Úvod

Bourne **A**gain **SH**ell je standardní interpret příkazů v Linuxu založený na Bourne shell. Funguje jako rozhraní mezi uživatelem a systémem. Jelikož je součástí GNU projektu, nebylo problémem ho portovat na unixové systémy, takže se jeho znalost uplatní i mimo Linux. Jeho funkce můžeme rozdělit na 3 základní části.

V interaktivním režimu čeká na zadání příkazu od uživatele. Příkazy mohou být buď přímo zabudované v shellu nebo samostatné programy napsané téměř v libovolném programovacím jazyku.

Pomocí systémových proměnných umožňuje přizpůsobení pracovního prostředí. Některé z těchto proměnných jsou přednastaveny systémem, ostatní nastavuje uživatel např. v inicializačních souborech při spuštění shellu.

Je to také velice mocný programovací nástroj. Když nám chybí nějaký program nemusíme ho hned psát v kompilovaném jazyku (C, C++, Ada, Java), ale je možné vyřešit náš problém vytvořením skriptu. Můžeme si tím ušetřit hodně práce a nebo právě naopak. Nejprve musíme důkladně analyzovat náš problém a zvolit správné řešení.

Zjistěte, jestli máte jako implicitní shell nastaven opravdu BASH. Možností je hned několik. Poslední příkaz zjistí, jaký shell používá implicitně váš systém.

```
$ echo $SHELL
/bin/bash
$ cat /etc/passwd | grep $USER
fuky:x:1000:1000:Jan Fuchs,,,:/home/fuky:/bin/bash
$ ls -l /bin/sh
lrwxrwxrwx 1 root root 4 čen 19 02:24 /bin/sh -> bash
```

Jestliže nemáte `/bin/bash` nastaven jako implicitní shell, napravte to následujícím příkazem a poté spusťte BASH, protože změny se projeví až po přihlášení.

```
$ usermod -s /bin/bash $USER
$ /bin/bash
```

Začínáme

Vypsání hodnot všech proměnných známých aktuálnímu interpretu příkazů (uvedl jsem jen některé z nich, je jich samozřejmě mnohem víc" class="emo">).

```
$ set
BASH=/bin/bash
LANG=cs_CZ
PS1='$ '
PS2='> '
PS4='+ '
_=noclobber
```

Proměnná `$PS1` definuje tvar primárního promptu (zobrazuje se, když shell čeká na zadání příkazu). V definici tvaru proměnných `$PSn` (n = 1, 2, 3, 4) můžeme použít sekvence se speciálním významem. Ukážeme si pouze některé z nich.

- `\t` - systémový čas (HH:MM:SS)
- `\u` - uživatelské jméno
- `\W` - název pracovního adresáře
- `\$` - pro roota #, jinak $

```
$ PS1='[\t] \W\$ '
```

```
[17:20:20] bash$
```
Proměnná `$PS2` je tvar sekundárního promptu (zobrazuje se když shell čeká na dokončení příkazu). Chcete-li pokračovat v zadávání příkazu na dalším řádku, stačí napsat \ a stisknout ENTER. Středník použijete při spušťení více příkazů najednou.

```
$ echo "Na dalším řádku je výpis příkazu who"; \
> who
Na dalším řádku je výpis příkazu who
fuky    tty1    Oct 18 10:37
```
Vypsání nastavení různých módů interpretu (uvedl jsem jen dva, je jich opět mnohem víc). Druhý příkaz zapíná mód `vi` a poslední ho znovu vypíná (takovýmto způsobem lze nastavit všechny módy).

```
$ set -o
history on
emacs on
$ set -o vi
$ set +o vi
```
Běžící program můžeme ukončit stiskem `CTRL+c` a standardní vstup (např. v níže uvedeném příkladu) můžeme ukončit stiskem CTRL+d, ale nejdříve musíme přejít na nový řádek.

```
$ wc
První, druhé, třetí,
čtvrté, páté, šesté,
sedmé
     3     7     48
```

Procesy a signály

Každý proces má svůj jedinečný identifikátor PID. Spuštěný proces je závislý na svém rodiči (na procesu, ze kterého byl spuštěn). Při ukončení rodiče budou automaticky ukončeni i všichni potomci. Pomocí příkazu `nohup` zajistíme nezávislost pro nově spouštěný proces a pomocí `&` ho spustíme na pozadí.

```
$ nohup ./skript.sh &
[5] 3043
$ nohup: appending output to `nohup.out'
```
V případě, že nyní ukončíme shell, bude proces s PID 3043 (náš skript) dál pracovat. Proces můžeme ukončit zasláním `SIGTERM` (dovolí procesu uložit data na disk a dobrovolně se ukončit), ale tento signál může proces ignorovat. Existují dva signály, které ignorovat nemůže, `SIGSTOP` (pozastaví proces) a `SIGKILL` (bez milosti proces zabije). Pro zaslání signálu můžeme použít `kill` nebo `killall` (POZOR ukončí všechny procesy zadaného názvu!). Použití ukazují následující příkazy (použijeme jeden z nich).

```
$ kill -SIGKILL 3043
$ killall -KILL skript.sh
[2]    Zabit (SIGKILL)      ./skript.sh
```
Stiskem CTRL+z zašleme právě běžícímu procesu signál `SIGSTOP`, zadáním příkazu `fg` ho opět probudíme a je-li proces na pozadí, umístí ho na popředí. Příkazem `bg` přesuneme naopak proces na pozadí. Nezadáme-li identifikátor procesu, je použit poslední použitý identifikátor v rámci aktivního shellu.

```
$ mp3blaster

[1]+  Stopped        mp3blaster
```

```
$ fg
mp3blaster
$ ./skript.sh

[2]+  Stopped                 ./skript.sh
$ bg
[2]+ ./skript.sh &
$ fg 2
./skript.sh
```

Speciální soubory

- `/etc/shells` - použitelné přihlašovací shelly
- `/etc/adduser.conf` - výchozí hodnoty pro adduser
- `/etc/profile` - načítaný při přihlášení
- `$HOME/.bash_profile` - načítaný při přihlášení
- `$HOME/.bashrc` - načítaný při startu interpretu
- `$HOME/.bash_logout` - načítaný při odhlášení
- `$HOME/.bash_history` - evidence naposledy prováděných příkazů

Editace příkazové řádky

Lze jí editovat jako ve dvou nejpoužívanějších (dle mého názoru i nejlepších) textových editorech vi, Emacs (není to "pouze" textový editor). Implicitní je mód `emacs`  " class="emo">. Zmíním pouze několik příkazů jako ukázku, zbytek si můžete dohledat v dokumentaci.

