovku.

Vedle vstupů a výstupů popsaných v článku existují ještě další mechanismy přesměrování, jako např. "Here Documents" a "Here Strings". Bližší informace o nich lze nalézt v manuálové stránce (**man bash**). Stránkovač **less** je velice mocný a poskytuje zkratkové klávesy z editoru **vi**. Například hledání regulárních výrazů pomocí lomítka je velmi praktické. Alternativami k programu **less** jsou programy **more** a **most**. Program **more** je nejjednodušší variantou a je velmi podobný tomu z operačního systému MS-DOS. Naopak **most** je vysoce sofistikovaný a poskytuje vysoký komfort.

7.díl

Pravidelné čtenáře vítám u již sedmého pokračování seriálu o zřejmě nejrozšířenějším textovém příkazovém interpretru, o Bashi. Minule jsme si pověděli o přesměrování výstupu a rourách. Dnes toto téma uzavřeme tím, že si představíme pojmenované roury. Potom si řekneme, co jsou to aliasy. Čím dál častěji slýchám, že bychom se v tomto seriálu měli posunout ke skriptování, tedy k programování pomocí Bashe. To je samozřejmě cílem celého mého snažení. Nejprve ale bylo potřeba vysvětlit si některé základní principy práce s Bashem a procvičit si je.

Pojmenované roury

Roury, o kterých jsme mluvili v minulých odstavcích, jsou jako nestabilní červí díry. Jeden průlet a díra zmizí. Co ale kdybychom chtěli nějakou rouru využívat opakovaně? Linux nám k tomu nabízí mechanismus zvaný pojmenovaná roura (**named pipe**).Pojmenovaná roura je speciálním typem souboru existujícího v souborovém systému. K jejímu vytvoření slouží program **mkfifo**. Syntaxe je jednoduchá. Stačí zadat jen její jméno, např. **mkfifo roura**. Že nejde o běžný soubor, si můžeme ověřit příkazem **ls -l roura**. Na začátku bloku s přístupovými právy vidíme "**p**" – pipe. Nyní můžeme do roury něco poslat, třeba přesný čas: **date > roura**. Nelekejte se, pokud se neobjevil prompt. To je v pořádku. Proces čeká na vyprázdnění roury. Spusťme tedy jinou konzoli nebo terminál, přejděme do adresáře, kde leží naše pojmenovaná roura a zadejme třeba **cat roura**. Ve druhém Bashi budou vypsány příslušné údaje a v prvním se objeví prompt. Funguje to i obráceně. Můžeme nechat nějaký proces číst z prázdné roury (**cat roura**). Ten bude čekat až do doby, kdy se roura zaplní. To provedeme třeba informacemi o našem systému (**uname -a > roura**) na jiném terminálu. Až nyní **cat** svou práci dokončí. Tohoto chování lze využít v mnoha situacích. Dostaneme se k nim, až budeme tvořit skripty.

Alias

Možná jste se již setkali s příkazem **ll**. Bash jej vykonává na většině systémů (mimo třeba Debianu a Slackwaru). Tento příkaz však nereprezentuje žádný spustitelný soubor na disku ani interní příkaz Bashe. Kde se tedy bere? Je to alias. Ve skutečnosti jde o zkratku zápisu **ls -l**. Říkáte si, že je v těch příkazech pěkný ... ? Nelekejte se, není to zas takový nepořádek. Pro všechny proveditelné příkazy funguje doplňování pomocí [**tab**]. Spustitelné soubory najdeme v adresářích zapsaných do proměnné PATH (**echo $PATH**). Interní funkce Bashe v manuálové stránce (**man bash**). No a seznam aliasů vyvoláte příkazem **alias** (interní příkaz Bashe). Příkaz zapsaný bez parametrů vyvolá seznam přidělených přezdívek. V seznamu představuje každý řádek definici jednoho aliasu. Vlevo od rovnítka stojí přezdívka. Vpravo, většinou v apostrofech, příkaz nebo skupina příkazů, které se provedou po zadání přezdívky jakožto příkazu pro Bash. Výpis mých aliasů je tento:

```
alias l.='ls -d .* --color=tty'
alias ll='ls -l –color=tty'
alias ls='ls –color=tty'
alias vi='vim'
```

Tak vidíme, že **ll** není jen zkratka **ls -l**, ale přidána je ještě podpora barev pomocí **–color=tty**. Nebo zde zjišťujeme, že zadáním **vi** ve skutečnosti spouštíme program **vim**. Zajímavý je příkaz **l.**, který vypíše všechny soubory se jménem začínajícím tečkou. Ve třetím řádku je příklad toho, že alias může přepsat stejnojmenný příkaz. Pokud nám třeba vadí, že **cp** se pořád dokola ptá, zda může přepsat existující cílový soubor, můžeme nadefinovat **alias cp='cp -f'**. Pozor ovšem na nebezpečnost tohoto mechanismu. Dal by se dost dobře zneužít pro trojské koně. Podobně jako pojmenovanou rouru poznáme mezi soubory pomocí příznaku "**p**", tak symbolické odkazy mají "**l**" (link), adresáře "**d**" (directory), znaková zařízení **"c"** (character), bloková zařízení "**b**" (block) apod. Vřele vám doporučuji udržovat si v pojmenovaných rourách pořádek. Neměli bychom proto zapomínat na úklid nepotřebného potrubí. Pojmenovanou rouru smažeme podobně
jako běžný soubor: **rm roura**. Věřte, že problémy s těmito rourami vám nepomůže vyřešit ani hodně dobrý instalatér. Pod pojmem alias najdeme ve slovníku význam "falešné jméno" nebo "přezdívka". Při označování osob má stejný význam jako "nick name" nebo zkráceně "nick". S pojmem "a**lias**" se v unixovém systému setkáváme často. V oblasti elektronické pošty značí více jmen pro jeden poštovní účet. U WWW serveru se používá pro virtuální domény. V **/etc/hosts** znamenají více jmen pro jeden počítač. V **/etc/modprobe.conf** zase přiřazují jména zařízení. Je dobré si proto dávat pozor, v jaké souvislosti o aliasu mluvíme. Parametr **-d** v definici aliasu **l**. je velmi užitečný. Samotné **ls .*** by vedlo k nechtěným výsledkům. Výpis všeho začínajícího na tečku by zobrazil obsah adresářů začínajících tečkou, tedy včetně obsahu .., tj. nadřazeného adresáře. Zmíněné **-d** tomu zamezí, neboť přikazuje vypsat pouze informace o adresářích a ne jejich obsah.

8.díl

A je tady revoluční osmý díl. Dnes se přeneseme ze světa zadávání jednotlivých příkazů našemu oblíbenému interpretru Bash do světa programátorů. Nelekejte se, neopouštíme Bash, jen si ukážeme, jak pro něj tvořit skripty. Co jsou to skripty? Je to prostě jen seřazený seznam příkazů, které se mají vykonat. Takový nákupní seznam nebo výrobní plán. Předpokládejme, že nějakou posloupnost příkazů používáte často v úplně stejné nebo jen malinko odlišné podobě. Otravuje vás ji stále hledat v historii nebo ji chcete využívat na více počítačích či spouštět pravidelně např. pomocí programu **cron**? Není nic jednoduššího než si tuto posloupnost uložit do textového souboru – do skriptu. Skripty slouží k jednoduchým věcem a pro většinu věcí, které si běžný uživatel má čas sám napsat, to stačí. Navíc má skriptování i své velké výhody. Takový skript nemusíte kompilovat a spustíte jej na jakémkoli počítači (PC, Mac, Atari, Amiga, Sun, Sgi, ...) a operačním systému, kde běží Bash (Linux, BSD, Windows, MiNT, AIX, Solaris, …).

Formát souboru

Skripty ukládáme do standardních textových souborů, tedy nikoli dokumentů typu OpenOffice.org. Pro editaci doporučuji textové editory, nikoli textové procesory, takže žádné OOo, KWord či Abiword. Nezáleží v celku na tom, zda zvolíte konzolový (vi, emacs, joe, pico, ...) či GUI (nedit, kedit, kwrite, kate, ...) textový editor nebo dokonce vývojový nástroj (Quanta+). Doporučuji používat režim unixových konců řádků (tj. pouze znak LF) a kódování češtiny, které máte nastaveno v textové konzoli (většinou **ISO 8859-2** nebo **UTF-8**). První řádek souboru by měl obsahovat právě tuto posloupnost znaků:

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

Znak křížek (též "mřížka" nebo dokonce jsem slyšel i verzi "vězení" příp. "kanál") říká interpretru, že od tohoto místa až do konce řádku je to poznámka a tudíž si textu tam uvedeného nemá všímat. Naopak vykřičník zde slouží jako uvození informace o tom, kde najde systém interpretr pro spuštění daného skriptu.

Ukázka skriptu

Řekněme, že máte často potřebu zjistit stav svého systému. Používáte **uname** (identifikace systému), **uptime** (jak dlouho systém běží) a **ping** (test, zda běží server 192.168.0.1 a internet – např. www.linux.cz). Skript by mohl vypadat takto:

```
#!/bin/bash
# Skript pro testování stavu systému
uname -a
uptime
ping -c 3 192.168.0.1
ping -c 3 www.linux.cz
```

Způsob spouštění

Tím, že text někam vepíšeme a uložíme jej do souboru (pojmenujme jej "stav.sh") jsme ještě nedostali plnohodnotný skript. Je to textový soubor obsahující příkazy Bashe, to ano. Nejde ovšem přímo spustit. Pokud jste nedočkaví, můžete jej spustit nepřímo – zadejte příkaz **bash stav.sh**, čímž spustíme nový Bash s parametrem v podobě našeho skriptu. Mnohem pohodlnější by byla možnost udělat z našeho textíku spustitelný soubor. To ale v unixovém systému není problém. Stačí nastavit přístupová práva, např.:

```
chmod a+x stav.sh
```

I tak ale budeme muset zadat vždy cestu k souboru. Máme ho totiž uložen někde v domovském adresáři, který (většinou) není předurčen (není v proměnné **PATH**) k hostování spustitelných souborů. Přesto si již můžeme činnost skriptu vyzkoušet, např. standardní konstrukcí **./stav.sh**. Pokud bychom chtěli skript spouštět odkudkoli prostým zadáním jeho jména, musíme jej nakopírovat do některé ze systémových cest (uvedených v proměnné **PATH**). Takovým místem bývá podadresář **bin** v domovském adresáři každého uživatele. Pokud neexistuje, vytvořte si jej a skript tam přesuňte
(**mv stav.sh ~/bin/**). Pokud byste chtěli, aby byl skript dostupný všem uživatelům systému, musíte jej umístit do veřejně přístupného systémového adresáře (např. **/usr/local/bin**). K tomu ale potřebujete oprávnění uživatele **root**.

Závěr

Tak, a úvod do skriptování máme za sebou. Pro dnešek toho necháme, ať si to můžete do příště zkoušet. Věci zde probrané budu považovat za samozřejmé a již se k nim nebudu vracet. Od dalšího dílu se vrhneme na věci, které se ve skriptech dají použít mnohem efektivněji než na příkazové řádce (podmínky, čtení z klávesnice, cykly, …). V našem případě je interpretrem Bash, jehož spustitelným souborem je **/bin/bash**. Pokud byste ovšem psali skript třeba v Perlu, byla by tam cesta k němu, tj. např. **#!/usr/bin/perl**.
Zda máme v seznamu cest podadresář **bin** svého domovského adresáře, zjistíme jednoduše vypsáním obsahu proměnné **$PATH**, tedy **echo $PATH**. Pokud tam není, můžeme to napravit zadáním:

```
PATH=$PATH:$HOME/bin
```

a potom

```
export PATH
```

9.díl

Posledně jsme si ukázali, jak vytvořit jednoduchoučký skript, kam jej uložit a jak spustit. Již tedy víme, jak zapsat často vkládanou posloupnost příkazů do textového souboru a vyvolat ji jednoduše zapsáním jména skriptu. To sice může někomu stačit, my ale budeme náročnější. Skriptování přináší možnosti, kterých na příkazovém řádku dosáhnout vůbec nelze nebo jen velmi těžko. Příkazy ve skriptu totiž nemusejí být otrocky vykonány od prvního do posledního, každý právě jednou. Některé části mohou být provedeny vícekrát, jiné jen za určitých podmínek, v průběhu provádění lze volat další skripty, chování lze ovlivňovat proměnnými.

cyklus for

Dnes pokročíme tím, že se budeme pohybovat v kruhu. Přesněji řečeno v cyklu. To je právě způsob, jak nechat určitou část skriptu provést vícekrát, třeba s různými parametry. Bash zná několik druhů cyklů. My začneme zřejmě nejjednodušším z nich – **for**. Ten možná mnozí z vás znají, protože se v různých obměnách vyskytuje ve většině počítačových jazyků – od C přes Basic až po PHP. Za klíčovým slovem **for** je nutné uvést parametry cyklu. Bash za tímto účelem nabízí dva typy syntaxe s různou funkčností. Na další řádek nebo několik řádků uvedeme všechny příkazy, které se mají v průběhu každého cyklu vykonat, přičemž před prvním z nich musí být uvedeno slovo **do** (anglicky "udělej, proveď"). Seznam příkazů v cyklu zakončíme dalším klíčovým slovem – **done** (anglicky "uděláno, provedeno"). Popis začneme syntaxí, která je možná jednodušší, ale méně používaná. Vychází ze tří aritmetických výrazů a ukážeme si ji na jednoduchém příkladu:

```
for (( a=1 ; $a-4 ; a=$a+1 ))
do echo $a
done
```

Smyčka začíná samozřejmě příkazem **for**. Na druhém řádku je za do příkaz **echo**, který vytiskne hodnotu proměnné **a**. Slovem **done** smyčka končí. Hned za **for** vidíme pravidla cyklu uzavřená do dvou kulatých závorek. Taková konstrukce v Bashi předznamenává vykonávání aritmetických operací.
První z nich přiřazuje proměnné a hodnotu **1**. Provede se pouze na začátku prvního průchodu. Druhý výraz slouží k rozhodování, zda se má smyčka vykonat. K vykonání dojde, pokud bude mít zde uvedený výraz hodnotu různou od nuly. V prvním průchodu je **a=1**, takže **a-4** není nula. Před každým dalším průchodem ale bude proveden třetí výraz, kde se hodnota a zvýší vždy o **1**. Ve druhém průchodu tak nabude hodnoty **2** a ve třetím **3**. Po skončení třetího průchodu bude opět aplikován třetí výraz a hodnota a se tak zvýší na čtyři. Nyní je ovšem výsledkem druhého výrazu (**a-4**) nula, takže čtvrtý průchod smyčkou již neproběhne.
Pokud zapíšete výše uvedenou smyčku do skriptu, uvidíte po jeho spuštění číslice 1, 2 a 3, což je důkazem, že proběhly právě tři průchody smyčkou s těmito hodnotami proměnné **a**.
Před spuštěním smyčky je dobré si podmínky ještě jednou promyslet, neboť se může snadno stát, že se druhá podmínka jaksi netrefí do nuly a smyčka tak poběží do nekonečna. Pozorní čtenáři si samozřejmě pro takový případ z prvního dílu pamatují chvat první pomoci [**Ctrl+c**]. Další důležitou věcí, na kterou bych měl upozornit, je, že Bash se většinou nekamarádí s desetinnou čárkou, takže nedoporučuji používání desetinných čísel, dělení apod.

for in

Další syntaxí smyčky je výraz **for in**. Ta postupně přiřazuje určené proměnné hodnoty ze zadaného seznamu. Seznam může nabývat nejrůznějších podob. Začněme opět příkladem:

```
for a in A B C D
do echo $a
done
```

Smyčka proměnné a přiřadí postupně hodnoty A, B, C a D. V každém průchodu jednu v pořadí, jak jsou zadány. Smyčka proběhne právě tolikrát, kolik hodnot seznam obsahuje. Výsledkem bude vypsání písmen A, B, C, D, každé na samostatném řádku (příkaz **echo** totiž defaultně vkládá konec řádku na konec svého výstupu). Jako seznam lze zadat přímo jména souborů, a to včetně zástupných znaků. Toho lze velmi dobře využít při hromadném zpracování více souborů programem, který umí v parametrech přijmout pouze jeden vstupní soubor. Příkladem budiž program **recode**, který mění kódování znaků v textových souborech. Mějme adresář obsahující neurčité množství souborů se jménem končícím
na .txt, které jsou v kódovaní CP1250 s konci řádků CRLF. Chceme-li z nich udělat ISO-8859-2 soubory s unixovými konci řádků, stačí se přesunout do onoho adresáře a spustit skript s obsahem:

```
mkdir ISO
for a in *.txt
do echo Konvertuje se $a ...
recode 1250..l2 $a > ISO/$a
done
```

