Jihočeská univerzita v Českých Budějovicích

Pedagogická fakulta

Katedra informatiky

Tvorba balíku skriptů pro správu uživatelských účtů v UNIXu

bakalářská práce

Autor: **Vladimír Rajnyš**

Vedoucí bakalářské práce: **Mgr. Jiří Pech, Ph.D.**

České Budějovice 2007

Prohlášení:

Prohlašuji, že jsem bakalářskou práci na téma *Tvorba balíku skriptů pro správu uživatelských účtů v UNIXu* vypracoval samostatně s použitím literatury, která je citována v seznamu literatury na konci bakalářské práce.

V Českých Budějovicích dne 27.4.2007

Vladimír Rajnyš

Abstrakt

Hlavním cílem této bakalářské práce je vytvořit sadu skriptů, které mohou výrazně pomoci administrátorům unixových serverů se správou uživatelů a popsat praktické zkušenosti s používáním těchto skriptů. Dalším cílem je popis vhodných skriptovacích nástrojů, které může využít administrátor unixového serveru.

Vytváření skriptů probíhalo v prostředí linuxu distribuce Debian. Jako skriptovací jazyky jsem používal Bash a Perl, protože jsou standardní součástí všech hlavních linuxových distribucí.

Tato práce je rozdělena do tří částí. V první části jsou popisovány vhodné skriptovací nástroje, které lze v unixových operačních systémech použít. V druhé části se zabývám popisem skriptů, které jsem vytvořil a které výrazně zjednodušily správu uživatelů v systému. V poslední části popisuji praktické zkušenosti s používáním těchto skriptů.

Abstract

The main aim of this Bachelor Thesis is to create a set of scripts, that can be very helpful to administrators of unix servers when administrating users, and describe practical experience with the use of those scripts. Second aim is to describe suitable scripting tools, which can be used by unix server administrator.

Creation of the scripts took place in the environment of Debian linux distribution. Bash and Perl were used as scripting languages, because they are standard elements of each prime linux distribution.

This Thesis is divided into three parts. In the first part there are described suitable scripting tools that are possible to be used in unix operation systems. In the second part I describe the scripts created by myself, which can simplify administration of users in a system. In the last part I describe practical experience with the use of those scripts.

Obsah

Úvod

Každý správce jakéhokoliv serveru musí řešit otázky správy uživatelů v systému. Musí najít nejvhodnější způsob zálohování uživatelských dat, kontrolovat, zda uživatelé nepřekračují přidělené diskové kvóty apod. K tomu, aby mohl tyto činnosti vykonávat si musí připravit nějaké nástroje, které mu tuto práci zjednodušší a zrychlí.

Cílem této bakalářské práce je popsat vhodné skriptovací nástroje v Unixu, které umožňují správcům takového serveru zautomatizovat často prováděné úlohy a zefektivnit správu uživatelských účtů. Dalším cílem je vytvořit balík skriptů, které umožní snažší manipulaci s uživatelskými účty.

Pro vytváření skriptů jsem zvolil prostředí linuxu (distribuce Debian), protože je nejčastěji používaným unixem. První část této bakalářské práce popisuje nástroje Bash a Perl, které jsou pro takovou činnost velmi vhodné a často používané. Tento popis má být hlavně popisem konstrukcí, které lze běžně v těchto nástrojích používat, nemá být úplným popisem těchto nástrojů. V rámci popisu skriptovacího jazyka Bash se také zmiňuji o nástrojích Sed a Awk, které jsou v tomto jazyce nejvíce používanými pro práci s textem. Tento popis obsahuje také to, jak lze zachytávat signály, které posílá operační systém procesům.

Ve druhé části je podrobný popis funkcí jednotlivých skriptů, obsažených v balíčku. V tomto popise je vysvětleno, jakým způsobem tyto skripty pracují, jakých nástrojů využívají a jaký je jejich výstup. Mnoho z těchto skriptů lze využít ve spolupráci programu cron, který umožňuje jejich pravidelné spouštění. To umožňuje zautomatizovat mnoho činností a včas upozornit správce serveru v případě neočekávaného problému.

Poslední část této bakalářské práce popisuje praktické zkušenosti s používáním tohoto balíčku skriptů.

BASH

Úvod

BASH (Bourne Again Shell) je dnes dostupný v každé standardní distribuci linuxu. Je to interpret příkazů a skriptovací jazyk. Znalost tohoto jazyka je samozřejmostí každého správce linuxového serveru.

Každý skript musí začínat hlavičkou ve tvaru:

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

Je to instrukce, která říká operačnímu systému, který interpret se má použít.

Jako v každém běžném skriptovacím jazyku lze psát do zdrojového kódu také komentáře, které začínají znakem #.

Je dobrým zvykem pojmenovávat skripty psané v shellu s příponou *.sh*.

Proměnné

Proměnné jsou místa v paměti označená identifikátorem. Proměnné se deklarují přiřazením hodnoty. Pokud již nechceme dále proměnnou používat, tak můžeme použít příkaz *unset.*

Příklad deklarace proměnné:

```
#!/bin/bash
prom1=1
readonly prom2=2
```

První řádek je hlavička. Druhý řádek je deklarace tzv. lokální proměnné. Při deklaraci nesmí být kolem rovnítka žádné mezery. Na třetím řádku je ukázána deklarace proměnné jen pro čtení pomocí příkazu *readonly*. Proměnné můžeme také exportovat do podřízeného shellu příkazem *export*.

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

```
prom1=1                 # deklarace proměnné prom1 na hodnotu 1
export $prom1           # export proměnné prom1 do podřízeného
                        # shellu
```

Z tohoto příkladu je také vidět, jak přistupujeme k proměnným. Pokud do proměnné chceme přiřadit nějakou hodnotu, tak jej píšeme *název_proměnné=hodnota_proměnné*. Pokud chceme dále proměnnou používat, tak před ní musíme napsat znak *$*. Např. e*xport $název_proměnné*.

Při spuštění každého skriptu jsou již deklarované některé proměnné.

$HOME	domovský adresář uživatele, který spustil skript
$PATH	seznam adresářů, ve kterých se mají hledat příkazy;tyto adresáře jsou oddělené dvojtečkami
$0	název spuštěného skriptu
$#	počet předaných argumentů skriptu
$$	id procesu skriptu
$IFS	seznam znaků, které oddělují slova
$*	seznam všech argumentů předaných skriptu, argumenty jsou oddělené prvním znakem v proměnné $IFS
$@	seznam všech argumentů předaných skriptu, ale nepoužívá $IFS
$1 … $9	seznam argumentů předaných skriptu, pokud jich bylo předáno více můžeme využít příkaz *shift*

V shellu bash lze také pracovat s jednorozměrnými poli. Vícerozměrná pole lze simulovat vložením proměnné typu pole jako položku jiného pole. Deklarace pole se provádí podobně jako deklarace ostatních proměnných přiřazením. Index pole se zapisuje do hranatých závorek.

```
pole[0]="hodnota první položky pole"
pole[1]="hodnota druhé položky pole"
# jiný způsob zápisu pole
pole=("hodnota první položky" "hodnota druhé položky")
```

Chceme-li vypsat všechny položky pole, můžeme použít konstrukci *${pole[*]}*.

Expanze proměnných

Expanze proměnných je vrácení její hodnoty ve výrazu.

```
# pokud je proměnná promenna definovaná, tak se vypíše její hodnota
echo ${promenna}
# pokud je promenna nedefinovaná nebo prázdná, tak se vytiskne
# slovo, jinak se vytiskne hodnota proměnné
echo ${promenna:-slovo}
# pokud je proměnná nedefinovaná, tak se jí přiřadí hodnota a
# vytiskne se slovo
echo ${promenna:=slovo}
# pokud je proměnná definovaná, tak se vytiskne slovo
echo ${promenna:+slovo}
# vytiskne délku řetězce uloženého v proměnné
echo ${#promenna}
# odstraní od konce nejkratší vyhovující příponu
echo ${promenna%slovo*}
# odstraní od konce nejdelší vyhovující příponu
echo ${promenna%%slovo*}
# odstraní od konce nejkratší vyhovující předponu
echo ${promenna#slovo*}
# odstraní od konce nejdelší vyhovující předponu
echo ${promenna##slovo*}
```

Uvozovky a apostrofy

V shellu bash se píší uvozovky k označení řetězce, který se může skládat z několika slov. Pokud bychom napsali *mkdir ahoj svete*, tak se vytvoří dva adresáře *ahoj* a *svete*. Pokud napíšeme

mkdir "ahoj svete", tak vznikne pouze jeden adresář. Pokud bude součástí řetězce proměnná, tak se na její místo dosadí její hodnota.

Apostrofy mají podobnou funkci jako uvozovky. Rozdíl spočívá v tom, že je řetězec interpretován přesně tak, jak byl zadán.

```
x=5
echo "Hodnota x je $x"          # vytiskne: Hodnota x je 5
echo 'Hodnota x je $x'          # vytiskne: Hodnota x je $x
```

V shellu bash můžeme také psát obrácený apostrof. Jeho funkce je taková, že se napřed vyhodnotí výraz, který tyto obrácené apostrofy uzavírají. Často nahrazuje tuto konstrukci:

```
x=5
x=$(expr $x + 1)
```

Druhý řádek můžeme přepsat takto:

```
x=`expr $x + 1`
```

Roury a přesměrování

Roura slouží k přesměrování výstupu jednoho programu na vstup druhého. Zapisuje se symbolem |. Například seznam všech uživatelů v systému můžeme získat pomocí tohoto příkazu:

```
sed "s/:/ /g" /etc/passwd | awk '{print $1}'
```

Příkaz sed nahradí všechny znaky dvojtečka mezerou a takto upravený soubor předá na vstup programu awk, který vypíše první slovo z každého řádku.

K přesměrování vstupu a výstupu slouží tyto operátory:

> přesměruje standardní výstup do souboru (pokud tento soubor již existuje, tak bude přepsán)

>> přesměruje standardní výstup na konec souboru

< presměruje standardní vstup do souboru

Často je také potřeba přesměrovat standardní chybový výstup na standardní výstup. V takovém případě můžeme před operátor přesměrování použít deskriptor souboru.

```
# přesměrování standardního výstupu do souboru „soubory“
ls > soubory
# přesměrování chybového výstupu do souboru chyby
cat /bin 2> chyby
# přesměrování chybového výstupu na standardní výstup
cat /bin 2>&1
# záměna standardního a chybového výstupu
cat /bin > chyby 3>&2 2>&1 1>&3
```

Podmínky

Podmínka je jednou z nejčastějších příkazů. K provádění podmíněných příkazů používáme příkazů *if*, *elif*, *else* a *fi* nebo *case*, *in* a *esac*. Např.:

```
if test ${USER} == "root“; then
      echo "Můžeš dělat cokoliv"
elif [ ${USER} == "jiny_uzivatel" ]; then
      echo "Nemůžeš dělat cokoliv"
else
      echo "Nesprávný uživatel"
      exit
fi
```

Tento příklad interpretujeme takto: Jestli má proměnná *$USER* hodnotu *root*, potom vypiš *Můžeš*

dělat cokoliv, jinak jestliže má proměnná *$USER* hodnotu *jiny_uzivatel*, potom vypiš *Nemůžeš dělat cokoliv*, jinak vypiš *Nesprávný uživatel* a ukonči program. Hranaté závorky v druhé podmínce nahrazují příkaz test. Kolem těchto závorek musí být mezery. Každá podmínka musí končit příkazem *fi*.

Tento příklad by se dal také upravit pomocí příkazu case.

```
case "$USER" in
      "root" )
                  echo "Můžeš dělat cokoliv"
      ;;
      "jiny_uzivatel" )
                  echo "Nemůžeš dělat cokoliv"
      ;;
      * )
                  echo "Nesprávný uživatel"
                  exit
      ;;
esac
```

Příkaz *case* lze plně nahradit příkazem if, ale v některých případech je lépe čitelný.

Cykly

V bourne again shellu můžeme použít několik druhů cyklů:

1. *while ... do ... done*
2. *until ... do ... done*
3. *for ... in ... do ... done*

While … do … done

Tento cyklus lze použít ve všech případech, ale není vždy z hlediska přehlednosti a pracnosti

nejvhodnější.

```
while true; do
      echo Pro ukončení programu stiskněte Ctrl+C
      sleep 5
done
```

Toto je tzv. nekonečný cyklus, který lze zastavit pouze ukončením programu pomocí kombinace kláves Ctrl+C. Cyklus je tvořen klíčovým slovem *while*, za kterým následuje podmínka. Pokud je podmínka splněna, tak se vykoná tělo cyklu, které je zakončeno klíčovým slovem *done*. Tento program bude každých 5 sekund vypisovat na obrazovku text *Pro ukončení programu stiskněte Ctrl+C*, dokud uživatel tento skript nepřeruší. Program *true* v podmínce cyklu se často nahrazuje dvojtečkou, která má stejný význam, ale nespouští další proces.