Než začnete zkoušet klávesové zkratky, ověřte si, jestli máte zapnutý mód `emacs`, případně ho zapněte.

```
$ set -o emacs
```

- `ESC b` - posun o jedno slovo zpět
- `ESC f` - posun o jedno slovo vzad
- `ESC d` - smazání následujícího slova
- `CTRL+Y` - vložení naposledy smazané položky
- `CTRL+K` - smazání textu do konce řádku
- `CTRL+R` - postupné vyhledávání v historii příkazů
- `ESC <` - posun na první řádek historie příkazů
- `TAB` - pokus o obecné dopnění textu
- `TAB TAB` - jestliže existuje více možností doplnění, vypíše je
- `ESC ~` - pokus o doplnění jména uživatele
- `CTRL+X ~` - vypíše možné alternativy doplnění jména uživatele
- `CTRL+X $` - vypíše možné alternativy doplnění jména proměnné
- `CTRL+X @` - vypíše možné alternativy doplnění jména počítače
- `ESC TAB` - pokusí se doplnit text z předchozích příkazů v historii

Klávesových zkratek a příkazů je opravdu hodně. Ale nemějte strach, časem vám přejdou do krve a znalost těch nejdůležitějších (z vašeho úhlu pohledu) se pro vás stane samozřejmostí. Když provádíte nějakou činnost v systému, vždy se ji snažte provádět co nejefektivnějším a nejrychlejším způsobem. Zkuste se zamyslet, jestli by to nešlo udělat lépe.

Své nové objevy si poznamenávejte na papír, který nechávejte ležet vedle klávesnice. Budete to mít všechno hezky po ruce, než si to potřebné pro vás častým používáním zapamatujete. Tak vám z papíru budou postupně některé věci ubývat a nové zase přibývat.

Základní příkazy, roury a přesměrování

Popis příkazů nebudu rozebírat do podrobností, od toho máme manuálové stránky. Jen stručně nastíním k čemu jednotlivé příkazy slouží. Abyste věděli, pod kterým příkazem se skrývá vámi požadovaná činnost, a měli jste se na začátku čeho chytit.

Základní příkazy

- `cp` - kopíruje soubory
- `rm` - ruší soubory
- `mkdir` - vytváří adresáře
- `rmdir` - ruší prázdné adresáře
- `ln` - vytvoří odkazy na soubory
- `chmod` - změní přístupová práva k souborům
- `ls, dir, vdir` - vypíše obsah adresářů
- `find` - vyhledávání souborů
- `which` - zobrazí absolutní cestu k programu
- `df` - vypisuje informace o připojených FS
- `ps` - informace o spuštěných procesech
- `cat, less` - výpis souboru na obrazovku
- `xargs` - spustí zadaný příkaz a zbylé argumenty čte ze standardního vstupu
- `grep` - tiskne řádky, které odpovídají zadanému vzoru
- `wc` - vypíše počet písmen, slov a řádků
- `sort` - setřídí řádky

Příklad použití archivačního programu tar (je to standardní nástroj, takže ho naleznete snad v každé distribuci).

```
$ tar zcvf archiv.tgz ./adresar
./adresar/
./adresar/obesenec.sh
./adresar/oggwavmp3.sh
./adresar/archmed.sh
$ tar zxf archiv.tgz
$ tar jcf archiv.tar.bz2 ./adresar
$ tar jxf archiv.tar.bz2
```

Mimo archivace tar použije i kompresi `z` - gzip, `j` - bzip2. Volba `x` - rozbalí archív, `c` - vytvoří archív, `v` - vypisuje informace.

Roury

Příkazy dostávájí opravdovou moc teprve díky rourám a přesměrováním. Roura (značí se pomocí operátoru `|`) připojuje výstup jednoho procesu na vstup druhého procesu.

Přesměrování

Operátory přesměrování.

- `>` - přesměrování standardního výstupu do souboru, jestliže soubor existuje bude přepsán
- `>>` - jako předchozí, ale data přídá na konec souboru
- `<` - přesměrování standardního vstupu do souboru

- `<<text` - jako předchozí, ale při výskytu řetězce `text` zašle znak konce souboru

Chcete-li zabránit přepsání souboru při přesměrování, můžete toto implicitní nastavení změnit následujícím příkazem.

```
$ set -o noclobber
```

Před operátorem přesměrování můžeme použít deskriptor souboru.

- `0` standardní vstup
- `1` standardní výstup
- `2` standardní chybový výstup

Dvě ukázky přesměrovaní standardního výstupu a standardního chybového výstupu do stejného souboru. Jako soubor použijeme `/dev/null` (o všechno, co do tohoto speciálního souboru přesměrujeme, přijdeme). Zkuste si příklad upravit tak, aby se vám na obrazovku vypisoval jen standardní chybový výstup a pak jen standardní výstup. Před zkoušením si nastavte jako aktuální adresář nějaký, který obsahuje podadresáře a soubory.

```
$ find | xargs cat &> /dev/null
$ find | xargs cat > /dev/null 2>&1
```

První příklad použití programu tar by šel zapsat i následujícím způsobem za použití roury a přesměrování do souboru.

```
$ tar cv ./adresar/ | gzip > archiv.tgz
./adresar/
./adresar/obesenec.sh
./adresar/oggwavmp3.sh
./adresar/archmed.sh
```

Praktický příklad

Potřebujeme vytvořit kontrolní součet všech souborů v aktuálním adresáři a jeho podadresářích za pomocí md5sum a uložit do souboru `md5sum.txt` (u tohoho souboru nebudeme provádět kontrolní součet).