Na prvním řádku, který není součástí smyčky, takže se provede jen jednou, vytvoříme adresář ISO. V prvním řádku smyčky vypíše **echo** informaci o tom, který soubor se zpracovává. Dále **recode** provede konverzi původního souboru a výsledek uloží (díky přesměrování výstupu) pod stejným jménem do adresáře ISO. Na konci bychom tak měli mít adresář ISO zaplněn všemi soubory *.txt z aktuálního adresáře převedenými do ISO-8859-2. Seznam může být také výsledkem činnosti nějakého příkazu. Ten musí být uzavřen do závorek uvozených znakem **$**. Zde se často využívá **seq**, který generuje číselné řady dle zadaných pravidel. Mějme skupinu obrázků na vzdáleném serveru www.masinky.cz/ pod jmény obr1.jpg až obr9.jpg, které chceme stáhnout. K tomu využijeme program **wget**.

```
for a in $( seq 9 )
do wget http://www.masinky.cz/obr${a}.jpg
done
```

Po provedení skriptu s touto smyčkou by se v pracovním adresáři mělo objevit 9 stažených obrázků (samozřejmě pokud máte funkční připojení k internetu a cílový server i soubory na něm existují). Při stahování většího objemu dat se hodí takový skript spouštět s časovým zpožděním, tedy v době, kdy spímespíme a linky jsou méně vytížené. Zde by se mohly hodit příkazy **sleep** (viz 1. díl seriálu) nebo **at** (viz **man at**).
Další možností je použít jako seznam proměnnou, která obsahuje několik skupin znaků oddělených mezerou nebo tabulátorem. To je tak triviální, že to nebudeme ani prezentovat na příkladu. Zajímavou možností, která za ukázku stojí, je využití pole (viz 5. díl seriálu) jako seznamu. Mějme pole lodicky definované zápisem:
lodicky=(člun plachetnice parník). Komentovaný přehled získáme smyčkou:

```
for a in ${lodicky[*]}
do echo Součástí flotily je $a
done
```

Smyčka v jednom řádku

Ve skriptech se často používají dlouhé smyčky, které obsahují mnoho jiných příkazů, podmínek nebo i vnořených smyček. Proto je nutné rozepsat jejich jednotlivé komponenty do více řádků, jak jsme si ukázali na příkladech. Někdy ovšem chceme jen opakovaně provést jeden příkaz s různými argumenty. Představme si situaci, kdy nám na nějakém serveru vzniklo v určitém adresáři sto tisíc souborů se jmény soubor000000.txt až soubor100000.txt. My je chceme smazat a tak zadáme **rm soubor*.txt** a ouvej. Odpovědí nám bude chybová hláška: **bash: /bin/rm: Příliš dlouhý seznam argumentů.** Z toho vidíme, že Bash neměl problém nahradit konstrukci soubor*.txt sty tisíci odpovídajících výsledků. Program **rm** ovšem nebyl schopen takovou várku zpracovat. Adresář nemůžeme smazat celý, protože jsou v něm i jiné důležité soubory. Pomůže nám jednoduchá smyčka zapsaná do jednoho řádku s použitím středníků:

```
for a in soubor*.txt ; do rm -f $a ; done
```

Závěr

Tak jsme dnes při sezení u počítače konečně udělali také něco pro zdraví – stali jsme se cyklisty. Vedle cyklů **for** nabízí Bash ještě další dva: **while** a **until**. Ty se provádějí na základě platnosti resp. neplatnosti zadané podmínky. Proto si tyto

typy smyček představíme až po té, kdy probereme podmínky. A na podmínky se vrhneme hned příště. Příkaz **seq** slouží ke generování číselných řad. Přijímá jeden až tři parametry. Pokud dostane jen jeden, počítá od jedničky po jedné až do zadané hodnoty. Má-li parametry dva, počítá od prvního po jedné ke druhému. Jsou-li tři, potom se první zvyšuje o hodnotu druhého (je-li druhý záporný, snižuje se), než dosáhne hodnoty třetího.
Program **wget** je zajímavý nástroj ke stahování souborů z internetu. Dokáže obsloužit protokoly HTTP a FTP. Při chybě sám inicializuje opakování a dovede navázat na přerušené stahování. Díky rekurzivnímu stahování umí vytvářet místní kopie vzdálených adresářových struktur. Pro bližší informace doporučuji **man wget**.

10.díl

Dnes si ukážeme podmínky. Je to další programátorská konstrukce, která umožňuje větvení programu, resp. skriptu, který jsme se již v tomto seriálu naučili vytvářet (viz 8. díl). Podmínky nám umožňují provedení určité části skriptu pouze v případě, kdy nastanou určité okolnosti.

Návratový kód

Aby program (skript) mohl být něčím víc, než jen seznamem po sobě prováděných příkazů, musejí si tyto být schopny mezi sebou předávat nějaké zprávy. K tomu se hodí proměnné, o kterých jsem už také psal (viz 4. a 5. díl).
Bash používá jednu takovou s názvem **PIPESTATUS** a ukládá do ní zprávu o výsledku práce posledního provedeného příkazu. Tato zpráva má podobu jednobytového čísla (0-255) a nazývá se návratový kód. Zpravidla se nástroje GNU a interní příkazy Bashe chovají tak, že návratový kód 0 znamená úspěšné provedení operace a nenulový návratový kód chybu. V příkladu se budeme snažit vypsat obsah adresářů – jednoho existujícího a jednoho neexistujícího.

Znak **$** zastupuje **prompt**:

```
$ ls /home
bohdan pavel petr

$ echo $PIPESTATUS
0

$ ls /mohe
ls: /mohe: není souborem ani adresářem

$ echo $PIPESTATUS
1
```

Konstrukce || a &&

S ampersandem (znak **&**) jsme se seznámili už ve 2. díle, kdy jsme s jeho pomocí spouštěli úlohy na pozadí.
Se svislítkem (znak **|**) jsme zase v 6. díle vytvářeli nepojmenované roury. Pokud ale najdete ve skriptu tyto znaky zdvojené, znamená to něco úplně jiného – jde o jednoduché podmínkové konstrukce. Před dvojicí znaků i za ní se nachází příkaz. Druhý příkaz se však provede v závislosti na hodnotě návratového kódu příkazu prvního.
příkaz1 && příkaz2 – příkaz2 bude proveden pouze v případě, že příkaz1 skončí s návratovým kódem 0.
To znamená, že příkaz2 se provede, pokud příkaz1 skončí úspěchem.
V následujícím příkladu si necháme v závislosti na výsledku operace zobrazit veselou emotikonu:

```
$ ls /home && echo ':-)'
bohdan pavel petr
:-)

$ ls /mohe && echo ':-)'
ls: /mohe: není souborem ani adresářem
```

V prvním případě proběhl příkaz správně, a proto se emotikona zobrazila. Ve druhém případě tomu tak nebylo. A nyní ke svislítkům: **příkaz1 || příkaz2** – příkaz2 bude proveden pouze v případě, že příkaz1 skončí s návratovým kódem různým od nuly. Druhý příkaz tedy bude proveden za předpokladu, že ten první skončí nezdarem. Vyzkoušíme stejný příklad, avšak v obráceném gardu a se smutnou emotikonkou:

```
$ ls /home || echo ':-('
bohdan pavel petr
```

```
$ ls /mohe || echo ':-('
ls: /mohe: není souborem ani adresářem
:-(
```

Emotikona se ukázala ve druhém případě, kdy je název adresáře zadán chybně. Mohli bychom chtít, aby nám emotikona nahradila (nikoli doplnila) chybovou hlášku, a proto by bylo vhodné se jí zbavit. K tomu nám poslouží přesměrování výstupu (viz 6. díl):

```
$ ls /mohe 2>/dev/null || echo ':-('
:-(
```

Obě konstrukce můžeme kombinovat na jednom řádku. Takže třeba pokud skončí zdárně příkaz1, provede se příkaz2, pokud ne, provede se příkaz3. V našem případě by to vypadalo následovně:

```
$ ls /home 2>/dev/null && echo ':-)' || echo ':-('
deti milarb petr test
:-)

$ ls /mohe 2>/dev/null && echo ':-)' || echo ':-('
:-(
```

Nutno podotknout, že příkaz1 může mít mezi svými argumenty proměnné, takže jeho výsledek není na první pohled tak zřejmý jako ve výše uvedených příkladech. Na závěr si proto ukážeme kraťoučký skript, který ve smyčce **for** (viz 9. díl) přiděluje proměnné **ADRESAR** různé hodnoty. Ty se pak příkaz vyzdobený konstrukcemi **&&** a **||** pokusí vypsat.

```
for ADRESAR in home opt user var; do
echo "Adresář /${ADRESAR}:"
ls /${ADRESAR} 2>/dev/null && echo ':-)' || echo ':-('
echo
done
```

Testy a hranaté závorky

Zatím jsme si předvedli, že v bashovém skriptu můžeme nechat provést jinou operaci, pokud nějaký příkaz dopadne pozitivně, a jinou, když se nezdaří. Co ale kritéria jako existence určitého souboru, hodnota proměnné nebo výsledek výrazu? I ty lze pro větvení použít, a to pomocí stejného mechanismu – návratové hodnoty. Existuje totiž specializovaný příkaz pro vyhodnocování těchto situací a jmenuje se **test**. Test vrací (stejně jako jiné příkazy) hodnotu **0**, pokud vše proběhlo v pořádku, tj. vyhodnocovaná podmínka je platná. Hodnotu **1** vrací tehdy, když podmínka neplatí. Vzniká tak trochu paradoxní situace, kdy pro Bash je **0** pravda a **1** nepravda. S tím si ale nemusíme příliš lámat hlavu.
Dobré může být vědět, že když dojde k chybě (např. špatná syntaxe), vrací test návratový kód **2**.
Ukažme si funkci příkazu test na příkladu. Budeme zjišťovat existenci souboru /etc/passwd. K tomu u příkazu test slouží přepínač **-e**. Výsledek ověříme vypsáním hodnoty proměnné **PIPESTATUS**. Potom zkusíme totéž se souborem /var/passwd, který většinou neexistuje:

```
$ test -e /etc/passwd
$ echo $PIPESTATUS
0

$ test -e /var/passwd
$ echo $PIPESTATUS
1
```

S příkazem test se můžeme velmi často setkat v dosti zvláštní podobě – otevírací hranaté závorky. Pokud voláme test jako závorku, musíme ji po vypsání hodnocené podmínky zase uzavřít. Za otevírací i před uzavírací závorkou doporučuji dělat mezeru.

Příklad:

```
$ [ -e /etc/passwd ]
```

Ve spojení s ampersandy to může vypadat následovně:

```
$ [ -e /etc/passwd ] && echo 'DB uživatelů existuje'
```

Obdobně dobře je test vybaven i pro práci s proměnnými (řetězci) a výrazy. Nejčastější použití je asi při srovnání hodnoty proměnné s nějakým řetězcem. Např chceme zjistit, zda se proměnná licence rovná řetězci "**GPL**": **["$licence" = "GPL"]**. Všimněte si, že kolem rovnítka jsou mezery. Kdyby tam nebyly, šlo by o přiřazení hodnoty, nikoli

o porovnávání. Obě porovnávané hodnoty také doporučuji uzavírat do uvozovek. Vyhneme se tak chybné interpretaci prázdných řetězců a řetězců s více slovy. Pokud chcete zjistit, zda se řetězce nerovnají, stačí těsně před rovnítko přidat vykřičník(**["$licence" != "GPL"]**). Když vás zajímá, zda proměnná existuje a není prázdná, stačí zadat [**"$licence"**]. Chcete-li testovat, zda neexistuje (resp. je prázdná), pomůže vám [**-z "$licence"**]. Test dokáže s podmínkami provádět také logické operace. Logickému A (AND) odpovídá přepínač **-a**. Chceme-li smazat soubor jen tehdy, když máme práva zápisu a proměnná **SMAZAT** má hodnotu ANO, zadáme:

```
[ -w soubor -a "$SMAZAT" = "ANO" ] && rm soubor
```

K logickému NEBO (OR) se analogicky používá přepínač **-o**.

Závěr

Tento díl byl trochu hutnější. Jsem si toho vědom a doufám, že nikoho z pravidelných čtenářů neodradil.
Těm nepravidelným doporučuji přečíst si i předcházející díly. Mohu však slíbit, že příští díl bude více opakovací.
Ke zopakování dosud probrané látky využijeme ukázkového skriptu. Znaky **|** a **&** nebývají běžně na českých klávesnicích dostupné. Řešení je přepnutí klávesnice na anglickou. Pokud se vám to ale nechce dělat, máte i další možnosti.
V textových (virtuálních) konzolích lze těchto znaků dosáhnout pomocí levého Altu a ASCII kódu zadaného přes numerickou klávesnici. Pro **&** je to **Alt+38**, pro **| Alt+124**. V prostředí X (např. KDE) lze využít pravý Alt a klávesy alfanumerické části klávesnice – pro **& Alt+7**, pro **| Alt+w**. Význam konstrukcí **&&** a **||** lze odvodit také z formální logiky, kde **&&** představuje logické AND (tj. příkaz1 a současně příkaz2). **||** je pak ztvárněním logického OR, tedy příkaz1 nebo příkaz2. Co se týká hodnocení souborů, nemusíme zůstávat jen u jejich existence. Testovat lze typ (např. běžný soubor, adresář, odkaz, ...), práva, vlastníka apod.

Zde je seznam:

-b soubor – existuje a je to blokový speciální soubor;
-c soubor – existuje a je to znakový speciální soubor;
-d soubor – existuje a je to adresář;
-e soubor – existuje;
-f soubor – existuje a je to normální soubor;
-g soubor – existuje a má právo set-group-id;
-k soubor - existuje a má nastavený sticky bit;
-L soubor – existuje a je to symbolický odkaz;
-p soubor – existuje a je to pojmenovaná roura (FIFO);
-r soubor – existuje a je čitelný;
-s soubor – existuje a má délku větší než nula;
-S soubor – existuje a je to soket;
-u soubor – existuje a má nastaven set-user-id bit;
-w soubor – existuje a je zapisovatelný;
-x soubor – existuje a je proveditelný;
-O soubor – existuje a je vlastněný efektivním user id;
-G soubor – existuje a je vlastněný efektivním group id.

11.díl

V tomto díle budeme pokračovat ve větvení skriptu. Slíbené velké opakování jsem se rozhodl nechat až na příště. Navážeme na minulou část, kde jsme se seznámili s podmínkami. Představíme si podmínkový příkaz **if** a různé způsoby jeho použití. To nám umožní lépe rozlišovat, které části skriptu budou provedeny při výskytu určitých okolností. Opakování to ale přece jen do značné míry bude, protože **if** dělá v podstatě stejnou práci jako konstrukce, které jsme probírali minule.

Základní konstrukce if, then, fi

Již dříve jsme si předvedli, jak v bashovém skriptu nechat provést jinou operaci, pokud nějaký příkaz dopadne pozitivně, a jinou, když se nezdaří. Nebo podmínit provedení určitého příkazu existencí nějakého souboru, hodnotou proměnné nebo výsledkem výrazu. Příkaz **if** nám dává možnost provést za daných okolností celou skupinu příkazů. O provedení dalších příkazů rozhoduje **if** na základě návratového kódu příkazů zadaných mu pro rozhodování. Trochu se to podobá konstrukci **$příkaz1 && příkaz2$**, kterou známe z 10. dílu. Nejjednodušší syntaxe **if** má podobu:

```
$if příkaz1; then příkaz2; fi$
```

Činnost **&&** jsme si ukázali na výpisu obsahu adresáře home. Použili jsme tehdy zápis **$ls home && echo ':-)‘$**.
U stejného příkladu zůstaneme i dnes, ale s **if**:

```
$ if ls /home ; then echo ‘:-)‘ ; fi
```
bohdan pavel petr
:-)

```
$ if ls /mohe ; then echo ‘:-)‘ ; fi
```
ls: /mohe: není souborem ani adresářem

Vidíme, že se **if** chová zcela stejně jako **&&**, a to v případě vyhovění i nevyhovění sledované podmínce. Podobně jako v případě ampersandů se také **if** používá nejčastěji společně s příkazem test.

