B33POS: Počítačové systémy

Lekce 02. Shell dokončení

Petr Štěpán
stepan@fel.cvut.cz

27. září, 2025

Obsah

Spuštění programu

Zpětný apostrof provede spuštění příkazu a výsledek uloží do proměnné

- `A=`date``
- `A=`cat a.txt``
  - uloží obsah souboru a.txt do proměnné, i kdyby obsah byl velmi velký
- stejný význam má i konstrukce `A=$(date)`
- navíc tato konstrukce umožňuje vkládat spouštěné příkazy do sebe
  - `A=$(ls $(pwd))`

Cykly

Minulou přednášku jsme si ukázaly základní verzi for cyklu:

```
for i in *.txt
do
  cp $i zaloha\$i
done
```

Seznam podle kterého děláme for cyklus můžeme i vypsat

```
for i in a.txt b.txt c.txt
do
  cp $i zaloha\$i
done
```

Cykly

Pokud jsou v seznamu i soubory s mezerou v názvu, pak při jejich použití musíme použít uvozovky:

```
for i in *.txt
do
  cp "$i" "zaloha\$i"
done
```

Seznam si můžeme předpřipravit v proměnné

```
FILES=`ls -1 *.txt`
OLD_IFS=IFS
IFS=$'\n\t'
for i in $FILES
do
  stat "$i"
done
IFS=OLD_IFS
```

Kvůli mezerám musíme změnit proměnnou IFS, která definuje, které znaky oddělují položky v seznamu.

Pole

Bash umožňuje vytvoření pole hodnot

- vytvoření prázdného pole `declare -a POLE`
- nebo `POLE=()`
- nebo vytvoříme pole přímo s prvky `POLE=(a.txt b.txt 'a b.txt')`
- případně přidáváme prvky do pole `POLE+=("$file_name")`
  - pokud proměnná file_name obsahuje jméno s mezerou je nutné použít uvozovky

Pole

Používání pole

- k poli pak lze přistupovat k jednotlivým prvkům `${POLE[3]}`
- zjistit velikost pole `${#POLE[@]}`
- nebo jako seznamu souborů pro for cyklus, případně příkaz
  - `for file in "${POLE[@]}"`
  - `tar czf output.tgz "${POLE[@]}"`
  - opět nesmíte zapomenout na uvozovky, pokud jsou názvy souborů s mezerou nebo citlivým znakem
  - k obsahu pole lze přistoupit i operátorem *, ale dojde k problémům s položkami obsahující mezeru, hvězdičku apod.
    `tar czf output.tgz "${POLE[*]}"`
    NEFUNGUJE vždy!

Pole

Proměnná IFS je důležitá i pokud chcete vytvořit pole jako výstup programu, např.

```
OLD_IFS=IFS
IFS=$'\n'
pole=( `ls` )
IFS=OLD_IFS
```

Cykly varianta C/Python

Pomocí konstrukce `(( ))` vytvoříte C variantu for cyklu:

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

Seznam si můžeme předpřipravit i programem seq (není to POSIX standard)

```
for i in `seq -f "%03g" 0 2 6`
do
  touch data_$i.bin
done
```

Vytvoří prázdné soubory data_000.bin, data_002.bin, data_004.bin, data_006.bin

While smyčka

Opakuj dokud platí podmínka – while cyklus

```
i=2
while [ "${#i}" -lt 4 ]
do
  i=0$i
done
echo $i
```

Rozumíte, co tento program dělá, co vytiskne?

A 2

B 3

C 2000

D 0002

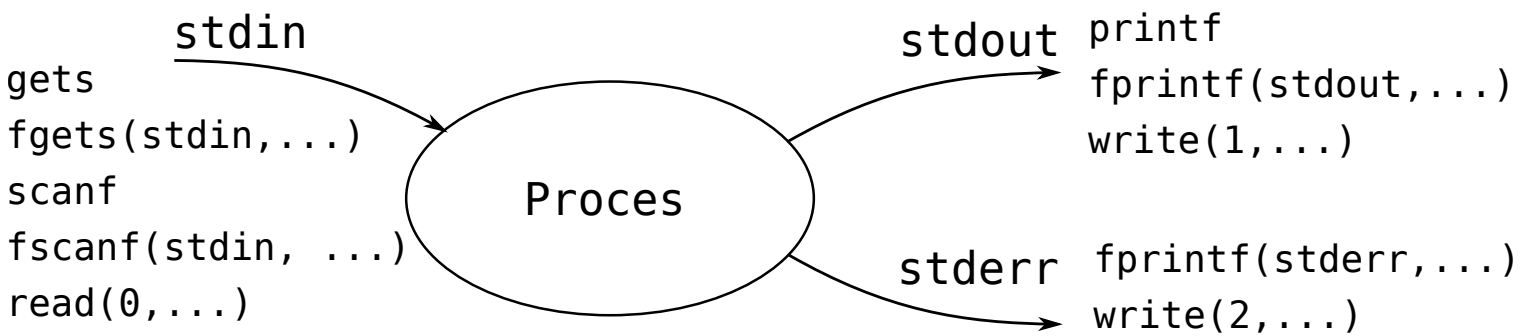
Kde Bash studovat?