Until … do … done

Cyklus until je velmi podobný cyklu while s tím rozdílem, že until provádí tělo cyklu dokud podmínka není splněna.

```
x=5
until [ ${x} -eq 10 ]; do
      echo x není rovno 10
      x=$(($x+1))
done
```

Tento cyklus bude probíhat dokud proměnná *$x* nebude rovna 10. Jak zde můžeme vidět, tak v cyklech stejně jako v podmínkách často využíváme programu test. Zápis *x=$(($x+1))* nahrazuje příkaz *expr*. Stejného výsledku dosáhneme i tímto zápisem *x=$(expr $x + 1)*.

For … in … do … done

Struktura for se nejčastěji používá k procházení seznamu.

```
for cislo in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20; do
        case ${cislo} in
                5 )   continue
                ;;
                13 )  break
        esac
        echo ${cislo}
done
```

Tento cyklus prochází čísla od 1 do 20 a ukládá je do proměnné *cislo*. Pokud najde číslo 5, tak přeskočí zbytek těla cyklu a pokračuje následující hodnotou. Pokud najde číslo 13, tak přeruší celý cyklus a na čísla 14-20 se nedostane. Výstupem tohoto cyklu budou pod sebou vypsané hodnoty 1,2,3,4,6,7,8,9,10,11,12.

Operátory

Matematické:

=	přiřazení
id++ ; id-- ; ++id ; --id	inkrementace a dekremenatce; pokud je operátor uveden za názvem proměnné, pak se napřed vyhodnotí výraz, jehož je součástí, a až poté se proměnná inkrementuje (popř. dekrementuje) a naopak
+ ; - ; * ; / ; %	sčítání, odečítání, násobení, dělení, modulo
**	umocnění
<= ; >= ; < ; >	operátory porovnání - menší nebo rovno, větší nebo rovno , je menší než, je větší než
== ; !=	operátory porovnání - je rovno, není rovno
,	operátor čárky, spojuje několik aritmetických operací, ale jako výsledek celého výrazu je vrácen pouze poslední z nich
+= ; -= ; *= ; /= ; %=	a+=2 => stejné jako a=a+2

Logické:

&&	logické a
||	logické nebo
výraz1 **?** výraz2 **:** výraz3	Ternární operátor, pokud je vyhodnocen výraz1 jako pravdivý, tak se provede výraz2, jinak se provede výraz3

Bitové:

&	Bitové a

| Bitové nebo
<< ; >> Bitový posun

Funkce a návratové hodnoty

Hlavním úkolem funkcí je činit zdrojový kód přehlednější a zabránit duplicitě kódu. Jejich zpracování je obvykle mnohem rychlejší, neboť si je shell udržuje již předzpracované v paměti. Každá funkce musí být deklarována ještě předtím než bude použita. Deklarace funkce se skládá z názvu funkce, kulatých závorek a bloku příkazů. Blok příkazů je označen složenými závorkami.

Příklad:

```
1. #!/bin/bash
2. ahoj()
3. {
4.   echo "Ahoj světe."
5. }
6. ahoj
```

Tento příklad vypíše na standardní výstup text *Ahoj světe.* Na řádcích 2-5 je deklarována funkce, jejíž název je *ahoj*. Na řádcích 3-5 je blok příkazů uzavřený ve složených závorkách. Na řádku 6 je vidět, jak tuto funkci voláme. Funkce můžeme volat i s argumenty tím způsobem, že je napíšeme za název funkce. Při volání funkce jsou tzv. poziční parametry skriptu ($*,$@,$#,$1,$2,...) nahrazeny paramtery funkce.

```
#!/bin/bash
vypis()
{
      echo $1
}
vypis "Ahoj světe."
```

Zde je ukázka volání funkce s argumenty. Tento kód provede to samé jako předcházející, ale je zde vidět, jak přistupujeme k předaným argumentům.

Každá funkce může vracet nějakou hodnotu. Mohli bychom sice ve funkci, která vypisuje nějaký text na standardní výstup, použít konstrukci *vysledek="$(ahoj)"*, ale to je nepřehledné a těžkopádné. K tomu, aby funkce mohla vrátit nějakou hodnotu se používá příkaz *return*. Pokud ve funkci nedojde k příkazu *return*, tak funkce vrací návratový kód naposledy prováděného příkazu.

Každá funkce může přistupovat k proměnným shellu. Každá funkce si také může pomocí klíčového slova *local* deklarovat své vlastní proměnné. Pokud má lokální proměnná stejný název jako proměnná shellu, tak tuto proměnnou v dané funkci potlačí.

```
#!/bin/bash
# deklarace funkce promenne
promenne()
{
      echo "volání funkce promenne"
      # deklarace lokální proměnné, která má stejný název jako
      # globální
      local prom="lokalni promenna"
      echo "Hodnota proměnné prom: $prom"
      # inkrementace globální proměnné
      g_prom=$(($g_prom+1))
      echo "Hodnota proměnné g_prom: $g_prom"
      echo "konec volání funkce"
}
prom="globální proměnná"
echo "Hodnota proměnné prom: $prom"
g_prom=5
echo "Hodnota g_prom před voláním funkce promenne: $g_prom"
# volání funkce promenne
```

```
promenne
echo "Hodnota proměnné prom: $prom"
echo "Hodnota proměnné l_prom: $l_prom"
echo "Hodnota g_prom po volání funkce promenne: $g_prom"
```

A takto vypadá výsledek:

```
Hodnota proměnné prom: globální proměnná
Hodnota g_prom před voláním funkce promenne: 5
volání funkce promenne
Hodnota proměnné prom: lokalni promenna
Hodnota proměnné g_prom: 6
konec volání funkce
Hodnota proměnné prom: globální proměnná
Hodnota proměnné l_prom:
Hodnota g_prom po volání funkce promenne: 6
```

Signály

Signál je událost, kterou posílá procesu operační systém, uživatel nebo jiná aplikace. Po přijetí signálu může tento proces provést nějakou akci. K zachytávání signálů v shellu slouží vestavěný příkaz trap. Tento příkaz přijímá dva argumenty. Prvním je akce, kterou má provést. Touto akcí je nejčastěji volání nějaké funkce. Druhým argumentem je název signálu. Všechny signály, které může proces přijmout lze vypsat pomocí příkazu *trap* s paramtrem *-l* (*trap -l*). Název každého signálu začíná písmeny SIG, které se však při volání příkazu trap nezapisují. Pokud tedy chceme zachytávat signál SIGINT, tak pouze zapíšeme *trap akce INT*. Místo názvu signálů lze také použít jejich číselnou hodnotu, např.: *trap akce 2*.

```
#!/bin/bash
# příkaz trap zajistí, že při stisku kláves Ctrl+C se zavolá funkce
```

```
# sig_int
trap sig_int INT
sig_int()
{
    echo -n "Oparvdu chcete ukončit tento program? [ano,ne]: "
    read konec
    if [ "$konec" == "ano" ]; then
        exit 0
    fi
}
while :;do
    echo "Stiskni Ctrl+C"
    sleep 5
done
```

Zde je ukázka zachytávání signálu, *SIGINT*. Tento skript bude na standardní výstup vypisovat text *Stiskni Ctrl+C*, do té doby, než se uživatel pokusí o ukončení programu. Poté, co uživatel stiskne klávesy Ctrl+C, tak se zavolá funkce *sig_int*, která se zeptá uživatele, zda chce ukončit tento program. Pokud zadá uživatel *ano*, tak se program ukončí, v opačném případě bude pokračovat ve vypisování textu.

Metaznaky shellu a regulární výrazy

Metaznaky jsou výrazy, které zastupují jiný znak, popř. jiné znaky nebo žádný.

*	Zastupuje libovolný řetězec, který může být i prázdný
?	Zastupuje libovolný jeden znak
~	Zastupuje cestu k domovskému adresáři
~+	Zastupuje aktuální pracovní adresář
~-	Zastupuje předcházející pracovní adresář
~uzivatel	Zastupuje domovský adresář uživatele *uzivatel*

[abcdef]	Zastupuje jakýkoliv ze znaků uvedených mezi [a], vykřičník za znakem [znamená negaci; potřebujeme-li zapsat více znaků, tak můžeme použít zápis intervalu, např.: [a-zA-Z] => znamená, jakýkoliv znak anglické abecedy

Příklad:

linux:~/test# **ls**

abcde a125u 756 5p

linux:~/test# **ls [0-9]**

ls: [0-9]: není souborem ani adresářem

linux:~/test# **ls [0-9]***

756 5p

linux:~/test# **ls a***

abcde a125u

linux:~/test# **ls a[0-9]***

a125u

Z tohoto příkladu můžeme vidět, jak lze metaznaky použít. Příkaz ls nám říká, jaké soubory máme v aktuálním adresáři. Pokud zadáme příkaz *ls [0-9]*, tak shell hledá soubor, který je tvořen jedním číselným znakem. Takový v aktuálním adresáři není, tak vypíše chybu. Následující příkaz *ls [0-9]**, již proběhne v pořádku, protože hledá jakýkoliv soubor v aktuálním adresáři, jehož název začíná jakýmkoliv číslem a pokračuje libovolným znakem. Příkaz *ls a** vypíše na standardní výstup všechny soubory, které začínají znakem *a*. Poslední příkaz vypisuje všechny soubory, začínající znakem *a*, za kterým následuje číslo a jakýkoliv znak.

Regulární výrazy jsou vzory, jejichž pomocí jsme schopni definovat společné rysy hledaného řetězce. Například, pokud hledáme v textu řetězec, který by mohl být emailovou adresou, lze ho vyjádřit pomocí regulárních výrazů tímto způsobem:

```
^[_a-zA-Z0-9-]+(\.[_a-zA-Z0-9-]+)*@[a-zA-Z0-9-]+(\.[a-zA-Z0-9-]
+)*(\.[a-zA-Z]{2,3})$
```

Speciální znaky používané v regulárních výrazech:

.	Znak tečka zastupuje jakýkoliv znak mimo znaku nového řádku
*	Zastupuje libovolný počet opakování předchozího znaku
^	Zastupuje začátek řetězce
$	Zastupuje konec řetězce
\	Escapuje metaznaky: např. Chceme-li vyjádřit znak tečky napíšeme \.
?	Znamená maximálně jednou
+	Znamená minimálně jednou
{n}	Právě n-krát
{m,n}	Minimálně m-krát a maximálně n-krát
{m,}	Minimálně m-krát, maximálně neomezeno
[abcdef]	Jakýkoliv ze znaků uvedených mezi [a]. Speciální znaky zde mají normální význam. Potřebujeme-li zapsat negaci, použijeme za znakem [znak stříšky (^)

Příklad s emailovou adresou tedy interpretujeme takto:

^[_a-zA-Z0-9-]+ na začátku řetězce musí být aspoň jeden ze znaků písmen anglické abecedy (nezáleží na velikosti písmen), číslo, znak – nebo znak _

*(\.[_a-zA-Z0-9-]+)** následuje libovolný počet znaků tečka, po které musí následovat alespoň jeden za znaků anglické abecedy, číslic, znaků – nebo _

@[a-zA-Z0-9-]+ za znakem zavináč musí následovat alespoň jeden znak anglické abecedy, číslo nebo pomlčka

*(\.[a-zA-Z0-9-]+)** opět libovolný počet znaků tečka, po které musí následovat alespoň jeden znak anglické abecedy, číslice nebo pomlčka

(\.[a-zA-Z]{2,3})$ na konci řetězce musí být tečka, za kterou následují 2-3 znaky anglické abecedy

V shellu se často používají regulární výrazy ve spojení s programy grep, sed, awk apod.

```
linux:~/test# ls
```

```
abcde  a125u  5p  756
linux:~/test# ls | grep -E '^a(.*)e$'
abcde
linux:~/test# ls | grep -E '^a[0-9]+'
a125u
```

Na tomto příkladu je vidět, jak lze použít regulární výrazy k omezení výpisu souborů. Příkaz *ls | grep -E '^a(.*)e$'* vypíše pouze názvy souborů, které začínají znakem *a* a končí znakem *e*. Příkaz *ls | grep -E '^a[0-9]+'* vypíše všechny soubory, které začínají znakem *a*, za kterým následuje alespoň jedno číslo.