Použití testu v if

Pro zopakování uvedu, že test vrací (stejně jako jiné příkazy) návratový kód 0, pokud vše proběhlo v pořádku, tj. vyhodnocovaná podmínka je platná. Hodnotu 1 vrací tehdy, když podmínka neplatí. Také připomínám, že s příkazem **test** se můžeme velmi často setkat v podobě otevírací hranaté závorky. Že test umí hodnotit typ (např. běžný soubor, adresář, odkaz, ...), práva, vlastníka a další parametry souboru, také víme. Seznam testovatelných charakteristik a příslušné parametry jsou shrnuty v tabulce. Najdete v ní také možnost porovnávání stáří souborů dle času poslední změny a operátory pro práci s proměnnými (řetězci) a výrazy.

Ve spojení s ampersandy jsme si ukázali tento zápis:

```
$ [ -e /etc/passwd ] && echo "DB uživatelů existuje"
```

Stejná situace s **if** bude vypadat následovně:

```
$ if [ -e /etc/passwd ] ; then echo "DB uživatelů existuje" ; fi
```

Z dosud uvedených příkladů je vidět, že zápis pomocí **if** je delší než s použitím **&&**. Proč si tedy komplikovat život a zbytečně natahovat skripty? Nejde o komplikaci, ale o správnou volbu prostředků.

Složitější konstrukce

Zatímco konstrukce **&&** (a **||**) jsou určeny k provedení jednoho nebo několika málo podmíněných příkazů, **if** slouží k tvorbě složitých konstrukcí, vnořených podmínek apod. **&&** a **||** využijete spíše na příkazové řádce, **if** asi častěji ve skriptech. Volba je na vás. Ukažme si krátký příklad použití **if** ve skriptu.

```
if [ -e /etc/passwd ] ; then
   echo "DB uživatelů existuje"
   echo "Kteřípak používají Bash?"
   grep "/bin/bash$" /etc/passwd
fi
```

Pokud existuje soubor **$etcpasswd$**, vypíší se hlášky na druhém a třetím řádku. Následně příkaz **grep** na vybere z **$etcpasswd$** všechny řádky končící řetězcem **$binbash$**.

Jinak – else

No jo, namítnete možná, ale co konstrukce **$příkaz1 && příkaz2 || příkaz3$**, která provede **příkaz2** v případě platnosti podmínky a **příkaz3**, když neplatí? Na tuto možnost samozřejmě **if** pamatuje také. Využívá k tomu pomocný příkaz **else**. Jako příklad použijeme zase hledání v **$etcpasswd$** pomocí **grep**, ale jeho výstup přesměrujeme do černé díry (**/dev/null**). Zajímá nás jen návratový kód, tedy zda takové řádky existují. Pokud ano, vypíšeme (pomocí programu **awk**) pouze jména příslušných uživatelů. Pokud ne, vypíše se (pomocí **echo**), že nikdo:

```
if grep "/bin/bash$" /etc/passwd > /dev/null ; then
   echo "Bash mají nastaveni:"
   awk -F ":" ,$7 == "/bin/bash" {print $1}‘ /etc/passwd
else
   echo "Nikdo :-("
fi
```

Nořímе se hlouběji

Tak a teď ke vnořeným podmínkám. To je možnost, kde má **if** oproti dvojitým ampersandům a rourám jednoznačně navrch. V následujícím výpise propojíme oba výše uvedené příklady:

```
if [ -e /etc/passwd ] ; then
   echo "DB uživatelů existuje"
   echo "Kteří pak používají Bash?"
   if grep "/bin/bash$" /etc/passwd > /dev/null ; then
      echo "Bash mají nastaveni:"
      awk -F ":" ,$7 == "/bin/bash" {print $1}‘ /etc/passwd
   else
     echo "Nikdo :-("
  fi
fi
```

Ještě jinak – elif

Další věcí, kterou má **if** oproti jiným rozhodovacím mechanismům navíc, je element **elif**. Ten umožňuje postupné rozhodování v průběhu jedné konstrukce. Jak lze z jeho názvu vyvodit, je to něco mezi **else** a **if**. Přesněji řečeno: pokud neplatí podmínka hodnocená přímo v **if**, lze se rozhodovat ještě na základě jiného kritéria. Za příklad vezměme opět počet příznivců Bashe mezi uživateli lokálního systému. Pokud se mezi nimi nenajde nikdo, nebude je hned odsuzovat. Podíváme se ještě, zda náhodou někdo z nich nemá přednastavenu odlehčenou verzi zvanou sh:

```
if grep "/bin/bash$" /etc/passwd > /dev/null ; then
   echo "Bash mají nastaveni:"
    awk -F ":" ,$7 == "/bin/bash" {print $1}‘ /etc/passwd
elif grep "/bin/sh$" /etc/passwd > /dev/null ; then
   echo "Bash sice nikdo, ale tito mají sh:"
   awk -F ":" ,$7 == "/bin/sh" {print $1}‘ /etc/passwd
else
   echo "Nikdo :-("
fi
```

Závěr

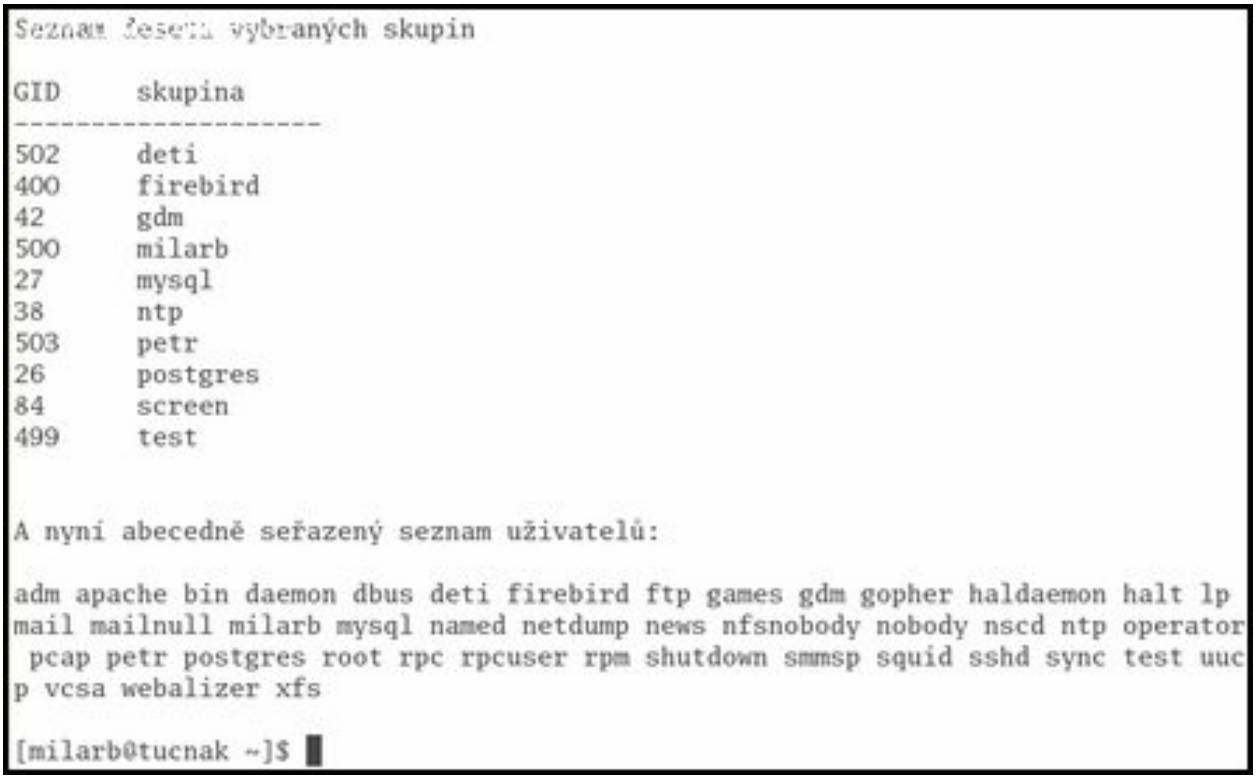
Tak nakonec ani tento díl nebyl tak úplně oddechový. Příští už ale opravdu opakovací bude. Už si na vás připravuji skript, do kterého zakomponuji co nejvíce probraných věcí. Celá konstrukce **if** je zakončena slůvkem **fi**, což nepředstavuje nic jiného než **if** napsané obráceně. Jak uvidíme později, není to jediný případ takto tvořeného názvu příkazu v Bashi.
Povšimněte si, že příkazy uzavřené dovnitř podmínky odsazuji o několik znaků (resp. o tabulátor). Příkazy uzavřené do vnořené podmínky ještě více atd. Je to dobrý zvyk, který značně zpřehledňuje delší skripty. Patří to do programátorské etiky podobně jako komentáře ve zdrojovém kódu. Pokud byste chtěli použít **if** jako náhradu dvojitých svislítek, tj. ve smyslu pokud neplatí, stačí před hodnocenou podmínku umístit vykřičník.
Např. **if ! [-e /etc/passwd] ; then …**

Seznam operátorů:

-b soubor	Existuje a je to blokový speciální soubor.
-c soubor	Existuje a je to znakový speciální soubor.
-d soubor	Existuje a je to adresář.
-e soubor	Existuje.
-f soubor	Existuje a je to normální soubor.
-g soubor	Existuje a má právo set-group-id.
-k soubor	Existuje a má nastavený sticky bit.
-L soubor	Existuje a je to symbolický odkaz.
-p soubor	Existuje a je to pojmenovaná roura (FIFO).
-r soubor	Existuje a je čitelný.
-s soubor	Existuje a má délku větší než nula.

-b soubor	Existuje a je to blokový speciální soubor.
-S soubor	Existuje a je to soket.
-u soubor	Existuje a má nastaven set-user-id bit.
-w soubor	Existuje a je zapisovatelný.
-x soubor	Existuje a je proveditelný.
-O soubor	Existuje a je vlastněný efektivním user id.
-G soubor	Existuje a je vlastněný efektivním group id.
soubor1 -nt soubor2	Pravda, když je soubor1 novější než soubor2.
soubor1 -ot soubor2	Pravda, když je soubor1 starší než soubor2.
soubor1 -ef soubor2	Pravda, když soubor1 a soubor2 mají shodný inode na stejném disku.
-z řetězec	Pravda, když je řetězec prázdný.
-n řetězec	Pravda, když je délka řetězce nenulová.
řetězec1 = řetězec2	Pravda, když řetězce jsou stejné.
řetězec1 != řetězec2	Pravda, když řetězce nejsou stejné.
! výraz	Pravda, když výraz je nepravdivý.
výraz1 -a výraz2	Pravda, když jak výraz1 tak výraz2 jsou pravdivé.

12.díl



Minule jsem přislíbil nerozvádět nová témata, ale pozastavit se nad již probranou látkou a znázornit ji na delším skriptu. Projdeme si v něm vše od proměnných, rour a smyček až po podmínky. Celý skript je uveden pod tímto textem. První řádek uvozuje skript pro Bash. Další dva (prázdné řádky nepočítáme, jsou jen pro zpřehlednění) definují hodnoty proměnných **SKUP** a **UZIV**. Nenechte se splést. Hodnotou budou řetězce **/etc/group** a **/etc/passwd**, nikoli např. obsah těchto souborů. V dalším řádku testujeme (příkaz **test**, resp. "**[**"), zda jsou oba (**-a**) soubory (tj. cíl cesty uvedené v řetězci obou proměnných) přístupné pro čtení (**-r**). Název proměnné se prostě při vykonávání příkazu nahradí jeho hodnotou. Pokud čitelné nejsou (**||**), bude skript ukončen s návratovou hodnotou **2** (**exit 2**). Pokud byla podmínka

splněna, pokračujeme dál ke třem řádkům **echo**. První dva z nich používají přepínač **-e**, který umožní využívat speciálních znaků. V našem případě jsou to znaky **\n** (nový řádek) a **\t** (horizontální tabulátor). Tabulátor využijeme k zarovnávání sloupců. Poslední z trojice **echo** pak vykreslí řadu pomlček. Vznikne tak nadpis a záhlaví tabulky.
Dostáváme se k nejsložitějšímu řádku skriptu. Již na první pohled je zřejmé, že jde o začátek smyčky **for-do-done**. Proměnná **g** bude v jejím průběhu nabývat hodnot, které vygeneruje posloupnost příkazů uzavřená v kulatých závorkách. Prvním příkazem v závoce je **cut**. Jde o klasický GNU nástroj pro vyřezávání částí řádků textových souborů. Volbou **-d ":"** mu říkáme, že jednotlivá pole v řádcích souboru jsou oddělena znakem dvojtečka. Volba **-f 1** pak znamená, že chceme vypsat hodnotu prvního pole. Vstupním souborem je hodnota proměnné **$SKUP**, tedy **/etc/group**.
Za normálních okolností by takto zadaný **cut** vypsal hodnoty všech prvních polí souboru **/etc/group**, tedy seznam všech skupin definovaných v systému pod sebou. My ale pomocí **|** přesměrováváme tento výstup ke zpracování příkazu **tail**. Ten slouží k výpisu "ocásku" (tj. posledních několika řádků) vstupního souboru (tedy výstupu z **cut**). Délka ocásku je stanovena pomocí **-n 10** na deset řádků. Výsledkem příkazu **tail** bude seznam jmen posledních deseti skupin ze souboru **/etc/group**. Ani to nebude finální, protože za **tail** se objevuje další roura. Mezivýsledek tak poputuje ještě do třídičky **sort**. Ta se postará o to, že náš seznam deseti vyvolených (skupin) bude (abecedně) seřazen.
Takže výsledkem celé závorky v řádku **for g** ... bude abecedně seřazený seznam jmen deseti posledních skupin ze souboru **/etc/group**. Proměnná **g** pak při každém průchodu cyklem nabyde postupně hodnotu jednoho z nich.
Tak a co vlastně děje uvnitř cyklu? Nejprve si naplníme proměnnou **cislo**. Ta bude opět nabývat hodnotu toho, co vypadne z příkazů seřazených v závorce. Prvním je **grep,** který vypíše ze zadaného souboru pouze ty řádky zadaného souboru (zde **$SKUP**, tedy **/etc/group**), ve kterých se vyskytuje zadaný řetězec. Hledaným řetězcem je zde hodnota proměnné **$g**, tedy jméno skupiny zpracovávané v tom kterém průchodu smyčkou. Protože by se v souboru **/etc/group** nemělo nacházet více skupin stejného jména, měl by být vypsán právě jeden řádek. Vypsaný řádek je opět s použitím roury přenesen na standardní vstup příkazu **cut**. O jeho funkcionalitě jsme se již zmiňovali, proto jen shrnu, že z dodaného řádku souboru **/etc/group** vyřízne třetí pole, tj. číslo hledané skupiny. To se stane pro tento průchod smyčkou hodnotou proměnné cislo. Na dalším řádku vypíšeme pomocí **echo** vedle sebe aktuální (při každém průchodu smyčkou budou jiné) hodnoty proměnných **$cislo** a **$g**. Díky přepínači **-e** můžeme opět použít tabulátor (**\t**), aby se sloupečky zarovnávaly hezky pod příslušné nadpisy v hlavičce. A jsme na konci první smyčky, protože vidíme "patník" **done**. Od ní se skript bude vracet k **for g** ... tak dlouho, dokud bude **g** nabývat nových hodnot (tj. do deseti). Po posledním průchodu bude pokračovat dál a dostane se k **echo**. To bez parametrů udělá prostě prázdný řádek.
Nyní přichází "odpočívadlo". Aby si pomalý uživatel mohl tabulky alespoň rychle přelétnout očima. Skript si zde na dlouhé dvě sekundy (tj. několik miliard procesorových cyklů) odpočine (**sleep 2**). Pokračovat bude vynecháním dalšího řádku (**echo**), zobrazením nadpisu další části a vynecháním ještě jednoho řádku. Když přidělíme proměnné hodnotu **""**, znamená to, že ji vlastně vyprazdňujeme. Hodnoty proměnných lze totiž uzavírat do uvozovek, a pokud uvedeme dvě uvozovky těsně vedle sebe (tj. není v nich nic, ani mezera), nabývá proměnná hodnotu nic, tj. Nijakou, tedy žádnou, takže je prázdná. A právě to jsme udělali s proměnnou seznam. A propracovali jsme se k další smyčce **for-do-done**. Tentokrát budeme naplňovat proměnnou u a opět výsledkem příkazů uzavřených do závorek. Nám již dobře známé **cut** vyřízne ze souboru **/etc/passwd** rozděleného na pole dvojtečkami (**-d ":"**) pole první (**-f 1**), tj. jméno uživatele. Výsledek této řezničiny, jak už je u mne zvykem, poputuje rourou. Na její druhé straně čeká netrpělivě **sort**, aby příchozí řetězce (jména uživatelů systému) setřídil. Smyčka se tedy bude opakovat tolikrát, kolik uživatelů je v daném systému definováno (kolik řádků má soubor **/etc/passwd**). Vnitřek druhé smyčky je velmi skromný. Definujeme hodnotu proměnné **seznam** tak, aby na konci obsahovala jména všech uživatelů oddělená mezerou. Kdybychom zadali **seznam=$u**, tak by se v každém průchodu smyčkou přemazala novou hodnotou. Chceme-li hodnotu přidat, řekneme Bashi, aby se nová hodnota rovnala staré hodnotě (**$seznam**), jedné mezeře a hodnotě proměnné **$u**. Tímto způsobem si v každém průchodu smyčkou ponechá **seznam** svou hodnotu z minula (na začátku byl prázdný) a rozšíří se o aktuální hodnotu proměnné **$u** (tj. další jméno v pořadí). Nyní můžeme smyčku ukončit pomocí **done**. Následuje prosté vypsání hodnoty proměnné seznam, jeden obligátní prázdný řádek a řádné ukončení skriptu pomocí **exit 0**.

```
#!/bin/bash
SKUP=/etc/group
UZIV=/etc/passwd
[ -r $SKUP -a -r $UZIV ] || exit 2
echo -e "\nSeznam deseti vybraných skupin"
echo -e "\nGID\tskupina"
echo "--------------------"
for g in $( cut -d ":" -f 1 $SKUP | tail -n 10 | sort ) ; do
      cislo=$( grep $g $SKUP | cut -d ":" -f 3 )
      echo -e "$cislo \t$g"
done
echo
sleep 2
echo
echo "A nyní abecedně seřazený seznam uživatelů:"
echo
seznam=""
for u in $( cut -d ":" -f 1 $UZIV | sort ) ; do
      seznam="$seznam $u"
done
echo $seznam
echo
exit 0
```