- naše materiály – https://psy.pages.fel.cvut.cz/labs/02-bash/bash/
- Advance Bash Scripting Guide – https://tldp.org/LDP/abs/html/

Obsah

Vstupy a výstupy programů

- Každý program (i ten co napíšete Vy) používá 3 základní soubory
  - stdin – 0 – standardní vstup
  - stdout – 1 – standardní výstup
  - stderr – 2 – standardní chybový výstup
- Uvedené soubory může používat každý program



Výstupy programů

Co můžete použít na výstup v jazyce C:

- Vysoko úrovňové funkce
  - printf - vytiskne/zapíše uvedený řetězec do souboru 1 stdout
  - fprintf(stdout,...) - vytiskne/zapíše uvedený řetězec do souboru 1 stdout
  - fprintf(stderr,...) - vytiskne/zapíše uvedený řetězec do souboru 2 stderr
- Nízko úrovňové funkce
  - write(1,b,l) - zapíše l bajtů z pole b do souboru 1 stdout
  - write(2,b,l) - zapíše l bajtů z pole b do souboru 2 stderr

Vstupy programů

Co můžete použít na vstup v jazyce C:

- Vysoko úrovňové funkce
  - scanf - čte data zadaného formátu ze souboru 0 – stdin
  - fscanf(stdin,...) - čte data zadaného formátu ze zadaného vstupu, zde stdin
  - gets - přečte jeden řádek ze vstupu - problém nekontroluje velikost bufferu, kam zapisuje
  - fgets - přečte jeden řádek ze vstupu, maximálně však zadané množství bajtů
- Nízko úrovňové funkce
  - read(0,b,l) - přečte l bajtů ze souboru 0 - stdin do pole b
    - nekontroluje zda řádka končí a případně čeká na l bajtů neomezeně

Přesměrování v bash

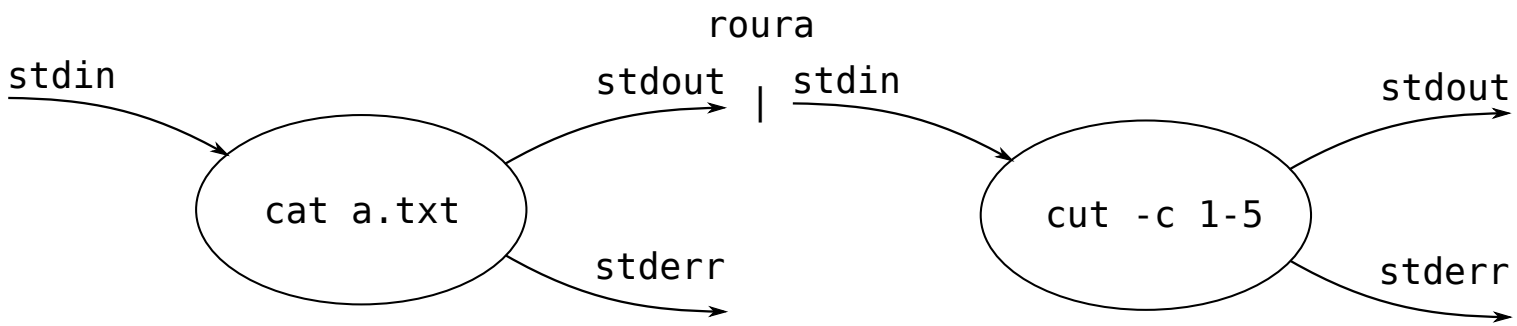
- cat
  - vytištění obsahu i více souborů na standardní výstup
  - pokud nezadáte vstupní soubor(y), pak program čte ze standardního vstupu a kopíruje ho na standardní výstup.
- výstup můžete přesměrovat do souboru konstrukcí:
  - `cat a.txt b.txt >uloz.txt`, obsah souborů a.txt a b.txt se uloží do souboru uloz.txt
  - `cat a.txt b.txt >>uloz.txt`, obsah souborů a.txt a b.txt se připojí na konec souboru uloz.txt
  - v obou případech, pokud soubor uloz.txt neexistuje, tak se vytvoří (pokud na to máte právo)
  - `cat a.txt b.txt >>uloz.txt 2 >err.txt`, obsah souborů a.txt a b.txt se připojí na konec souboru uloz.txt a všechny chybové hlášky (tištěné do souboru 2 – stderr) se uloží do souboru err.txt
  - `cat a.txt b.txt >>uloz.txt 2 >/dev/null`, soubor – zařízení /dev/null slouží k odstranění obsahu, který je do něj vytisknut, tištěný text se prostě smaže