Ladění skriptů

V případě, že dojde k syntaktické chybě, tak shell vypíše číslo řádku a popis chyby na standardní výstup. Pokud nic nevypíše, ale skript vrací nesprávné výsledky, tak je dobré si kolem problematického místa vypsat hodnoty proměnných příkazem echo. Pokud ani to nepomůže, nabízí nám shell příkaz *set*, pomocí kterého si můžeme nastavit, co vše bude shell vypisovat. Příkazem *set -o volba* zapínáme určitý způsob výpisu. Tyto výpisy se dají vypnout příkazem *set +o volba*. Volby mohou být:

noexec	neprovádí žádné příkazy, pouze kontroluje syntaktické chyby
verbose	před zpracováním vypíše příkaz
xtrace	podobné jako verbose, ale příkazy vypíše až po zpracování
nounset	v případě, že shell nalezne nedefinovanou proměnnou, tak ukončí běh skriptu a vypíše chybovou hlášku

SED

Sed (Stream Editor) je velmi výkoný program pro textové transformace. Jeho výkonnost spočívá ve způsobu, jakým je vstup zpracováván. Většina editorů provádí příkazy na celý text, ale sed zpracovává vstup po řádcích, a proto dokáže zpracovat i velké soubory. Sedovské příkazy vycházejí z editoru ed. Sed se nejčastěji používá k nahrazení nějakého textu jiným. Některé z následujících příkladů jsou převzaty z http://sed.sourceforge.net/sed1line_cz.html.

sed G	*# přidá za každý řádek volný řádek*
*sed '/^ *$'/d*	*# odstraní všechny prázdné řádky nebo řádky, které # jsou vyplněny mezerami*
sed '/regex/{x;p;x;G}'	*# vloží prázdný řádek před i za každý řádek, který # obsahuje regex*
sed = soubor | sed 'N;s/\n/ /'	*# očísluje všechny řádky v souboru*
sed 's/regulární výraz/náhrada/g'	*# nahradí všechny výskyty regulárního výrazu # náhradou*

AWK

Awk je programovací jazyk, který je navržen pro zpracování textových dat. Tento program může načítat data stejně jako sed ze standardního vstupu nebo ze souboru. Tento jazyk můžeme napsat jako argument programu awk, nebo může být uložen v souboru.

```
# příklad spuštění s programem jako argument
awk program [soubory]
# příklad spuštění s programem uloženým v souboru
awk -f soubor [soubory]
```

Program pro awk je tvořen sekvencí *vzor {akce}*. Vzor může být regulární výraz. Najde-li awk řádek, který vyhovuje vzoru, tak se provede akce. Vzor nebo akce může být vynechána, ale ne obojí. Pokud chybí vzor, tak se akce provede pro každý řádek. Pokud chybí akce, tak se vypíše každý řádek, kterému odpovídá vzor. Například, pokud chceme vypsat všechny uživatele v systému, tak napíšeme *awk -F : '{print $1}' /etc/passwd*. Protože uživatelská jména jsou v souboru /etc/passwd oddělená dvojtečkou, musíme použít přepínač -F při spuštění awk. Chceme vypsat všechny uživatele, tak můžeme vynechat vzor a napsat přímo akci. Akce musí být uzavřená ve složených závorkách. Příkaz print je příkaz jazyka awk a v proměnné $1 je uložený text prvního sloupce. Kdybychom napsali *print $0*, tak vypíšeme celý řádek. *Awk -F: '{print $1>"uzivatele";print $6>"adresare"}' /etc/passwd* tímto příkazem rozdělíme soubor /etc/passwd na dva. V souboru *uzivatele* budeme mít uložena všechna uživatelská jména a v souboru *adresare* všechny domovské adresáře. Pokud bychom chtěli vypsat pouze uživatelská

jména začínající na b, tak napíšeme *awk -F: '/^b/ {print $1}' /etc/passwd*.

PERL

Úvod

Perl je interpretovaný skriptovací jazyk, který se používá pro krátké programy i velké projekty. Umožňuje procedurální i objektové programování a existuje pro něj mnoho modulů. Perl je velmi rychlý při práci s textem a regulárními výrazy. Každý perlovský skript musí, stejně jako skript v shellu bash, začínat hlavičkou:

```
#!/usr/bin/perl
```

Cesta se může v každé distribuci linuxu lišit. Tuto cestu lze zjistit příkazem:

```
linux:~# whereis perl
perl: /usr/bin/perl /etc/perl /usr/lib/perl /usr/share/perl
/usr/share/man/man1/perl.1.gz
```

Komentáře se v perlu píší pomocí znaku #. Vše od tohoto znaku do konce řádky je považováno za komentář (kromě první řádky). Každý příkaz v perlu musí končit znakem středník.

Perlovské skripty jsou pojmenovávány s příponou *.pl*.

Proměnné a pole

V perlu rozeznáváme několik typů proměnných. Jsou to skalární proměnné, proměnné typu pole a asociativní pole. Skalární proměnné zapisujeme pomocí znaku *$*, po kterém následuje název proměnné. Proměnné typu pole se zapisují znakem @ a název pole.

Příklad inicializace skalární proměnné a práce s proměnnými typu pole:

```
1. #!/usr/bin/perl
2. $skalarni = "toto je skalární proměnná";
```

```
3. @pole = ("a","b","c","d","e","f","g","h");
4. $pocet = @pole;
5. print @pole."";print "\n";
6. print @pole;print "\n";
7. print "@pole";print "\n";
```

Výsledek:

```
8
abcdefgh
a b c d e f g h
```

Na řádku č.1 je vidět hlavička, která je povinná pro všechny perlovské skripty. Řádky č.2 a 3 nám ukazují, deklaraci skalární proměnné a proměnné typu pole. Počet prvků pole získáme tak, že přiřadíme celé pole do skalární proměnné. Příkaz na řádcích 5-7 *print* je vestavěná funkce perlu, která své argumenty vypisuje na standardní výstup. Operátor tečka je operátor zřetězení. Na 5. řádku tento skript vypisuje počet prvků v poli. Pokud proměnnou typu pole nezřetězíme, tak výstupem jsou všechny prvky v poli. Jestliže napíšeme *"@pole"*, tak výstupem budou všechny prvky pole oddělené mezerou.

Perl nabízí také některé funkce pro práci s poli. Pro spojení polí slouží funkce *push*. Pro odstranění poslední položky pole slouží funkce *pop* a pro odstranění první položky funkce *shift*. Pokud potřebujeme otočit celé pole můžeme využít funkce *reverse*.

Příklad:

```
@pole = ("a","b","c","d","e","f","g","h");
@pole2 = ("i","j","k","l","m","n");
$p = push(@pole,@pole2,"o","p");
print @pole;print "\t$p\n";
$p = pop(@pole);
```

```
print @pole;print "\t\t$p\n";
$p = shift(@pole);
print @pole;print "\t\t$p\n";
@p = reverse(@pole);
print @pole;print "\t\t@p\n";
```

Výsledek:

```
abcdefghijklmnop16
abcdefghijklmno      p
bcdefghijklmno       a
bcdefghijklmno       o n m l k j i h g f e d c b
```

Z tohoto příkladu je vidět, že funkce *push* spojí pole do svého prvního argumentu, argumenty mohou být i skalární proměnné a návratová hodnota je velikost pole. Návratová hodnota funkcí *pop* a *shift* je odstraněná položka. Funkce *reverse* jako jediná z těchto funkcí má návratovou hodnotu otočené pole a svůj argument nezměnila.

K jednotlivým prvkům pole přistupujeme podle indexů, které se píší za název proměnné v hranatých závorkách. Tyto položky jsou skalární proměnné, proto se místo symbolu zavináče píše symbol dolaru. Indexy začínají od nuly. Poslední index pole je uložen v proměnné *$#nazev_pole*. V perlovských skriptech je také možné používat pole spolu se skalárními proměnnými.

```
1. ($prvek1,$prvek2) = ($p1,$p2);
2. ($prvek1,$prvek2) = @pole;
3. ($prvek1,@pole2) = @pole;
4. (@pole2,$prvek) = @pole;
5. print $pole[0];
6. print $pole[2];
7. print $pole[$#pole];
```

Na prvním řádku dojde k přiřazení hodnoty proměnné *$p1* do proměnné *$prvek1* a proměnné *$p2* do proměnné *$prvek2*. Na druhém řádku se do proměnných *$prvek1* a $prvek2 přiřadí hodnoty prvních dvou prvků z pole *@pole*. Na třetím řádku se do proměnné *$prvek1* přiřadí první prvek pole *@pole* a do pole *@pole2* zbylé prvky. V posledním případě se všechny prvky z pole *@pole* přiřadí do pole *@pole2* a $prvek zůstane nedefinovaný.

Na řádcích 5-7 je ukázáno, jak lze v perlu přistupovat k jednotlivým prvkům pole. Jestliže platí *@pole=("a","b","c","d","e","f","g","h","i","j","k","l","m","n","o","p")*, potom 5. řádek vytiskne hodnotu *a*, 6. řádek vytiskne *c* a poslední řádek vytiskne hodnotu *p*.

Perl obsahuje několik speciálních proměnných. Některé z nich jsou vysvětleny v následující tabulce.

$_	Implicitní vstup a implicitně prohledávaná proměnná
$.	Číslo řádky naposledy čteného souboru
$"	Oddělovač prvků pole při spojování do řetězce
$0	Jméno souboru spuštěného skriptu
$$	Číslo procesu
$ARGV	Jméno právě čteného souboru při čtení ze standardního vstupu
@ARGV	Argumenty, se kterými byl spuštěn skript (neobsahuje jméno souboru)

Asociativní pole

Asociativní pole se od obyčejného pole liší tím, že místo indexů umožňuje přistupovat k položkám pomocí klíčů. Asociativní pole se také zapisuje jiným způsobem. Na začátku je znak procento, po něm název pole a klíč se zapisuje do složených závorek.

```
%asoc_pole=("Česká republika","Praha",
          "Francie","Paříž","Itálie","Řím");
# vytiskne Paříž
print $asoc_pole{"Francie"};
# konverze asociativního pole na běžné pole
@pole = %asoc_pole;
```

```
# konverze běžného pole na asociativní
%asoc_pole2 = @pole2;
```

Pro přístup ke všem klíčům asociativního pole slouží funkce *keys* a ke všem prvkům asociativního pole funkce *values*. Existuje také funkce *each*, která vrací dvojici klíč/hodnota jako dvouprvkové pole. Tyto funkce se často požívají pro procházení pole v cyklech.

Podmínky

Pro podmíněné příkazy používáme v perlu klíčová slova *if*, *elsif* a *else*.

Synatxe příkazu if:

```
if(podmínka1)
{
      # příkazy, které se mají provést, pokud je podmínka1 splněna
}
elsif(podmínka2)
{
      # příkazy, které se mají provést, pokud nebyla splněna
      # podmínka1, ale byla splněna podmínka2
}
else
{
      # příkazy, které se mají provést, pokud nebyla splněna žádná z
      # předchozích podmínek
}
```

Větve *elsif* a *else* jsou nepovinné. Větví *elsif* může být i více. Závorky *{* a *}* označují blok příkazů, které se mají vykonat, je-li podmínka splněna. Pokud není splněna žádná z podmínek ve

větvích *if* nebo *elsif*, tak se provedou příkazy z bloku *else*, je-li uveden. Blok *elsif* musí vždy následovat po bloku *if* nebo *elsif* a blok *else*, pokud je uveden, musí být poslední.

Perl umožňuje také podmíněné provedení příkazu s podmínkou na konci.

Příklad:

```
$cislo=5;
print $cislo if $cislo>0;
```

Tato konstrukce vytiskne hodnotu proměnné *$cislo* na standardní výstup, jen tehdy je-li proměnná *$cislo* kladná. Obecný zápis vypadá tak, že se napřed napíše příkaz, poté klíčové slovo *if*, podmínka a středník. V této konstrukci není možné zapsat více příkazů do bloku. Pokud potřebujeme vykonat více příkazů, tak lze využít funkce *do*, která jako svůj argument přebírá blok příkazů, které má vykonat.