Ukáži vám dvě řešení. To druhé jsem vytvořil, než jsem se v konferenci dozvěděl o příkazu `xargs`.

```
$ find . \! -path './md5sum.txt' -type f | xargs -i md5sum {} >
md5sum.txt
```

Program `find` předá programu `xargs` cestu ke všem souborům (na každém řádku je cesta k jednomu souboru), ten vezme řádek, dá ho do uvozovek a předá jako argument programu `md5sum`, načte další řádek... Dokud nezpracuje celý vstup. Standardní výstup programu `md5sum` se přesměruje do souboru `md5sum.txt`.

`find`

- `\!` - neguje následující podmínku
- `-path './md5sum.txt'` - najde soubory, jejichž jména odpovídají `'./md5sum.txt'`
- `-type f` - jsou nalezeny běžné soubory

`xargs`

- `-i` - všechny výskyty dvojice znaků `{}` jsou nahrazeny cestou k souboru ze standardního vstupu, mezery neuzavřené v uvozovkách nejsou považovány za ukončení argumentu

Je zbytečné psát takhle dlouhý příkaz, když ho budeme často používat. Proto si do souboru `~/.bashrc` přídáme alias.

```
alias md5sumr='find . \! -path './md5sum.txt' -type f | xargs -i
```

```
md5sum {} > md5sum.txt'
```
Po dalším spustění BASHE stačí, když zadáte jen `md5sumr`.

Druhé řešení je vytvoření skriptu `md5sumr.sh`. Je to jen pro ukázku, aby bylo vidět, že to jde udělat i mnohem složitějším způsobem.

```
#!/bin/bash

koren=$(pwd)
vystup="md5sum.txt"
cesta="./"

Md5sum() {
  local tmp

  for soubor in *; do
    if [ "$soubor" == "*" ]; then
      break
    fi

    if [ -d "$soubor" ]; then
      cd "./$soubor"
      tmp="$cesta"
      cesta="$cesta$soubor/"
      Md5sum
      cd "../"
      cesta="$tmp"
    else
      if [ "$soubor" != "$vystup" ] || [ "$cesta" != "./" ]; then
        pwd=$(pwd)
        cd "$koren"
        md5sum "$cesta$soubor" >> "./$vystup"
        cd "$pwd"
      fi
    fi
  done
}

Md5sum
```

Na příště si připravte svůj oblíbený editor. Jestli žádný takový ještě nemáte, určitě vyzkoušejte Emacs a vi a jeden z nich si vyberte, časem určitě oceníte jejich kvality. Výše uvedený kód jsem úmyslně nekomentoval. Až dočtete tento seriál, měli byste ho pochopit.

Proměnné

Jsou pouze jednoho datového typu - řetězec znaků. Některé z nich mohou být určeny jen pro čtení. Proměnné můžeme rozdělit do tří částí.

1. Vnitřní proměnné shellu. O jejich inicializaci se stará shell.

```
$ echo $USER
fuky
$ echo $OSTYPE
linux-gnu
$ echo $LANG
cs_CZ
$ echo $SHELLOPTS
```

```
braceexpand:hashall:histexpand:monitor:
history:interactive-comments:emacs
```

2. Uživatelské proměnné. Jako výše uvedené proměnné se skládají pouze z alfanumerických znaků.
3. Proměnné speciálního významu, skládají se ze speciálních znaků. Například:
    - `$$` - PID shellu
4. `$!` - PID posledního procesu, který byl spuštěn na pozadí
5. `$?` - návratová hodnota posledního dokončeného procesu.

Proměnnou můžeme exportovat příkazem `export` do podřízeného shellu a příkazem `readonly` zajistíme, že bude určena pouze pro čtení (POZOR, toto omezení se nepřenáší do podřízeného shellu). Když chceme získat hodnotu proměnné, napíšeme před ni znak `$`. Ale když jí např. hodnotu přiřazujeme, nepíšeme před ní znak dolaru. Pro odstranění proměnné použijeme příkaz `unset`.

```
$ jedna="Lokální proměnná"
$ export DVA="Proměnná exportovaná do podřízeného shellu"
$ readonly TRI="Proměnná určená pouze pro čtení, ale jen na
lokální urovni"
$ export TRI
$ export
declare -x DVA="Proměnná exportovaná do podřízeného shellu"
declare -rx TRI="Proměnná určená pouze pro čtení, ale jen na
lokální urovni"
$ readonly
declare -rx TRI="Proměnná určená pouze pro čtení, ale jen na
lokální urovni"
$ echo $jedna
Lokální proměnná
$ TRI="Nová hodnota"
bash: TRI: readonly variable
$ bash
$ TRI="Nová hodnota"
$ echo $jedna

$ echo $DVA
Proměnná exportovaná do podřízeného shellu
$ echo $TRI
Nová hodnota
$ unset TRI
```

První skript

Nadešel čas pro napsání a spuštění našeho prvního skriptu. Pak se ještě na chvíli vrátíme k proměnným. Pojmenujeme ho `prvni.sh`.

```
#!/bin/bash

# Tento skript nepotřebuje žádné komentáře
echo "Náš první skript byl právě spuštěn a za 3 vteřiny bude
ukončen."
sleep 3
echo "Konec."
```

`exit 0`

Prvním řádkem zajistíme, že náš skript bude opravdu interpretován BASHEM. To je jediná výjimka při použití znaku #, řádka začínající tímto znakem je ignorována a slouží k okomentování zdrojového kódu. Každý správný programátor používá ve svých kódech komentáře. Když se k němu po čase vrátí, dříve ho pochopí a také zjednoduší pochopení ostatním. Potřeba naučit se správnému používání komentářů přijde časem sama. Uvidíte, kde jsou zbytečné a kde naopak velice důležité (POZOR, komentáře se píší ihned se zdrojovým kódem - podle mě není dobrý zvyk je psát až po dokončení programu). A nakonec samozřejmě nezapomeneme vrátit návratový kód `exit 0`.