Přesměrování v bash

- vstup můžete přesměrovat na čtení ze souboru konstrukcí:
  - `cat <cti.txt >>uloz.txt`, obsah souboru cti.txt připoj na konec souboru uloz.txt
  - `cat <cti.txt >uloz.txt 2 >err.txt`, konstrukce přesměrování lze použít pro vstup i oba výstupy
- přesměrování vstupu můžete využít pro vytvoření dočasných nepojmenovaných souborů v rámci tzv. process substitution
  - `cat a.txt <(echo ahoj) b.txt`, na standardní výstup se uloží obsah souboru a.txt, pak výstup programu echo a obsah programu b.txt
  - `diff <(ls adr1) <(ls adr2)`, porovná programem diff obsahy dvou adresářů adr1 a adr2
- když v bashi chceme tisknout chybové hlášky, potřebujeme přesměrovat standardní výstup na standardní chybový výstup konstrukcí:
  - `echo Chyba: Nelze se ucit >&2`
    text "Chyba: Nelze se ucit" se vytiskne na chybový výstup

Roura

- Roura znak | je virtuální soubor, do kterého jeden proces zapisuje a druhý z něj čte.
- Jedná se vlastně o buffer, kam lze data uložit a současně i číst a tím uvolňovat
  - příklad `cat a.txt | cut -c 1-5` vytiskne na standardní výstup prvních 5 znaků z každé řádky
  - program cut umí vyříznout znaky nebo slova s danou pozicí na řádce
  - programy o sobě nevědí, každý používá jen svůj vstup (fgets), výstup (printf)
  - klidně se může jednat i o binární data, obrázky, nebo i video stream



Roura

- Rourami můžete propojit i více programů:
  - `cat a.txt | grep a | cut -c 1-5 | sort`
    vytiskne na standardní výstup prvních 5 znaků z každé řádky souboru a.txt, která v sobě obsahuje písmeno a (klidně i mimo prvních 5 znaků) a výstupy seřadí abecedně
  - program grep propustí pouze řádky, které obsahují zadaný řetězec
  - program cut umí vyříznout znaky nebo slova s danou pozicí na řádce
  - program sort seřadí všechny řádky podle abecedy

Obsah

Parametry programu

- Parametry programu jsou vlastně pole bez jména, kdy pro přístup k prvkům nepoužíváme hranaté závorky
  - `$1` – první argument programu – `$9` devátý argument programu
  - `${10}` – pro 10-tý a další argumenty musíme použít složené závorky
  - `$#` – počet zadaných argumentů – odpovídá velikosti pole
  - `$@` – seznam všech argumentů, obdoba `${POLE[@]}`
  - `$*` – seznam všech argumentů, POZOR problém s mezerami, stejně jako při `${POLE[*]}`
  - `shift` – odstraní argument `$1` a posune všechny ostatní argumenty o jedno místo dopředu

Zpracování parametrů

```
while getopts ":abchn:s:" opt; do
    case $opt in
    a|b|c) echo "Byl aktivovan prepinac -${opt}"   ;;
    n) echo "Prepinac -n ma hodnotu ${OPTARG}"     ;;
    s) echo "Prepinac -s ma hodnotu ${OPTARG}"     ;;
    h) echo "Pouziti skriptu:"
    echo "$(basename "$0") [-a] [-b] [-c] [-h] [-n <argument>] [-s <argument>]" ;;
    ?) echo "Byl zadan neplatny prepinac -${OPTARG}" ;;
    esac
done
shift $((OPTIND - 1))
while [ $# -gt 0 ]; do
    echo "Ostatni parametry skriptu: $1"
    shift
done
```

Vestavěná funkce uvnitř bash `getopts` postupně zpracovává argumenty programu a podle nastaveného schématu `":abchn:s:"` se snaží rozpoznat přepínač a,b,c,h bez parametrů a přepínače n,s s parametrem (pozná se to podle znaku :).