Další podmínkou, kterou v perlu můžeme použít je *unless*. *Unless* se používá místo podmínky *if* a má opačný význam. Pokud není podmínka splněna, tak se provede blok příkazů za ním. Za touto podmínkou je možné dále používat *elsif* a *else* stejně jako u podmínky *if*. Podmínka *if* může plně nahradit podmínku *unless*, ale v některých případech je její použití přehlednější. *Unless* lze používat také k podmíněnému provedení příkazu s podmínkou na konci.

Příklad:

```
#!/usr/bin/perl
$cislo=10;
do
{
    print $cislo;
    print "\n" ;
    $cislo--;
} unless $cislo<0;
print $cislo."\n";
```

Na tomto příkladu je vidět příkaz s podmínkou *unless* na konci. Protože je potřeba vykonat více příkazů najednou, tak je využita funkce *do*. Podmínka *unless* je až za blokem příkazů, který je argumentem funkce *do*, a je zakončena středníkem. Příkaz se vykoná, protože podmínka je vyhodnocena jako nepravda. Výstupem tohoto programu bude číslo 10, které bylo vypsáno při vykonávání funkce *do*. Během této funkce je proměnná *$cislo* dekrementována a proto dalším výstupem tohoto programu bude číslo 9.

Cykly

V perlu rozeznáváme několik druhů cyklů. Jsou to cykly *for*, *foreach*, *while*, *until*.

For

Obecný zápis cyklu for je:

```
for (inicializace ; podmínka ; příkaz, který se má provés po každé iteraci)
{
      # blok příkazů, který se má vykonat
}
```

Když program dojde k cyklu *for*, tak napřed provede inicializaci, poté vyhodnotí podmínku a pokud je podmínka splněna, vykoná blok příkazů. Po vykonání bloku příkazů provede příkaz za druhým středníkem, potom opět vyhodnotí podmínku a provádí blok příkazů do té doby, než je podmínka vyhodnocena jako nepravda.

Příklad:

```
for ($i=0 ; $i < 20 ; $i+=2)
{
      print $i;
      print "\n";
}
```

Tento cyklus bude pod sebou vypisovat všechna sudá čísla od 0 do 18. Při prvním průchodu se nejprve provede příkaz *$i=0* a vyhodnotí se podmínka. Protože je podmínka pravdivá, tak se na standardní výstup vytiskne hodnota proměnné *$i* a znak nového řádku. Poté, co je vykonán blok příkazů se provede příkaz *$i+=2* a opět se vyhodnotí podmínka. Při druhé a další iteraci se již neprovádí inicializace.

Foreach

Cyklus *foreach* se obvykle používá k procházení polem. Obecný zápis je:

```
foreach $promenna (@pole)
{
      # příkazy, které se mají vykonat pro každý prvek pole
}
```

Cyklus prochází pole a pro každý prvek vykoná blok příkazů.

Příklad:

```
%asoc_pole = ("adam",35,"jitka",24,"eva",30);
foreach $clovek (keys %asoc_pole)
{
      print $clovek." má ".$asoc_pole{$clovek}." let.\n";
}
```

Tento skript bude procházet všechny klíče asociativního pole, ukládat je do proměnné *$clovek* a následně vypisovat na standardní výstup, kolik je komu let.

While

Syntaxe cyklu *while*:

```
while(podmínka)
{
      # blok příkazů
}
```

Cyklus *while* provádí blok příkazů do té doby, dokud platí podmínka. Předchozí příklad lze upravit takto:

```
%asoc_pole = ("adam",35,"jitka",24,"eva",30);
while( ($clovek,$vek) = each (%asoc_pole) )
{
      print $clovek." má ".$vek." let.\n";
}
```

Tento cyklus postupně ve své podmínce pomocí funkce *each*, která vrací dvojici *klíč/hodnota*, přiřazuje klíče do proměnné *$clovek* a hodnoty do proměnné *$vek* tak dlouho, dokud jsou v asociativním poli ještě nějaké nezpracované záznamy.

Until

Cyklus *until* je stejný jako cyklus *while*, ale testuje opak. Dokud neplatí podmínka, tak provádí blok příkazů

Cykly *while* a *until* je možné používat stejně jako příkazy s podmínkou na konci. Potom je jejich význam takový, že napřed provedou příkaz, potom testují podmínku a zjišťují, zda mají provést další iteraci.

Příklad:

```
$i=0;
print $i++."\n" until $i>=10;
```

Tato část skriptu bude pod sebe vypisovat čísla 0-9. Na prvním řádku se inicialzuje proměnná přiřazením hodnoty 0. Na dalším se vypíše její hodnota a poté se provede inkrementace. Takto se bude proměnná vypisovat, dokud nedosáhne hodnoty 10. Pokud potřebujeme vykonat více příkazů, tak opět využijeme funkce *do*.

Perl obsahuje příkazy pro předčasné ukončení cyklu (příkaz *last*) nebo pro přeskočení zbytku cyklu a pokračování další iterací (příkaz *next*).

```
#!/usr/bin/perl
for($i=0;$i<=20;$i++)
{
    last if $i==10;
    next if $i%2==0;
    print $i;
}
```

Tento cyklus prochází po jedné čísla od 0 do 20, ale k hodnotám vyšším než 10 se nikdy nedostane kvůli příkazu *last*. Na standardní výstup budou vypsány pouze lichá čísla od 1 do 9, protože příkaz *next* zabrání výpisu sudých hodnot.

Funkce

Pokud potřebujeme definovat v perlu funkce, tak musíme použít klíčové slovo *sub*, po kterém následuje název funkce a blok příkazů. Volání funkce se provádí pomocí znaku *&* a názvu funkce.

Příklad:

```
#!/usr/bin/perl
&vypis("Ahoj");
sub vypis
{
```

```
    print $_[0];
}
&vypis(" světe.\n");
```

Z tohoto příkladu je patrné, že můžeme funkci zavolat ještě před její deklarací. K argumentům ve funkci přistupujeme pomocí pole @_. K tomu, aby mohla funkce vrátit nějakou hodnotu se používá klíčové slovo *return*. Následující skript zjišťuje maximální hodnotu z čísel, které uživatel zadal jako argumenty programu a čísel, které byly předány jako parametry funkce.

```
#!/usr/bin/perl
sub max
{
    $max = shift(@ARGV);
    foreach $a (@ARGV)
    {
        $max=$a if $max<$a;
    }
    foreach $a (@_)
    {
        $max=$a if $max<$a;
    }
    return($max);
}

$maximum = &max (1010,1550,65,32);
print $maximum."\n";
```

Nejprve je definovaná funkce *max*, která provádí samotné hledání maxima. Na začátku funkce definujeme proměnnou *$max* přiřazením první hodnoty argumentu zadaného na příkazové řádce a

odebráním jej z pole *@ARGV* pomocí funkce *shift*. Následují dva totožné cykly. První z nich prochází pole *@ARGV* a druhé prochází pole @_. V každém z těchto cyklů je příkaz s podmínkou na konci. Tento příkaz se tedy vykoná právě tehdy, je-li podmínka splněna.

Ve funkcích lze definovat lokální proměnné pomocí klíčového slova *local*. Takto definovaná proměnná má platnost pouze v daném bloku a ve funkcích, které jsou z tohoto bloku volány. Jestliže definujeme proměnnou pomocí klíčového slova *my*, pak vytvoříme také lokální proměnnou, ale pokud z tohoto bloku budeme volat jinou funkci, tak v ní tato proměnná nebude definovaná. K definici proměnných můžeme také využít klíčové slovo *our*, které definuje proměnnou jako globální.

Příklad:

```
#!/usr/bin/perl
$p1="proměnná 1"; $p2="proměnná 2"; $p3="proměnná 3";
sub funkce_a {
    print "Ve funkci a:\n";
    local $p1 = "local p1"; my $p2 = "my p2"; our $p3 = "our p3";
    print $p1."\n"; print $p2."\n"; print $p3."\n";
    &funkce_b();
}

sub funkce_b {
    print "Ve funkci b:\n"; print $p1."\n"; print $p2."\n"; print
$p3."\n";
}
&funkce_a();
print "Mimo funkce\n"; print $p1."\n"; print $p2."\n"; print
$p3."\n";
```

Výstup:

```
Ve funkci a:
local p1
my p2
our p3
Ve funkci b:
local p1
proměnná 2
our p3
Mimo funkce
proměnná 1
proměnná 2
our p3
```

Nejprve definujeme proměnné *$p1*, *$p2*, *$p3* přiřazením hodnoty. Pak zavoláme funkci *funkce_a*, která nadefinuje proměnnou *$p1* jako *local*, proměnnou *$p2*, která je definována pomocí klíčového slova *my* a proměnnou *$p3*, kterou definujeme pomocí klíčového slova *our*. Hodnoty těchto proměnných jsou následně vypsány. Z funkce *funkce_a* je volána funkce *funkce_b*, která pouze vypisuje hodnoty proměnných. Zde je vidět, že proměnná definovaná jako *local* má stejnou hodnotu jakou měla ve funkci *funkce_a*. Proměnná definovaná pomocí klíčového slova *my* má zpět svou původní hodnotu a proměnná *$p3* si ponechala hodnotu z funkce *funkce_a*. Z výpisu, který byl proveden mimo funkce po jejich zavolání vidíme, že proměnné definované jako lokální mají zpět svou hodnotu, ale proměnná, která je definovaná pomocí klíčového slova *our*, má hodnotu přiřazenou ve funkci *funkce_a*.

Práce s řetězci

Základem práce s řetězci jsou v perlu regulární výrazy. Regulární výrazy se píší uzavřené mezi lomítky. Porovnání se provádí pomocí operátorů =~ a *!*~. První z těchto operátorů testuje, zda se v řetězci vyskytuje daný regulární výraz, druhý testuje opak. Při použití výchozí proměnné *$_* není

nutné tyto operátory používat.

Přehled symbolů, používané v regulárních výrazech:

.	Znak tečka zastupuje jakýkoliv znak, kromě znaku nové řádky
[abcdef]	Hranaté závorky označují skupinu znaků, musí odpovídat právě jeden znak. K negaci se používá znak ^ za otevírací závorkou
\d \D	Malé **d** označuje jakoukoliv číslici, velké je negace k malému
\w \W	Jakýkoliv znak abecedy, podtržítko nebo číslice, Velké **W** označuje negaci
\s \S	Mezera, CR,LF nebo tabelátor. Velké **S** označuje negaci.
(…)	Skupina znaků, na kterou se lze později odkázat pomocí **\1-\9**
^	Označuje začátek řetězce
$	Označuje konec řetězce
\n \r \t \b \B	LF, CR, tabelátor, hranice slova, velké **B** je negací k malému **b**
|	Označuje alternativu
+	Alespoň jeden výskyt předchozího znaku
*****	Žádný, jeden nebo více výskytů předchozího znaku
?	Žádný nebo jeden výskyt předchozího znaku
{N,M}	Minimálně N-krát a maximálně M-krát opakování předchozího znaku

Příklad:

```
#!/usr/bin/perl
print "Zadej číslo: ";
$_ = <STDIN>;
if(/^(\d+)$/) {
    print "Správně. Napsal jsi $1\n";
}
else {
    print "Špatně.\n";
}
```

Tento program nejprve požádá uživatele o zadání čísla. Tento vstup uloží do implicitně prohledávané proměnné, kterou potom testuje v podmínce *if*. Regulární výraz */^(\d+)$/* hledá v řetězci alespoň jedno číslo. Znak stříška říká, že řetězec musí začínat tímto znakem a znak dolar, že řetězec musí číslem také končit. Na tomto příkladu můžeme vidět použití implicitně prohledávané proměnné *$_* a také to, jak načítat uživatelský vstup pomocí ovladače <*STDIN*>. To, že je toto číslo uzavřené v kulatých závorkách nám umožňuje s ním dále pracovat. V rámci regulárního výrazu k němu přistupujeme pomocí *\1*. Mimo regulární výraz máme nastavenou speciální proměnnou *$1*.

Substituce jednoho řetězce druhým se provádí pomocí funkce *s*. Její syntaxe vypadá takto: *s/regulární_výraz/náhrada/přepínače*. Tato funkce nahradí první řetězec odpovídající regulárnímu výrazu náhradou. Pokud potřebujeme nahradit všechny řetězce, tak použijeme přepínač *g*. Při volání této funkce záleží na velikosti písmen, toto lze potlačit pomocí přepínače *i*. Návratovou hodnotou této funkce je počet změněných řetězců. Pokud není použit žádný z operátorů =~ nebo *!*~, tak se nahrazování provádí na implicitní proměnné. Proměnná, ve které je nahrazován řetězec je po skončení této funkce změněna.