Nyní si skript spustíme, ale nejprve musíme přidat právo pro spuštění, protože textový editor toto právo standardně k nově vytvořeným souborům nepřidává.

```
$ ls -l
-rw-r--r--    1 fuky    fuky       114 říj 19 14:43 prvni.sh
$ chmod +x ./prvni.sh
$ ls -l
-rwxr-xr-x    1 fuky    fuky       114 říj 19 14:43 prvni.sh
$ ./prvni.sh
Náš první skript byl právě spuštěn a za 3 vteřiny bude ukončen.
Konec.
```

Proměnné - dokončení

Nyní, když umíme spouštět skripty, tak si ukážeme na skriptu `promenne.sh` ještě několik zajímavých věcí.

```
#!/bin/bash

prvni="Níže uvedený zá"
echo "${prvni}pis umožní oddělit proměnnou od okolního textu"

# Kdyby byla $druha definována, byla by vrácena její hodnota,
# jelikož není, bude vrácen "náhradní výraz"
echo ${druha-"náhradní výraz"}
echo $druha

# To samé jako předchozí, ale $treti nezůstane nedefinovaná
echo ${treti="náhradní výraz"}
echo $treti

ctvrta="Příšerně žluťoučký kůň úpěl ďábelské ódy."

# Vrátí "náhradní výraz" je-li proměnná definována, jinak
# se nevrací žádná hodnota
echo ${ctvrta+"náhradní výraz"}
echo $ctvrta

# Vypíše délku $ctvrta
echo ${#ctvrta}

# Od konce odstraní nejkraší část $ctvrta, která odpovídá e*
echo ${ctvrta%e*}

# Od konce odstraní nejdelší část $ctvrta, která odpovídá e*
echo ${ctvrta%%e*}

# Od začátku odstraní nejkraší část $ctvrta, která odpovídá *e
echo ${ctvrta#*e}
```

```
# Od začátku odstraní nejdelší část $ctvrta, která odpovídá e*
echo ${ctvrta##*e}

exit 0
```

Ještě si skript spustíme pro lepší pochopení.

```
# ./promene.sh
Níže uvedený zápis umožní oddělit proměnnou od okolního textu
náhradní výraz

náhradní výraz
náhradní výraz
náhradní výraz
Příšerně žluťoučký kůň úpěl ďábelské ódy.
41
Příšerně žluťoučký kůň úpěl ďáb
Příš
rně žluťoučký kůň úpěl ďábelské ódy.
lské ódy.
```

V shellu si ještě vyzkoušíme několik příkazů, abychom pochopili, jak je to s uvozovkami, apostrofy a expanzí.

```
$ echo $promenna
./promenne.sh ./prvni.sh
$ echo '$promenna'
$promenna
$ echo "${promenna}vni.sh"
./*vni.sh
$ echo ${promenna}vni.sh
./prvni.sh
$ echo ${promenna}vni.pdf
./*vni.pdf
$ echo "$(echo $promenna) - výpis adresáře"
./promenne.sh ./prvni.sh - výpis adresáře
```

Podmínky

Skript `if.sh` nám ukáže použití konstrukce

```
if výraz; then příkazy elif výraz; then příkazy else příkazy fi
```

```
#!/bin/bash

if [ "$USER" == "root" ]; then
  echo "Ahoj admine";
fi

if [ "$USER" == "root" ]; then
  echo "Ahoj admine";
else
  echo "Ahoj uživateli";
fi

if [ "$USER" == "root" ]; then
  echo "Ahoj admine";
elif [ "$USER" == "fuky" ]; then
```

```
  echo "Ahoj Honzíku";
else
  echo "Ahoj uživateli";
fi

exit 0
```

POZOR, mezera za `[` je důležitá! Znak `[` je totiž program a to, co následuje za ním, jsou jeho argumenty.

```
$ which [
/usr/bin/[
```

Jak jsem už jednou říkal, všechny proměnné v shellu jsou jednoho datového typu. To vysvětluje, proč se řetězce a čísla porovnávají níže popsaným způsobem (`výraz`, `výraz1`, `výraz2` vrací řetězec a teprve když ho chceme porovnávat jako číslo, tak ho shell bere jako číslo, jinak to je stále řetězec).

- `[ výraz ]` - délka řetězce je nenulová
- `[ -z výraz ]` - délka řetězce je nulová
- `[ výraz1 == výraz2 ]` - řetězce jsou shodné
- `[ výraz1 != výraz2 ]` - řetězce jsou různé
- `[ výraz1 -eq výraz2 ]` - čísla jsou shodná
- `[ výraz1 -le výraz2 ]` - výraz1 <= výraz2
- `[ výraz1 -lt výraz2 ]` - výraz1 < výraz2
- `[ výraz1 -ge výraz2 ]` - výraz1 >= výraz2
- `[ výraz1 -gt výraz2 ]` - výraz1 > výraz2
- `[ výraz1 -ne výraz2 ]` - čísla jsou různé

Testování souborů.

- `[ výraz1 -ef výraz2 ]` - soubory sdílejí stejný i-uzel
- `[ výraz1 -nt výraz2 ]` - první soubor je novější
- `[ výraz1 -no výraz2 ]` - první soubor je starší
- `[ -e výraz ]` - soubor existuje
- `[ -d výraz ]` - soubor je adresář
- `[ -f výraz ]` - soubor je obyčejný soubor
- `[ -L výraz ]` - soubor je symbolický odkaz
- `[ -w výraz ]` - soubor je zapisovatelný
- `[ -x výraz ]` - soubor je spustitelný

Místo `[` můžete používat `test`. Jsou to stejné programy svázané pevným odkazem.

```
$ if test /usr/bin/test -ef /usr/bin/\[; then echo "Je to opravdu
tak..."; fi
Je to opravdu tak...
$ if [ /usr/bin/test -ef /usr/bin/\[ ]; then echo "Je to opravdu
tak..."; fi
Je to opravdu tak...
```

Podmínky samozřejmě můžete spojovat pomocí operátorů `&&` (a zároveň platí) a `||` (nebo platí).

```
# if [ $USER == "root" ] && [ $LANG == "cs_CZ" ]; then
> echo "Jsi český admin"
> fi
Jsi český admin
```