Závěr

Tento skript může na první pohled vypadat děsivě. Věřím však, že pro pozorné čtenáře seriálu není po mikroskopické analýze ničím nepochopitelným. Většina skriptů nebývá tak zamotaná jako tento. Abyste ale rozuměli čemukoli, na co můžete narazit, máme před sebou ještě řádný kus práce. Příště se podíváme na další typy cyklů, které Bash vedle **for-do-done** nabízí. Soubor **/etc/passwd** obsahuje informace o účtu uživatele, nikoliv hesla. Soubor **/etc/group** zase definuje skupiny, do kterých uživatelé patří. Oba soubory může měnit jenom root, a to ještě přesně definovaným postupem. Špatným zápisem do souboru si můžete celý systém nenávratně poškodit, protože uživatelé jsou i programy, nejen lidé. Pokud by program přestal patřit do skupiny, kterou potřebuje, odmítne pracovat. Týká se to samozřejmě i životně důležitých procesů, které startují systém. Známá, ale o to víc nepříjemná chyba byla v jedné starší verzi programu **KUser** (aplikace pro editaci uživatelů v KDE), která takto, spouštěna pod rootem, spolehlivě poškodila v podstatě celý systém.

13.díl

Vítám příznivce Bashe u prvního dílu druhého ročníku našeho seriálu. Pevně doufám, že jste se v opakovacím skriptu z minulého dílu neztratili. Dnes nás čekají další dva typy cyklů - **while** a **until.**

Smyčka while

Smyčka má obecně tvar **while list1; do list2; done**, kde **list1** a **list2** jsou seznamy příkazů. Tělo smyčky se provádí (tj. příkazy v seznamu **list2** jsou vykonávány) tak dlouho, dokud poslední příkaz ze skupiny **list1** (která se také vykonává před každým průchodem smyčkou) zasílá návratovou hodnotu 0. Pomocné výrazy **do** a **done** se používají obdobně jako v případě smyčky **for**. Pro případ, že by se vám z toho motala hlava, zkusíme krátký příklad, který naleznete v rámci.

příkazy	1. průchod	2. průchod	3. průchod	4. průchod
a=""	""			
while ["$a" != "AAA"]	""	A	AA	AAA
do a=${a}A	A	AA	AAA	
echo "a = "$a	A	AA	AAA	
done	A	AA	AAA	
exit 0				AAA

První řádek není součástí smyčky. Při spuštění skriptu pouze nastaví hodnotu proměnné **a** na "", tedy prázdný řetězec. Jako **list1** se často používá podmínka (test) uzavřená do hranatých závorek. Zde v podmínce zkoumáme, zda se hodnota proměnné a NErovná řetězci AAA. Pokud ne (1. průchod "", 2. průchod A, 3. průchod AA), nabývá test hodnotu 0 a **list2** může být proveden.
První příkaz uvnitř smyčky (hned za **do**) přidává k aktuální hodnotě proměnné a znak A. Při prvním průchodu se tak změní z "" na A, při druhém z A na AA a při posledním z AA na AAA. Bližší popis tohoto mechanismu najdete ve 4. části seriálu. Další řádek jen pro kontrolu vypíše hodnotu proměnné **a**. Podle toho zjistíme, kolikrát smyčka opravdu proběhla. Závěrečná klauzule **done** vrací běh skriptu opět k **while**. Ten vyhodnotí podmínku a dle výsledku rozhodne, co dál. Po třetím průchodu má a hodnotu AAA, proto při pokusu o průchod čtvrtý **while** vstup do smyčky nepovolí a skript pokračuje za **done**, kde jej čeká konec v podobě **exit 0**.

Smyčka until

Smyčka **until** je téměř dvojčetem **while**. Mají stejnou syntaxi a plní stejnou funkci. Jediným rozdílem je, že zadanou podmínku (resp. výsledek operace **list1**) vyhodnocuje obráceně. Mohli bychom říci, že **while** znamená pro počítač "vykonávej, pokud", zatímco **until** překládá ve smyslu "vykonávej, dokud neplatí". Pro úpravu našeho skriptu na **until** bychom museli druhý řádek změnit na until **["$a" = "AAA"]**, který můžeme popsat slovy "Vykonávej, dokud nenastane

situace, kdy se hodnota proměnné a rovná AAA". Průběh smyčky i hodnoty proměnné v jednotlivých průbězích jsou shodné s případem použití **while**.

Závěr

Opět jsme se ve znalostech nejpoužívanějšího textového příkazového interpretru posunuli kousek dál.
Na příště si chystám představení jednoho velmi důležitého pomocníka, kterým je příkaz **read**. Umožní nám interakci s uživatelem pomocí vstupu z klávesnice v průběhu provádění skriptu.
Cykly **while** a **until** jsou jistou formou kombinace smyčky **for** (viz 9. díl) a podmínkové konstrukce **if** (viz 11. díl).
Čtvrtý díl seriálu vyšel v dubnovém čísle 4/2005, ale už dlouho dobu jej naleznete na našem webu:
http://www.linuxexpres.cz/praxe/bash-4-dil
Názvy příkazů samozřejmě vycházejí z angličtiny – **while** je "zatímco" a **until** je "dokud, dokud ne, do té doby než".

14.díl

Možná máte ze skriptů prezentovaných v tomto seriálu divný pocit, jako by to nebyly plnohodnotné programy, jako by jim něco chybělo. Abstrahujeme-li od grafického rozhraní, zjistíme, že schází jakákoli komunikace s uživatelem.
Ano, všechny zde dosud prezentované skripty byly neinteraktivní. Dnes tento nedostatek odstraníme seznámením se s příkazem **read**.

Řádky a slova

Nejprve si prosvištíme trochu teorie. Řádek je v podání Bashe sekvence znaků zakončená znakem (klávesou) [**Return**] ([**Enter**]). Slova jsou části řádku oddělené speciálními znaky. Tyto znaky se nazývají vstupní oddělovače polí (Input Field Separator – IFS) a jejich seznam je uchováván v proměnné **IFS**. Pokud si ovšem zkusíte vypsat její obsah (**echo $IFS**), pravděpodobně nic neuvidíte. To proto, že výchozím oddělovačem slov je mezera a její vypsání na prázdný řádek budí stále dojem prázdného řádku. Zkuste ji tedy obklíčit třeba písmeny a (**echo a${IFS}a**).

Příkaz read

Úkolem **read** je číst standardní vstup a takto získanými daty naplňovat zadané proměnné. Přesněji řečeno přečte vždy jeden řádek vstupu, rozdělí jej na slova a ta postupně přiřadí zadaným proměnným. Nejjednodušším zápisem je samotný příkaz **read**. V tomto případě je celý řádek načten do proměnné **REPLY**. Pokud chceme vstupem naplňovat konkrétní proměnnou, zadáme jednoduše její jméno za **read**, např. **read login** naplní proměnnou **login** zadaným řetězcem.
Pro naplnění dvou proměnných by vypadal zápis takto: **read jmeno prijmeni**.
Pokud je na vstupu zadáno více slov, než je počet proměnných, dosadí se do poslední z nich všechna zbývající slova včetně oddělovačů. Pokud je naopak v řádku slov méně, jsou zbývající proměnné naplněny prázdným řetězcem.
Read přichází také s několika přepínači. Vybíráme z nich:

-a pole – místo do proměnných ukládá **read** načtené hodnoty do pole zadaného jména (počínaje hodnotou [0]) – viz 5. díl;
-n x – nečte řádek, ale jen x znaků, což se hodí pro zadávání hodnot přesně dané délky nebo prostě pro čekání na stisk klávesy (**read -n 1**);
-s – způsobí, že zadávané znaky nebudou zobrazovány, čehož lze využít např. při zadávání hesla nebo jiných důvěrných údajů;
-t sekundy – read bude čekat na vstup z klávesnice jen stanovený počet sekund.

Použití ve skriptech

Čekání na stisk klávesy:

```
while true; do
    echo "Stiskněte klávesu"
    read -n 1
    echo -n "Právě je "; date
done
```

Zadávání jména a hesla:

```
echo -n "Jméno: "; read jmeno
echo -n "Heslo: "; read -s heslo
echo "Heslo pro $jmeno zadáno."
```

Příkaz **read** může pracovat i s jinými vstupy než klávesnice. Za upozornění však stojí, že při použití konstrukce roura nepracuje tak, jak většina uživatelů očekává (viz 6. díl). Místo:

```
echo "aa bb" | read a b ; echo $a $b
```

Je třeba použít:

```
echo "aa bb" | ( read a b ; echo $a $b )
```

Častější je však použití pro načítání víceřádkových souborů. Ukažme si, jak ze souboru **/etc/fstab** (seznam přípojných bodů) vybrat jen údaje o discích a přípojných bodech:

```
grep -v "^#" /etc/fstab | while read disk bod zbytek
    do echo "$disk na $bod"
done
```

V prvním řádku jsme odfiltrovali řádky začínající křížkem a výsledek jsme předali smyčce **while**, která bude probíhat tak dlouho, dokud ji bude **grep** zásobovat vyhovujícími řádky z **/etc/fstab**. V každém průběhu smyčkou uloží **read** první pole zpracovávaného řádku do proměnné **disk**, druhé do **bod** a **zbytek** řádku do zbytek. Zbytek smyčky je triviální. Read je anglické sloveso, jehož základním významem je číst. Velké slovníky však přinášejí více než deset dalších významů (např. vysvětlovat, přednášet, rozumět, studovat, projednávat, …). Je to sloveso nepravidelné, které se ve všech třech tvarech stejně píše, ale v minulém čase a trpném rodě vyslovuje odlišně. Mimo to lze read použít jako podstatné jméno (četba) a přídavné jméno (sečtělý).

15.díl

Pokud jste sami nepřišli na to, jak s pomocí **read** (který jsme brali minule) vytvářet textové rozhraní pro komunikaci s uživatelem, bude se vám jistě hodit příkaz **case**, který si ukážeme tentokrát.

Příkaz case

Case nám podobně jako **if** umožňuje vytvořit konstrukci, ve které se budou provádět za různých okolností různé příkazy nebo skupiny příkazů. Ve spojení s minule probíraným **read** vytváří ideální prostředí pro tvorbu interaktivních skriptů. Read zprostředkuje vstup od uživatele tím, že naplní nějakou proměnou např. znaky z klávesnice a **case** na základě hodnoty této proměnné provede příslušnou akci. Tuto situaci bychom samozřejmě mohli řešit také použitím konstrukce **if**, **elif**, **elif**, **else**, **fi**. Ta má ovšem složitější formu zápisu a hodí se při rozhodování podle složitějších kritérií. Syntaxe příkazu **case** začíná slovy **case slovo in**, kde místo slova vkládáme nejčastěji hodnotu proměnné, tj. jméno proměnné uvozené znakem dolar (**$**), např. **case $promenna in**. Za uvozující formulí následuje seznam hodnot (v závorce, resp. zakončených zavírací závorkou), kterým když proměnná vyhovuje, vykoná se první sada příkazů. Příkazy v sadě jsou odděleny středníkem (nebo koncem řádku) a celá sada je potom zakončena dvěma středníky. Následuje další podmínka a sada příkazů atd. Nakonec můžeme vložit obecnou podmínku znakem hvězdička, což znamená **cokoli, co nevyhovovalo předchozím podmínkám**. Celá konstrukce je zakončena slovem **esac** (tj. pozpátku **case**, podobně jako u **if** a **fi**).

Použití ve skriptech

Je mi jasné, že předchozí popis vyžaduje praktickou ukázku. Uvedu krátký skript, který na základě zadání čísla od jedné do pěti pomocí příkazu **read -n 1** (viz minulý díl) zobrazí slovní hodnocení školního prospěchu. Hodnotu známky budeme načítat do proměnné **znamka**. Příkazem **read** tedy načteme právě jeden znak (**read -n 1**) z klávesnice a uložíme ho do proměnné **znamka**. **Case** potom na základě její hodnoty (**case $znamka in**) vypíše příslušný text. Nabývá-li hodnot 1 až 5, zobrazí se slovní význam dané známky. Jakýkoli jiný znak (zařizuje *) je označen za neplatnou známku. A nakonec nesmíme zapomenout **esac**. Porovnávat můžeme i řetězce delší než jeden znak. Pokud chceme dát do jedné podmínky více hodnot, oddělujeme je svislou čarou (znak **|**). Lze využívat také jiných zástupných znaků než hvězdička. Jednoduchým je otazník, který nahrazuje jakýkoli jeden znak, podobně jako ve jménech souborů.

```
#!/bin/bash
echo "Slovní hodnocení prospěchu"
echo -n "Zadejte známku: "
read -n 1 znamka
echo
case $znamka in
   1) echo "Výborný" ;;
   2) echo "Chvalitebný" ;;
   3) echo "Dobrý" ;;
   4) echo "Dostatečný" ;;
   5) echo "Nedostatečný" ;;
   *) echo "Neplatná známka" ;;
esac
```

Zajímavé jsou hranaté závorky, do kterých lze přímo vypsat přesný seznam znaků, které má závorka nahrazovat. Tak např. [**aeiouy**] znamená jakákoli malá krátká samohláska. V hranatých závorkách ale můžeme použít i rozsahy ve smyslu od-do. Kupř. [**0-9**] značí všechny číslice, [**a-z**] potom všechna malá písmena. Zde ovšem pozor. V závislosti na nastavení jazyka ve vašem systému (**echo $LANG**) bude tento rozsah zahrnovat i znaky s diakritikou. Pokud tam chcete mít opravdu všechny, měl by zápis mít podobu [**a-ž**]. Pojďme si ukázat krátký skriptík na určování osoby osobního zájmena.

```
#!/bin/bash
echo "Určení osoby zájmena"
echo -n "Zadejte osobní zájmeno: "
read zajmeno
case $zajmeno in
   já|my) echo "1. os." ;;
   [vt]y) echo "2. os." ;;
   on|on[aioy]) echo "3. os." ;;
   *) echo "Chyba" ;;
esac
```

Ve třetí osobě se nám mísí použití svislé čáry a hranatých závorek. Věřím, že se vám použití těchto znaků v případě rozhodovacích podmínek **case** nebude plést s jejich použitím jakožto nepojmenované roury (viz 6. díl) a testu (viz 10. díl). Zápis **on|on[aioy]** čte Bash jako on nebo ona nebo oni nebo ono nebo ony. **Case** je anglické sloveso s významem případ, a to jak ve smyslu obecné situace, tak jako odborný termín např. v oblasti lékařské nebo právní. Známé slovní spojení **case study** označuje případovou studii. **Case** se dá použít také slangově pro označení výstředního člověka ve smyslu "to je ale případ".

16.díl

V tomto díle seriálu nás čeká poslední z větvících konstrukcí – příkaz **select**. Umožní nám snadno vytvářet textové nabídky, ze kterých si bude moci uživatel vybírat.

