Úvod do UNIXu – Honza „Stínovlas“ Musílek

Standardní vstup, výstup a chybový výstup

Standardní vstup a výstup, resp. chybový výstup jsou vstupní a výstupní kanály, kterými proces komunikuje se svým prostředím (s shellem). Standardní vstup `stdin` je obvykle přijímán z klávesnice, standardní výstup `stdout` a standardní chybový výstup `stderr` jsou vypisovány na obrazovku. `stderr` slouží především pro výpis chybových hlášení.

Každý vstup, resp. výstup má přiřazeno číslo. Toto číslo souvisí se *souborovým deskriptorem* v jazyce C.

```
0 – stdin
1 – stdout
2 – stderr
```

Přesměrování standardního vstupu a výstupů nám umožňuje odpojit je od klávesnice a obrazovky a místo toho číst vstup, resp. vypisovat výstup jinam – do souboru, na standardní vstup jiného příkazu apod. Díky přesměrování spolu mohou jednotlivé programy komunikovat. Jedná se o naprosto zásadní techniku – jsou-li jednotlivé příkazy základními kameny, které tvoří budovu programování v shellu, pak přesměrování je maltou, která je drží pohromadě.

Přesměrování `stdin`

Standardní vstup lze načíst ze souboru místo z klávesnice:

```
příkaz < soubor
wc -l < /etc/passwd   ... vypíše počet řádek souboru /etc/passwd
wc -l 0< /etc/passwd  ... to samé, ale s explicitním zmíněním čísla deskriptoru
                          u stdin se v praxi téměř nepoužívá
```

Jaký je rozdíl mezi voláním `wc -l < /etc/passwd` a `wc -l /etc/passwd`? V druhém případě vypíše `wc` kromě počtu řádek i název souboru. V prvním případě ale data přišla na standardní vstup – `wc` neví o tom, že pochází ze souboru `/etc/passwd` a ani ho to nezajímá. Vypíše tedy pouze počet řádek.

Přesměrování `stdout`

Standardní výstup můžeme mimo jiné přesměrovat do souboru. Přesměrování do souboru má dvě varianty – podle toho, zda chceme cílový soubor přepsat, nebo jen připsat data na jeho konec. V každém případě, pokud přesměrováváme výstup do souboru, který ještě neexituje, bude automaticky vytvořen.

```
date > datum.txt   ... zapíše do souboru datum.txt aktuální datum a čas; původní
obsah se přepíše
date >> datum.txt  ... připíše na konec souboru datum.txt aktuální datum a čas
```

Oba předchozí způsoby můžeme opět zapsat i s explicitním zmíněním čísla deskriptoru. Ani u standardního výstupu číslo deskriptoru obvykle neuvádíme, je však dobré si uvědomit, co se za `>` skutečně skrývá:

```
date 1> datum.txt
date 1>> datum.txt
```

Přesměrování `stderr`

Standardní chybový výstup můžeme stejně jako `stdout` přesměrovat do souboru a stejně jako v případě `stdout` se můžeme rozhodnout, zda soubor přepsat, či připisovat na jeho konec. V tomto případě však už skutečně musíme číslo deskriptoru uvést – žádný zkrácený zápis neexistuje.

```
ls /blabla 2> chyby.log
ls /blabla 2>> chyby.log
```

Výstup však nemusíme přesměrovat jenom do souboru. Někdy může být zajímavé přesměrovat `stderr` na `stdout` či naopak.

```
ls /bla > vystup.txt 2>&1   ... stderr je přesměrován na stdout a ten pak do souboru vystup.txt
ls /bla 2> error.txt 1>&2   ... stdout je přesměrován na stderr a ten pak do souboru error.txt
```

Roura

Roura je mechanismus, který vezme dva příkazy a přesměruje standardní výstup prvního z nich na standardní vstup druhého. Zapisuje se pomocí `|`.

```
ls -1 ~ | wc -l  ... spočítá počet souborů v domovském adresáři
```

Pokud chceme, můžeme rouru opsat pomocí obyčejných přesměrování a dočasného souboru. Ten pak ale po sobě musíme uklidit, takže roura je obvykle praktičtější (a rychlejší). Výhodou je, že si mezi těmito voláními můžete vypsat obsah souboru `docasny_soubor` a snáze tak pochopíte, proč roura dělá to co dělá. Předchozí příkaz by se tedy dal provést i takto:

```
ls -1 ~ > ~/docasny_soubor
wc -l < ~/docasny_soubor
```

Expanze `* ? []`

Expanze je činnost prováděná na úrovni shellu, která předtím, než spustí příkaz a předá mu parametry nahradí některé znaky či jejich posloupnosti něčím jiným. Prozatím se budeme bavit jen o expanzi na jména souborů. Obecné pravidlo je, že žádný z expanzních znaků `* ? []` se neexpanduje na řetězec obsahující lomítko. Pokud chceme vybrat soubory v podadresářích, musíme napsat `/` explicitně. Stejně tak žádný z expanzních znaků `* ? []` zapsaný na začátku vzoru neodpovídá skrytému souboru (začínajícímu tečkou). Pokud chceme vybrat skryté soubory, musíme `.` napsat explicitně.