Na skriptu `case.sh` se podíváme na použití konstrukce

`case slovo in vzory ) příkazy;; ... esac`:

```
#!/bin/bash

case "$USER" in
  root )
    echo "Ahoj admine"
    ;;
  fuky )
    echo "Ahoj Honzíku"
    ;;
  * )
    echo "Ahoj uživateli"
    ;;
esac

case "$USER" in
  root | fuky )
    echo "Ahoj Honzíku"
    ;;
  * )
    echo "Ahoj uživateli"
    ;;
esac

exit 0
```

Cykly

Pro tento díl poslední skript `cykly.sh` nás zasvětí do používání cyklů `for`, `while` a `until`. Podle mě je dobrým zvykem uzavírat proměnné v podmínkách do uvozovek, protože kdyby proměnná obsahovala např. mezeru nebo nic, došlo by k chybě.

```
#!/bin/bash

# Vypíše všechny soubory v adresáři s příponou sh
for file in *.sh; do
  # Soubor je samozřejmě i adresář a co když nějaký šílenec
  # pojmenuje adresář jmeno_adresare.sh
  if [ -f "$file" ]; then
    echo $file
  fi
done

# Do $cislo bude postupně dosazovat čísla
for cislo in 10 20 30 40 50 60 70 80 90 100; do
  echo $cislo
done

cislo=0
# Podmínka je splněna jestliže $cislo != 100
while [ "$cislo" -ne 100 ]; do
  # Konstrukci $(()) zavedl shell ksh a je rychlejší a méně
  # náročná na systémové zdroje než příkaz expr
  cislo=$((cislo + 10))
  echo $cislo
done

cislo=0
# Cyklus pokračuje dokud není splněna podmínka
until [ "$cislo" -eq 100 ]; do
  cislo=$((cislo + 10))
  echo $cislo
```

```
done

exit 0
```

Informace o názvu skriptu, počtu předaných argumentů a argumenty samotné jsou uloženy ve speciálních proměnných.

- `$0` - název skriptu
- `$#` - počet předaných argumentů
- `$IFS` - seznam znaků, který je použit k oddělování slov atp., např. když shell čte vstup
- `$1` až `$9` - první až devátý argument předaný skriptu
- `${n}` - libovolný n-tý argument předaný skriptu
- `$*` - obsahuje všechny argumenty oddělené prvním znakem z `$IFS`
- `$@` - jako předchozí, ale k oddělení se nepoužívá první znak z `$IFS`

Skript `argumenty.sh` nám poslouží jako ukázka.

```
#!/bin/bash

echo "Název skriptu: $0"
echo "Počet argumentů: $#"
echo "Všechny argumenty: $@"
echo "První argument: $1"
echo "Desátý argument: ${10}"

exit 0
```

Nyní skript spustíme s 10 argumenty.

```
$ ./argumenty.sh jedna dva tři čtyři pět šest sedm osm devět deset
Název skriptu: ./argumenty.sh
Počet argumentů: 10
Všechny argumenty: jedna dva tři čtyři pět šest sedm osm devět
deset
První argument: jedna
Desátý argument: deset
```

Funkce

Provádění funkcí je mnohem rychlejší než provádění skriptů, protože funkce si shell udržuje trvale předzpracované v paměti. Funkce musí být definována dříve než bude použita. Příkaz `export` lze použít i pro funkce, ale musí být zapnutý mód `allexport`.

```
$ set -o allexport
$ prvni_funkce() {
> echo "Jsem první funkce a vypisuji text"
> }
$ export prvni_funkce
$ prvni_funkce
Jsem první funkce a vypisuji text
$ bash
$ prvni_funkce
Jsem první funkce a vypisuji text
```

Funkcím můžeme předávat argumenty stejně jako skriptům a získáváme je stejným způsobem jako u

skriptů. Příkaz `return` ukončí funkci a vrací její návratovou hodnotu ve formě celočíselného argumentu. Po dokončení funkce jsou poziční argumenty skriptu (`$#`, `$@` ...) obnoveny (u starších shellů to tak být nemusí).

```
$ funkce_s_argumenty() {
> echo "Počet argumentů: $#"
> echo "Všechny argumenty: $@"
> echo "První argument: $1"
> return 0
> }
$ funkce_s_argumenty první druhý
Počet argumentů: 2
Všechny argumenty: první druhý
První argument: první
```

Budeme-li chtít vrátit řetězcovou hodnotu, můžeme to udělat např. níže uvedeným způsobem.

```
#!/bin/bash

vrat_retezec() {
  echo "Řetězec"
}

promena=$(vrat_retezec)
echo $promena

exit 0
```

Pomocí klíčového slova `local` můžeme také vytvořit lokální proměnné funkce. Jestliže bude existovat globální proměnná se stejným názvem, bude ve funkci potlačena.

```
#!/bin/bash

jedna="První globální proměnná"
dva="Druhá globální proměnná"

lokalni_promena() {
  local jedna="První lokální proměnná"

  echo $jedna
  echo $dva
}

lokalni_promena

echo $jedna
echo $dva

exit 0
```

Příkazy

Příkazy můžeme rozdělit na zabudované a normální. Zabudované příkazy nemůžeme spustit jako externí programy, ale většinou mají své ekvivalenty ve formě externích programů. Normální příkazy jsou externí programy a jejich vykonání je pomalejší než u zabudovaných příkazů.