Příklad:

```
#!/usr/bin/perl
$_ = "Abc ABC dEF DeF";
$poc = (s/(\b.+\b) \1/\1/gi);
print $poc."\n".$_;
```

Výstupem této funkce bude výpis proměnné *$poc*, která po skončení funkce *s* bude obsahovat hodnotu 2, protože se změnily dva různé řetězce. Protože není použit žádný z operátorů =~ nebo *!*~, tak se použije implicitní proměnná, která je vypsána na standardní výstup a její hodnota je *Abc dEF*. Tento skript vyhledá dvě stejná slova, jdoucí po sobě a nahradí je prvním z nich. Přepínač *g* zajistí, že se nahrazení provede v celém řetězci a přepínač *i*, že nebude záležet na veliksti písmen.

Je-li potřeba nahradit některé znaky v řetězci jinými, tak můžeme využít funkci *tr*. Například chceme-li nahradit všchny znaky *a* znakem *b* a všechny znaky *c* znakem *d*, tak napíšeme *tr/ac/bd/*. Tato funkce změní hodnotu svého argumentu a vrátí počet změněných znaků. Většina speciálních znaků regulárních výrazů se v této funkci nedá použít. Například následující kód vrátí počet znaků

hvězdička v řetězci *tr/*/*/*. Ve funkci *tr* lze použít znak pomlčka pro určení rozsahu, např. *tr/a-z/A-Z* nahradí všechna malá písmena velkými.

Další funkcí, kterou můžeme v perlu při práci s řetězci využít je funkce *split*. Tato funkce rozdělí řetězec podle vzoru, který je regulárním výrazem, a vrátí pole prvků. Funkce split přebírá jako druhý argument řetězec, který se má rozdělit. Pokud je volána pouze s jedním argumentem, tak pracuje s implicitní proměnnou.

Práce se soubory

Se soubory lze v perlu pracovat pomocí tzv. ovladačů. Mezi předdefinované ovladače patří např. <*STDIN*> - standardní vstup, <*STDOUT*> - standardní výstup, <*STDERR*> - standardní chybový výstup, <*ARGV*> - čte soubory předané na příkazové řádce. K otevření souboru slouží funkce *open*, která jako svůj první argument přijímá název ovladače bez < a >. Jako svůj druhý argument přijímá název souboru. Před název souboru můžeme doplnit znak >, který říká, že soubor bude otevřen pro výstup, >> - říká, že soubor bude otevřen pro rozšíření a < - říká, že soubor bude otevřen pro vstup (stejné, jako když se žádný z těchto znaků neuvede). Místo názvu souboru můžeme napsat znak '-', který způsobí otevření standardního vstupu, a znak '>-', který otevře standardní výstup.

Příklad:

```
#!/usr/bin/perl
open(HESLA,"</etc/passwd");
while(<HESLA>)
{
    chop($_);
    @ret = split(/:/);
    print $ret[0]."\t\t".$ret[5]."\n";
}
close(HESLA);
```

Tento příklad otevře soubor */etc/passwd*, ve kterém jsou uložena jména uživatelů. Pomocí cyklu *while* čte soubor po řádcích. Protože tento řetězec není ukládán do žádné proměnné, tak se použije implicitní proměnná. Příkaz *chop* v cyklu odstraní poslední znak z řetězce (v tomto případě znak konce řádku). Tato funkce změní svůj argument a vrátí odstraněný znak. Pomocí funkce *split* je řetězec rozdělen podle znaku dvojtečka a všechny části uloží do pole *@ret*. Z tohoto pole jsou následně vypsána uživatelská jména a jejich domovské adresáře. Každý otevřený soubor by měl být opět uzavřen pomocí příkazu *close*, který jako svůj argument přijímá název ovladače.

Ke zjišťování informací o souborech v perlu existuje funkce *-X*. Za *X* dosadíme příslušné písmeno podle toho, co chceme zjistit. Například *-r* zjišťuje, zda je soubor, předaný v parametru, možné číst. Těchto příkazů je mnoho a lze je všechny nalézt na http://perldoc.perl.org/functions/-X.html.

Další užitečné příkazy

Třídění

Pro třídění v perlu existuje funkce *sort*. Jako svůj argument přijímá pole. Tato funkce třídí prvky ve vzestupném pořadí. Nejprve čísla potom velká písmena a na konec malá písmena. Čísla nejsou tříděny numericky, ale jako řetězce. Pokud bychom chtěli třídit pole podle své vlastní funkce, tak příkaz *sort* může přijímat jako svůj první argument název této funkce nebo anonymní funkci (nebo také jen blok příkazů) a až jako druhý argument pole, které má být stříděno. Pokud definujeme svou vlastní funkci, tak tato funkce se volá pro každé dva prvky seznamu. Porovnávané hodnoty jsou uloženy jako globální proměnné *$a* a *$b*. Tato porovnávací funkce musí vracet *-1*, pokud má být hodnota proměnné *$a* umístěna před *$b*, *0* pokud jsou hodnoty obou proměnných stejné a *1* pokud má být hodnota *$a* umístěna za *$b*.

Příklad:

```
#!/usr/bin/perl
sub trid {return $a<=>$b;}
@pole = (1,2,34,5,4,5,23,34645,7,3,32,534,5,2,23,323,34);
@setridene = sort trid  @pole;
print "@setridene";
```

Nejprve deklarujeme funkci *trid*. Operátor <=> vrací hodnoty *-1,0,1* podle velikosti čísla. Tento příklad se dá napsat zkráceně s použitím bloku příkazů. Potom příkaz *sort* bude vypadat takto: *sort {return $a<=>$b;} @pole;*. Lze také třídit asociativní pole podle klíčů tak, že funkci *sort* předáme jako první paramater funkci *keys* a jako druhý asociativní pole.

Funkce eval

Funkce *eval* slouží k dynamickému vyhodnocování řetězců. Pokud například máme v proměnné *$p* hodnotu 'print "Ahoj světe";', tak po zavolání funkce *eval* s argumentem, kterým je proměnná *$p*, bude na standardní výstup vypsán text *Ahoj světe.*

Funkce *eval* může také dobře posloužit, potřebujeme-li odchytit výjimku.

Příklad:

```
#!/usr/bin/perl
eval {$nula = 0; $chyba = 15/$nula;};
print $@;
print "pokračování programu ...";
```

Funkce pro práci s časem

Pro práci s časem perl nabízí funkce *time*, *gmtime* a *localtime*. Funkce *time* zjišťuje počet sekund od 1.1.1970. Funkce *gmtime* konvertuje čas do Greenwich času a funkce *localtime* konvertuje čas do lokálního časového pásma. Poslední dvě funkce nevrací počet sekund, ale pole, které obsahuje sekundy, minuty, hodinu, den, měsíc a rok v tomto pořadí.

Formátování výstupu

Pro formátování výstupu máme v perlu možnost využít funkci *printf* nebo použít tzv. *formáty*. Funkci *printf* většinou využijeme na jednoduché jednořádkové výstupy. Její syntaxe je: *printf ovladač (formát,seznam hodnot)*. Pokud není uveden ovladač, tak se řetězec vypíše na standardní výstup. Podobnou funkcí je *sprintf* s tím rozdílem, že řetězec je vrácen jako návratová hodnota a nevypíše se na standardní výstup.

Příklad:

```
#!/usr/bin/perl
printf ("Znak s ascii hodnotou 65 je: %c\n",65.64568787);
printf ("Toto je řetězec: %s\n",65.64568787);
printf ("Toto je celé číslo: %d\n",65.64568787);
printf ("Toto je celé číslo šestnáctkově: %x\n",65.64568787);
printf ("Toto je desetinné číslo: %f\n",65.64568787);
printf ("Toto je číslo v semilogaritmickém tvaru: %e\n",65.64568787);
```

Výsledek:

```
Znak s ascii hodnotou 65 je: A
Toto je řetězec: 65.64568787
Toto je celé číslo: 65
Toto je celé číslo šestnáctkově: 41
Toto je desetinné číslo: 65.645688
Toto je číslo v semilogaritmickém tvaru: 6.564569e+01
```

Formáty slouží k uspořádání výstupních dat do sloupců. Můžeme pomocí nich například uvést hlavičku. Formát definujeme pomocí klíčového slova *format* a názvu formátu. Následuje rovnítko, formátovací řetězec, hodnoty a je zakončený tečkou.

Znaky, které se používají ve formátovacím řetězci:

@	Začíná položku hodnoty
^	Začíná položku postupně vkládané hodnoty
>	Zarovnání doprava
<	Zarovnání doleva
|	Zarovnání na střed
…	Zobrazí se pokud se hodnota nevejde na vymezený prostor
#	Zobrazení čísel

0	Zarovnání nulami zleva
.	Desetinná tečka
~	Nebudou vytištěny prázdné řádky
~~	Vypisuje text po řádcích
@*	Vypisuje libovolně dlouhý text
^*	Vypisuje libovolně dlouhý řádek textu

Příklad:

```
#!/usr/bin/perl
format STDOUT =
----- HLAVIČKA ---------
Zarovnáno doleva        Zarovnáno na střed       Zarovnáno doprava
@<<<<<<<<<<<<<<         @|||||||||||||||||       @>>>>>>>>>>>>>>>>
$vlevo, $nastred, $vpravo
.
$vlevo="zarovnano doleva";
$vpravo="zarovnano doprava";
$nastred="zarovnano na střed";
write STDOUT;
```

Příkazem *write* způsobíme vypsání dat. Název formátu je svázán s názvem ovladačů. Tzn., že pokud nazveme formát *STDOUT*, tak se zobrazení provede na standardní výstup. Pokud chceme zapisovat do souboru, tak do funkce *open* na místo ovladače napíšeme název formátu.

Odkazy

V perlu rozlišujeme dva druhy odkazů. Jsou to pevné odkazy a odkazy symbolické. Hodnota pevných odkazů je místo v paměti. Symbolické odkazy mají jako hodnotu název jiné proměnné. Například *$cislo=125;$promenna="cislo";* potom se lze pomocí *$$promenna* odkázat na hodnotu proměnné *$cislo*. Druhý dolar je v tomto případě dereferenční operátor. První dolar je označení proměnné. Kdyby měl odkaz směřovat na pole, pak se zapíše jako *@$promenna*. Pevné odkazy

vytváříme pomocí obráceného lomítka před proměnnou nebo hodnotou.

Příklad:

```
#!/usr/bin/perl
$cislo = 5;
$odkaz1 = \$cislo;
$odkaz2 = \"naka hodnota";
@pole = ("a","b","c","d","e","f","g","h");
$odkaz3 = \@pole;
// tisk proměnných
print $$odkaz1."\n";
$$odkaz1 = "jina hodnota";
print $cislo."\n";
print $$odkaz2."\n";
print ${$odkaz3}[0]."\n";
print $$odkaz3[4]."\n";
print $odkaz3->[7]."\n";
```

Výsledek:

```
5
jina hodnota
naka hodnota
a
e
h
```

Z tohoto příkladu je vidět definice odkazů. *$odkaz1* odkazuje na proměnnou *$cislo*. *$odkaz2*

odkazuje na anonymní proměnnou (místo v paměti, které nemá identifikátor). *$odkaz3* odkazuje na proměnnou typu pole. Pokud změníme hodnotu *$$odkaz1* tak se změní hodnota proměnné *$cislo*. Poslední tři řádky ukazují, jak lze přistupovat k odkazu na pole. Protože můžeme vytvořit také pole odkazů, tak *$$odkaz3[4]* je nepřehledné. Nemusí být na první pohled zřejmé, zda se má napřed dereferencovat *$odkaz3* a potom použít index pole nebo naopak. Proto lze použít složené závorky nebo operátor ->. Hodnota *$$odkaz2* nelze změnit, lze pouze přesměrovat odkaz na jinou anonymní hodnotu. Lze používat také odkazy na funkce, např.: *$p = \&funkce;*, za název funkce nepíšeme závorky. Ty se píšou až při volání dané funkce: *&$p();* nebo *&$p->();*. Existuje funkce *ref*, která nám říká, jakého typu je předaný argument: *SCALAR* – skalární proměnná*, ARRAY*– proměnná typu pole*, HASH*– proměnná typu asociativního pole*, CODE*– odkaz na funkci*, REF*– odkaz na jinou proměnnou typu odkaz.