Příkaz select

Mějme seznam nějakých možností a chtějme, aby si z nich uživatel mohl vybrat. To zadáni jako šité na míru pro příkaz **select**. Příbalový leták (v našem případě manuálová stránka) nám radí postupovat takto:

```
select jmeno [ in seznam ] ; do prikazy ; done
```

Na místo **jmeno** doplníme jméno proměnné bez znaku dolar. Hranaté závorky říkají, že slůvko **in** a následující seznam hodnot oddělených mezerami nejsou povinné. My je však uvádět budeme a **select** nám z nich posléze sestaví přehlednou nabídku. Pomocné **do** již známe, následuje seznam příkazů oddělených středníkem nebo nový řádek a konstrukci zakončuje **done**.

Použití ve skriptech

Nejlépe si to samozřejmě ukážeme na příkladu. Začíná jaro, ve vzduchu vznáší různé věci, třeba volby nebo chřipka. Zeptejme se uživatelů, co nejčastěji na chřipku používají:

```
select lek in multivitamin tamiflu viagra ; do echo Vybráno: $lek ; done
```

Po zadání tohoto řádku do Bashe se zobrazí číslovaný seznam o třech položkách (multivitamin, tamiflu, viagra) a pod ním výzva k odpovědi (čekající kurzor a zvlaštní forma promtu). Odpovíme číslem a klávesou [**Enter**] (**Return**). Do proměnné **lek** je vložena příslušná položka seznamu (nikoli číslo), jak můžeme vidět na výpisu jejího obsahu příkazem **echo**. Možná vás zaskočí, že konstrukce se neukončí a ptá se znovu. Opravdu se chová jako smyčka. Při vkládání z klávesnice ji nezastaví ani prázdný řetězec, tedy pokud naprázdno odentrujete. Ostatně vložíme-li cokoli nad rozsah seznamu, nabude proměnná hodnotu prázdného řetězce. Vkládání ukončíme kombinací kláves [**Ctrl+d**]. Chováni **selectu** jakožto smyčky můžeme potlačit uvedením **break** jako poslední v seznamu prováděných příkazů před závěrečným **done**. Ukážeme si to na dalším příkladu, tentokrát rozepsaném na více řádků. Mnohé lidi postihne viróza v důsledku špatně těsnících oken. Zeptejme se jich tedy, kudy nejvíce táne:

```
echo "Zvolte typ oken:"
select okno in dřevěná plastová ikspé; do
   echo "Nejděravější okna jsou "$okno
   break
done
```

Select má své výhody i nevýhody. Mezi klady patří jistě skutečnost, že seznam nejen automaticky očísluje, ale také jej naformátuje do sloupců, pokud by byl příliš dlouhý. Pohodlné je v některých případech zadávat seznam položek pomocí pole (viz. 5. díl). Nevýhodou **case** je naopak zase to, že si nemůžeme zvolit vlastní způsob označování jednotlivých položek. Horší však je, že bez ohledu na odpověď vykonává vždy stejné příkazy.

Alternativa

Pokud bychom chtěli, aby se při různých odpovědích vykonával různé sady příkazů, museli bychom to udělat pomocí konstrukce **case** vložené mezi příkazy uvnitř konstrukce **select**. Když k tomu připočtu svou lenost, tedy neochotu enterovat po každém výběru, jsem již krůček od toho, abych si seznam vytvořil ručně. Přepis výše uvedeného výběru by tak mohl vypadat třeba takto:

```
echo "Zvolte typ oken:"
echo "a) dřevěná"
echo "b) plastová"
echo "c) ikspé"
echo -n "#> "
read -n 1 okno
echo
case $okno in
    a) echo "Zkuste plast" ;;
    b) echo "To mne překvapuje" ;;
    c) echo "Aha..." ;;
    *) echo "Nesprávný výběr" ;;
esac
```

Nejprve jsem pomoci **echo** vytvořil seznam i s promtem pro kurzor. Položky jsou označeny písmeny.
Potom pomoci **read -n 1** (viz 14. díl) načítám právě jeden znak do proměnné **okno**. Na základě její hodnoty se pak s pomocí **case** (viz 15. díl) vypíše text.

Závěr

Dnes jsme probrali poslední z konstrukcí, které nám Bash pro skriptování nabízí. Nebuďte ale smutní, ještě několik interních příkazů známe. Také jsme neprobrali lecjaké fígle. Příště si ukážeme některé takové se zvláštním druhem proměnných. A další díl pak bude opakovací. **Select** v angličtině znamená vybrat, zvolit a vyvolit. V roli přídavného jména pak výběrový a exkluzivní. Osobně jsem se s tímto slovem setkal prvně na osmibitových počítačích Atari, kde byl **select** prostřední z pěti systémových kláves.

17.díl

Tentokrát se naučíme pracovat s parametry, které uživatel posílá skriptu z příkazové řádky. Navážeme tak na předchozí části, kdy jsme se seznámili s příkazy, které nám umožnily komunikovat s uživatelem během provádění skriptu.

Poziční parametry

Poziční parametry jsou něčím, co zná každý, kdo pracuje s příkazovou řádkou. Jsou to jednoduše všechny takové ty přepínače a parametry, které přidáváme za různé příkazy. Např. zadáním **ls** jsme nepředali žádný parametr. V případě **ls /home** je jediným parametrem **/home**. Při zadání **ls /hoem /opt** jsou dvěma parametry **/home** a **/opt**. Parametrem je ale také **--help**, pokud zadáme **ls --help**. Dvěma parametry u **ls -l /home** jsou pak **-l** a **/home**. V mnoha případech (např. u příkazů **cp** a **mount**) záleží na tom, v jakém pořadí se který parametr nachází, resp. na kolikáté pozici je. Z toho důvodu se těmto parametrům říká poziční parametry. Bash před spuštěním příkazu od sebe oddělí jednotlivé parametry a naplní jimi zvláštní sadu proměnných. První zadaný poziční parametr tak spuštěný program (nebo skript) najde v proměnné **$1**, druhý v **$2**, třetí v **$3** atd. Uveďme malý příklad. V něm bude skript vypisovat hlášku "uživatel nechce pomoc", pokud nebude prvním parametrem **-h**. V případě, že jej uvedete, zobrazí se pomoc pro uživatele. Hodnotu prvního pozičního parametru čteme z proměnné **$1**:

```
#!/bin/bash
if [ "$1" = "-h" ]; then
    echo "pomoc pro uživatele!"
else
    echo "uživatel nechce pomoc!"
fi
```

Nyní si tento skript uložte, nastavte mu práva spouštění (viz 8. díl) a zkuste jej spustit bez parametru, s parametrem **-h**, příp. i s jinými parametry. Trochu odlišné je volání desátého a vyšších pozičních parametrů. Pokud totiž zadáme: **echo $10**, zobrazil by se obsah prvního pozičního parametru (tj. hodnota **$1**) a znak nula (0). Abychom získali desátý poziční parametr, musíme zadat **echo ${10}**.

Související proměnné

Zajímavým případem je **$0**. Uvádí jakoby nultý poziční parametr, tedy parametr předcházející tomu prvnímu. Na příkazovém řádku je ale před prvním parametrem napsán samotný příkaz. A právě přesné jméno (včetně případně zadané cesty) je náplní této proměnné. K čemu to je dobré? Program tak může snadno zjistit, jakým způsobem byl spuštěn a jak se vlastně jmenuje. Protipólem nultého parametru jsou **$*** a **$@**. Oba zobrazí seznam všech pozičních parametrů. V prvním případě oddělených předdefinovatelným oddělovačem (problematiku interních proměnných probereme v některém z následujících dílů), ve druhém případě mezerou. Pokud bychom tak chtěli zjistit, jak přesně byl spuštěn skript, dosáhli bychom toho příkazem **echo $0 $@**. Hodit se může také znalost celkového počtu pozičních parametrů. K tomu slouží speciální proměnná **$#**.

Příkazy shift

S pozičními parametry souvisí interní příkaz Bashe zvaný **shift**. Není to jen pojmenování přeřaďovače na klávesnici, ale především anglické slovo s významem posunout. A to je také podstatou tohoto příkazu. Posunuje poziční argumenty. Sám přitom přijímá jediný argument, kterým je číslo. Jednoduše řečeno, když zadáme např. **shift 2**, budou vypuštěny první dva poziční parametry a ostatní budou posunuty tak, že původní parametr číslo 3 bude nyní 1, z **$4** se stane **$2**, z **$5** bude **$3** atd. Pokud zadáte **shift** bez parametru (bez čísla), bude efekt stejný jako při zadání **shift 1**. Předveďme si to opět na příkladu:

```
#!/bin/bash
echo "Na začátku: "$@
echo "Počet parametrů: "$#
while [ "$#" != "0" ]; do
    shift
    echo "Po shiftu: "$@
    sleep 1
done
```

Skript můžete spustit s libovolným počtem parametrů. Nejprve budou všechny vypsány (**$@**) a určen jejich počet (**$#**). Potom bude zahájena smyčka **while** (viz 13. díl), která bude probíhat, dokud bude počet parametrů různý od nuly. Uvnitř smyčky se nejprve provede samotný **shift** (bez parametru, tj. o jedno místo) a následně se se vypíše seznam parametrů po shiftu. Aby byl zdůrazněn průběh jednotlivými cykly, je každý ukončen vteřinovým spánkem (**sleep 1**). Už umíme vytvořit skript reagující na parametry zadané z příkazové řádky. Příští díl pak bude opakovací.

18.díl

A máme tu další opakovací díl. Stejně jako minule (tím myslím díl č. 12) si předvedeme dosud probraná témata na ukázkovém skriptu. Za vzor nám poslouží mírně modifikovaný skript Tomáše Hanuska pro převod souborů WAV na OGG (viz LinuxEXPRES 10/2005), za který mu tímto ještě jednou děkuji. Cílem skriptu je získat z audio CD digitální záznam všech skladeb, jejich seznam z CDDB, skladby zkomprimovat do formátu OGG Vorbis, ze záznamů CDDB vytvořit ID3 tagy a jména souborů a uklidit dočasné soubory.

Rozbor skriptu

První podmínka **if** zjišťuje, zda byl zadán první poziční parametr. Pokud ano, naplní jím proměnnou **device**, pokud ne, uloží do této proměnné výchozí hodnotu **cdrom**. Podobně druhá podmínka naplňuje proměnnou **directory** buď hodnotou druhého pozičního parametru nebo defaultním **new_dir**. Třetí podmínka zjišťuje, zda položka stejného jména, jako je hodnota proměnné **directory**, již existuje v aktuálním adresáři. Pokud ano a nejde o adresář, celý skript se ukončí. Pokud adresář již existuje, vypíše se jen upozornění a skript pokračuje dál.
Příkaz **cdda2wav** stáhne z CDDB názvy skladeb a uloží je do souborů pojmenovaných **track<číslo>.inf**, které budeme dále využívat pro ID3 tagy a jména souborů ogg. Dále vytvoří soubory **audio.cddb** a **audio.cdindex** s popisem celého disku, které si na konci skriptu uschováme. Příkazy **cdparanoia** provedou vlastní převod všech audio stop z CD do aktuálního adresáře po skladbách. Vznikne tak tolik vsw souborů, kolik bylo melodií na CD. Někdy však mechanika nesprávně vytvoří ještě prázdný soubor **track00.cdda.wav**. Toto jméno jsem proto vložil do proměnné **err_track**.
V syntaxích **cdda2wav** a **cdparanoia** využíváme parametr pro označení zařízení, ze kterého se má číst (tj. naše CD-ROM mechanika). Protože z příkazového řádku se očekává pouze např. **hdb** nebo **cdrom2**, doplňujeme před hodnotu **$device** ještě adresář **/dev/**,takže z toho vzejde např. **/dev/hdb** nebo **/dev/cdrom2**.
Na dalším řádku je podmínka ve zjednodušeném zápisu (bez **if**). Zjišťuje, zda soubor jména odpovídajícího hodnotě proměnné **err_track** v aktuálním adresáři existuje. Pokud ano (**&&**), smaže jej. Kdybychom to neudělali, nesouhlasilo by nám číslování skladeb. Před začátkem procesu konvertování souborů z wav na ogg nastavíme počítadlo skladeb (**soubor_idx**) na hodnotu 1. A nyní zahájíme konvertovací smyčku příkazem **for**. Ta bude postupně obměňovat obsah proměnné **soubor** tak dlouho, až v něm vystřídá všechna jména souborů z aktuálního adresáře, které odpovídají masce ***.wav**, tj. projde postupně všechny naše nagrabované skladby.
Další dva řádky zavádějí novou proměnnou **soubor_index**, což bude dvoumístné pořadové číslo skladby na CD. Vycházíme z proměnné **soubor_idx** (která se na konci každého průchodu cyklem zvyšuje o 1). Pokud je **soubor_idx** menší než deset, bude **soubor_index** sestávat z nuly a hodnoty **soubor_idx**, tj. i čísla pod 10 budou přidáním nuly dvojmístná.
Pokud je **soubor_idx** větší nebo roven 10, převezme se tato hodnota do **soubor_index**. Následující podmínka zkoumá, zda máme pro právě zpracovávanou skladbu soubor s popisem stažený z CDDB, tj. zda existuje soubor jména **audio_$soubor_index.inf**. Máme-li jej, vybere z něj postupně tři hodnoty pro proměnné **nazev_song**,**nazev_artist** a **nazev_album**.
Výběr hodnot provádíme tak, že každé proměnné přiřadíme výsledek práce posloupnosti příkazů **grep** (vyhledávání řetězce v obsahu souboru) a **sed** (proudový editor textu). Tato posloupnost je spojena rourou, takže řádek, který **grep** v zadaném souboru najde, pošle ke zpracování **sedu**. Ten z něj vybere potřebnou posloupnost znaků (řetězec mezi apostrofy) a ten se pak stane hodnotou příslušné proměnné.
Další čtyři řádky postupně utvářejí jméno budoucího souboru a využívají k tomu proměnnou novy. Nejprve ji naplníme spojením hodnoty **soubor_index** (číslo skladby), pomlčky, **nazev_song** (název skladby) a koncovky .ogg.
Lze očekávat, že **nazev_song** obsahuje mezery, lomítka, dvojtečky, uvozovky a jiné znaky, které bychom ve jménech souborů raději neměli. Proto řetězec uložený v proměnné **novy** v následujících třech řádcích profiltrujeme.
Použijeme k tomu podobnou metodu jako při vytahování jména skladby, autora a alba z inf souboru. Proměnné **novy** přiřadíme (vlastně její hodnotu nahradíme) řetězcem, který vypadne z posloupnosti příkazů zadaných ve zpětných apostrofech. Nejprve vypíšeme příkazem **echo** původní hodnotu proměnné **novy** a potom ji skrze rouru pošleme na zpracování konkrétnímu filtru.

```
#!/bin/bash
if [ "$1" ]; then device=$1; else device="cdrom"; fi
if [ "$2" ]; then directory=$2; else directory="new_dir"; fi
if [ -e $directory ]; then
       if [ ! -d $directory ]; then
              echo "$directory již existuje a není to adresář!"
              exit 1
       else
              echo "Adresář $directory již existuje!"
       fi
else
       mkdir $directory
fi
cdda2wav -D /dev/$device -L 0 -J -v titles
cdparanoia -d /dev/$device -Q -v
cdparanoia -d /dev/$device -B
```

```
err_track="track00.cdda.wav"
[ -e $err_track ] && rm -f $err_track
soubor_idx=1
for soubor in *.wav ; do
      [ $soubor_idx -lt 10 ] && soubor_index=0$soubor_idx
      [ $soubor_idx -ge 10 ] && soubor_index=$soubor_idx
      if [ -e "$PWD/audio_$soubor_index.inf" ]; then
            nazev_song=`grep Tracktitle audio_$soubor_index.inf |sed "s/[^']*'\
(.*\)'[^']*/\1/g"`
            nazev_artist=`grep Performer audio_$soubor_index.inf |sed "s/[^']*'\
(.*\)'[^']*/\1/g"`
            nazev_album=`grep Albumtitle audio_$soubor_index.inf |sed "s/[^']*'\
(.*\)'[^']*/\1/g"`
            novy="$soubor_index-$nazev_song.ogg"
            novy=`echo $novy |tr ,[]&=|/\\:;, , ,{}+------._'`
            novy=`echo $novy |sed -e "s/-_/-/g" -e "s/_-/-/g" -e "s/\._/./g" -e
"s/_\././g"`
            novy=`echo $novy |sed -e "s/[!\"#$%'()*<>?@{}]//g"`
            echo "Převádí se $soubor_index:$nazev_song,$nazev_album,$nazev_artist"
            oggenc -q 6 -t "$nazev_song" -a "$nazev_artist" -l "$nazev_album" -N
$soubor_index -n "%n %t.ogg" "$soubor" -o "$directory/$novy"
      else
            echo "Převádí se $soubor_index"
            novy=`echo $soubor |sed ,s/wav/ogg/'`
            oggenc -q 6 "$soubor" -o "$directory/$novy"
      fi
      soubor_idx=$[$soubor_idx+1]
done
echo "Přejete si odstranit dočasné soubory inf a wav z aktuálního adresáře?"
read -n 1 y
echo
case $y in
      [yYaAjJ])
            mv *.cddb *.cdindex *.log $directory
            rm -f *.inf *.wav
            ;;
esac
exit 0
```