- `break` - vyskočí z cyklu
- `:` - nulový příkaz
- `continue` - spustí další iteraci cyklu

- `.` - provede příkaz v aktuálním shellu
- `eval` - vyhodnotí zadaný výraz
- `shift` - posune poziční parametry
- `read` - načte uživatelský vstup, jako argument se použije název proměnné, do které se má uložit
- `stty` - mění a vypisuje charakteristiky terminálové linky
- `exec` - spustí nový shell nebo jiný zadaný program a nebo upraví deskriptor souboru
- `exit n` - ukončení skriptu s návratovým kódem n (n = 0 - úspěšné ukončení, n = 1 až 125 - chyba, ostatní n jsou rezervovány)
- `printf` - není dostupný ve starých shellech a při vytváření formátovaného výstupu byste mu měli dávat přednost před příkazem `echo` podle specifikace X/Open

Na skriptu `prikazy.sh` si ukážeme použití některých výše uvedených příkazů.

```
#!/bin/bash

for i in 10 20 30 40 50; do
  if [ $i -eq 40 ]; then
    break
  elif  [ $i -eq 20 ]; then
    continue
  else
    :
  fi
  echo $i
done

a="abc"
nazev_promene="a"

promena='$'$nazev_promene
echo $promena

eval promena='$'$nazev_promene
echo $promena

while [ "$1" ]; do
  echo $1
  shift
done

exec date

echo "Tato část již nebude provedena!"

exit 0
```

Nezapomeneme skript spustit s několika argumenty.

```
$ ./prikazy.sh první druhý třetí
10
30
$a
abc
první
druhý
třetí
St říj 22 16:08:36 CEST 2003
```

Nyní si ukážeme interaktivní skript `read.sh`, který požádá uživatele o zadání přihlašovacího jména

a hesla. Heslo se nebude vypisovat na obrazovku.

```
#!/bin/bash

echo -n "Přihlašovací jméno: "
read jmeno

echo -n "Heslo: "

# Vypne výpis vstupních znaků
stty -echo

read heslo

# Zapne výpis vstupních
stty echo
echo

if [ "$jmeno" == "fuky" ] && [ "$heslo" == "heslo" ]; then
  echo "Kód: Příšerně žluťoučký kůň úpěl ďábelské ódy"
else
  echo "Nemáte oprávnění k vypsání kódu"
fi

exit 0
```

Zadáme-li správné údaje, získáme kód.

```
$ ./read.sh
Přihlašovací jméno: fuky
Heslo:
Kód: Příšerně žluťoučký kůň úpěl ďábelské ódy
```

Na závěr tohoto dílu si ukážeme použití konstrukce
`select proměnná in hodnota1 ... hodnotaN; do příkazy; done`.

```
#!/bin/bash

echo "Zadejte vaše pohlaví"

select pohlavi in muž žena; do
  if [ "$pohlavi" ]; then
    echo "Jste $pohlavi"
    break
  else
    echo "$REPLY je nedefinovaná odpověď"
  fi
done

exit 0
```

Po spuštění příkazu `select` je uživatel vyzván, aby zadal číslo jedné z hodnot (`hodnota1 ... hodnotaN` v našem případě `muž` nebo `žena`). proměnná `$REPLY` obsahuje vždy hodnotu uživatelského vstupu. proměnná `$pohlavi` obsahuje hodnotu pouze v případě, že číslo odpovídá jedné z voleb. Dotaz se opakuje, dokud se neprovede v těle příkaz `break`.

```
$ ./select.sh
Zadejte vaše pohlaví
1) muž
2) žena
#? 3
```

```
3 je nedefinovaná odpověď
1) muž
2) žena
#? 1
Jste muž
```

Dokumenty here

Umožňují předat vstup příkazu ze samotného skriptu. Ukážeme si to na skriptu `here.sh`.

```
#!/bin/bash

cat <<EOF
\$USER=$USER
\$HOME=$HOME
\$SHELL=$SHELL
EOF

cat <<"EOF" | egrep 'J|u'
Jestliže nechceme expandovat proměnné, uzavřeme příznak určující
konec vstupu do uvozovek ($USER, $HOME, $SHELL).
EOF

exit 0
```

Ještě si skript spustíme.

```
# ./here.sh $USER=root
$HOME=/root
$SHELL=/bin/bash
Jestliže nechceme expandovat proměnné, uzavřeme příznak určující
konec vstupu do uvozovek ($USER, $HOME, $SHELL).
```

Metaznaky shellu

Lze je použít k neúplnému zadání jména souboru.

POZOR neztotožňujte metaznaky shellu s regulárními výrazy, jsou to dvě různé věci. Metaznaky expanduje přímo shell. A proto když chceme nějakému programu předat regulární výraz, musíme ho uzavřít například do apostrofů.

- `*` - libovolný řetězec (může být i nulové délky)
- `?` - libovolný jeden znak
- `~` - domovský adresář ($HOME)
- `~UJ` - domovský adresář uživatele UJ
- `~+` - aktuální pracovní adresář ($PWD)
- `~-` - předchozí pracovní adresář ($OLDPWD)
- `[abc...]` - jakýkoliv znak uvedený v [], lze použít - k zápisu intervalu znaků např a-z, 0-9
- `[!abc...]` - opak předchozího (tj. jakýkoliv znak mimo uvedených znaků v [])

První příkaz smaže zálohy souborů (soubory končící na ~). Znak ~ nebude v tomto případě expandován.

```
$ rm *~
```

```
$ ls dil*.html
dil2.html  dil3.html  dil4.html  dil5.html  dil6.html
$ ls [di]*.html
dil2.html  dil3.html  dil4.html  dil5.html  dil6.html  index.html
```

Regulární výrazy

Jsou (mými slovy, přesná definice je "trochu" složitější 😊) vzory, s jejichž pomocí lze definovat společné rysy několika různých řádků a tím pádem je reprezentovat jako jeden regulární výraz. Níže uvedené speciální znaky jsou použitelné např. v `grep`, `egrep`, `sed`, `ed`, `ex`, `awk`.

- `.` - jakýkoliv znak (mimo znaku nového řádku)
- `*` - libovolný počet (i nulový) opakování předchozího znaku (lze použít i regulární výraz)
- `^` - následující výraz musí odpovídat začátku řádku
- `$` - předchozí výraz musí odpovídat konci řádku
- `\` - vypíná speciální význam následujícího znaku
- `[]` - jakýkoliv znak uvedený v hranatých závorkách, speciální znaky zde mají normální význam, mimo − tu lze použít pro zápis intervalů (a-z atd.) a znak `^` uvedený jako první způsobí negaci (tj. jakýkoliv znak neuvedený v ...)