Expanze `*`

Znak `*` se expanduje na libovolnou (i prázdnou) posloupnost znaků. Lépe to bude vidět na příkladu:

```
ls -d *             ... vypíše všechny neskryté soubory v pracovním adresáři
ls -d *.txt         ... vypíše všechny neskryté soubory s příponou .txt
ls -d *\ *          ... vypíše všechny neskryté soubory, které obsahují v názvu mezeru
ls -d /etc/*.conf   ... vypíše všechny neskryté soubory v adresáři /etc s příponou .conf
ls -d */*.jpg       ... vypíše všechny neskryté soubory s příponou jpg, které se nachází
                        v podadresáři pracovního adresáře
ls -d .*            ... vypíše všechny skryté soubory
```

Expanze `?`

Znak `?` se ve vzoru nahradí právě jedním libovolným znakem.

```
ls -d ???           ... vypíše všechny neskryté soubory s třípísmenným názvem
ls -d /tmp/*.??     ... vypíše všechny neskryté soubory v /tmp s dvoupísmennou
příponou
```

Expanze `[]`

Dovnitř závorek `[]` můžeme napsat libovolnou množinu znaků. Závorky pak odpovídají jednomu znaku z dané množiny. Pokud je ale prvním znakem v množině `!`, závorky naopak odpovídají libovolnému znaku, který není ze zadané množiny. Místo vyjmenování jednotlivých znaků můžeme zadat i rozsah (`[a-z]`, `[0-9]`, …). Třetí možností je použít některou z předdefinovaných znakových tříd:

```
[:alnum:] – písmena a číslice
[:lower:] – malá písmena
[:space:] – mezery
[:alpha:] – písmena
[:digit:] – číslice
[:print:] – tisknutelné znaky (tj. ty, které jsou při vypsání vidět)
[:upper:] – velká písmena
[:blank:] – bílé znaky (tj. ty, které při vypsání nejsou vidět)
[:punct:] – interpunkce
... a další
```

Použití `[]`:

```
ls -d *[0-9]*       ... vypíše všechny neskryté soubory, jejihž jméno obsahuje
číslici
ls -d *[!0-9]*      ... vypíše všechny neskryté soubory, jejihž jméno obsahuje
alespoň jednu nečíslici
```

Oprávnění

Upozornění: Domovské adresáře v labu jsou umístěny na síťovém filesystému AFS, který používá jiný systém práv, než popisuji dále (konkrétně ACL). Všechny pokusy a cvičení s právy tedy doporučuji provádět v adresáři `/tmp`, který je umístěn na lokálním disku a práva na něm fungují dle očekávání.

Každý soubor v UNIXu má nastavena *práva*, která určují kdo může se souborem manipulovat a jak. Každý soubor má nějakého *vlastníka* (owner), obvykle je jím ten, kdo soubor vytvořil, a*skupinu* (group). Vlastníka a skupinu, stejně jako práva souboru zjistíme pomocí `ls -l soubor`:

```
musilekj@u-pl0:~$ ls -ld /etc/passwd /etc/shadow /dev/null /tmp ~
drwxr-xr-x 53 musilekj nofiles      6144 23. úno 22.32 /afs/ms/u/m/musilekj
-rwxr-xr-x  1 root     root       693240 18. kvě  2013 /bin/bash
crwxrwxrwx  1 root     root         1, 3 25. zář  2006 /dev/null
-rw-r--r--  1 root     root         4151 10. říj 16.30 /etc/passwd
-rw-------  1 root     root         1351 10. říj 16.30 /etc/shadow
drwxrwxrwt 61 root     root    196317184 23. úno 22.34 /tmp
```

Třetí pole výpisu určuje vlastníka (`musilekj`, příp. `root`), čtvrté pole skupinu. `root` je speciální uživatel, správce systému, který má oproti ostatním uživatelům výjimečné pravomoci. K práci s ním se ale v labu nedostaneme.

Z hlediska oprávnění nás bude zajímat první pole výpisu, skládající se z deseti znaků. První z nich určuje typ souboru – `-` značí obyčejný soubor, `d` je adresář, `c` speciální znakové zařízení. Existují i další, postupem času se s nimi seznámíme. Z následujících devíti znaků určují první tři oprávnění vlastníka, další tři oprávnění skupiny a poslední tři oprávnění všech ostatních (tedy uživatelů, kteří nejsou vlastníkem daného souboru, ani nejsou ve skupině). Interpretace písmenek je následující:

```
normální soubory
r – právo ke čtení
w – právo k zápisu
x – právo ke spuštění
s – setuid / setgid bit – program bude spuštěn s právy vlastníka / skupiny

adresáře
r – právo vypsat obsah adresáře
w – právo vytvářet a mazat soubory v adresáři
x – právo vstoupit do adresáře
s – setgid bit – nově vytvořené soubory budou mít stejnou skupinu jako adresář
t – sticky bit – soubory v adresáři může mazat či přesouvat pouze jejich
vlastník a root
```

Z bezpečnostních důvodů není možné vzdát se vlastnictví souboru. Jediný kdo může změnit vlastníka souboru je root. Vlastník souboru může změnit jeho skupinu příkazem `chgrp`, na některých UNIXech je ale nutné, aby byl sám členem skupiny, do které soubor přiřazuje. Seznam skupin, ve kterých jste zařazeni získáte zavoláním příkazu `groups` (není v POSIXu). Práva změníme pomocí příkazu `chmod`:

```
chmod kdo[+-=]práva

kdo: u – vlastník, g – skupina, o – ostatní, a – všichni
+-=: + – přidej práva, - – odeber práva, = – nastav práva
práva: r w x s t – viz výše
```

vyjadřuje postupně nastavení setuid, setgid a sticky bitu, práva pro vlastníka, skupinu a ostatní. Pokud nenastavujeme setuid, setgid ani sticky bit, můžeme první číslici vynechat.

```
setuid bit – 4     r – 4
setgid bit – 2     w – 2
sticky bit – 1     x – 1

Příklad:
rwxr-xr-x = 755
rw-r----- = 640
rwxrwxrwt = 1777
rwsrwsrwt = 7777
```

Odkazy

Odkazy představují způsob, jak přistupovat k datům uložených v souborech z více různých míst. Umožní nám mít tentýž soubor umístěn ve více místech adresářového stromu zároveň.