V prvním případě hraje roli filtru příkaz **tr,** který nahradí všechny výskyty jednotlivých znaků uvedených v prvních apostrofech jejich ekvivalenty z druhých apostrofů, takže z hranatých závorek udělá složené, z **&** bude **+**, rovnítko, svislítko, lomítko, obráceného lomítko (to bylo nutné zadat zdvojeně) a dvojtečka se změní v pomlčku, středník a čárka v tečku a z mezery se stane podtržítko. Zbývající dva filtry na jméno souboru jsou poháněny programem **sed**. I **sed** zde používáme jen k nahrazování. V každém páru uvozovek je příkaz pro jedno nahrazení. Nahrazuje se vždy řetězec mezi prvním a druhým lomítkem řetězcem mezi druhým a třetím lomítkem. Jeden řádek tak změní konstrukce typu _-_ na pomlčku a _._ na tečku. V tom druhém změníme vypsané znaky (!\"#$%‘()*<>?@{}) na prázdný řetězec, tedy odstraníme je. Potom je tam řádek velmi jednoduchý. Pomocí **echo** nám oznámí, která skladba se bude převádět. Vlastní enkódování zajišťuje příkaz **oggenc**, který převezme dříve zjištěné informace o názvu skladby, jménu autora a alba do ID3 tagů. Výstup (**-o**) nasměruje do adresáře stanoveného proměnnou **adresar** a jméno souboru vezme z námi pracně upravené proměnné **novy**. Tím jsme ukončili tu část podmínkové konstrukce **if**, ve které pro zpracovávaný hudební soubor existoval příslušný soubor inf stažený z CDDB. Za **else** bude následovat posloupnost příkazů, které se provedou, když tento soubor přítomný není. Nejprve nám skript oznámí, který soubor bude převádět, a to výpisem jeho pořadového čísla. Potom si připraví proměnnou **novy** tak, že vezme původní jméno souboru (**echo $soubor**) a rourou se pošle programu **sed**, který v něm nahradí koncovku wav za ogg. A nakonec už stačí jen spustit **oggenc** s parametrem vstupního a výstupního souboru. Pomocným **fi** zakončujeme celou část vlastního překódovávání wavu do oggu. Než se ukončením cyklu **for-do-done** vrhneme na další soubor, zvýšíme ještě hodnotu **soubor_idx** o jedničku. Využíváme k tomu zápis **$[$soubor_idx+1]**, který podrobněji rozebereme v některém z následujících dílů seriálu. Až skript projde a překóduje všechny wav soubory v aktuálním adresáři, přistoupí k závěrečně fázi, kde se nás zeptá, zda má odstranit pracovní soubory inf a wav. Potom pomocí **read** přečte právě jeden znak z klávesnice. Jedno **echo** pro odřádkování a je tu **case**, kde budeme vyhodnocovat obsah proměnné **y** načtené zmíněným **read**. Pokud **y** nabyde jednu z hodnot **yYaAjJ**, budou soubory inf a wav odstraněny. Soubory *.cddb, *.cdindex a *.log budou přesunuty do cílového adresáře (**$directory**). A je vymalováno. Ani to nebolelo.

19.díl

Máme za sebou opakování a před sebou posledních několik dílů. Základy dávno dobře ovládáme a také skripty již umíme psát. Zbývá několik specialitek jako třeba speciální parametry a vnitřní proměnné Bashe.

Speciální parametry

Jistě si vzpomenete na poziční parametry, které jsme probírali v 17. díle. Speciálními parametry (anglicky special parameters) rozumí Bash vyhrazené proměnné, které lze pouze číst. Jejich jména jsou jednoznaká a často se vztahují k pozičním parametrům. O některých jsem se ostatně již tehdy zmínil. Byly to **$0, $+, $@ a $#**. Ze speciálních parametrů týkajících se pozičních parametrů jsem tedy nezmínil **$_**. Ten zobrazí poslední argument (poziční parametr) naposled provedeného příkazu. Ne však v podobě, jak jsme jej zadali, ale po expanzi, tj. nahrazení zástupných znaků a jmen proměnných jejich hodnotami. O tom si ale povíme zase příště. Užitečné je znát výstupní hodnotu naposled spuštěného příkazu. K tomu slouží konstrukce **$?**. Pro zkoušku můžeme nechat provést příkaz např. **ls /home**, který by měl v linuxovém systému souborů být čitelný pro všechny, takže by nemělo dojít k chybě. Hned po jeho skončení zkuste zadat **echo $?** a mělo by vám být odpovězeno 0, tedy návratový kód značící absenci chyby. Jinak tomu bude, když budeme zkoumat výsledek chybového příkazu, dejme tomu **ls /homer**. Za předpokladu, že nemáte adresář **homer** v kořenu svého systému, skončí tento příkaz chybou. Výstupem **echo $?** bude návratový kód 1, tedy chyba. Více o návratovém kódu najdete v 10. dílu tohoto seriálu. Také se vám občas stává, že nějaký proces v textové konzoli nebo celá konzole takříkajíc zatuhne? V některých případech zbývá jako jediné řešení zabít (viz příkaz **kill** ve 2. díle) shell, který byl na dané konzoli spuštěn. Jak ale ve výpise běžících procesů (**ps aux**) poznat ten pravý? Jak to udělat, abychom nezabili zdravý Bash, ve kterém právě pracujeme? Stačí zjistit jeho PID. Zkuste **echo $$** a hned budete vědět, kterou větev nepodřezávat. Zmíním ještě vykřičník, tedy parametr **$!** — jeho hodnotou je také ID procesu. Ne však shellu, ale naposledy spuštěného příkazu na pozadí. Spouštění procesů na pozadí a jejich řízení Bashem jsme podrobněji probírali ve 2. dílu. Další poziční parametry najdete v manuálové stránce (**man bash**).

Proměnné shellu

Vedle pozičních a speciálních parametrů nastavuje Bash ještě spoustu dalších proměnných. Některé jsme si představili v 5. dílu. Jsou to proměnné běžného typu. Mají slovní jména (ač psaná velkými písmeny) a lze je i měnit. Jejich kompletní výčet najdete v dokumentaci k Bashi. Na ukázku jsem vybral proměnnou význačně pojmenovanou **BASH**. Udává absolutní cestu k binárnímu souboru, kterým byl právě běžící Bash spuštěn. Demonstrujme rozdíl od speciálního (pozičního) parametru **$0**, který udává cestu relativní:

```
[milarb@NBL1 ~]$ echo $BASH
/bin/bash

[milarb@NBL1 ~]$ ../../bin/bash
[milarb@NBL1 ~]$ echo $BASH
/home/milarb/../../bin/bash

[milarb@NBL1 ~]$ echo $0
../../bin/bash
```

Nejprve jsem stál ve svém domovském adresáři (**/home/milarb**) a dotázal jsem se na hodnotu proměnné **BASH**. Potom jsem spustil Bash voláním jeho binárky pomocí relativní cesty (**../../bin/bash**). V takto spuštěném Bashi (prompt se nijak nezměnil) jsem se pak tázal na hodnoty **$BASH** a **$0**. Zajímavůstkou je **BASH_COMMAND** vypisující zadání právě běžícího příkazu. Výsledkem **echo $BASH_COMMAND** je tak paradoxně **echo $BASH_COMMAND**. Proměnná **HISTCMD** zase udává, kolikátý v pořadí v rámci uchovávané historie provedených příkazů je ten právě prováděný. Pro zjištění **PID** rodičovského procesu, což může být vhodný doplněk **$!**, lze najít pod názvem **$PPID**. Podobně můžeme zjistit identifi kační číslo právě přihlášeného uživatele — **$UID**. A takto bychom mohli ve výčtu pokračovat. Příště si ukážeme, jaké triky Bash s proměnnými dokáže. Uzavřeli jsme výčet proměnných, které defi nuje Bash pro své vlastní potřeby. Za zmínku však stojí, že vedle nich Bash využívá ještě desítky dalších, tzv. systémových. Nejméně jednu z nich (**PATH**) jsme již blíže poznali.

20.díl

A je tady jubilejní dvacáté pokračování našeho seriálu. Nebude však oddechové. Naopak si ukážeme velmi zajímavou skupinu tzv. expanzních (rozšiřovacích) funkcí.

Expanze neboli rozšiřování

Tato označení mohou vyvolat mylnou představu o pravé podstatě funkcí, které se pod nimi skrývají. Nejde ani o snahu Bashe množit se a expandovat svévolně do našeho nebo i cizích počítačových systémů. Nejde ani o rozšíření o zásuvné moduly (tzv. **plug-ins**). Expanze v podání Bashe znamená soubor operací, které provede s obsahem příkazové řádky zadané uživatelem před tím, než začne vykonávat příkazy z ní vyplývající. Může jít např. o nahrazení zápisu **/home/bohdan/*txt** seznamem všech souborů v mém adresáři, jejichž jméno končí na **txt**. No a protože taková náhrada vede nejčastěji ke zvětšení zadaného zápisu (v našem případě dokonce z jednoho slova na několik), říká se jí **expanze** (česky rozšíření). Bash rozeznává sedm druhů rozšíření:

Rozšíření složených závorek (Brace Expansion)
Pokud si vzpomenete na hranaté závorky (viz 15. díl), tak ty složené fungují podobně. Pomocí hranatých jsme v řetězci na určitém místě mohli zadat více variant pro určitý znak, takže **/home/m[ai]rek** znamená **/home/mirek** a **/home/marek**. Složené závorky nabízejí možnost na určitém místě zadat více variant celé skupiny znaků. Například výsledkem příkazu **echo {ba,tc,z,a}sh** budou jména čtyř příkazových interpretrů: **bash tcsh zsh ash**. Tento mechanismus se často hodí při práci s adresáři, kde nám třeba zápis **/usr/X11R6/lib/X11/x{dm,edit,init,kb,server}** ušetří hodně psaní.

Vlnovková rozšíření (Tilde Expansion)
Je vcelku známá věc, že místo vlnovky (**~**) doplňuje Bash cestu k domácímu adresáři právě přihlášeného uživatele, tedy totéž co proměnná **$HOME**. V mém případě tak **~/bin** znamená **/home/milarb/bin**. Již méně se ví o konstrukcích s plus (**~+**) a mínus (**~-**). První z nich představuje aktuální pracovní cestu (**$PWD**), druhá pak předchozí pracovní cestu (**$OLDPWD**).

Rozšíření parametrů a proměnných (Parameter Expansion)
Zřejmě nejzajímavější oblast rozšíření. Dovoluje mimo jiné v řetězcích nahrazovat, vypisovat podřetězce apod. Toto téma si zaslouží vlastní díl a my mu věnujeme hned ten následující.

Nahrazení příkazů (Command Substitution)
Toto je také velmi užitečný nástroj. Umožňuje nám do příkazového řádku vložit řetězec, který je výsledkem provedení nějakého příkazu. Toho lze využít v případech, kdy potřebujeme výstup jednoho příkazu použít v rámci příkazu jiného, ale z nějakého důvodu to není možné provést přesměrováním či rourou. Typickým příkladem je situace, kdy potřebujete vědět, ze kterého RPM balíčku pochází určitý spustitelný soubor, např. **ls**. Nejprve musíme zjistit, kde binárka **ls** leží (**which ls**) a výsledek (**/bin/ls**) použít v dotazu **rpm -qf /bin/ls**. Pomocí nahrazení příkazů si však můžeme práci zjednodušit zápisem **rpm -qf $(which ls)**. Lze se setkat i se starším zápisem s pomocí obrácených apostrofů **— rpm -qf `which ls`**. Malý tip — místo **$(cat soubor)** je lepší použít **$(< soubor)**.

Aritmetická rozšíření (Arithmetic Expansion)
Ano, Bash umí počítat. Standardně zvládá sice jen celá čísla, ale to pro většinu případů postačuje. Komu ne, může si Bash buď překompilovat s podporou desetinné čárky nebo využít externích nástrojů (**bc**, **awk** apod.). Podobně jako v případě expanze proměnných i toto je téma na samostatný díl. Proto jen jednoduchý důkaz:

```
echo $((1 + 1))
```

Dělení na slova (Word Splitting)
Ve 14. díle jsme si představili proměnnou **IFS**, která uchovává tzv. oddělovače (většinou mezery, tabulátory apod.), s jejichž pomocí se dělí text na jednotlivá slova. Přesně to provádí Bash po dokončení rozšíření. Zkuste si třeba:

```
IFS=":" ; echo $(< /etc/passwd)
```

Rozšíření cest (Pathname Expansion)
Klasické doplňování jmen souborů, kde otazník (**?**) nahrazuje právě jeden znak a hvězdička (*****) libovolný nezáporný (tj. i nulový) počet znaků. Nahrazuje se i tečka, takže ***** znamená všechny soubory v daném adresáři, kdežto ***.*** pouze takové, které ve jméně mají alespoň jednu tečku. Ve jménech souborů lze použít i výčet znaků v hranatých závorkách (viz 15. díl).
Dnes jsme se na jednu skupinu funkcí Bashe podívali trochu z jiného úhlu – tak, jak ji prezentují autoři Bashe. Pro příští díl máme jasno. Rozebereme podrobně rozšíření parametrů a proměnných.

21.díl

Minule jsme si udělali přehled o zajímavé oblasti zvané rozšiřování neboli expanze. Jedním z typů expanzí bylo rozšíření parametrů a proměnných. To mimo jiné umožňuje práci s textovými řetězci přímo na úrovni Bashe. Této problematice se dnes budeme věnovat podrobněji.

Expanze parametrů a proměnných

Základním principem je nahradit jména parametrů (tímto termínem zde budeme označovat proměnné a poziční i speciální parametry) jejich hodnotou. Bash to udělá vždy, když před jméno parametru uvedeme znak dolar, např. **$PATH**, **$1**. Jak jsme si ale ukázali už dříve, je vhodné jména parametrů balit do složených závorek, aby nedošlo k jejich splynutí s okolním textem. Třeba místo **echo $cestabin/skript** se jistě hodí spíše **echo ${cesta}bin/skript**. Vedle prostého nahrazení jména parametru jeho hodnotou má však Bash v rukávu (či spíše ve složených závorkách) ještě tyto užitečné funkce:

• **${parametr:-slovo}** — za normálních okolností vypíše prostě hodnotu parametru (proměnné), ale pokud je tento prázdný, vypíše místo toho slovo (řetězec) uvedený za dvojtečkou a pomlčkou. Např. můžeme zadat:

```
echo "Desktop: "${DESKTOP:-žádný}
```

a Bash vypíše jméno pracovní plochy uživatele nebo slovo "žádný", pokud uživatel pracuje mimo grafiku.

• **${parametr:=slovo}** — funguje velmi podobně jako varianta s mínusem, takže také vypíše hodnotu parametru **a**, pokud je tento nulový (nenastavený), zobrazí místo něj zadaný řetězec (slovo). Přitom ale zároveň změní hodnotu parametru tak, že do něj dané slovo vloží.

• **${parametr:?slovo}** — další obdoba první varianty s tím, že slovo se vypíše v případě nulové hodnoty parametru, ne však jako běžný, nýbrž chybový výstup. Pokud to nastane v neinteraktivním shellu (např. při provádění skriptu), bude shell (i skript) ukončen.

• **${parametr:+slovo}** — a do čtvrtice zde máme opak mínusu. Pokud je parametr nulový nebo nenastavený, nic se nestane. Pokud však nějakou hodnotu obsahuje, zobrazí se slovo.

• **${parametr:offset}** — velké překvapení mne čekalo, když jsem za jméno proměnné zadal dvojtečku a nějaké malé číslo. Bash v takovém případě uřízne zadaný počet znaků ze začátku vypisovaného řetězce. Např. **echo ${SHELL}** vypíše **/bin/bash**, zatímco **echo ${SHELL:5}** jen bash, protože prvních 5 znaků ubral.