Použijeme programy `cat`, `grep` a všechno si poctivě vyzkoušíme.

```
$ cat << END > ./retezce.txt
> abclinuxu
> alfa
> aaa
> abcabcabc
> znak $
> a1a
> aAa
> END
$ cat ./retezce.txt | grep '.*'
abclinuxu
alfa
aaa
abcabcabc
znak $
a1a
aAa
$ cat ./retezce.txt | grep '.* \$'
znak $
$ cat ./retezce.txt | grep '^a[a-z]*a$'
alfa
aaa
$ cat ./retezce.txt | grep '^a[a-z0-9]*a$'
alfa
aaa
a1a
```

Filtry

Jsou programy, které ze vstupu podle zadaného vzoru odfiltrují jen námi požadovaná data a pošlou je na výstup. Jsou jimi např. `grep`, `egrep` (`grep -E`) a `fgrep` (`grep -F`), jsou to vlastně stejné programy. Pro nás je důležité, že `grep` používá pro zápis regulárních výrazů starší notaci a `egrep` naopak novější notaci. Níže uvedené speciální znaky patří do novější notace a chceme-li je použít ve filtru `grep`, musíme před ně zapsat znak `\`.

- `+` - jeden a více výskytů předchozího výrazu.
- `?` - jeden nebo žádný výskyt předchozího výrazu.
- `|` - předcházející nebo následující výraz.
- `()` - text odpovídající výrazu mezi závorkami se uloží do paměti a lze ho použít pomocí `\1` až `\9`, čísluje se od vnějších závorek směrem dovnitř (např. `((abc)linuxu) \1` = "abclinuxu") a `\2` = "abc". Nebo lze použít závorky k definování priority vyhodnocení.
- `{n,m}` - interval opakování předchozího výrazu, {n} - opakuje se n-krát, {n,} n-krát a více, {n,m} n-krát až m-krát

Pro lepší pochopení uvedu opět několik příkladů.

```
$ cat ./retezce.txt | grep '^a\+$'
aaa
$ cat ./retezce.txt | egrep '^a+$'
aaa
$ cat ./retezce.txt | egrep '^abcl?'
abclinuxu
abcabcabc
$ cat ./retezce.txt | egrep '^c|z'
znak $
$ cat ./retezce.txt | egrep '(abc)+'
abclinuxu
abcabcabc
$ cat ./retezce.txt | egrep '^(.*)\1\1$'
aaa
abcabcabc
$ cat ./retezce.txt | egrep '^a{3}$'
aaa
$ cat ./retezce.txt | egrep '^a{2,}$'
aaa
$ cat ./retezce.txt | egrep '^a{1,3}$'
aaa
```

Proudové editory

Z názvu je zřejmé, že slouží k proudové editaci dat. O načítání vstupu se starají sami. Mají k dispozici sadu příkazů, pomocí které data upravují (obvykle pracují s jedním řádkem), např. `sed` a nebo na složitější věci `awk`.

Sed

Syntaxe příkazu:

`Začátek,Konec!InstrukceArgumenty`

- `Začátek` - číslo řádku ($ značí poslední řádek) nebo /regulární výraz/
- `Konec` - číslo řádku nebo /regulární výraz/
- `!` - neguje předchozí body
- `Instrukce` - mají jedno písmeno
- `Argumenty` - k některým instrukcím

Není-li uveden `Začátek` a `Konec`, aplikuje se instrukce na každý vstupní řádek. Je-li uveden pouze `Začátek`, aplikuje se instrukce pouze na odpovídající řádek (či řádky) a je-li uvedeno obojí, tak od řádku odpovídajícímu `Začátek` se budou aplikovat instrukce a od řádku odpovídajícímu `Konec` se aplikovat přestanou. Níže jsou uvedeny některé `Instrukce` a jejich `Argumenty`.

- `s/vzorek/náhrada/příznaky` - nahradí první nalezený `vzorek náhradou`.

`Příznaky`: `n` - nahradí n-tý výskyt `vzorku` (1 až 512), `g` - nahradí všechny výskyty vzorku.

- `w soubor` - do souboru uloží vstupní řádek (řádky)
- `r soubor` - soubor načte do vstupu
- `p` - vypíše vstupní řádek na výstup
- `n` - přesune se na další vstupní řádek
- `d` - vstupní řádek je smazán
- `y/původní znaky/nové znaky/` - přeloží znaky (`man tr`)
- `:` - označí řádek skriptu pro odskok `Instrukcí t` nebo `b`
- `t` - byla-li provedena substituce, skočí na následující značku `:`, není-li uvedena, skočí na konec skriptu
- `{}` - zajistí aplikaci více příkazů na jednu adresu

```
$ cat ./retezce.txt | sed '2,$s/a/?/g'
abclinuxu
?lf?
???
?bc?bc?bc
zn?k $
?1?
?A?
$ cat ./retezce.txt | sed -n '2p'
alfa
$ cat ./retezce.txt | sed -n '1{
> n
> p
> }'
alfa
$ cat ./retezce.txt | sed '2p
> d'
alfa
$ cat ./retezce.txt | sed '4y/a/?/
> 4!d'
?bc?bc?bc
```

Na závěr uvedu ještě jeden příklad ve formě skriptu `sed.sh`.

```
#!/bin/bash

spojka="je bydliště"
cat <<EOF | sed \
"s/^\(.\+j\) \(.\+\)o:\(.\+\)\$/\3 $s \1e \2a/
 t
 s/^\(.\+j\) \(.\+\):\(.\+\)\$/\3 $s \1e \2a/
 t
 s/^\(.\+\) \(.\+\)o:\(.\+\)\$/\3 $s \1a \2a/
 t
 s/^\(.\+\) \(.\+\):\(.\+\)\$/\3 $s \1a \2a/"
Petr Novák:Praha
Viktor Igo:Brno
Blažej Vodník:Plzeň
Jan Hugo:Hradec Králové
Metoděj Sporák:Ostrava
EOF

exit 0
```

Výstup skriptu vypadá následovně.

```
$ ./sed.sh
```

```
Praha je bydliště Petra Nováka
Brno je bydliště Viktora Iga
Plzeň je bydliště Blaženy Vodníka
Hradec Králové je bydliště Jana Huga
Ostrava je bydliště Metoděje Sporáka
```

V případě, že bychom chtěli zajistit správné skloňování úplně pro všechny jména a přijmení, určitě by výše uvedené řešení nebylo to nejkratší a nejvhodnější, berte ho pouze jako ukázku.