Bohužel getopts neumí dlouhé přepínače (např. −−sort), k tomu je možné použít externí program getopt, který je ale složitější.

Zpracování parametrů

```
case $opt in
a|b|c) echo "Byl aktivovan prepinac -${opt}"   ;;
n) echo "Prepinac -n ma hodnotu ${OPTARG}"     ;;
?) echo "Byl zadan neplatny prepinac -${OPTARG}" ;;
esac
```

Konstrukce case odpovídá konstrukci switch–case v jazyce C, kdy je možno uvádět hodnoty proměnné `$opt` jako řetězce oddělené znakem `|` a ukončené znakem `)`, pro které se provedou zadané příkazy.
Znaky `;;` odpovídají konstrukci break v jazyce C, tedy ukončení bloku příkazů pro danou hodnotu.
Hodnota `?` je použita ve smyslu globbingu, tedy libovolná jiný jednopísmenný přepínač.

```
shift $((OPTIND - 1))
```

Příkaz shift umí odstranit i více vstupních parametrů programu, v našem případě je hodnota `$((OPTIND - 1))`, tedy o 1 menší než je nastavené v proměnné `OPTIND`, kterou využívá funkce `getopts`.

Zpracování parametrů

```
while [ $# -gt 0 ]; do
    echo "Ostatni parametry skriptu: $1"
    shift
done
```

Tento while cyklus na konec projde všechny zbývající parametry programu a v našem případě je pouze vytiskne.

Kvíz

Vytvoříme si soubor kviz.sh, který bude obsahovat tento kód:

```
echo Pocet argumentu je $#
#co udela tato radka
cat $0
```

Co bude vypsáno, pokud spustíme skript bez argumentů, pouze `bash kviz.sh`

A Vypíše počet argumentu je 0 a pak nic

B Vypíše počet argumentu je 0 a pak chybu, protože argument nebyl zadán

C Vypíše počet argumentu je 0 a pak svůj program

D Vypíše počet argumentu je 1 a pak nic

Obsah

Vytvoření funkce

Vytvořit funkci lze dvěma konstrukcemi

```
function jmeno
{
  prikzy
}
```

nebo

```
jmeno()
{
  prikzy
}
```

Obě varianty vytvoří funkci `jmeno`. Nezadává se, jaké argumenty funkce očekává, protože argumenty pro funkci se předávají stejným způsobem, jako argumenty programu.

Vytvoření funkce

```
function vypis
{
  A=$A:$1
  echo `ls $1`
}

A=''
while [ $# -gt 0 ]; do
    vypis $1
    shift
done
echo $A
```

Všimněte si, že proměnná $1 má uvnitř funkce a mimo jiné hodnoty. Proměnná A je ale společná – globální jak pro základní skript, tak pro volání funkce.

Funkce – lokální proměnné

```
function vypis
{
  local A
  A=( `ls $1` )
  echo "${A[@]}"
}
A=1
while [ $# -gt 0 ]; do
    vypis $1
    shift
done
echo $A
```

Použiti klíčového slova `local` umožní vytvořit proměnnou, která je nastavena pouze uvnitř funkce.
Je to důležité, pokud vytváříte funkci, tak abyste omylem nezměnili hodnotu proměnné ze skriptu, který Vaši funkci zavolá.

Funkce – návratová hodnota

```
function vypis
{
  local A
  A=`ls $1`
  ok=$?
  echo "$A"
  return $ok
}
while [ $# -gt 0 ]; do
    vypis $1
    echo Navratova hodnota je $?
    shift
done
```

Návratová hodnota může být předána pomocí klíčového slova `return`. Je to ale stejná hodnota, jako návratová funkce, omezená na hodnoty jednoho bajtu.

Funkce – návratová hodnota

```
function vypis
{
  local A
  A=( `ls $1` )
  ok=$PIPESTATUS
  echo "${#A[@]}"
  return $ok
}
while [ $# -gt 0 ]; do
    A=`vypis $1`
    echo Predana hodnota je $A vysledek je $?
    shift
done
```

Složitější hodnoty můžeme vracet na standardním výstupu, v tomto případě se výsledek uloží do proměnné A.
Chybový výstup se do proměnné A neuloží a pokud by nám vadil, můžeme ho přesměrovat do zařízení null.

```
A=`vypis $1 2>/dev/null`
```

Funkce – usnadnění práce

Vytvoříme si soubor mcd.sh, který bude obsahovat definici funkce mcd

```
function mcd
{
  if [ $# -ge 1 ]
  then
    mkdir -p $1
  fi
  cd $1
}
```