Softlinky

Softlinky (symbolické odkazy) jsou speciálním typem souborů (pro účely příkazů `find` a `ls -l` se jedná o typ `l`). Můžeme je přirovnat k pointerům na data. Softlink je malý soubor, který v sobě obsahuje informaci o tom, na který jiný soubor ukazuje. Pokud onen soubor, na který se odkazujeme smažeme, softlink sice bude nadále exitovat, ale bude ukazovat na neexistující soubor a pokus o jeho vypsání skončí chybou. Softlinky můžeme vytvářet pomocí příkazu `ln -s soubor link`. Lze vytvářet i softlinky na adresáře.

Hardlinky

Hardlinky se chovají jinak. Jedná se o normální soubory, které pouze ukazují na stejné místo v paměti. Nelze říct, který z nich je původní a který kopie, oba jsou identické a změna obsahu jednoho změní i druhý. Ke každému inode patří počitadlo hardlinků. Tento počet vidíme v druhém sloupci výpisu `ls -l`. Adresáře musí mít vždy alespoň dva hardlinky – jeden z nadřazeného adresáře a jeden ze sebe sama (soubor `.`). Vždy když smažeme soubor, sníží se počitadlo hardlinků u jeho inode o jedna. Teprve ve chvíli kdy klesne na nulu jsou data skutečně nenávratně smazána a ztracena.

Hardlink vytvoříme pomocí příkazu `ln soubor hardlink`. Na současných systémech obvykle není možné vytvářet hardlinky na adresáře (z bezpečnostních důvodů).

Aritmetika v shellu

Aritmetika pro začátečníky

Kdykoliv shell narazí na výskyt `$(( ... ))`, nahradí text uvnitř závrek jeho aritmetickým vyčíslením. Uvnitř se smí používat proměnné, celočíselné konstanty, aritmetické operátory `+-*/%`, bitové operátory `&|^`, `>>`, `<<`, logické operátory `&&` a `||`, ternární operátor `expr?expr:expr`, operátory porovnání a přiřazení a libovolné množství mezer.

Uvnitř `$(( ... ))` nemusíme (ale můžeme) používat `$` před jménem proměnné, protože jediné řetězce znaků, které se smí uvnitř vyskytnout jsou právě jména proměnných. Aritmetická expanze umí pracovat pouze s celými čísly. I dělení je pouze celočíselné.

```
i=$((i+1))    ≈    i=$(($i+1))
x=$((2*3+1-i))
max=$(( a>b ? a : b ))
```

Aritmetika pro pokročilé

Potřebujeme-li pracovat s desetinnými čísly, mocnit je nebo dělat další operace, které aritmetická expanze nezvládá, můžeme se obrátit na program `bc`. Jedná se o kalkulátor s libovolnou přesností. Pokud spustíte `bc` bez parametrů, přepne se do interaktivního módu a očekává vstup z klávesnice. Můžeme mu ale také poslat výraz na standardní vstup a výsledek si přečíst ze standardního výstupu. Jediné co `bc` oproti aritmetické expanzi postrádá jsou bitové operátory.

Standardně počítá `bc` s přesností na 0 desetinných míst. Použití přepínače `-l` ji zvýší na 20. Pokud chceme počítat na jiný počet desetinných míst, můžeme to `bc` dát na vědomí nastavením scale:

```
echo "2*3-5+$y" | bc
printf "%s\n" 'scale = 50; 104348/33215' | bc
```

Návratové hodnoty

Každý unixový proces má návratovou hodnotu, kterou vrátí ve chvíli svého ukončení. Tuto hodnotu můžeme testovat např. příkazem v konstrukci `if` nebo `while`. Jaké konkrétní hodnoty vrací který příkaz se můžeme dočíst v jeho manuálové stránce, v sekci `EXIT STATUS`. Obvykle platí, že návratová hodnota `0` znamená úspěšné ukončení, jakákoliv kladná hodnota chybu. Např. v manuálové stránce příkazu `ls` se dočteme:

```
EXIT STATUS
       0      if OK,
       1      if minor problems (e.g., cannot access subdirectory),
       2      if serious trouble (e.g., cannot access command-line argument).
```

Logické spojky

Jednotlivé příkazy (nebo sekvenci příkazů oddělených rourou) můžeme oddělit pomocí logických spojek `&&` a `||`. To nemá smysl pouze v podmínkách, ale i pro normální zřetězení dvou příkazů. V UNIXovém shellu totiž funguje tzv. zkrácené vyhodnocování. Pokud je použita logická spojka `&&` a návratová hodnota prvního příkazu je false, UNIX ví, že bez ohledu na hodnotu druhého příkazu už bude výsledek false a tak jej vůbec neprovede. Stejně tak, pokud je použita logická spojka `||` a návratová hodnota prvního příkazu je true, UNIX ví, že výsledek bude true a tak druhý příkaz vůbec

neprovede.

```
mkdir a 2>/dev/null || echo "Adresář se nepodařilo vytvořit."
cd /root && ls
[ "$a" = "$b" ] && echo "$a a $b jsou si rovny."
```