Příklady:

```
# Vytvoření pole odkazů:
@pole1 = (1,2,3);
@pole2 = (4,5,6);
@pole_odkazu = \(@pole1,@pole2);
# stejné jako @pole = (\@pole1,\@pole2);

# Vytvoření pole odkazů pomocí anonymních dat:
@pole_odkazu = ([1,2,3],[4,5,6]);

# Práce s polem odkazů
print @{$pole_odkazu[0]};
print ${$pole_odkazu[0]}[1];
print $pole_odkazu[0]->[1];
```

Signály

Perl umožňuje reagovat na signály zasílané operačním systémem. V asociativním poli *%SIG* jsou uloženy odkazy na funkce. Tyto funkce jsou spuštěny tehdy, je-li zaslán signál.

Příklad:

```
#!/usr/bin/perl
$SIG{INT} = \&sig_int;
sub sig_int
{
    local $p;
    do
    {
        print "Opravdu chcete ukončit tento program??? [a|n]:";
        $p = <STDIN>;
    }until($p =~/^[an]$/);

    if($p =~ /^a$/)
    {
        exit 0;
    }
}
while(true)
{
    print "Stiskni Ctrl+C.\n";
    sleep 5;
}
```

Tento příklad vypisuje každých 5 sekund text *"Stiskni Ctrl+C"*. Toto vypisování je přerušeno stiskem kláves *Ctrl+C* a je vyvolán signál *SIG_INT*. Jeho původním významem je ukončení procesu, ale protože je v asociativním poli *%SIG* přiřazen odkaz na funkci, tak je zavolána tato funkce. Ta se zeptá, zda uživatel skutečně chce ukončit program a pokud stiskne klávesu *a*, tak se ukončí pomocí příkazu *exit*.

Balíčky a moduly

Modul je kód ve zvláštním souboru. Soubory s tímto kódem mají příponu .pm. Tento kód lze potom použít ve více programech. Aby názvy proměnných v modulu nekolidovaly s jiným kódem, tak se používají tzv. balíčky. Balíček je definován klíčovým slovem *package* a názvem balíčku. Pokud neuvedeme na začátku programu název balíčku, tak se implicitne použije balíček *main*. Jestliže potom nadefinujeme v takovém programu nějakou proměnnou, tak se na ní z ostatních balíčků můžeme odvolat pomocí plně kvalifikovaného názvu. *$main::název_proměnné*. Chceme-li využít nějaký modul, tak použijeme příkaz *use*.

Příklad:

```
#!/usr/bin/perl
use Net::Ping;
$p = Net::Ping->new();
$host = "10.0.0.138";
print "$host is alive.\n" if $p->ping($host);
$p->close();
```

Tento příklad je převzatý z http://perldoc.perl.org/Net/Ping.html. V některých případech je vhodné využít příkaz *require*, který vloží část kódu ze souboru, definovaného v parametru, do souboru, ve kterém je tento příkaz.

Popis vytvořených skriptů

Zálohování uživatelských dat

Pro zálohování uživatelských dat jsem vytvořil několik skriptů. Všechny jsou napsané v jazyku Bash, protože využívají jiné programy jako tar nebo rsync. První z nich vytváří úplné a inkrementální zálohy pomocí nástroje tar. Protože jsou všechny z těchto skriptů určeny pro pravidelné spouštění pomocí programu cron, tak je jim jako argument předána cesta ke konfiguračnímu souboru. Tento soubor obsahuje co a kam se má zálohovat (těchto částí může být i více), počet záloh, které se mají udržovat, a cestu k souboru, který bude vytvářen jako zámek, aby nedošlo ke spuštění dalšího zálohovacího procesu ještě předtím než skončí předchozí. V každém z těchto skriptů se kontroluje, zda byl předán skriptu jako argument platný soubor. Pokud ne, tak skript vypíše chybovou hlášku a ukončí se. Následně se přečte konfigurační soubor a pomocí nástroje grep se z něj odstraní prázdné řádky nebo řádky vyplněné pouze bílými znaky a řádky, které začínají znakem #, protože ten označuje komentář. Komentáře lze v konfiguračním souboru psát vždy na zvláštní řádek. Potom tento soubor čte po řádcích a pomocí nástroje awk rozdělí na dvě části a podle první z nich uloží hodnoty do proměnných. Po zpracování konfiguračního souboru zkontroluje, zda je skript spuštěn uživatelem root a existuje-li soubor, který má sloužit jako zámek. Pokud existuje, tak vypíše chybovou hlášku a ukončí se, pokud ne, tak ho vytvoří a začne zálohovat příslušné adresáře. Pro každý adresář, který se má zálohovat, kontroluje, zda existuje. Také zkontroluje, jestli existuje adresář do kterého se bude zálohovat a pokud ne, tak jej vytvoří i s adresáři na vyšších úrovních. Adresář, do kterého se má zálohovat vždy obsahuje další soubory, které jsou potřeba pro správné zálohování. Do prvního se ukládá počet provedených záloh a druhý je určen pro nástroj tar, který si do něj ukládá seznam souborů a adresářů, které již byly zazálohovány. Tento soubor pak využije při vytváření inkrementálních záloh. Pokud je tedy vytvářena první záloha, tak je tato záloha úplná a následující zálohy jsou inkrementální. Jestliže je tento skript spuštěn a již je vytvořena poslední záloha (počet záloh se nastavuje v konfiguračním souboru), tak tyto zálohy odstraní a vytvoří novou úplnou zálohu.

Zdrojový kód:

```
#!/bin/bash
ID=/usr/bin/id
ECHO=/bin/echo
TOUCH=/bin/touch
```

```
RM=/bin/rm
TAR=/bin/tar

if [ ! -e "$1" ] ; then
    $ECHO "Správný formát je" `basename $0` " configuracni_soubor."
    exit
fi


CO_I=0
KAM_I=0

for i in `cat $1 | grep -v -E '^ *$' | grep -v -E '^#.*$'` ; do
    p=`echo $i | awk -F = '{print $1}'`
    h=`echo $i | awk -F = '{print $2}'`
    if [ "$p" = "CO" ]; then
      CO[$CO_I]=$h
      CO_I=$(($CO_I + 1))
    elif [ "$p" = "KAM" ]; then
      KAM[$KAM_I]=$h
      KAM_I=$(($KAM_I + 1))
    elif [ "$p" = "LOCKFILE" ]; then
      LOCKFILE=$h
    elif [ "$p" = "POCET_ZALOH" ]; then
      POCET_ZALOH=$h
    fi
done

if [ "$CO_I" -lt "$KAM_I" ]; then
    POCET=$CO_I
else
    POCET=$KAM_I
fi

if(( `$ID -u` != 0 )); then
    $ECHO "$0 Tento skript smi spoustet jen root."
fi

if [ -z "$LOCKFILE" ]; then
    $ECHO "Neni nastaven LOCKFILE v konfiguracnim souboru."
    exit
fi

if [ -z "$POCET_ZALOH" ]; then
    $ECHO "Neni nastaven POCET_ZALOH v konfiguracnim souboru."
    exit
fi

if [ -f "$LOCKFILE" ]; then
    $ECHO "Nelze udelat zalohu, protoze neskoncila predchozi"
    exit
fi

$TOUCH $LOCKFILE

I=0
```

```
while [ $I -lt $POCET ]; do

    K=${KAM[$I]}
    C=${CO[$I]}

    if [ ! -d "$C" ]; then
      $ECHO "$C nelze zazalohovat"
      I=$(($I + 1))
      continue
    fi

    if [ -e "${K}pocet" ]; then
      HOTOVYCH=`cat ${K}pocet`
    else
      HOTOVYCH=0
    fi

    if [ "${POCET_ZALOH}" -le "${HOTOVYCH}"  ]; then
      $RM -rf "${K}"
      HOTOVYCH=0
    fi

    NASLEDUJICI=$(($HOTOVYCH + 1))

    if [ ! -d "$K" ]; then
      mkdir -p -m 700 "$K"
    fi


    $TAR -czf "${K}${NASLEDUJICI}.tgz" -g "${K}snapshot" "${C}" 2>
/dev/null

    if [ -d "$K" ]; then
      $ECHO ${NASLEDUJICI} > ${K}pocet
    fi

    I=$(($I + 1))
done

$RM $LOCKFILE
```

Příklad konfiguračního souboru:

```
# komentare musi byt na samostatnem radku
# KAM urcuje kam se bude zalohovat
# CO urcuje co se bude zalohovat
# Pro prvni KAM odpovida prvni CO, pro druhe KAM druhé CO ...

KAM=/mnt/zalohy/tar/www/
CO=/home/swr/www/

# POCET_ZALOH urcuje kolik zaloh se ma udrzovat
POCET_ZALOH=4

# LOCKFILE urcuje jaky soubor ma byt pouzit jako zamek
```

```
LOCKFILE=/tmp/zal_tar.lock

# adresare musi koncit lomitkem!!!
```

Tento skript neřeší problémy v případě uzamčení některého souboru, ale plně se spoléhá na standardní nástroje, které jsou v systému a prošly již dlouhým vývojem. V případě, že se tento skript na takovém souboru zastaví příliš dlouho, tak je chráněn před dalším spuštěním tak, že při začátku každého zálohování se vytváří soubor. Pokud tento soubor existuje, tak skript vypíše chybovou hlášku na standardní výstup. Další problém, na který může administrátor serveru narazit při použití tohoto skriptu ja ten, že v případě nesprávných hodnot v konfiguračním souboru nebo neukončením adresářů lomítkem není jasné, jak se tento skript zachová. Ověřuje pouze zda existují adresáře, které se mají zálohovat a pokud neexistují adresáře, do kterých se má zálohovat, tak je vytvoří včetně všech nadřazených adresářů. Také může nastat problém v případě, že v konfiguračním souboru bude nastaveno, aby se dva zdroje zálohovaly do stejného adresáře.

Druhý způsob využívá nástrojů cp a rsync. Začátek tohoto skriptu je stejný jako předchozí, změněný je způsob zálohování. V cílovém adresáři vytváří číslované podadresáře. Pomocí nástroje rsync synchronizuje obsah zdrojového adresáře s adresářem označeným „1“. Při dalším vytváření záloh se k souborům v tomto adresáři vytváří pevné odkazy do adresáře s následujícím číslem a opět se spustí nástroj rsync na adresář označený „1“.

Zdrojový kód:

```
#!/bin/bash

ID=/usr/bin/id
ECHO=/bin/echo
RM=/bin/rm
MV=/bin/mv
CP=/bin/cp
TOUCH=/bin/touch
RSYNC=/usr/bin/rsync

if [ ! -e "$1" ] ; then
    $ECHO "Správný formát je" `basename $0` " configuracni_soubor."
    exit
fi

CO_I=0
KAM_I=0

for i in `cat hourly.conf | grep -v -E '^ *$' | grep -v -E '^#.*$'` ;
do
    p=`echo $i | awk -F = '{print $1}'`
```

```
    h=`echo $i | awk -F = '{print $2}'`
    if [ "$p" = "CO" ]; then
      CO[$CO_I]=$h
      CO_I=$(($CO_I + 1))
    elif [ "$p" = "KAM" ]; then
      KAM[$KAM_I]=$h
      KAM_I=$(($KAM_I + 1))
    elif [ "$p" = "LOCKFILE" ]; then
      LOCKFILE=$h
    elif [ "$p" = "POCET_ZALOH" ]; then
      POCET_ZALOH=$h
    fi
done

if [ "$CO_I" -lt "$KAM_I" ]; then
    POCET=$CO_I
else
    POCET=$KAM_I
fi

if(( `$ID -u` != 0 )); then
    $ECHO "$0 Tento skript smi spoustet jen root."
fi

if [ -z "$LOCKFILE" ]; then
    $ECHO "Neni nastaven LOCKFILE v konfiguracnim souboru."
    exit
fi

if [ -z "$POCET_ZALOH" ]; then
    $ECHO "Neni nastaven POCET_ZALOH v konfiguracnim souboru."
    exit
fi

if [ -f "$LOCKFILE" ]; then
    $ECHO "Nelze udelat zalohu, protoze neskoncila predchozi"
    exit
fi

$TOUCH $LOCKFILE

I=0
while [ $I -lt $POCET ]; do

    #$ECHO ${CO[$I]}
    #$ECHO ${KAM[$I]}
    #$ECHO

    K=${KAM[$I]}
    C=${CO[$I]}

    if [ ! -d "$C" ]; then
      $ECHO "$C nelze zazalohovat"
      I=$(($I + 1))
      continue
    fi
```

```
    if [ ! -d "$K" ]; then
      mkdir -p -m 700 "$K"
    fi

    if [ -d "${K}${POCET_ZALOH}" ]; then
      $RM -rf "${K}${POCET_ZALOH}"
    fi

    PZ=$(($POCET_ZALOH - 1))
    while [ "${PZ}" -gt 1 ]; do
      #echo $PZ
      NASL=$(($PZ + 1))
      if [ -d "${K}${PZ}" ]; then
          $MV "${K}${PZ}" "${K}${NASL}"
      fi

      PZ=$(($PZ - 1))
    done

    if [ -d "${K}1" -a "${POCET_ZALOH}" -gt 1 ]; then
      $CP -al "${K}1" "${K}2"
    fi

    $RSYNC -a --delete ${C} "${K}1"

    I=$(($I + 1))
done

$RM $LOCKFILE
```

Konfigurační soubor tohoto skriptu je stejný jako konfigurační soubor předchozího skriptu.