Příkazem `source mcd.sh` zavedeme definici funkce do aktuálního bashe. Nyní můžeme použít nový příkaz vytvoř adresář a přepni se do něj:

```
mcd adresar
```

Obsah

Program find

Příkazem `find` můžeme hledat soubory v zadaném adresáři a jeho podadresářích:

- základní použití `find [přepínače] počáteční_adresář [výrazy]`
  - program find začne prohledávat zadaný počáteční adresář a hledá v něm soubory, které splňují zadané výrazy
- Výrazy se týkají:
  - jména (souboru i adresáře): -name -path
  - velikosti: -size
  - času: -atime -ctime -mtime -amin -cmin -mmin
  - času porovnáním: -anewer -cnewer -newer
  - typu a vlastníka: -type -user -group -perm
  - a mnoho dalších (viz man stránky)

Příklady find

- `find / -name a.txt`
  - najdi kdekoliv se vyskytující soubor a.txt, začne v kořenovém adresáři a projde všechny adresáře
- `find ~ -name "*.txt"`
  - najdi v domácím adresáři a podadresářích soubory s koncovkou .txt (uvozovky jsou nutné, aby řetězec nebyl nahrazen jmény konkrétních souborů a pracovním adresáři)
- `find ~ -path "*/pr*/*sh -type f`
  - v rámci domácích adresářů najdi soubory, které končí na sh a jsou v podadresáři začínající na pr (type f jako file, nebo type d jako directory)

Příklady find

- `find / -name "*.txt"-size +1M`
  - najdi kdekoliv se vyskytující soubor s koncovkou .txt větší než 1M (+ znamená větší než, - menší než, bez znaménka přesná hodnota)
- `find ~ -name "*.sh"-mtime -1`
  - najdi v domácích adresářích všechny skripty (soubory končící .sh), které byly dnes modifikovány (čas modifikace je menší než 1*24 hodin tedy 1 den)
- `find ~ -path "*/pr*/*sh"-mmin -60`
  - v rámci domácích adresářů najdi soubory, které končí na sh a jsou v podadresáři začínající na pr a byly změněny před méně než 60 minutami

Příklady find

- `find ~ -path "*/pr*/*sh" -mmin -60 -exec ls -l {} \;`
  - u nalezených souborů vytiskni o nich podrobnosti (symboly {} se nahradí nalezeným jménem)
- `find ~ -path "*/pr*/*sh" -mmin -60 -exec rm {} \;`
  - smaž všechny soubory, které jsi našel (lze použít i příkaz -delete)

Obsah

shellcheck

Program shellcheck zkontroluje, zda Váš skript odpovídá striktním pravidlům pro přenositelnost

- spuštění `shellcheck muj_skript.sh`
- pokud je program v pořádku, na standardním výstupu se nic nezobrazí a $? je roven 0
- jinak se doporučení zobrazí na standardním výstupu, např:
- ```
  In skript.sh line 1:
  echo Pocet argumentu je $#
  ^-- SC2148: Tips depend on target shell and yours is unknown.
  Add a shebang.
  ```
  První řádka neobsahuje pro jaký shell jste skript psali, přidejte shebang (#!)
  ```

shellcheck

- ```
  In skript.sh line 4:
  cat $0
      ^-- SC2086: Double quote to prevent globbing and word splitting.
  Did you mean:
  cat "$0"
  ```

  Nejsou použity uvozovky což může vést k nahrazení jménem souboru (globbing) nebo rozdělení mezerou na více argumentů.
- Na konci je pak odkaz na webové stránky, kde jsou doporučená pravidla detailně rozebrána

  ```
  For more information:
  https://www.shellcheck.net/wiki/SC2148 -- Tips depend on target
  shell and y...
  https://www.shellcheck.net/wiki/SC2086 -- Double quote to prevent
  globbing ...
  ```
  ```

Bod aktivity

Co dělá tento skript:

```
#!/usr/bin/env bash

for file_name in *.h
do
   f=${file_name//./_}
   head -n4 "$file_name" | grep '#ifdef' >/dev/null 2>/dev/null
   if [ $? -ne 0 ]
   then
     cat <(echo -e "#ifdef __${f}__ \n#define __${f}__\n") \
                 "$file_name" <(echo -e "\n#endif\n") >tmp.txt
     mv tmp.txt "$file_name"
  fi
done
```

A všem souborům *.h připojí na začátek #ifdef ... #define ... a na konec #endif

B všem souborům *.h odstraní ze začátku #ifdef ... #define ...

C všem souborům *.h, které v prvních 4 řádkách neobsahují #ifdef připojí na začátek #ifdef ... #define ... a na konec #endif

D Nedělá nic.