For cyklus

V shellu neexistuje for cyklus standardní C-syntaxe. Místo toho umí opakovat blok příkazů pro všechny položky seznamu. Můžeme také použít už známou expanzi na jména souborů:

```
for name in Karel Jakub Honza; do
    printf "Jmenuji se %s.\n" "$name"
done

for file in *; do
    wc -c "$file"
done
```

Co asi budete používat nejčastěji je for pro nějaký pevný počet opakování. Toho snadno docílíte použitím příkazu `seq`, který vygeneruje seznam celých čísel od-do, s volitelným zadáním velikosti skoku:

```
for i in $(seq 10 -1 0); do
    echo $i
    sleep 1
done
echo "BUMMMM!"
```

Pokud napíšete pouze `for param; do ... done`, bude se do `$param` postupně dosazovat hodnota všech argumentů zadaných programu na příkazové řádce.

Příkaz test

Příkaz `test` je naprosto zásadní pro psaní while cyklů a podmínek. Vyhodnotí jestli je jeho argument pravdivý nebo ne a podle toho zvolí návratovou hodnotu. Daleko častěji než přímo s příkazem `test` se setkáme s jeho variantou, kdy je výraz k vyhodnocení uzavřen do hranatých závorek `[ ... ]`. Závorky musí být od příkazu odděleny mezerou!

```
[ "$a" = "$b" ]    ... true, když jsou řetězce stejné
[ "$a" != "$b" ]   ... true, když jsou řetězce různé
[ $a -eq $b ]      ... true, když je $a = $b (číselně)
[ $a -ne $b ]      ... true, když je $a ≠ rovno $b
[ $a -gt $b ]      ... true, když je $a > $b
[ $a -ge $b ]      ... true, když je $a ≥ $b
[ $a -lt $b ]      ... true, když je $a < $b
[ $a -le $b ]      ... true, když je $a ≤ $b

[ -e "$file" ]     ... soubor $file existuje
[ -d "$file" ]     ... soubor $file existuje a je adresář
[ -f "$file" ]     ... soubor $file existuje a je obyčejný soubor

[ -n "$str" ]      ... $str není prázdný řetězec
[ -z "$str" ]      ... $str je prázdný řetězec

[ ! expr ]         ... true, když je expr false
[ expr1 -a expr2 ] ... true, když jsou expr1 i expr2 true
[ expr1 -o expr2 ] ... true, když je expr1 nebo expr2 true
```

Můžeme také použít závorky `()` pro určení priorit.

Podmínka `if`

```
if blok příkazů 1; then
    ...
elif blok příkazů 2; then
    ...
else
    ...
fi
```

To, zda je podmínka splněna nebo ne závisí na návratové hodnotě posledního z příkazů v bloku příkazů 1. Větve `elif` a `else` lze pochopitelně vynechat. Podmínky bude ve většině případů reprezentovat příkaz `test`:

```
if [ $n -eq 1 ]; then
    echo "<th>$n</th>"
else
    echo "<td>$n</td>"
fi
```

While cyklus

```
while blok příkazů 1; do
    blok příkazů 2
done
```

While bude provádět blok příkazů 2, dokud je návratová hodnota bloku příkazů 1 rovna 0. Pokud se v bloku příkazů 1 nachází více příkazů oddělených středníkem, zajímá jej jenom poslední návratová hodnota. Obvykle bude oním příkazem příkaz `test`:

```
i=0
while [ $i -le 42 ]; do
    echo $i
    i=$((i+1))
done
```

Příkaz read

Příkaza `read` umí načítat text ze standardního vstupu a ukládat jej do proměnných. Jeho syntaxe je prostě `read x y z ...`, kde `x y z` jsou jména proměnných. `read` vždy načte celý řádek a do zadaných proměnných načte postupně obsah prvního pole, druhého pole, … až do poslední proměnné načte celý zbytek řádku (včetně oddělovačů).

Čím jsou oddělena pole? Jedním ze znaků uložených v proměnné `$IFS`. Její defaultní hodnota je `␣\t\n` (tedy mezera, tabulátor a nový řádek). Její hodnotu můžeme změnit, ale musíme to udělat opatrně – tak, abychom ovlivnili pouze příkaz `read` a žádný jiný. To nám umožní přiřazení do proměnné bezprostředně předcházející volání příkazu – `IFS=: read x y z`.