Tento skript je velmi podobný předchozím a tak má stejné nedostatky. Dalším problémem tohoto skriptu je ten, že v případě snížení počtu záloh v konfiguračním souboru se administrátor serveru musí postarat o to, aby se odstranily přebytečné adresáře, protože tento skript je sám neodstraní.

Oba tyto skripty byly vytvořeny tak, aby se vzájemně doplňovaly. Skript, který vytváří archivy byl navržen pro pravidelné denní zálohování, tak aby se dala udržovat záloha až týden stará, druhý skript byl navržen pro pravidelné hodinové zálohování.

Tyto skripty jsou již několik měsíců úspěšně využívány na webhostingovém serveru, kde umožňují zálohovat data zákazníků. Pokud tedy zákazník ztratí svá data, tak tyto skripty je umožní rychle obnovit. Velmi často se stává, že zákazník odstraní svá data a několik hodin poté je potřebuje obnovit. V takovém případě lze plně využít skriptu, který udržuje kopie a nevytváří archivy, neboť překopírování je mnohem méně výpočetně náročné než rozbalení archivu, který pak zabere stejný prostor na disku.

Kontrola překročení diskových kvót

Vytvořil jsem skript napsaný v perlu, který testuje překročení diskových kvót zákazníků webhostingového serveru. Protože uživatelé jsou zde virtuální, tak nelze využít programu setquota. Další překážkou bylo to, že nastavení diskových kvót musí jít pomocí webového rozhraní. Byl zvolen způsob, že se pomocí skriptů php bude generovat soubor v přesném formátu a tento skript jej zpracuje a zkontroluje kvóty. Pokud zjistí překročení limitu, tak vypíše hříšníky na standardní výstup. Protože je tento skript pravidelně spouštěn pomocí cronu, tak je tento výstup odeslán emailem správci serveru. Ten se pak dohodne se zákazníkem. Tento skript nesmí při překročení znemožnit uživateli přidávání nových dat.

Zdrojový kód:

```
#!/usr/bin/perl
use Fcntl ':flock';

my $file = "/var/www/admin.molbud.cz/db/.db_uzivatelu";

sub velikost_adresare
{
      my($dir) = @_;
      my $fh;
      my $velikost = 0;
      opendir($fh, $dir) or die "Nepodařilo se otevřít $dir: $!";

      while (my $pol = readdir $fh)
      {
            next if $pol eq "." or $pol eq "..";
            if (-f $dir."/".$pol)
            {
                $velikost += -s $dir."/".$pol;
            }
              elsif(-d $dir."/".$pol)
            {
              $velikost += -s $dir."/".$pol;
              $velikost += velikost_adresare($dir."/".$pol);
            }
      }

      $velikost += -s $dir;
      return $velikost;
}

sub web
{
      if(-d "/var/www/$_[1]")
      {
```

```
            $size = velikost_adresare("/var/www/$_[1]");
            return $size/1024;
      }

      return 0;
}

sub mysql
{
      if(-d "/var/lib/mysql/$_[1]")
      {
            $size = velikost_adresare("/var/lib/mysql/$_[1]");
            return $size/1024;
      }

      return 0;
}

sub mail
{
      ($user,$dom) = split(/@/,$_[1]);
      if(-d "/var/vmail/$dom/$user")
      {
            $size = velikost_adresare("/var/vmail/$dom/$user");
            return $size/1024;
      }

      return 0;
}

sub mail_domena
{
      if(-d "/var/vmail/$_[1]")
      {
            $size = velikost_adresare("/var/vmail/$_[1]");
            return $size/1024;
      }

      return 0;
}

open(ZAK, $file);
flock(ZAK,LOCK_EX);
my $lines = "";

$lines = join("",<ZAK>);

flock(ZAK,LOCK_UN);
close(ZAK);

@po_zak = split(/-----/,$lines);
my $poc=0;
##### PROCHAZENI ZAKAZNIKU
while(@po_zak)
{
      $poc++;
```

```
	$zak = shift(@po_zak);

	@pocitat = split(/;;;/,$zak);
	$pocet = @pocitat;
	$pocitat[0] =~  m/zak:\s([0-9]*)/m;
	$zakaznik = $1;
	$zakaznik =~ s/ //g;
	##### PROCHAZENI POCITADEL
	for ($i=1; $i<$pocet;$i++)
	{
		$pocitat[$i] =~ m/poc: (.*)/m;
		$co = $1;
		$pocitat[$i] =~ m/lim: (.*)/m;
		$lim = $1;
		$lim =~ s/ //g;
		$vysl = 0;
		@fce = split(/\+/,$co);
		$pocet_fce = @fce;
		##### PROCHAZENI FCI V POCITADLE
		for ($j=0;$j<$pocet_fce;$j++)
		{
			if($fce[$j] =~ /(.*)\((.*)\)/)
			{
				$funkce = $1;
				$param = $2;
				$funkce =~ s/ //g;
				$param =~ s/ //g;
				if($funkce =~ /web|mysql|mail|mail_domena/)
				{
					$vysl += &$funkce ($zakaznik,$param);
				}
			}
		}

		if($lim < $vysl)
		{
			print "Zakaznik: ".$zakaznik."\n\n";
			print "    pocitadlo: ".$co."\n";
			print "    zabira: ".$vysl."kB\n";
			print "    limit:  ".$lim."kB\n\n\n";
		}
	}
}
```

Příklad souboru, který je zpracováván:

```
zak: 200701001
;;;
poc: web(firma.cz)+mysql(firma)+mail(200701001@zakaznici.molbud.cz)
lim: 10240
```

```
;;;
poc: mail_domena(firma.cz)
lim: 5120
-----
zak: 200701002
;;;
poc: web(firma2.cz) + mail(200701002@zakaznici.molbud.cz)
lim: 20480
```

Tento skript využívá modulu Fcntl, díky kterému umožňuje zamknout soubor. Na začátku jsou definovány některé funkce. Funkce velikost_adresare zjišťuje velikost všech souborů v adresáři a jeho podadresářích a tyto velikosti sečte. Celková velikost je pak jeho návratovou hodnotou. Jako argument přijímá jeden parametr, kterým je cesta k adresáři. V cyklu while pak prochází všechny soubory a zjišťuje jejich velikost. Pokud narazí na adresář, tak rekurzivně volá sama sebe s argumentem, kterým je tento adresář. Během tohoto cyklu je potřeba ošetřit, zda adresář není „.“ nebo „..“, které označují aktuální a nadřazený adresář a pokud by to nebylo ošetřeno, tak se tento skript tzv. zacyklí. Následují čtyři podobné funkce web, mysql, mail a mail_domena. Tyto funkce přijímají dva argumenty. Prvním je označení zákazníka a druhý se pro každou funkci liší (je zapsán v souboru, který se zpracovává). První argument se ve funkcích nevyužívá a slouží pouze pro kontrolní výpisy. Funkce web přijímá jako druhý argument název domény, pro kterou byl zřízen hosting. Tato funkce zkontroluje, zda existuje příslušný adresář pro tuto doménu a pak volá funkci velikost_adresare s argumentem, kterým je tento adresář. Celková velikost je pak jeho návratovou hodnotou. Funkce mysql se chová podobně jako funkce web, ale jako argument přijímá název mysql databáze. Funkci velikost_adresare pak předává cestu, kde jsou uložené adresáře s mysql databázemi (obvykle /var/lib/mysql) a název této databáze. Funkce mail, je trochu odlišná. Jako argument přijímá emailovou adresu. Tu pak rozdělí pomocí funkce split podle znaku zavináč a sestaví cestu k uživatelské emailové schránce. Tuto cestu pak předá funkci velikost_adresare a vrátí celkovou velikost všech emailů ve schránce. Funkce mail_domena se chová podobně jako funkce web. Také přijímá jako svůj argument doménu a zjistí velikost všech emailových schránek pro danou doménu. Skript začíná otevřením souboru, ve kterém jsou uloženy limity zákazníků. Každý zákazník může mít nastaveno několik počítadel. Skript si tento soubor uzamkne, načte do proměnné pomocí funkce join a co nejdříve opět odemkne a uzavře. Proměnná, ve které je uložen celý soubor je následně rozdělena na několik částí pomocí funkce split. Nejprve je rozdělena podle zákazníků. Tyto zákazníky prochází pomocí cyklu while a poté si zjistí počet počítadel, které má zákazník

nastaveny. Pomocí cyklu for prochází všechna jeho počítadla. Funkce v počítadle jsou shodné s názvy funkcí skriptu. Pro každou z těchto funkcí je uložen její název do proměnné a to umožňuje využít tuto proměnnou jako symbolický odkaz. Pro případ, že by tato funkce neexistovala, tak je její volání podmíněno příkazem if. V případě, že skript zjistí překročení limitu, tak vypíše označení zákazníka a všechny funkce počítadla, ve kterém došlo k překročení, na standardní výstup. To umožňuje kontrolovat limity i na příkazovém řádku a mít výsledky okamžitě k dispozici. V případě pravidelného spouštění pomocí cronu, může být dle nastavení cronu výsledek odeslán emailem, nebo přesměrován do souboru.

Tento skript vyžaduje, aby soubor, ve kterém jsou uloženy diskové kvóty zákazníků, byl přesně v tomto formátu. V případě, že bude v tomto souboru chyba, tak tento skript může vracet neočekávané výsledky.

Vyhledávání velkých souborů

Při vytváření balíku skriptů byl požadavek na vytvoření skriptu, který odešle emailem cesty ke všem souborům, které jsou větší než 2GB. Na začátku skriptu jsou definovány proměnné, ve kterých je uložen email, na který se odešle zpráva s cestami k souborům, které jsou větší než zadaná velikost. Tato velikost je uložena v proměnné na začátku skriptu, aby její případná změna nemusela zasahovat hlouběji do skriptu. Poslední proměnnou je cesta k adresáři, od kterého začne prohledávání. Po deklaraci těchto proměnných začíná samotné tělo skriptu. Nejprve se zkontroluje, zda existuje adresář, od kterého má začít prohledávání. Pak se vytvoří dočasný soubor pomocí příkazu tempfile, do kterého se bude ukládat text emailu. Pro přehlednost jsem vytvořil funkci ukladani, která čte data ze standardního vstupu a ukládá je do dočasného souboru. Poté se volá příkaz find, který provede vlastní prohledání. Protože se mohou vyskytovat v adresářové struktuře také soubory, které obsahují bílé znaky a diakritická znaménka, tak všechny cesty k souborům, které nalezne, odděluje nulovým znakem. Tento výstup je předán pomocí roury na vstup programu xargs, který je podle tohoto nulového znaku rozdělí a postupně pomocí roury předává na vstup funkci ukladani. Text uložený v dočasném souboru je poté odeslán emailem a odstraněn. K tomu, aby příkaz find vracel cesty oddělené nulovým znakem jsem využil parametru -print0, pro příkaz xargs bylo potřeba doplnit parametr -0.