• **${parametr:offset:delka}** — pokud vám nestačí řezat pouze od začátku, můžete přidat další dvojtečku a za ni počet znaků, které chcete od offsetem zadaného místa zobrazit (nikoli uřezat zezadu). **echo ${SHELL:5:2}** potom ukáže jen "ba", tedy šestý a sedmý znak hodnoty **$SHELL**.

• **${!prefix*}** — zobrazí seznam jmen proměnných, která začínají na zadaný prefix. Chceme-li zjistit, jak egocentrický je Bash, pomůže nám v tom **echo ${!BASH*}**, vypisuje všechny proměnné se jménem začínajícím slovem BASH. Stejnou funkci plní **${!prefix@}**.

• **${!jmeno[*]}** — vypíše seznam položek pole (viz 5. díl seriálu). Zdůrazňuji seznam jmen položek, nikoli hodnot (to by udělalo **echo ${jmeno[*]}**). Jména položek (tj. označení v hranaté závorce) nemusí nabývat jen číselných hodnot, takže můžeme mít třeba pole **pc[lin]=10**, **pc[bsd]=3**, **pc[win]=0**.

• **${#parametr}** — udává délku parametru ve znacích, takže **echo ${#SHELL}** nám dá 9. **${#*} a ${#@}** ale zobrazí počet pozičních parametrů. Obdobně **${#jmeno[*]}** udává počet položek zadaného pole (**echo ${#pc[*]}** z předchozího bodu by vrátilo 3).

• **${parametr#slovo}** a **${parametr##slovo}** — slouží k uřezávání začátku hodnoty parametru, ne však podle délky (jako u offsetu), ale přímo zadaným řetězcem. Tento řetězec může navíc obsahovat zástupné znaky (*, ? apod.) Jedna mřížka značí co nejkratší řezání, dvě mřížky naopak co nejdelší. Kupř. **echo ${PATH#*:}** uřízne všechno od počátku řetězce až po první dvojtečku (včetně). Naopak **echo ${PATH##*:}** vezme vše až po poslední výskyt dvojtečky v seznamu cest.

• **${parametr%slovo}** a **${parametr%%slovo}** — obdoba předchozího, ale řeže se od konce. Abychom si to ukázali opět prakticky, použijeme speciální parametr **$0**, což by měla být cesta k aktuálnímu shellu (např. **/bin/bash**). **echo ${0%b*}** ukáže **/bin/**, protože celé slovo bash smazalo. Varianta **echo ${0%%b*}** vypíše jen **/**, protože smazala vše od prvního výskytu **b**.

• **${parametr/vyraz/retezec}** — a to nejlepší nakonec. Bash umí v řetězcích nahrazovat určený výraz zadaným řetězcem. Řekněme, že chceme při zobrazení zvýraznit všechny výskyty bin v proměnné **PATH** velkými písmeny.

Pak zadáme **echo ${PATH/bin/BIN}**. Chyba? Že se vám zvýraznil jen první výskyt? Máte pravdu. Pro náhradu všech musíme první lomítko zdvojit (**echo ${PATH// bin/BIN}**). Vyraz přitom nemusí být jen text, ale může obsahovat zástupné znaky (*,?,[] apod.)

Závěr

A to je k expanzi proměnných asi tak všechno. Je to mocná zbraň. Může vám ušetřit mnoho volání externích aplikací typu **cut**, **sed** nebo **awk**. A mocnější je ještě více, když si uvědomíme, že za výrazy a slova nemusíme dosazovat jen konkrétní řetězce, ale proměnné (např. **echo ${PATH:$odstup:$delka}**). Příště nám čeká mimo jiné počítání, tak si raději za domácí úkol zopakujte malou násobilku.

22.díl

Ve dvacátém díle jsme si představili oblast rozšiřování neboli expanzí. Jeden z typů expanzí (rozšíření parametrů a proměnných) jsme probírali minule. Dnes si ukážeme aritmetická rozšíření a celé téma expanzí uzavřeme.

Aritmetické expanze

Jak jsme si již řekli, expanze v Bashi znamená nahrazení nějakého zástupného řetězce jeho logickým významem (hodnotou) — např. hodnota proměnné, výsledek provedení nějakého příkazu nebo regulárního výrazu, doplnění cesty k souboru apod. V případě aritmetických expanzí jde o náhradu matematického výrazu jeho výsledkem. Výraz přitom zapisujeme do dvojitých závorek následovně: **$((vyraz))** — např. tak můžeme zadat **echo $((3 * 3))**.
Možná vám až dech vyrazí, jaké Bash umí výrazy. Vedle klasických **+, -, *, /** jsou to:

• **$((a++))**, resp. **$((a--))** — vypíše hodnotu proměnné a a následně ji zvýší (resp. sníží) o jedničku.

• **$((++a))**, resp. **$((--a))** — nejprve zvýší (resp. sníží) hodnotu proměnné a o jedničku a potom tuto novou hodnotu vypíše.

• **$((2**3))** — třetí mocnina dvou, tj. 2 na třetí (8).

• **$((7%2))** — výsledkem je zbytek po dělení 7/2. Jak jsem již zmínil, Bash umí pracovat jen s celými čísly, proto jako výsledek dělení 7/2 vrátí 3, zbytek (7%2) bude 1.

• **$((a==5))** — vyhodnotí, zda se hodnota proměnné a rovná číslu 5. Výsledkem tohoto výrazu je pravda (1) nebo nepravda (0). Obdobně funguje vyhodnocování nerovností (**!=, <=, >=, <, >**).

• **$((4<<1))** — provede bitový posun doleva, tedy 3. bit (hodnota 4) o 1, tj. na 4. bit (hodnota 8) a výsledkem bude 8.

• **$((4>>1))** — provede bitový posun doprava, tedy 3. bit (hodnota 4) o 1, tj. na 2. bit (hodnota 2) a výsledek je 2.

• **$((5&6))** — bitové **AND**, vypíše hodnotu společných bitů, v našem případě 5 (1. a 3. bit) a 6 (2. a 3. bit) je společný 3. bit, tedy hodnota 4.

• **$((5|6))** — bitové **OR**, vypíše hodnotu bitů, které se vyskytují alespoň na jedné straně, v tomto případě 5 (1. a 3.bit) a 6 (2. a 3. bit) jsou použity bity 1, 2 a 3, tedy hodnota 1+2+4=7.

• **$((5^6))** — bitové exkluzivní **OR**, vypíše hodnotu bitů, které se vyskytují pouze na jedné nebo pouze na druhé straně, zde tedy 5 (1. a 3. bit) a 6 (2. a 3. bit) má společný bit 3 a exkluzivní zůstávají 1 a 2, dohromady 3.

• **$((a&&b))** — logické **AND**, nabývá hodnoty 1, pokud jsou hodnoty a i b nenulové, jinak nabývá hodnoty 0.

• **$((a||b))** — logické **OR**, nabývá hodnoty 1, pokud alespoň jedna z hodnot a a b je nenulová, jinak nabývá hodnoty 0.

Další aritmetickou konstrukcí, kterou Bash zvládá, je kondiční operátor. Jeho zápis má podobu **vyraz1?vyraz2:vyraz3**. Funguje tak, že pokud platí podmínka ve **vyraz1** (výsledkem je nenulová hodnota), je proveden **vyraz2**, jinak se provede **vyraz3**. Např. **echo $((a==6?a:a++))** vypíše hodnotu **a** pokud se rovná 6, jinak také vypíše hodnotu **a** a poté ji zvýší o jedničku. Ještě zmíním skupinu konstrukcí označovaných za přiřazovací. V podstatě jde o většinu zde zmíněných operandů doplněných znakem rovnítko. Např. prostý zápis **a=5** přiřadí proměnné **a** hodnotu 5, zatímco **a+=3** vypíše hodnotu **a** poté, co k ní připočte číslo 3. K dispozici jsou vedle aritmetických (***= /= %= += -=**) také konstrukce logické (**<<= >>= &= ^= |=**).

Speciální znaky

V posledních třech dílech jsme se věnovali operacím, které provádí Bash se vstupem z příkazové řádky. Ukázali jsme si jednoduché i složitější konstrukce. Nepohovořili jsme však o třech znacích se zvláštním významem: obrácené lomítko (\), apostrof (,) a uvozovky ("). Bash tyto znaky zadané na příkazové řádce (tedy ne takové, které vznikly v důsledku expanzí) odstraní. Uvozovky slučují do jednoho bloku (pozičního parametru) skupinu znaků obsahující např. mezery, a přitom zaručují provedení expanzí (náhrada jmen proměnných za jejich hodnoty apod.) Apostrofy také slučují skupinu znaků dohromady, ale zabrání provedení případných expanzí, takže řetězec bude použit přesně tak, jak byl zadán. Obrácené lomítko potlačuje speciální význam bezprostředně následujícího zadaného znaku, kterým může být #, $, &, *, ale i závorka, mezera nebo další obrácené lomítko.
To je asi tak všechno, co Bash dělá se vstupem na příkazové řádce před samotným provedením zadaného příkazu. Protože tento seriál není překladem manuálové stránky ani referenční příručkou, neobsahuje úplně všechny mechanismy. Bádavý uživatel tak může najít mechanismy typu Process Substitution nebo bitová a logická negace. Příště si ukážeme, jak si v Bashi, resp. shellovém skriptu, nadefinovat vlastní funkce.

23.díl

Již po třiadvacáté se setkáváme na stránkách LinuxEXPRESu nad tématem Bash - nejrozšířenější textový příkazový interpretr v Linuxu. Probrali jsme všechna důležitá témata a dnes si představíme poslední- tvorbu vlastních funkcí.

Funkce

Jak jsme si předvedli, Bash má vlastních (interních) funkcí celkem dostatek. Navíc se můžeme ve skriptech odkazovat na jakýkoli linuxový program (**ls, ps, awk, wc, cut, ...**), jakých jsou v systému běžně nainstalovány stovky. Proč tedy definovat další funkce? Vlastní funkce se hodí v případě, kdy potřebujeme ve skriptu na více místech provést tutéž (resp. velmi podobnou) posloupnost příkazů. Ukažme si to na příkladu. Píšeme skript, který bude ukazovat různé informace o běžícím systému (třeba je bude číst z **/proc** nebo pomocí **uname, uptime, who** apod.) Na začátku každého výpisu chceme zobrazit hlavičku identifikující daný počítač. Např. seznam běžících procesů (**ps a**) začíná:

```
PID TTY STAT TIME COMMAND
4244 tty1 Ss+ 0:00 /sbin/mingetty tty1
4245 tty2 Ss+ 0:00 /sbin/mingetty tty2
```

My ale chceme, aby se před hlavičkou tohoto výpisu zobrazilo ještě: Počítač: Muj_Comp, čas: Pá lis 3 08:19:21 CET 2006

Toho dosáhneme např. trojicí příkazů:

```
echo -n "Počítač: $HOSTNAME, čas: "
date
echo
```

Pro potřeby našeho příkladu tedy uzavřeme tyto dva příkazy do funkce nazvané např. **tento_pocitac** a použijeme ji před každým výpisem. Definici funkce provedeme zápisem jejího jména následovaným prázdnou kulatou závorkou, otevřením složené závorky, posloupností příkazů na dalších řádcích a uzavřením složené závorky:

```
tento_pocitac()
{
   echo -n "Počítač: $HOSTNAME, čas: "
   date
   echo
}
```

Definice funkcí dáváme zpravidla na začátek skriptu. Potom je můžeme v rámci tohoto skriptu volat prostým uvedením jejich jména jako příkazu. Takto by vypadal skript, ve kterém danou funkci nadefinujeme a potom ji použijeme před výpisem volného místa na discích (**df**) a seznamem přihlášených uživatelů (**who**):

```
!#/bin/bash
tento_pocitac()
{
   echo -n "Počítač: $HOSTNAME, čas: "
   date
}
echo "Obsazenost disků:"
```

```
tento_pocitac
df
echo
echo "Přihlášení uživatelé:"
tento_pocitac
who
```

Takto samozřejmě vypadá skript dost primitivně. Pokud mu ale vytvoříte nějaké textové menu a příkazem **read** (14. díl) budete hlídat stisknutou klávesu, můžeme (např. pomocí **case** - viz 15. díl) vypisovat různé informace o systému, uvozené vždy jménem počítače a aktuálním časem. Posloupnost příkazů, které takto vytvořená funkce popisuje, je Bashem zařazena do skriptu na místo, kde bylo uvedeno jméno funkce. Z toho vyplývá, že (na rozdíl od mnoha programovacích jazyků) zde není nutné předávat funkci nějaké proměnné či jiné parametry a očekávat od ní návratové kódy nebo něco podobného. Návratovou hodnotou je jednoduše návratová hodnota posledního provedeného příkazu ve funkci.

Tečka

Symbolickou tečku za výčtem možností Bashe v tomto seriálu udělá příkaz tečka. Představme si situaci, že jsme zkušenými skriptéry a časem jsme si vytvořili několik vlastních funkcí, které ve svých skriptech často používáme. Bylo by jistě nepohodlné tento zdrojový text vkládat na začátek každého našeho skriptu. Např. by bylo dost pracné opravit nějakou chybu nebo vylepšit některou z již déle používaných funkcí. Toho se naštěstí bát nemusíme. Bash umožňuje na libovolné místo skriptu vložit posloupnost příkazů definovaných v jiném souboru. K tomu právě slouží příkaz tečka (**.**). Své oblíbené funkce si tak mohu vyčlenit a udržovat ve zvláštním souboru (třeba **funkce.sh**). Kdykoli pak chci do svého nového skriptu tyto funkce zařadit, zapíšu někam na jeho začátek . **funkce.sh** . Pokud by uvedený soubor neležel v aktuálním adresáři, bude hledán (jakožto soubor spustitelný) ve všech adresářích definovaných proměnnou **PATH**. Místo tečky lze použít také slovo **source**. Jde však o zcela jiný příkaz než **exec** (viz 3. díl)! Tečka (resp. **source**) pouze provede obsah zadaného souboru, jako by šlo o příkazy obsažené ve stávajícím skriptu.

Závěr

Tím jsme probrali vše, co jsem vám chtěl v Bashi předvést. On toho samozřejmě umí ještě mnohem více. Další podrobnosti najdete v manuálové stránce a jiné dokumentaci. Ve dvou zbývajících dílech si ukážeme probranou látku v praxi – na konkrétních (systémových) skriptech.

24.díl

V tomto předposledním díle seriálu o nejpoužívanějším textovém příkazovém interpretru – Bash – se budeme věnovat jeho konfiguračním souborům. Nejsou totiž ničím jiným než bashovými skripty.

Spuštění Bashe

Bash může být spuštěn v několika režimech. Nejdůležitější z nich jsou:
• přihlašovací (**login**) — pokud je spuštěn s parametrem **--login** nebo nultý poziční parametr začíná pomlčkou (není to např. jméno souboru);
• interaktivní — mezi parametry jsou jen přepínače (ne třeba jména souborů) jiné než **-c** nebo kdykoli, kdy je mezi parametry **-i** .

Interaktivní Bash je samozřejmě takový, který má vstup i výstup nasměrovaný na terminál, takže jej klasicky ovládáme klávesnicí a on zobrazuje výsledky práce a prompt na obrazovce. Opačnou situací je, když byl Bash spuštěn pouze pro provedení skriptu, tedy neinteraktivně. Na způsobu spuštění Bashe závisí, jakým způsobem čte své konfigurační soubory.
• Pokud je Bash spuštěn jako interaktivní a přihlašovací nebo jako neinteraktivní s přepínačem **--login**, provádí příkazy z **/etc/profile**. Potom Bash hledá postupně **~/.bash_profile**, **~/.bash_login** a **~/.profile**, a pokud existují, provede příkazy v nich uvedené.
• Pokud je Bash volán jako interaktivní, a není přitom přihlašovací, načte pouze soubor **~/.bashrc** a provede příkazy v něm.
• V okamžiku ukončování Bashe, který byl spuštěn jako přihlašovací, hledá a spouští příkazy v souboru **~/.bash_logout** .

Pokud některý ze jmenovaných souborů existuje, ale není přístupný pro čtení, Bash to ohlásí jako chybu. Výše popsané způsoby načítání konfiguračních souborů lze měnit různými přepínači. Existují také další, zde nepopsané, způsoby spouštění Bashe. Ukažme si tedy, jak vypadá typický start Bashe na mnou používaném Mandriva Linuxu 2006.