Čtení ze souboru

Čtení ze souboru není žádná magie, stačí nám přesměrovat soubor na standardní vstup příkazu `read`: `read x y < soubor.txt`. Pokud ale chceme vytvořit while cyklus, který přečte všechny řádky souboru, musíme si pomoci jinak:

```
while IFS=: read x _ y _; do
    printf "%s -- %s\n" "$x" "$y"
done < /etc/passwd
```

Signály a jejich odchytávání

Signály jsou jedním ze způsobů (vedle např. standardního vstupu), jak může vnější prostředí komunikovat s procesem po dobu jeho běhu. Ve chvíli, kdy proces zachytí zaslaný signál, nastane přerušení jeho běhu – nejprve je obsloužen daný signál a poté teprve pokračuje běh programu tam, kde byl přerušen. Seznam všech signálů můžeme získat pomocí `kill -l`.

kill

Signály můžeme posílat programem `kill`. Syntaxe je `kill [-signal] pid`, kde `signal` je buď jméno, nebo číslo signálu a `pid` je číslo procesu (uložené v proměnné `$$` a zjistitelné pomocí `ps`). Pokud neuvedeme číslo signálu, zašle se 15 `SIGTERM`. POSIX zaručuje existlenci následujících signálů:

```
1  SIGHUP    (hang up)
2  SIGINT    (interrupt)
3  SIGQUIT   (quit)
6  SIGABRT   (abort)
9  SIGKILL   (kill)
14 SIGALRM   (alarm)
15 SIGTERM   (terminate)
```

S některými z nich už jste se mohli setkat. Pokaždé, když ukončujete běžící program pomocí `<Ctrl-C>`, posíláte mu signál `SIGINT`, když ukončujete vstup pomocí `<Ctrl-D>`, posíláte `SIGQUIT`.

trap

Všechny výše zmíněné signály (kromě `SIGKILL`) můžeme odchytávat pomocí příkazu `trap`. To znamená, že zadefinujeme jakým způsobem se má přerušení zpracovat – např. při zavolání `SIGINT` můžeme nejdřív uložit rozdělanou práci a poté teprve program ukončit. Nebo jej dokonce můžeme ignorovat a neukončit se vůbec.

Syntaxe je `trap 'příkazy' signály`, kde `signály` jsou posloupností čísel nebo jmen příkazů, bez úvodního `SIG`. Pokud uvedeme `trap '' TERM`, signál se bude ignorovat.

Nesmíme zapomenout, že ve chvíli kdy odchytáváme signály sami, už se nezpracovávají standardním způsobem – tj. pokud si přejeme ukončit program, musíme to udělat sami příkazem `exit`.

`SIGKILL` je neodchytitelný signál. Každý proces můžete ukončit pomocí `kill -9 pid`. Na druhou stranu z toho plyne, že takový proces nemá šanci po sobě uklidit dočasné soubory apod., takže je vždycky lepší jít na něj po dobrém a poslat mu nejdříve `SIGTERM`.

Awk, 1. díl

Awk je programovací jazyk, orientovaný na práci s textem. Text, který zpracovává dostává stejně jako `sed` buď na standardním vstupu, nebo ze souboru.

Když programujete v awk, můžete buď psát samostatné awkové skripty (podobně jako pro program `sed`). Takové skripty započnete shabangem `#!/usr/bin/awk -f`. Druhou možností je spouštět awk přímo z příkazové řádky pomocí příkazu `awk 'text skriptu'`.

Základní struktura

Základní příkazy v `awk` mají podobu `VZOR {pravidlo}`. Příklady:

```
/foo/ { print }   ... vypíše pouze řádky odpovídající regulárnímu výrazu
BEGIN { sum=0 }   ... před načtením vstupu nastaví proměnnou sum na 0
END { print sum } ... po ukončení vstupu vypíše obsah proměnné sum
{ print $1 }      ... vypíše 1. pole ze všech řádků (VZOR je prázdný)
x==1 { print $2 }      ... vypíše 2. pole záznamu, pokud je proměnná x rovna 1
```

Vzorů a pravidel můžeme mít samozřejmě v jednom skriptu více – pro každý záznam se provedou všechna pravidla, která odpovídají vzorům, tedy nejen první vyhovující. Bloky `BEGIN` a`END` se vždy provádí na začátku, resp. na konci skriptu – nezáleží na tom, jak jsou umístěny vůči ostatním pravidlům.

Záznamy a pole

V předchozích příkladech jsme používali pojmy záznam a pole. Awk načítá vstupní text po záznamech a pro každý záznam je jeho i-té pole dostupné v proměnné `$i` (tedy první pole v `$1`, druhé v `$2`, atd. Pozor, tyto proměnné nemají nic společného s pozičními parametry předanými shellu na příkazové řádce, jen používají stejný název.

Pokud neurčíme jinak, awk bude za záznamy považovat jednotlivé řádky a za pole jednotlivá slova (řetězce oddělené mezerami). Chceme-li říci můžeme nastavit proměnnou `RS`, příp. `FS`, která specifikuje jaký znak od sebe odděluje jednotlivé záznamy, resp. pole záznamu. Pokud nastavíme `RS=""`, bude se text číst po odstavcích a pole budou jednotlivé řádky (bez ohledu na nastavení `FS`. Příklad:

```
BEGIN { RS=""; FS="\n" }
{ print $1 }
```

Speciální proměnné

- `RS` – record separator – oddělovač záznamů ve vstupním textu.
- `FS` – field separator – oddělovač polí ve vstupním textu.
- `ORS` – output record separator – oddělovač záznamů ve výstupu.
- `OFS` – output field separator – oddělovač polí ve výstupu.
- `NR` – number of record – číslo aktuálně zpracovávaného záznamu.
- `NF` – number of fields – počet polí v aktuálně zpracovávaném záznamu.