Zdrojový kód:

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

```
# na jaky email se ma napsat zprava
mail=root@server.cz
# cesta, ve ktere se ma hledat
cesta=/home/

# velikost je v kB
velikost=2097152

if [ ! -d "$cesta" ]; then
    echo "$cesta neni platny adresar."
    exit
fi


export tmpf=`tempfile`
export pocet=0
ukladani()
{
    while read n; do
      pocet=$(($pocet + 1))
      echo "$n" >> "$tmpf"
    done
    echo $pocet
}

echo "Soubory vetsi nez $velikost kB v $cesta:" > "$tmpf"
echo "" >> "$tmpf"

pocet=`find "$cesta" -size "+${velikost}k" -type f -print0 | xargs -0
-n2 ls | ukladani`
mail -s "Velke soubory" "$mail" < "$tmpf"
rm -f $tmpf
```

Vyhledávání souborů podle přípony

Tento skript je velmi podobný předchozímu. Na začátku jsou opět definované proměnné, které označují na jaký email se má odeslat výsledek a odkud se má začít vyhledávat. Je zde definovaná také nová proměnná typu pole, ve které jsou uloženy soubory, které se mají vyhledat. Názvy souborů mohou obsahovat metaznaky (například hvězdičku). Toto pole procházím pomocí cyklu for a pro každý z těchto souborů je spuštěn příkaz find, jehož výstupem jsou cesty k souborům oddělené nulovými znaky. Tento výstup je pak předán příkazu xargs a ten ho postupně předává předtím definované funkci ukladani. Dalším řešením by mohlo být vyhledávat všechny soubory najednou pomocí parametru -or v příkazu find, ale chtěl jsem, aby byl výstup setříděn podle vyhledávaných souborů.

Zdrojový kód:

```
#!/bin/bash

# na jaky email se ma napsat zprava
mail=root@server.cz
# cesta, ve ktere se ma hledat
cesta=/home/
# nazvy souboru, ktere se maji vyhledat (nezalezi na velikosti
pismen)
typ[0]="*.avi"
typ[1]="*.mp3"
typ[2]="*.mpg"

if [ ! -d "$cesta" ]; then
    echo "$cesta neni platny adresar."
    exit
fi


export tmpf=`tempfile`
ukladani()
{
    pocet=0
    while read n; do
     pocet=$(($pocet + 1))
     echo "$n" >> "$tmpf"
    done
    echo $pocet
}

for i in ${typ[*]}; do
    echo "Soubory, odpovidajici vyrazu $i v $cesta:" >> "$tmpf"
    echo "" >> "$tmpf"

    pocet=`find "$cesta" -iname "$i" -type f -print0 | xargs -0 -n2
ls | ukladani`
    echo "" >> "$tmpf"
    echo "" >> "$tmpf"
done
mail -s "Soubory podle vzoru" "$mail" < "$tmpf"
rm -f $tmpf
```

Vyhledávání core souborů

Core soubory jsou soubory, které se vytváří při pádu některého programu. Jeho název je core následovaný tečkou a několika číslicemi. Pro vytvoření skriptu, který bude tyto soubory vyhledávat, jsem vycházel ze skriptu, který vyhledával velké soubory. Na začátku jsou definované pouze dvě proměnné. Hodnotou prvního je email, kam se odešlou cesty k těmto souborům. Hodnotou druhé proměnné je cesta k adresáři, od kterého se má začít s vyhledáváním. Pro vyhledání core souborů

jsem opět využil příkazu find propojeného s příkazem xargs. Příkazu find je předáván parametr -regex a po něm regulární výraz: .*/core\.[0-9]+. Tento regulární výraz znamená, že se budou vyhledávat všechny soubory, které mohou být v několika podadresářích, začínají znaky core a tečkou a po nich následuje alespoň jedna číslice. Po průchodu výstupu příkazu find jsou příkazem xargs předávány na standardním vstupu pomocí roury funkci ukladani cesty k jednotlivým core souborům a ukládány do dočasného souboru, který byl vytvořen pomocí příkazu tempfile. Tento soubor je pak, stejně jako v předchozích skriptech, odeslán emailem. Dočasný soubor je po odeslání emailu odstraněn.

Zdrojový kód:

```
#!/bin/bash

# na jaky email se ma napsat zprava
mail=root@server.cz
# cesta, ve ktere se ma hledat
cesta=/home/

if [ ! -d "$cesta" ]; then
    echo "$cesta neni platny adresar."
    exit
fi

export tmpf=`tempfile`
ukladani()
{
    pocet=0
    while read n; do
      pocet=$(($pocet + 1))
      echo "$n" >> "$tmpf"
    done
    echo $pocet
}

echo "Core soubory v $cesta:" >> "$tmpf"
echo "" >> "$tmpf"
pocet=`find $cesta -regex ".*/core\.[0-9]+" -type f -print0 | xargs
-0 -n2 ls | ukladani`
mail -s "Core soubory" "$mail" < "$tmpf"
rm -f $tmpf
```

Všechny tyto vyhledávací skripty se spoléhají na správnou funkci příkazů find a xargs. Problém může nastat ve chvíli, kdy tyto skripty spustí jiný uživatel než root a bude se snažit vyhledávat v adresáři, do kterého nemá přístup. Dalším problémem může být použití programu mail ve chvíli, kdy bude v systému nesprávně nastaven smtp server. Také se nekontroluje, zda je emailová adresa zadána ve správném formátu.

Praktické zkušenosti s použitím vytvořeného balíku skriptů

Většina skriptů se používá již několik měsíců. Zejména skripty pro zálohování se velmi osvědčili. Správné zálohování je na serveru nezbytné a tyto skripty umožňují pravidelné úplné i inkrementální zálohování a zajišťují bezproblémový chod serveru. Pro každodenní zálohování se používá skript, který vytváří archivy a udržuje inkrementální zálohy po dobu jednoho týdne, poté vytvoří novou úplnou zálohu. Pro hodinové zálohování se používá skript, který vytváří pevné odkazy. Tento skript využívá programu rsync, který umožní správnou synchronizaci zálohovaných dat s již vytvořenou zálohou. Tímto způsobem je zajištěno minimální vytížení serveru jak během zálohování, tak při případné obnově dat. Tento skript je spouštěn jednou za dvě hodiny a udržuje zpětně čtyři zálohy. To vytváří dostatečné možnosti pro obnovu dat, které jsou staré několik dní nebo třeba jen několik hodin.

Protože oba tyto skripty se spouští s argumentem, kterým je konfigurační soubor, tak změna způsobu zálohování je velmi jednoduchá. Pokud administrátor preferuje pouze zálohování do archivů, ale potřebuje některá data zálohovat jiným způsobem, tak se právě v takovém okamžiku uplatní výhoda, že se konfigurační soubor předává jako argument při jejich spouštění a není pevně nastaven ve zdrojových kódech skriptů. Tyto skripty jsou určeny pro pravidelné spouštění pomocí cronu, tak aby se administrátor serveru nemusel jimi vůbec zabývat.

Tyto nástroje byly vytvořeny pro konkrétní případ. Jsem si vědom některých menších nedostatků, které se mohou vyskytnout při použití v jiném prostředí. Dořešení těchto nedostatků je věcí dalšího vývoje těchto skriptů. Největším problémem těchto skriptů je ten, že neřeší zálohování uzamčených souborů. Při zkoumání tohoto problému se zjistilo, že pokud skript narazí na zamknutý soubor, tak se jeho činnost zastaví a čeká, dokud není soubor opět odemknutý. Takto může skript čekat hodně dlouhou dobu i takovou, že je mezitím spuštěn pomocí cronu ještě jednou. To řeší zálohovací skripty tak, že při každém spuštění vytvoří soubor, který funguje jako zámek, a v případě spuštění před dokončením předchozího zálohování je pouze vypsáno chybové hlášení na standardní výstup a nehrozí, že by se spustilo příliš procesů, které by operační systém nestačil provádět.

Také se velmi uplatnil skript, který testuje překročení diskových kvót. Tento skript testuje překročení diskových kvót virtuálních uživatelů systému, kteří mají přístup k souborovému systému přes protokol ftp, mají k dispozici databázi mysql a vlastní emailové schránky. Umožňuje každému uživateli mít nastaveno několik počítadel. Zároveň je velmi flexibilní, protože umožňuje kontrolovat diskové kvóty zvlášť pro mysql, ftp i email. Soubor, ve kterém jsou uloženy diskové

kvóty, je generován pomocí php přes webové rozhraní. Díky tomu je správa serveru mnohem jednodušší. V případě překročení diskové kvóty je na standardní výstup vypsán zákazník a počítadlo, ve kterém došlo k překročení. Protože je tento skript pravidelně spouštěn pomocí programu cron, tak může být veškerý výstup tohoto skriptu odeslán emailem správci serveru. V případě, že je potřeba tento výstup ukládat do souboru, tak lze přesměrovat výstup pomocí operátorů shellu Bash. Protože tento skript v případě překročení kvóty nezakáže uživateli přidávat další data, tak je toto předmětem dalšího vývoje.

Funkčnost tohoto skriptu je velmi závislá na správnosti formátu souboru, ze kterého se načítají diskové kvóty. Toto by se dalo řešit propojením s mysql databází, což je předmětem dalšího vývoje. Zároveň by tím odpadlo řešení uzamykání souboru, ve kterém jsou uloženy kvóty zákazníků, při jeho načítání skriptem a jeho editaci v php.

Pro některé fakulty Jihočeské Univerzity jsem do tohoto balíčku přidal také skripty pro vyhledávání tzv. core souborů, souborů větších než 2GB a vyhledávání souborů podle různých kritérií.

Skript, vyhledávající core soubory, se používá pro vyhledávání souborů ve tvaru „core.“ a číslice. Tyto soubory generují některé programy při neočekávaném ukončení. Cesty k těmto souborům jsou následně odeslány emailem a nejsou automaticky mazány. Díky tomu má administrátor serveru přehled, které programy jsou nejméně stabilní a může navrhnout nejvhodnější řešení.

Skript pro vyhledávání souborů větších než 2GB se používá, aby se předešlo problémům při kopírování souborů přes NFS. Cesty k těmto souborům jsou opět odeslány emailem správci serveru.

Pro přehled, čím zaplňují studenti svůj disk na Jihočeské univerzitě jsem napsal skript, který vyhledává soubory podle zadaných kritérií. Protože tento skript nebyl ještě příliš využit, tak zatím nejsou k dispozici žádné přehledy. Co se týče využívání svého prostoru na disku studenty, tak je předmětem dalšího vývoje, aby tento skript generoval statistiky zaplněnosti a zjistil kolik procent z toho jsou právě soubory, které nesouvisí se studiem.

Závěr

Operační systémy založené na Unixu jsou velmi často využívané jako servery. Proto je potřeba, aby byl zajištěn bezproblémový chod pro všechny uživatele takového systému. Tento bezproblémový chod musí zajistit jeho administrátor a ten potřebuje mít k dispozici takové nástroje, které mu umožní dobře vykonávat svou práci. K vytvoření takových nástrojů může využít jednoduchých, ale velmi efektivních skriptovacích jazyků, které jsou součástí většiny operačních systémů, založených na Unixu.

Balík skriptů, který jsem vytvořil, zjednodušuje správu uživatelů a umožnil zautomatizovat některé často prováděné úlohy. Současně zajistil bezpečné zálohování dat uživatelů a umožnil administrátorovi serveru více se zabývat důležitějšími úkoly správy systému.

Velkým zjednodušením je kontrola překročení diskových kvót pro virtuální uživatele systému, kdy nelze využít standardních nástrojů (edquota) jako v případě fyzických uživatelů. Tito uživatelé mají přístup k disku pouze přes protokol FTP, využívají databáze mysql a využívají služeb mailserveru. Tento skript kontroluje překročení limitu pro každou z těchto služeb. Limity lze nastavovat přes webové rozhraní a skript z tohoto balíčku pouze vypíše uživatele, kteří tento limit překročili. Tento seznam lze pak dále zpracovávat.

Většina těchto skriptů je pravidelně spouštěna pomocí nástroje cron. Způsob, jakým jsou tyto skripty napsány zajišťuje velkou flexibilitu při správě uživatelských účtů. Zároveň umožňuje jejich další vývoj. Díky tomu, že tyto skripty využívají nástroje, které jsou vyvíjeny již mnoho let, je zajištěna jejich stabilita a rychlé zpracování dat.

Seznam použité literatury

- FLICKENGER R.: Linux server na maximum, 100 tipů a řešení pro náročné, Computer Press, Brno 2005
- http://docs.linux.cz/programming/interpreted/bashdoc-1.4/
- http://www.abclinuxu.cz/clanky/navody/bash-ii
- http://cs.wikibooks.org/wiki/Bash#Pole_.28array.29
- http://sed.sourceforge.net/sed1line_cz.html
- http://cs.wikipedia.org/wiki/Sed
- http://www.ics.muni.cz/zpravodaj/articles/33.html
- http://cs.wikipedia.org/wiki/AWK
- http://www.root.cz/clanky/uvod-do-awk-promenne-operatory-a-prvni-kroky/
- http://perldoc.perl.org/
- http://perl.eurohost.cz/
- http://cs.wikipedia.org/wiki/Perl
- http://www.cpan.cz/
- http://www.linuxsoft.cz/article_list.php?id_kategory=210