Odchytávání signálů

Signály zaslané skriptu můžeme odchytávat pomocí příkazu `trap`.

- `trap příkaz signál` - jestliže jako `příkaz` uvedeme znak "`-`", nastaví se pro `signál` původní akce a když `''`, neprovede se nic (`trap -l` vypíše signály, které lze odchytnout).

Vyzkoušejte skript `trap.sh`.

```
#!/bin/bash

konec() {
  echo -n "Uklízím"

  i=0
  while [ "$i" -le 10 ]; do
    i=$((i + 1))
    echo -n "."
    sleep 0,1
  done

  echo
  echo "Konec"
}

trap '' INT
echo "Ctrl+C neudělá nic"
sleep 3

trap - INT
echo "Ctrl+C ukončí skript"
sleep 3

trap 'konec; exit 0' INT
echo "Ctrl+C spustí funkci konec a ukončí skript"
sleep 3

konec

echo "Skript proběhl až do konce"

exit 0
```

Ladění skriptů

Následující módy shellu nám mohou usnadnit ladění.

- `verbose` - před vykonáním příkaz vypíše

- `xtrace` - jako předchozí, ale napřed provede expanzi; `$PS4` na začátku řádku určuje stupeň expanze
- `nounset` - je-li použita nedefinovaná proměnná, ukončí běh skriptu a vypíše chybovou hlášku

```
#!/bin/bash

set -o verbose
echo $PWD

set -o xtrace

echo $PWD

echo $(pwd)

set +o verbose
set +o xtrace

set +o nounset
echo $nedefinovana_promena

set -o nounset
echo $nedefinovana_promena

echo "Tento řádek se již nevypíše"

exit 0
```

Nyní si skript `ladeni.sh` spustíme a podíváme se na jeho výpis.

```
$ ./ladeni.sh
echo $PWD
/root/fuky/clanky/bash

set -o xtrace

echo $PWD
+ echo /root/fuky/clanky/bash
/root/fuky/clanky/bash

echo $(pwd)
pwd
++ pwd
+ echo /root/fuky/clanky/bash
/root/fuky/clanky/bash

set +o verbose
+ set +o verbose
+ set +o xtrace

./ladeni.sh: nedefinovana_promena: unbound variable
```

Praktické příklady

Úkol 1

Máme libovolnou adresářovou strukturu a v ní jsou uložené soubory *.wav, *.ogg a *.mp3.

- *.wav chceme převést do *.ogg a uložit do podadresáře `ogg`
- *.ogg chceme nahradit *.wav
- *.mp3 chceme nahradit *.wav

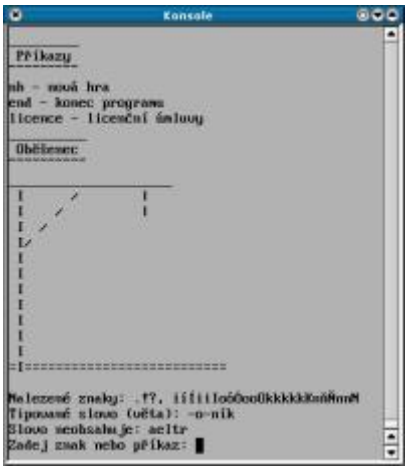
Vytvoříme si skript `oggwavmp3.sh`.

```
#!/bin/bash

case "$1" in
  *.wav )
    #cesta="${1%/*}/"
    cesta=$(echo $1 | sed
            's/^\(.\+\/\)\([^/]\+wav\)$/\1/')

    if [ -d "${cesta}ogg" ]; then
      :
    else
      mkdir "${cesta}ogg"
    fi

    #soubor="${1%.*}.ogg"
    #soubor="${soubor##*/}"
    soubor=$(echo $1 | sed
             's/^\(.\+\/\)\([^/]\+\)\.wav$/\2.ogg/')

    oggenc "$1" -Q -b 192 -o "${cesta}ogg/$soubor"
    ;;

  *.ogg )
    #soubor="${1%.*}.wav"
    soubor=$(echo $1 |
             sed 's/\(.\+\)\.\(ogg\)$/\1.wav/')

    if [ "$soubor" ]; then
      sox "$1" "$soubor"
      rm "$1"
    fi

    ;;

  *.mp3 )
    #soubor="${1%.*}.wav"
    soubor=$(echo $1 | sed
             's/\(.\+\)\.\(mp3\)$/\1.wav/')

    if [ "$soubor" ]; then
      mpg123 "$1" -q -w "$soubor"
      rm "$1"
    fi

    ;;

  * )
    ;;
esac

exit 0
```

Do souboru `~/.bashrc` si přidáme alias a po dalším spuštění shellu můžeme začít využívat náš nový příkaz.

```
alias oggwavmp3='find -type f | xargs -i ~/bash/oggwavmp3.sh {}'
```

Úkol 2



Na závěr si zkuste udělat hru Oběšenec. Nastavení hry se provádí pomocí konfiguračního souboru. Lze vybrat, zda se bude rozlišovat diakritika či velikost písmen. Slova či věty jsou uloženy v souborech podle složitosti a jedna z možností nastavení je právě výběr jen některých souborů se slovy (větami). A nakonec je zde i možnost časového omezení při zadávání znaku (příkazu) - v případě, že čas vyprší, bude hráč ihned oběšen. Samozřejmě, že vám dám k dispozici i mé řešení obesenec-1.1.tgz, ale podívejte se na něj až problém sami vyřešíte nebo budete-li opravdu v úzkých.

Literatura

- Linux začínáme programovat (Richard Stones, Neil Matthew)
- Linux příručka českého uživatele (Vilém Vychodil)
- Linux v kostce (Ellen Siever a další) [recenze]
- Učíme se RedHat Linux (Bill McCarty)

Závěr

A to je z našeho seriálu opravdu vše. Máte poslední možnost zeptat se na věci, které se do našeho seriálu nevešly, nebo se na ně zapomnělo. Děkuji všem, kteří seriál průběžně komentují.