For cyklus

For cyklus v awk je totožný s for cyklem v jazyce C. Má tři části oddělené středníky – blok který se provede před prvním cyklem, ukončovací podmínku a blok, který se provede po každém cyklu. Příklad:

```
BEGIN {for(i=0;i<10;i++){print i}}
```

Awk, 2. díl

Pokračování vzorů

Už známe vzory ve formě regulárních výrazů `/^...$/`. Takto zapsaný vzor se porovnává oproti celému načtenému záznamu `$0`. Vzor můžeme omezit tak, aby vyhledával v jiném řetězci, operátorem `~`. Např. `$1 ~ /^...$/` odpovídá záznamům, jejichž první pole obsahuje právě 3 znaky. Naopak, vzor `$1 !~ /^...$/` odpovídá všem ostatním.

Vzory, ať už jsou to regulární výrazy, porovnávání proměnných, nebo ještě něco jiného, můžeme kombinovat logickými operátory `&&`, `||` a `!`, podobně jako v shellu.

Můžeme také provést nějakou akci pro rozsah záznamů. Pokud zapíšeme vzor ve formě `/start/,/stop/`, provede se akce pro všechny záznamy počínaje záznamem vyhovujícím vzoru `/start/` a konče záznamem vyhovujícím vzoru `/stop/`.

Další řídící konstrukce

Řídící konstrukce v awk jsou hodně podobné těm v C a jiných vyšších programovacích jazycích, proto je nebudu popisovat podrobně, ale uvedu jen přehled:

```
if (podmínka) příkaz [ else příkaz ]
while (podmínka) příkaz
do příkaz while (podmínka)
for (proměnná in pole) příkaz     # provede cyklus pro všechny prvky pole,
                                    v náhodném pořadí
break                             # ukončí provádění cyklu
continue                          # přeskočí na další iteraci cyklu
next                              # přeskočí na zpracování dalšího záznamu
delete pole[index]
exit                              # ukončí program (načte konec vstupu)
```

Pole

Pole v awk se indexují hranatými závorkami. Nemusíme je předem deklarovat, stačí rovnou zapisovat na libovolnou pozici uvnitř pole. Např. `{ x[NR] = $0 }` načte všechny záznamy do pole `x`. můřeme jednoduše používat i vícerozměrná pole, stačí od sebe indexy oddělit čárkou, např. `M[i,j]`. Poslední zajímavostí je, že indexy polí nemusejí být pouze čísla, můžeme použít také libovolné řetězce.

Další speciální proměnné

Kromě proměnných řídících načítání vstupu, o kterých jsme se bavili minule, má awk ještě řadu dalších užitečných proměnných.

```
ARGC     ... počet argumentů zadaných na příkazové řádce
ARGV     ... pole argumentů zadaných na příkazové řádce
FILENAME ... jméno právě zpracovávaného souboru, nebo - pro standardní
             vstup; v bloku BEGIN není definována
OFMT     ... výstupní formát čísel, defaultně %.6g
```

Funkce

Podobně jako v shellu, funkce nedeklarujeme, ale přímo definujeme. definice vypadá

způsobem `function jméno(seznam parametrů) { příkazy }`. Konkrétně např. `function plus(a, b) { return a+b; }`. Návratovou hodnotu vracíme klíčovým slovem return. Je potřeba dát pozor na jednu zradu – uvnitř funkcí nelze definovat žádné lokální proměnné. Tento problém řešíme tak, že přidáme seznam pomocných proměnných do hlavičky funkce, při volání je ale neuvádíme. Bývá zvykem oddělit pomocné proměnné od zbytku větším množstvím mezer. Příklad:

```
#!/usr/bin/awk -f
function sum(pole,   i, s) {
    s=0
    for (i in pole) s+=pole[i]
    return s
}

BEGIN {
    p[0]=1
    p[1]=2
    p[2]=3
    print sum(p)
}
```

V awk existuje množství předdefinovaných funkcí – matematických (sin, cos, exp, rand, …), řetězcových (length, sub, gsub, substr, split, …) a jiných. Jejich popis najdete v manuálové stránce awk.

Přepínače awk

Velmi užitečným přepínačem je `-v`. Umožňuje dovnitř skriptu dostat proměnnou zvenku. Např. při zavolání `awk -v x=10 '... skript ...'` bude uvnitř skriptu nastavena proměnná `x` na hodnotu 10. Funguje i pro `RS`, `FS` a další interní proměnné.

getopts

Tento příkaz pro nás znamená velké zjednodušení při zpracování argumentů předaných našim shellovým skriptům, neboť nám umožní zpracovat přepínače. Volání getopts prakticky vždy vypadá následovně:

```
while getopts dh:m opt
do
    case $opt in
        d) day=1;;
        h) help=$OPTARG;;
        m) month=1;;
        *) echo "Invalid argument";;
    esac
done
shift $(($OPTIND - 1))
```

Do proměnné `$opt` je opakovaně přiřazováno jméno zadaného přepínače, v proměnné `$OPTARG` je pak vždy uložena jeho hodnota, pokud ji dostává. Za přepínače, které očekávají hodnotu, napíšeme při předávání `getopts` dvojtečku. Příkaz `shift` na konci zaručí, že z `$@` zmizí všechny přepínače (které musí být uvedeny jako první) a zbudou další argumenty, pokud skript nějaké očekává.

xargs

Příkaz `xargs` čte seznam parametrů ze standardního vstupu a předává jej svému argumentu. Typickým vstupem příkazu `xargs` je příkaz `find`. Zatímco při použití `find -exec`bychom spouštěli příkaz pro každý soubor znovu, `find . -name '*.txt' | xargs rm` spustí `rm` pouze jednou a smaže všechny soubory s příponou `.txt` znatelně rychleji.