/etc/profile

Nejprve se provede systémový skript **/etc/profile**:

```
loginsh=1
```

Nastaví proměnnou **loginsh**, která se nám bude hodit v jiném souboru.

```
[ "$UID" = "0" ] && ulimit -S -c 1000000 > /dev/null 2>&1
```

Pro uživatele kromě roota vypne generování tzv. core souborů.

```
if ! echo ${PATH} |grep -q /usr/X11R6/bin ; then
   PATH="$PATH:/usr/X11R6/bin"
fi
```

Pokud operace "hledej řetězec **/usr/X11R6/bin** v obsahu proměnné **PATH**" má nulový návratový kód (tj. daný řetězec mezi systémovými cestami není), rozšíří se obsah **PATH** o tuto cestu.

```
if [ "$UID" -ge 500 ] && ! echo ${PATH} | grep -q /usr/games ; then
   PATH=$PATH:/usr/games
fi
```

Obdoba předchozího pro adresář **/usr/games**, ale provede se jen pro uživatele s **UID** vyšším než **500**, tj. normální uživatelé.

```
umask 022
USER=‘id -un‘
LOGNAME=$USER
MAIL="/var/spool/mail/$USER"
HISTCONTROL=ignoredups
HOSTNAME=‘/bin/hostname‘
HISTSIZE=1000
```

Nastaví **umask** a důležité systémové proměnné.

```
if [ -z "$INPUTRC" -a ! -f "$HOME/.inputrc" ]; then
   INPUTRC=/etc/inputrc
fi
```

Pokud není nastavena proměnná **INPUTRC** a neexistuje soubor **.inputrc** v domovském adresáři, bude nastavena hodnota **INPUTRC** na **/etc/inputrc** .

```
NLSPATH=/usr/share/locale/%l/%N
export PATH PS1 USER LOGNAME MAIL HOSTNAME INPUTRC NLSPATH
export HISTCONTROL HISTSIZE
```

Nastavení další proměnné a vyexportování důležitých proměnných do systému.

```
for i in /etc/profile.d/*.sh ; do
   if [ -x $i ]; then
    . $i
   fi
done
unset i
```

Postupné spuštění všech spustitelných souborů s koncovkou **.sh** z adresáře **/etc/profile.d/**

bash_profile

Dále se provede soubor **.bash_profile** z domovského adresáře uživatele:

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

Pokud existuje soubor **.bashrc** v domovském adresáři, je spuštěn (viz níže).

```
PATH=$PATH:$HOME/bin
```

Obsah proměnné **PATH** je rozšířen o podadresář **bin** domovského adresáře uživatele.

```
export PATH
unset USERNAME
```

Vyexportování nového obsahu proměnné **PATH** a zrušení proměnné **USERNAME**. Dále lze do tohoto souboru přidat seznam příkazů a nastavení dle přání uživatele.

/etc/bashrc

Dále by měly být provedeny skripty **~/.bash_login** a **~/.profile**, které ovšem Mandriva ve standardní konfiguraci nepoužívá. Z **.bash_profile** byl ovšem spuštěn ještě **.bashrc**, který nedělá nic jiného než, pokud tento existuje, spustí **/etc/bashrc**.

```
if [ "‘id -gn‘" = "‘id -un‘" -a ‘id -u‘ -gt 99 ]; then
 umask 002
else
 umask 022
fi
```

Pokud má uživatel **UID** větší než **99**, tj. většinou běžní uživatelé, nastaví se mu **umask 002**, jinak **022**.

```
if [ "$PS1" ]; then
```

Následující příkazy se provedou pouze tehdy, je-li Bash spuštěn interaktivně.

```
case $TERM in
   xterm*)
    PROMPT_COMMAND='echo -ne "\033]0;${USER}@${HOSTNAME}:${PWD}\007"'
     ;;
    *)
     ;;
esac
```

Pokud je terminálem **xterm**, nastaví se příkaz pro **prompt**.

```
[ "$PS1" = "\\s-\\v\\$ " ] && PS1="[\u@\h \W]\\$ "
```

Nastaví novou podobu promptu **PS1**.

```
if [ -z "$loginsh" ]; then
```

Podle hodnoty proměnné **loginsh** (nastavené v **/etc/profile**) zjistíme, zda běží Bash jako přihlašovací.

```
if [ -n "${BASH_VERSION}${KSH_VERSION}${ZSH_VERSION}" ]; then
   for i in /etc/profile.d/*.sh; do
    if [ -x $i ]; then
     . $i
    fi
   done
fi
```

Pokud není nastavena ani jedna z proměnných **BASH_VERSION, KSH_VERSION** nebo **ZSH_VERSION**, budou postupně spuštěny všechny spustitelné soubory s koncovkou **.sh** z adresáře **/etc/profile.d/** . Následuje ukončení dřívějších podmínek a zrušení proměnné **loginsh**.

```
   fi
fi
unset loginsh
```

Závěr – start

A teprve potom je Bash nastartován. Nutno znovu zdůraznit, že zde uváděný proces se může v různých distribucích značně lišit. Příště, než se s Bashem definitivně rozloučíme, si ukážeme látku probíranou v dřívějších dílech na rozboru jednoho ze systémových skriptů.
Jak již bylo řečeno v Úvodu, všechny výše jmenované konfigurační soubory mají formát bashového skriptu. V některých případech může jít jen o přiřazení hodnot nějakým proměnným, jindy to bude složitý skript s funkcemi, smyčkami a podmínkami.
Ještě bych chtěl připomenout, že **~/.bashrc**, a tedy i **/etc/bashrc**, se spouští také samostatně (bez **/etc/profile** a **~/.bash_profile**), pokud je Bash volán jako interaktivní a přitom ne přihlašovací.

25.díl

Tak a je tu poslední díl seriálu o textovém příkazovém interpretru Bash. Podobně jako minule si ukážeme dříve probraná támata na příkladu reálného skriptu. Tentokrát jsem vybral systémový skript používaný programem **Init**.

```
1       #!/bin/bash
2       . /etc/init.d/functions
3       . /etc/sysconfig/crond
4       t=${CRON_VALIDATE_MAILRCPTS:-UNSET}
5       [ "$t" != "UNSET" ] && export CRON_VALIDATE_MAILRCPTS="$t"
6       prog="crond"
7       start() {
8                       echo -n $"Starting $prog: "
9                       if [ -e /var/lock/subsys/crond ]; then
10                              if [ -e /var/run/crond.pid ] && [ -e /proc/`cat
/var/run/crond.pid` ]; then
11                                      echo -n $"cannot start crond: crond is already
running.";
12                                      failure $"cannot start crond: crond already running.";
13                                      echo
14                                      return 1
15                              fi
16                      fi
17                      daemon crond $CRONDARGS
18                      RETVAL=$?
19                      echo
20                      [ $RETVAL -eq 0 ] && touch /var/lock/subsys/crond;
21                      return $RETVAL
22      }
23      stop() {
24                      echo -n $"Stopping $prog: "
25                      if [ ! -e /var/lock/subsys/crond ]; then
26                              echo -n $"cannot stop crond: crond is not running."
27                              failure $"cannot stop crond: crond is not running."
28                              echo
29                              return 1;
30                      fi
31                      killproc crond
32                      RETVAL=$?
33                      echo
34                      [ $RETVAL -eq 0 ] && rm -f /var/lock/subsys/crond;
35                      return $RETVAL
36      }
37      rhstatus() {
38                      status crond
39      }
40      restart() {
41                      stop
42                      start
43      }
44      reload() {
45                      echo -n $"Reloading cron daemon configuration: "
46                      killproc crond -HUP
47                      RETVAL=$?
```

```
48                  echo
49                  return $RETVAL
50      }
51      case "$1" in
52              start)
53                      start
54                      ;;
55              stop)
56                      stop
57                      ;;
58              restart)
59                      restart
60                      ;;
61              reload)
62                      reload
63                      ;;
64              status)
65                      rhstatus
66                      ;;
67              condrestart)
68                      [ -f /var/lock/subsys/crond ] && restart || :
69                      ;;
70              *)
71                      echo $"Usage: $0 {start|stop|status|reload|restart|condrestart}"
72                      exit 1
73      esac
```

Spouštění systému

Nebudu se zde podrobně zabývat problematikou spouštění operačního systému. Pro naše potřeby postačí vědět, že po zapnutí počítače naběhne BIOS, ten zavolá správce spouštění (boot manager) a v něm si vybereme požadovaný operační systém. V případě Linuxu se načte jádro, osahá si hardware a předá činnost hlavnímu procesu — programu **Init**.
Init se stará o spouštění všech dalších programů. Nejdříve to jsou různé služby běžící na pozadí (démoni), nakonec potom správce přihlášení pro textové nebo grafické prostředí. Nás budou zajímat právě démoni. Ty totiž **Init** nespouští přímo, ale pomocí skriptů uložených v adresáři **/etc/init.d/**. Ukážeme si, jak pracuje spouštěcí skript pro démona **crond**. Ten se stará o opakované provádění nastavených operací v určeném čase (např. jednou denně nebo týdně).
Tento skript najdete ve svém systému pravděpodobně pod jménem **/etc/init.d/crond**.

Rozbor skriptu

Příklad skriptu **/etc/init.d/crond** z Fedora Core 6 je uveden v výše. Pro usnadnění orientace má očíslované řádky, takže jej nebudu kouskovat přímo do textu, ale podle těchto čísel se na jeho jednotlivé části odvolávat.
První řádek určuje, že skript má být proveden Bashem. Druhý řádek volá skript **/etc/init.d/functions**, pomocí něhož se nadefinují některé funkce, které budou dále použity (viz 23. díl). Podobně řádek 3 volá skript, který nastaví některé proměnné (viz 4. díl) pro konfiguraci Cronu.
Čtvrtý řádek definuje proměnnou **t** a využívá k tomu expanzi parametrů (viz 21. díl). Pokud je (např. ze souboru **/etc/sysconfig/crond**) nastavena proměnná **CRON_VALIDATE_MAILRCPTS**, převezme **t** její hodnotu. Pokud tomu tak není, vloží se do **t** řetězec **UNSET** (anglicky nenastaveno).
V pátém řádku se nachází jednoduchá podmínka s testem (viz 10. díl). Pokud se obsah proměnné **t** nerovná řetězci **UNSET**, bude proměnná **CRON_VALIDATE_MAILRCPTS** nastavena na hodnotu **t** a zpřístupněna ostatním procesům pomocí **export**. Šestý řádek prostě nastavuje proměnné **prog** hodnotu **crond**.
Na řádcích 7 až 22 se nachází definice funkce (viz 23. díl) **start()**, tj. posloupnost příkazů prováděných při spouštění C**ronu**. Nejprve (řádek 8) se vypíše hláška **Starting crond:**, pak (řádky 9-16) následuje podmínka (viz 11. díl). Ta se provede jen za předpokladu, že existuje soubor **/var/lock/subsys/crond** (viz 10. díl). Pokud se podmínka vykonává, dostáváme se k další, která leží na řádcích 10 až 15. Hodnotí, zda existuje soubor **/var/run/crond.pid** (vytvářený
při spouštění démona). Pokud ano, potom ještě (**&&**), zda existuje v adresáři **/proc/** soubor (nebo adresář), jehož jméno je stejné jako obsah souboru **/var/run/crond.pid**. K tomu bylo použito mechanismu nahrazování příkazů (viz 20. díl). Tím si skript ověří, zda už náhodou nějaký Cron neběží. Pokud **cron** již běží, vypíše se o tom varovná hláška (řádek 11). Dál se provede funkce **failure** (anglicky selhání, řádek 12) definovaná ve skriptu **/etc/init.d/functions**, která zapíše vzniklý problém do logu. V řádku 13 se pomocí **echo** odřádkuje a 14. řádek nastaví návratovou hodnotu na **1** (viz 10. díl).

V řádku 17 dochází konečně k samotnému spuštění Cronu. Slouží k tomu funkce (viz 23. díl) **daemon** definovaná v **/etc/init.d/functions**. Jako poziční parametry (viz 17. díl) přijímá jméno spouštěného démonu a seznam jeho parametrů, v našem případě obsah proměnné **$CRONDARGS** definované v konfiguračním souboru **/etc/sysconfig/crond**.
V řádku 18 nabývá proměnná **RETVAL** hodnotu posledního návratového kódu (**$?** — viz 19. díl).
Echo na řádku 19 odřádkuje a dostáváme se k podmínce (viz 10. díl) na řádku 20. Ta v případě, že **RETVAL** má hodnotu 0, vytvoří prázdný soubor (zámek) **/var/lock/subsys/crond**. Definice funkce **start()** (viz 23. díl) končí předáním návratové hodnoty **RETVAL**.
Na řádcích 23 až 36 je definována funkce **stop()**, která se funkci **start()** dosti podobá. Na řádku 24 se vypíše hlášení o zastavování Cronu. Řádky 25 až 30 jsou podmínkou prováděnou v případě, že neexistuje soubor (zámek) **/var/lock/subsys/crond**. Tehdy se vypíší varovné hlášky, že Cron nelze zastavit, když neběží, a vrátí se chybová hodnota 1. Řádek 31 volá další funkci definovanou skriptem **/etc/init.d/functions**. **Killproc** se postará o korektní zastavení zadaného démonu, v našem případě **crond**. Pak následuje přiřazení hodnoty návratového kódu proměnné **RETVAL**. Pokud je to 0 (bez chyby), tak se v řádku 34 odstraní soubor (zámek) **/var/lock/subsys/crond**.
Funkce **rhstatus()** definovaná na řádcích 37 až 39 volá pouze funkci status definovanou ve skriptu **/etc/init.d/functions**. Ta jen vypíše, zda zadaný démon (v našem případě samozřejmě **crond**) běží.
Na řádcích 40 až 43 najdeme definici funkce **restart()**. Je velmi prostá, neboť prostě vykoná posloupnost dříve definovaných funkcí **stop()** a **start()**.
Řádky 44 až 50 ukrývají obsah funkce **reload()**. Ta, na rozdíl od **restart()**, Cron neukončí, ale pouze jej donutí znovu načíst konfigurační soubory. Nejprve o tom vypíše hlášku v řádku 45. Potom zavolá funkci **killproc**, která procesu jménem **crond** pošle signál (viz 2. díl) **-HUP**, což je ekvivalent **kill -1** (viz **man 7 signal**). Pak už následuje jen definice návratového kódu.
Tím jsme ukončili definici funkcí ve skriptu a dostáváme se k jeho hlavní části (která se např. v jazyce C označuje jako **Main**). Ta se rozkládá na řádcích 51 až 73 a je tvořena jedinou konstrukcí **case** (viz 15. díl). Jednotlivé části **case** budou provedeny v závislosti na hodnotě prvního pozičního parametru (**$1** — viz 17. díl). Záleží tedy na tom, s jakým parametrem byl námi zde analyzovaný skript **/etc/init.d/crond** spuštěn. Jak se asi dovtípíte, v závislosti na tom může sloužit nejen ke spouštění, ale i zastavování a dalším způsobům ovládání **Cronu**. Pokud byl skript volán s parametrem **start** (52. řádek), bude provedena funkce **start()** (53. řádek) a sekci ukončuje dvojice středníků (54. řádek). Podobně následují situace pro hodnoty pozičního parametru **stop**, **restart**, **reload** a **status**.
Na řádku 68 je složená podmínka (viz 10. díl) pro parametr **condrestart**. Pokud existuje soubor (zámek) **/var/lock/subsys/crond**, provede se **restart**. Pokud ne (**||**), bude vrácen nulový návratový kód, k čemuž slouží nicnedělající interní příkaz dvojtečka (**:**). Před uzavřením konstrukce **case** zbývá vypořádat se s případy, kdy byl skript spuštěn s jiným parametrem, než jaké jsme dosud jmenovali. K tomu slouží sekce v řádcích 70 až 72 uvozená hvězdičkou. V tom případě se vypíše krátká informace o použití (anglicky Usage) se jménem skriptu (**$0** — viz 17. díl) a seznamem přípustných parametrů a skript je zakončen návratovým kódem **1**.
A to je opravdu vše. Pravidelným čtenářům děkuji za přízeň. Všem zájemcům o Bash přeji mnoho úspěchů při používání tohoto mocného nástroje a tvorbě vlastních skriptů.

Odkazy

http://www.gnu.org/software/bash **BASH–GNU Projekt**

http://www.abclinuxu.cz/clanky/show/46130 **Seriál Bash**

http://www.root.cz/clanky/drobnosti-ze-shelloveho-zapisniku/ **Drobnosti ze shellového zápisníku**

http://tldp.org/LDP/Bash-Beginners-Guide/html/ **Bash Guide for Beginners**

http://tldp.org/LDP/abs/html/ **Advanced Bash-Scripting Guide**