`xargs` má několik přepínačů, které omezují počet nebo délku argumentů, pro které zavolá jeden příkaz. Viz `man 1p xargs`.

Příkaz find

`find` je velice mocný nástroj pro vyhledávání souborů a manipulaci s nimi. Pokud spustíme `find` bez parametrů (nebo s parametrem `.`), vypíše rekurzivně obsah adresářového stromu nacházejícího se pod aktuálním adresářem.

Přidáváním dalších parametrů (pozor, ačkoliv začínají pomlčkou, nejsou to přepínače – píšeme je až za seznam prohledávaných adresářů) můžeme tyto soubory filtrovat a vybírat z nich jenom některé.

```
find . -name '*.txt'    ... soubory s příponou .txt
find . -user root       ... soubory jejichž vlastníkem je root
find . -type f          ... obyčejné soubory
find . -type d          ... adresáře
find . -size +10000c    ... soubory větší než 10000 bajtů
find . -newer file      ... soubory novější než soubor file
find . -mtime -7        ... soubory změněné v posledních 7 dnech

find . -prune           ... hledá jenom zadané soubory, nerozbaluje adresáře
```

```
find . -depth            ... zanořuje se do podadresářů (defaultní chování)
```
Pokud zapíšeme více filtrů najednou, find vypisuje jenom ty soubory, které vyhovují všem. Pokud bychom chtěli vypsat ty soubory, které splňují alespoň jeden filtr, můžeme použít parametr `-o`. Příklaz který vypíše všechny obyčejné soubory s příponou `.txt` nebo `.sh`, které byly buď změněny během posledních 14 dnů nebo jsou větší než 1kB by vypadal takto:

```
find . -type f \( -name '*.txt' -o -name '*.sh' \) \( -mtime -14 -o -size +1024c \)
```
Všimněte si zpětných lomítek před závorkami upravujících prioritu. Závorky mají v shellu speciální význam a proto jej musíme potlačit zpětným lomítkem. Libovolný filtr můžeme znegovat tím, že před něj napíšeme `!`

Parametr -exec

Teď teprve přichází ta pravá magie. `find` umí soubory nejenom vyhledávat, ale také pro ně provádět nejrůznější akce – přejmenovávat, mazat, vypisovat začátky, … Za parametr `-exec`můžeme napsat jeden shellovský příkaz. Dvojice znaků `{}` se nahradí jménem zpracovávaného souboru (i s cestou) a příkaz musí být ukončen sekvencí `\;`. Opět proto, že znak `;` má v shellu speciální význam. Příklad, který vypíše první řádek ze všech nalezených souborů:

```
find . -type f -name '*.sh' -mtime -14 -exec head -n1 {} \;
```
Pokud chceme pro daný soubor provést více příkazů, máme dvě možnosti. První je použít více parametrů `-exec` napsaných za sebou. Druhou je zavolání příkazu `sh -c 'příkazy'`, což je jeden příkaz, který spustí shell a zavolá v něm všechny zadané příkazy. Z hlediska findu se ale tváří jako jeden příkaz a proto ho můžeme předat parametru `-exec`

Užitečné nePOSIXové parametry

```
find . -maxdepth n    ... soubory v hloubce nejvýše n
find . -mindepth n    ... soubory v hloubce alespoň n
find . -size +10M     ... soubory větší než 10 megabajtů
find . -delete        ... smaž všechny odpovídající soubory
```

Regulární výrazy

Velmi pěkný a obsáhlý popis najdete v seriálu o regulárních výrazech od pana Satrapy. Zmínky o Perlu ignorujte, jenom by vás mátly. Tady poskytnu pouze stručný přehled nejdůležitějších bodů syntaxe jak pro základní regulární výrazy, tak pro rozšířené.

```
Basic RE      Extended RE      význam
.             .                libovolný znak
[a-fxyz]      [a-fxyz]         jeden znak z množiny
[^a-z]        [^a-z]           jeden znak mimo množinu
*             *                libovolný počet opakování (i 0x)
              +                alespoň jedno opakování
              ?                nejvýše jedno opakování
\{n,\}        {n,}             alespoň n opakování
\{n,m\}       {n,m}            alespoň n a nejvýš m opakování
\(...\)       (...)            skupina znaků
\2            \2               obsah 2. zapamatované skupiny
              ab|cd|ef         jedna z alternativ
^             ^                začátek řetězce
$             $                konec řetězce
```

Pokud má některý ze znaků speciální význam (např. `.`, `[`, …) a my chceme hledat jeho normální hodnotu, předřadíme mu zpětné lomítko.

Nástroje

Příkaz `grep` filtruje řádky odpovídající základnímu regulárnímu výrazu. `grep -E`, nebo také `egrep` filtruje řádky odpovídající rozšířenému regulárnímu výrazu.

Textový editor `sed` filtruje standardně řádky podle základních regulárních výrazů. Rozšířené regulární výrazy můžeme zapnout přepínačem `-r`.

Sed

Příkazy sedu načítáme z řádky nebo ze souboru:

```
sed 'příkazy' soubor1 soubor2 ...
sed -f 'soubor_s_příkazy' soubor1 soubor2 ...
sed -e 'příkazy' -f 'soubor_s_příkazy' soubor1 soubor2 ...
```

Důležitý přepínač je `-n` – nevypisovat defaultně na výstup. Další je `-r` zapínající extended regulární výrazy. Příkazy od sebe můžeme oddělovat středníky nebo znaky nového řádku. Syntaxe příkazů sedu je vždy následující:

```
[adresa1[,adresa2]]příkaz[parametry]
```

Pokud není uvedena adresa, provede se příkaz pro každý řádek. Pokud je uvedena pouze `adresa1`, provede se příkaz pro řádek odpovídající adrese. Pokud je uvedena i `adresa2`, provede se příkaz pro všechny řádky mezi `adresou1` a `adresou2` (včetně). Adresa může být číslo řádku, `$` pro poslední řádek nebo `/regexp/`.

Sed pracuje se dvěma prostory – pattern space (PS), do kterého se načítá každá nová řádka a hold space (HS), s jehož obsahem manipulujeme sami. Nic dalšího nemáme, žádné proměnné apod. Pouze příkazy pro manipulaci s obsahem pattern space a hold space. Seznam příkazů naleznete v manuálové stránce sedu. Nejpoužívanější jsou:

```
n ... vypsání současného PS (není-li -n) a načtení
```

```
      dalšího řádku do PS
N ... připojení dalšího řádku do PS
d ... smazání PS, začne od začátku
D ... smazání prvního řádku PS, začne od začátku,
      ale nenačítá vstup, je-li PS neprázdný
p ... vypíše PS na standardní výstup
P ... vypíše první řádek PS

h ... kopíruje PS do HS
H ... připoujuje PS do HS
g ... kopíruje HS do PS
G ... připojuje HS do PS
x ... prohazuje obsah PS a HS

:l ... značka l
bl ... skočí na značku l
tl ... skočí na značku l, pokud došlo od posledního
       načtení řádku k úspěšnému nahrazení s///
Tl ... skočí na značku l, pokud nedošlo od posledního
       načtení řádku k úspěšnému nahrazení s///
s/// ... nahrazení (viz minulá hodina)
y/// ... transliterace (jako tr, bez rozsahů)

q ... ukončí zpracování vstupu a vypíše PS (není-li -n)
```

Příklady

Vypište pouze liché řádky souboru.

```
sed -n 'p;n'
```

Vypište pouze tělo e-mailu, bez hlaviček.

```
sed '1,/^$/d'
```

Vypište pouze hlavičky `From:`, `To:` a `Subject:` a poté tělo e-mailu, bez ostatních hlaviček.

```
sed -rn '/^(From:|To:|Subject:)/{p;d};1,/^$/!p'
```

Rozepište věty na vstupu tak, aby na každém řádku byla jedna věta. Věta se pozná tak, že začíná velkým písmenem a končí tečkou. Předpokládejte, že se nikde nevyskytují vícenásobné mezery, ani žádné tabulátory. Jednodušší varianta navíc předpokládá, že žádná věta není obsažena na dvou řádcích, těžší již nikoliv. Zkuste napsat alespoň nějaké řešení a potom nejelegantnější.

Řešení lehčí varianty:

```
sed -r 's#([.!?]) *#\1\n#g'
```

Řešení těžší varianty:

```
sed -rn 'H;${x;s#\n# #g;s#([.!?]) *#\1\n#g;p}'
```

Napište skript, který setřídí posloupnost 0 a 1 zadanou na jednom řádku vstupu.

```
sed ':l;s/10/01/g;tl'
```

Napište skript, který z textu vymaže slova délky nanejvýš 3. Zkuste napsat alespoň nějaké řešení a potom nejelegantnější.

```
sed -r 's/\b[A-Za-z]{1,3}\b//g'
```

Paralelismus v shellu

Jak jsme se dozvěděli v kapitolce o práci s procesy, pomocí & můžeme spouštět programy na pozadí. Toho můžeme vhodně využít k psaní paralelních programů. V shellovém neinteraktivním skriptu nelze použít příkazy fg a bg, obejdeme se však bez nich.

Příklad použití

Tento skript odpočítá čísla od jedné do deseti, jedno za vteřinu a vytvoří kvůli tomu deset samostatných procesů.

```
#!/bin/sh
sleep_and_echo() {
    sleep $1
    echo $1
}

for i in $(seq 1 10); do
    sleep_and_echo $i &
    echo "Spuštěn proces č. $i s PID $!"
done

wait
```

- Proces (klidně i vlastní funkci) na pozadí spustíme pomoc `&`
- PID posledního na pozadí spuštěného příkazu obsahuje proměnná `$!`
- Na skončení procesu můžeme počkat pomocí `wait $PID` (PID může být i více – pak se čeká na dokončení všech procesů).
- Příkaz `wait` bez parametrů počká na ukončení všech procesů běžících na pozadí.

Vyčkávací smyčka

Pokud bychom chtěli zpracovat velké množství záznamů, ale vždy maximálně 10 najednou, můžeme použít vyčkávací smyčku. Pomocí příkazu `jobs` zjistíme kolik nám aktuálně běží procesů na pozadí a další proces spustíme až tehdy, když tento počet klesne pod 10:

```
for vec in $veci; do
    while [ $(jobs | wc -l) -ge 10 ]; do
        sleep 1
    done
    zpracuj $vec &
done
```

Licence textu: