

Operační systémy

IOS 2024/2025

Tomáš Vojnar

`vojnar@fit.vutbr.cz`



Programování v UNIXu: přehled

Nástroje programátora

❖ Prostředí pro programování zahrnuje:

- API OS a různých aplikačních knihoven,
- CLI a GUI,
- editory,
- překladače a sestavovače/interprety,
- ladící nástroje,
- nástroje pro automatizaci překladu,
- ...
- dokumentace.

❖ CLI a GUI v UNIXu:

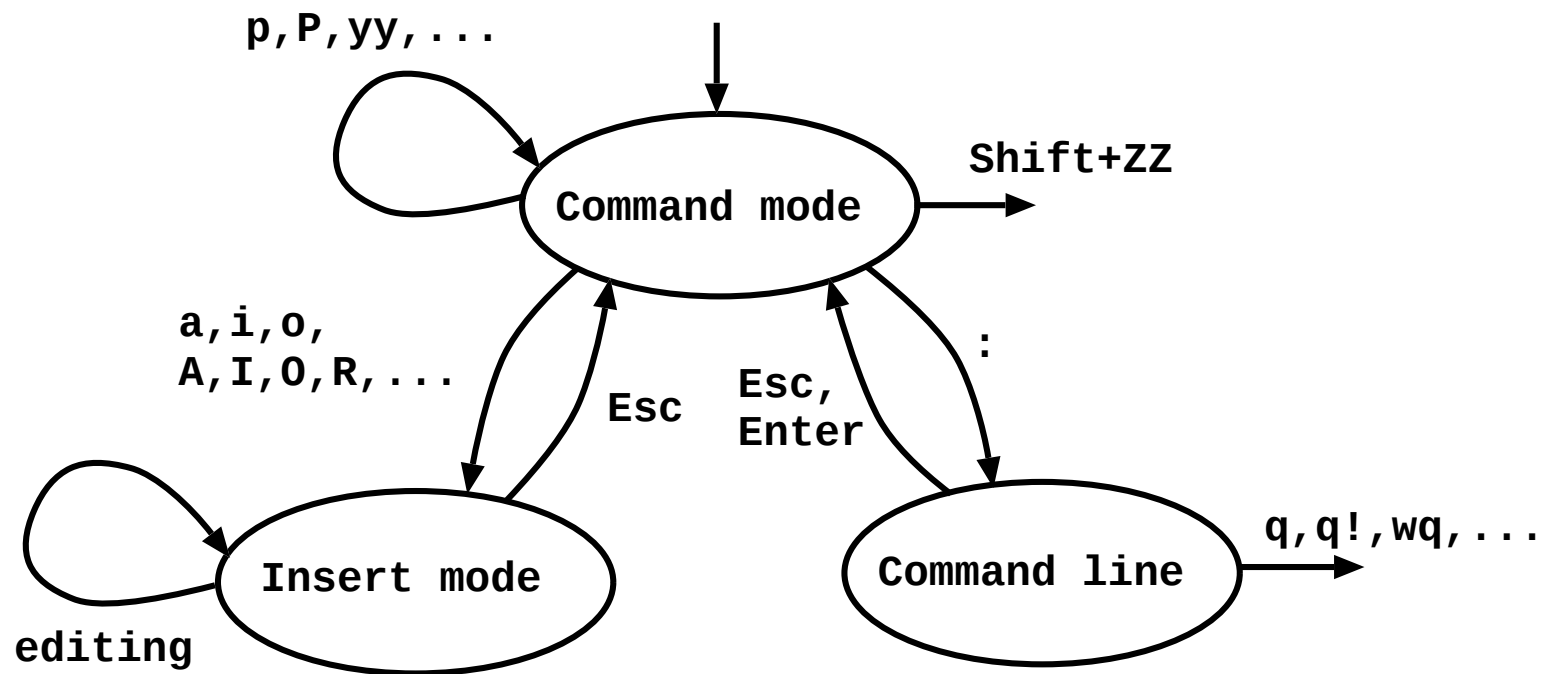
- CLI: [shell](#) (sh, ksh, csh, bash, dash, ...)
- GUI: [X-Window](#), [Wayland](#), [Quartz Compositor](#), ...

Editory, vim

❖ Textové editory běžné v UNIXu:

- v terminálu: vi, vim, emacs, pico, nano, ...
- grafické: gvim, xemacs, gedit, nedit, ...

❖ Tři režimy vi, vim:



Užitečné příkazy ve vim

❖ **Mazání** – smaže a vloží do registru:

- znak: x/X
- řádek: dd
- konec/začátek řádku: dEnd / dHome
- konec/začátek slova: dw / db
- konec/začátek odstavce: d} / d{
- do znaku: dt znak

❖ **Změna**: r, R a cc, cw, cb, c+End/Home, c}, ct+ukončující znak, ...

❖ **Vložení textu do registru**: yy, yw, y}, yt+ukončující znak, ...

❖ **Vložení registru do textu**: p/P

❖ **Bloky**: (v+šipky)/(Shift-v+šipky)/(Ctrl-v+šipky)+y /d, ...

- ❖ Vícenásobná aplikace: číslo+příkaz (např. 5dd)
- ❖ undo/redo: u /Ctrl-R
- ❖ Opakování posledního příkazu: . (tečka)
- ❖ Vyhledání: / regulární výraz
- ❖ Aplikace akce po vzorek: d/ regulární výraz

Regulární výrazy

❖ **Regulární výrazy** jsou nástrojem pro konečný popis případně nekonečné množiny řetězců. Jejich uplatnění je velmi široké – nejde jen o vyhledávání ve vim!

❖ **Základní regulární výrazy** (existují také rozšířené RV – viz dále):

znak	význam
obyčejný znak	daný znak
.	libovolný znak
*	0 – n výskytů předchozího znaku
[<i>množina</i>]	znak z množiny, např: [0-9A-Fa-f]
[^ <i>množina</i>]	znak z doplňku množiny
\	ruší řídicí význam následujícího znaku
^	začátek řádku
\$	konec řádku
[[: <i>k</i> :]]	znak z dané kategorie <i>k</i> podle locale

❖ **Příklad:** "`^ *\[0-9]\[0-9]* *\$`"

Příkazová řádka ve vim

- ❖ Uložení do souboru: `w`, případně `w!`
- ❖ Vyhledání a změna: řádky `s/ regulární výraz / regulární výraz (/g)`
- ❖ Adresace řádků: číslo řádku, interval (`x,y`), aktuální řádek (`.`), poslední řádek (`$`), všechny řádky (`%`), nejbližší další řádek obsahující řetězec (`/řetězec/`), nejbližší předchozí řádek obsahující řetězec (`?řetězec?`)
- ❖ Příklad – vydělení čísel 10:

```
:%s/\([0-9]*\) \([0-9]\)/\1.\2/
```

Základní dokumentace v UNIXu

❖ `man`, `info`, `/usr/share/doc`, `/usr/local/share/doc`, `HOWTO`, `FAQ`, ..., `README`, `INSTALL`, ...

❖ `man` je rozdělen do **sekcí** (`man n name`):

1. Executable programs or shell commands
2. System calls (functions provided by the kernel)
3. Library calls (functions within program libraries)
4. Special files (usually found in `/dev`)
5. File formats and conventions, e.g., `/etc/passwd`
6. Games
7. Miscellaneous (including macro packages and conventions)
8. System administration commands (usually only for root)
9. Kernel routines (Non standard)

❖ `apropos name` – kde všude se v `man` mluví o `name`.

Bourne shell

❖ Skriptování:

- **Interpret**: program, který provádí činnost programu, který je jeho vstupem.
- **Skript**: textový soubor s programem pro interpret.
- **Nevýhody**: pomalejší, je třeba interpret.
- **Výhody**: nemusí se překládat (okamžitě spustitelné), čitelný obsah programu.

❖ Spuštění skriptu v UNIXu:

```
sh skript.sh          # explicitní volání interpretu
```

```
chmod +x skript.sh    # nastaví příznak spustitelnosti
```

```
./skript.sh           # spuštění
```

```
. ./skript.sh         # spuštění v aktuálním shellu
```

❖ “Magic number” = číslo uvedené na začátku souboru a charakterizující jeho obsah:

- U spustitelných souborů jádro zjistí na základě magic number, jak soubor spustit: tj. u **binárních programů** jejich formát určující způsob zevedení do paměti, u **interpretovaných programů** pak, který interpret použít (první řádek: `#!/cesta/interpret`).
- Výhoda: možnost psát programy v libovolném skriptovacím jazyku.

❖ Příklady:

`#!/bin/sh` - skript pro Bourne shell

`#!/bin/ksh` - skript pro Korn shell

`#!/bin/csh` - skript pro C shell

`#!/usr/bin/perl` - skript v Perlu

`#!/usr/bin/python` - skript v Pythonu

`\177ELF` - binární program - formát ELF

Speciální znaky (metaznaky)

- ❖ Jsou interpretovány shellem, znamenají provedení nějaké speciální operace.
- ❖ Jejich speciální význam lze zrušit například znakem \ těsně před speciálním znakem.
- ❖ Poznámky:

znak	význam a příklad použití
#	poznámka do konce řádku echo "text" # poznámka

- ❖ Práce s proměnnými:

\$	zpřístupnění hodnoty proměnné echo \$TERM
----	--

❖ Přesměrování vstupu a výstupu:

znak	význam a příklad použití
>	přesměrování výstupu, přepíše soubor <pre>echo "text" >soubor # přesměrování stdout příkaz 2>soubor # přesměrování stderr</pre> příkaz [n]>soubor # n je číslo, implicitně 1 Čísla zde slouží jako popisovače otevřených souborů (file handle). Standardně používané a otevírané popisovače: • <code>stdin=0</code> standardní vstup (klávesnice) • <code>stdout=1</code> std. výstup (obrazovka) • <code>stderr=2</code> std. chybový výstup (obrazovka)

znak	význam a příklad použití
>&	duplikace popisovače pro výstup <pre data-bbox="353 427 1518 772"> echo "text" >&2 # stdout do stderr příkaz >soubor 2>&1 # přesm. stderr i stdout příkaz 2>&1 >soubor # stdout do souboru, # stderr na obrazovku m>&n # m a n jsou čísla </pre>
>>	přesměrování výstupu, přidává do souboru <pre data-bbox="353 986 900 1091"> echo "text" >> soubor příkaz 2>>log-soubor </pre>

znak	význam a příklad použití
<	přesměrování vstupu příkaz < soubor
<<token	přesměrování vstupu, čte ze skriptu až po <i>token</i>, který musí být samostatně na řádku – tzv. “here document” (expanze proměnných a vnořených příkazů) <pre>cat >soubor <<__END__ jakýkoli text, i \$PROMENNA kromě ukončovacího řádku __END__</pre> Varianty: <<\token quoting jako ', dále žádná expanze <<-token možno odsadit tabelátory

❖ Zástupné znaky ve jménech souborů:

znak	význam a příklad použití
*	zastupuje libovolnou sekvenci libovolných znaků mimo / a . na začátku jména (což lze ale ovlivnit proměnnými shellu), shell vyhledá všechna odpovídající jména souborů a nahradí jimi příslušný vzor – pokud žádné nenajde, expanzi neprovede (lze ovlivnit opět proměnnými shellu): <pre>ls *.c ls *archiv*gz ls .*/*.conf # soubory s příponou conf ve skrytých adresářích</pre> Poznámka: Programy nemusí zpracovávat tyto expanzní znaky samy. Poznámka: Pozor na limit délky příkazového řádku!
?	zastupuje 1 libovolný znak jména souboru (výjimky viz výše) <pre>ls x???.txt ls soubor-?.txt</pre>
[množina]	zastupuje jeden znak ze zadané množiny (výjimky viz výše) <pre>ls [A-Z]* ls soubor-[1-9].txt</pre>

❖ Skládání příkazů:

znak	význam a příklad použití
	přesměrování stdout procesu na stdin dalšího procesu, slouží pro vytváření <i>kolon</i> procesů-filtrů: <pre>ls more cat /etc/passwd awk -F: '{print \$1}' sort příkaz tee soubor příkaz</pre>
'příkaz'	je zaměněno za standardní výstup příkazu (command substitution) <pre>ls -l 'which sh' DATUM='date +%Y-%m-%d' # ISO formát echo Přihlášeno 'who wc -l' uživatelů</pre>

znak	význam a příklad použití
;	sekvence příkazů na jednom řádku <pre>ls ; echo ; ls /</pre>
	provede následující příkaz, pokud předchozí neuspěl (exitcode<>0) <pre>cc program.c echo Chyba překladu</pre>
&&	provede následující příkaz, pokud předchozí uspěl (exitcode=0) <pre>cc program.c && ./a.out</pre>

❖ Spouštění příkazů:

znak	význam a příklad použití
(příkazy)	spustí <i>subshell</i>, který provede příkazy <pre>(echo "Text: " cat soubor echo "konec") > soubor2</pre>
&	spustí příkaz <i>na pozadí</i> (pozor na výstupy programu) <pre>program &</pre>

❖ Rušení významu speciálních znaků (quoting):

- Znak `\` ruší význam jednoho následujícího speciálního znaku (i znaku "nový řádek").

```
echo \* text \*  \\  
echo "fhgksagdsahfgsjdagfjkdsaagjdsagjhfdsa\  
jhdsajfhdsaflljkshdafkjhadsk"  
echo 5 \> 2 text \$TERM
```

- Uvozovky `"` ruší význam speciálních znaků kromě: `$`proměnná, `'`příkaz`'` a `\`.

```
echo 3 * 4 = 12 # chyba, pokud  
                  # jsou v adresáři soubory  
echo "3 * 4 = 12"  
echo "Dnešní datum: 'date'"  
echo "PATH=$PATH"  
echo "\ <\\"test\> *** 'date' *** $PATH *** "
```

- Apostrofy `'` ruší speciální význam všech znaků v řetězci.

```
echo '$<>*' jakýkoli text kromě apostrofu '  
echo 'toto ->'\''<- je apostrof'  
echo '*\** \' $PATH 'ls' <> \'
```

Postup při hledání příkazů

❖ Po zadání příkazu postupuje shell následovně:

1. Test, zda se jedná o funkci nebo zabudovaný příkaz shellu (např. break), a případné provedení této funkce/příkazu.
2. Pokud se jedná o příkaz zadaný i s cestou (např. /bin/sh), pokus provést program s příslušným jménem v příslušném adresáři.
3. Postupné prohlížení adresářů v PATH.
4. Pokud program nenalezne nebo není spustitelný, hlásí chybu.

❖ **Poznámka:** Vlastní příkazy do \$HOME/bin a přidat do PATH.

Vestavěné příkazy

- ❖ Které příkazy jsou vestavěné závisí na použitém interpretu.
- ❖ Příklad: `cd`, `wait`, `break`, ...
- ❖ **Výhoda**: rychlost provedení.
- ❖ Ostatní příkazy jsou běžné spustitelné soubory.

Příkaz eval

❖ eval příkaz:

- Jednotlivé argumenty jsou načteny (a je proveden jejich rozvoj), výsledek je konkatenován, znovu načten (a rozvinut) a proveden jako nový příkaz.
- Možnost **za běhu sestavovat příkazy** (tj. program) na základě aktuálně čteného obsahu souboru, vstupu od uživatele apod.

❖ Příklady:

```
echo "text > soubor"  
eval echo "text > soubor"  
eval 'echo x=1' ; echo "x=$x"
```

Ukončení skriptu

❖ Ukončení skriptu: `exit` [číslo]

- vrací `exit-code` číslo nebo exit-code předchozího příkazu,
- vrácenou hodnotu lze zpřístupnit pomocí `$?`,
- možné hodnoty:
 - 0 – O.K.
 - `<>0` – chyba

❖ Spuštění nového kódu: `exec` příkaz:

- nahradí kód shellu prováděcího `exec` kódem daného příkazu,
- spuštění zadaného programu je rychlé – nevytváří se nový proces,
- bez parametru umožňuje přesměrování vstupu/výstupu uvnitř skriptu.

Správa procesů

ps	výpis stavu procesů
nohup	proces nekončí při odhlášení
kill	posílání signálů procesům
wait	čeká na dokončení potomka/potomků

❖ Příklady:

```
ps ax          # všechny procesy
nohup program  # pozor na vstup/výstup
kill -9 1234    # nelze odmítnout
```

Subshell

❖ Subshell se implicitně spouští v případě použití:

<code>./skript.sh</code>	spuštění skriptu (i na pozadí)
<code>(příkazy)</code>	skupina příkazů

❖ Subshell **dědí** proměnné prostředí, **nedědí** lokální proměnné (tj. ty, u kterých nebyl proveden `export`).

❖ Změny proměnných a dalších nastavení v subshellu se neprojeví v původním shellu!

❖ Provedení skriptu aktuálním interpretem:

- příkaz `.`
- např. `. skript`

❖ Posloupnost příkazů `{ příkazy }` – stejné jako `( )`, ale nespouští nový subshell.

❖ **Příklad** – možné použití { } (a současně demonstrace jedné z programovacích technik používaných v shellu):

```
# Changing to a log directory.
```

```
cd $LOG_DIR
```

```
if [ "`pwd`" != "$LOG_DIR" ] # or if [ "$PWD" != "$LOG_DIR" ]  
                           # Not in /var/log?
```

```
then
```

```
    echo "Cannot change to $LOG_DIR."
```

```
    exit $ERROR_CD
```

```
fi # Doublecheck if in right directory, before messing with log file.
```

```
# However, a far more efficient solution is:
```

```
cd $LOG_DIR || {
```

```
    echo "Cannot change to $LOG_DIR." >&2
```

```
    exit $ERROR_CD;
```

```
}
```

Proměnné

❖ Rozlišujeme proměnné:

- **lokální** (nedědí se do subshellu)

```
PROM=hodnota
```

```
PROM2="hodnota s mezerami"
```

- **proměnné prostředí** (dědí se do subshellu)

```
PROM3=hodnota
```

```
export PROM3    # musíme exportovat do prostředí
```

❖ Příkaz **export**:

- `export seznam_proměnných`
- exportuje proměnné do prostředí, které dědí subshell,
- bez parametru vypisuje obsah prostředí.

❖ Přehled **standardních proměnných**:

\$HOME	jméno domovského adresáře uživatele
\$PATH	seznam adresářů pro hledání příkazů
\$MAIL	úplné jméno poštovní schránky pro e-mail
\$USER	login jméno uživatele
\$SHELL	úplné jméno interpretu příkazů
\$TERM	typ terminálu (viz termcap/terminfo)
\$IFS	obsahuje oddělovače položek na příkazové řádce – implicitně mezera, tabelátor a nový řádek
\$PS1	výzva interpretu na příkazové řádce – implicitně znak \$
\$PS2	výzva na pokračovacích řádcích – implicitně znak >

❖ Další standardní proměnné:

\$\$	číslo = PID interpretu
\$0	jméno skriptu (pokud lze zjistit)
\$1..\$9	argumenty příkazového řádku (dále pak \${n} pro $n \geq 10$)
\$*/\$@	všechny argumenty příkazového řádku
"\$*"	všechny argumenty příkazového řádku jako 1 argument v ""
"\$@"	všechny argumenty příkazového řádku, individuálně v ""
\$#	počet argumentů
\$?	exit-code posledního příkazu
\$!	PID posledního příkazu na pozadí
\$-	aktuální nastavení shellu

❖ Příklady:

```
echo "skript: $0"  
echo první argument: $1  
echo všechny argumenty: $*  
echo PID=$$
```

❖ Použití proměnných:

<code>\$PROM text</code>	mezi jménem a dalším textem musí být oddělovací znak
<code>\${PROM}text</code>	není nutný další oddělovač
<code>\${PROM-word}</code>	word pokud nenastaveno
<code>\${PROM+word}</code>	word pokud nastaveno, jinak nic
<code>\${PROM=word}</code>	pokud nenastaveno, přiřadí a použije word
<code>\${PROM?word}</code>	pokud nenastaveno, tisk chybového hlášení word a konec (exit)

❖ Příkaz `env`:

- `env` nastavení_proměnných `program` [`argumenty`]
- spustí `program` s nastaveným prostředím,
- bez parametrů vypíše prostředí.

❖ Proměnné pouze pro čtení:

- `readonly seznam_proměnných`
- označí proměnné pouze pro čtení,
- `subshell` toto nastavení nedědí.

❖ Posun argumentů skriptu:

- příkaz `shift`,
- posune `$1 <- $2 <- $3 ...`

Čtení ze standardního vstupu

❖ Příkaz `read seznam_proměnných` čte řádek ze `stdin` a přiřazuje slova do proměnných, do poslední dá celý zbytek vstupního řádku.

❖ Příklady:

```
echo "x y z" | (read A B; echo "A='$A' B='$B'")
```

```
IFS=","; echo "x,y z" | (read A B; echo "A='$A' B='$B'")
```

```
IFS=":"; head -1 /etc/passwd | (read A B; echo "$A")
```

Příkazy větvení

❖ Příkaz if:

```
if seznam příkazů
then
    seznam příkazů
elif seznam příkazů
then
    seznam příkazů
else
    seznam příkazů
fi
```

Příklad použití:

```
if [ -r soubor ]; then
    cat soubor
else
    echo soubor nelze číst
fi
```

Testování podmínek

❖ Testování podmínek:

- konstrukce `test výraz` nebo `[ výraz ]`,
- výsledek je v `$?`.

výraz	význam
<code>-d file</code>	je adresář
<code>-f file</code>	je obyčejný soubor
<code>-r file</code>	je čitelný soubor
<code>-w file</code>	je zapisovatelný soubor
<code>-x file</code>	je proveditelný soubor
<code>-t fd</code>	deskriptor <code>fd</code> je spojen s terminálem
<code>-n string</code>	neprázdný řetězec
<code>string</code>	neprázdný řetězec
<code>-z string</code>	prázdný řetězec
<code>str1 = str2</code>	rovnost řetězců
<code>str1 != str2</code>	nerovnost řetězců

výraz	význam
<code>int1 -eq int2</code>	rovnost čísel
<code>int1 -ne int2</code>	nerovnost čísel
<code>int1 -gt int2</code>	<code>></code>
<code>int1 -ge int2</code>	<code>>=</code>
<code>int1 -lt int2</code>	<code><</code>
<code>int1 -le int2</code>	<code><=</code>
<code>! expr</code>	negace výrazu
<code>expr1 -a expr2</code>	and
<code>expr1 -o expr2</code>	or
<code>\(\)</code>	závorky

❖ Příkaz case:

```
case výraz in
    vzor { | vzor }* )
    seznam příkazů
    ;;
esac
```

Příklad použití:

```
echo -n "zadejte číslo: "
read reply
case $reply in
    "1")
        echo "1"
        ;;
    "2"|"4")
        echo "2 nebo 4"
        ;;
    *)
        echo "něco jiného"
        ;;
esac
```

Cykly

❖ Cyklus for:

```
for identifikátor [ in seznam slov ] # bez []: $1 ...  
do  
    seznam příkazů  
done
```

Příklad použití:

```
for i in *.txt ; do  
    echo Soubor: $i  
done
```

❖ Cyklus while:

```
while seznam příkazů # poslední exit-code se použije  
do  
    seznam příkazů  
done
```

Příklad použití:

```
while true ; do  
    date; sleep 1  
done
```

❖ Cyklus until:

```
until seznam příkazů # poslední exit-code se použije  
do  
    seznam příkazů  
done
```

❖ Ukončení/pokračování cyklu:

break, continue

❖ Příklady:

```
stop=ne
while [ "$stop" != ano ]; do
    echo -n "má skript skončit: "
    read stop
    echo $stop
    if [ "$stop" = ihned ] ; then
        echo "okamžité ukončení"
        break
    fi
done
```

Zpracování signálů

❖ Příkaz trap:

- `trap [příkaz] {signál}+`
- při výskytu signálu provede příkaz,
- pro ladění lze užít `trap příkaz DEBUG`.

❖ Příklad zpracování signálu:

```
#!/bin/sh
```

```
trap 'echo Ctrl-C; exit 1' 2 # ctrl-C = signál č.2
```

```
while true; do  
    echo "cyklíme..."  
    sleep 1  
done
```

Vyhodnocování výrazů

❖ Příkaz `expr` výraz:

- Vyhodnotí výraz, komponenty musí být odděleny mezerami (pozor na quoting!).

- Operace podle priority:

`\*` `/` `%`
`+` `-`
`=` `\>` `\>=` `\<` `\<=` `!=`
`\&`
`\|`

- Lze použít závorky: `\( \)`

❖ Příklady:

```
V='expr 2 + 3 \* 4' ; echo $V
```

```
expr 1 = 1 \& 0 != 1 ; echo $?
```

```
expr "$P1" = "$P2" # test obsahu proměnných
```

```
V='expr $V + 1'     # V++
```

❖ Řetězcové operace v expr:

`String : Regexp`

`match String Regexp`

– vrací délku prefixu řetězce, který vyhovuje `Regexp`, nebo 0

`substr String Start Length`

– získá podřetězec od zadané pozice

`index String Charlist`

– vrací pozici prvního znaku ze seznamu, který se najde

`length String`

– vrací délku řetězce

Korn shell – ksh

❖ Rozšíření Bourne shellu, starší verze ksh88 základem pro definici POSIX, jeho důležité vlastnosti jsou zabudovány rovněž v bash-i.

❖ Příkaz `alias`: `alias rm='rm -i'`.

❖ Historie příkazů: možnost vracet se k již napsaným příkazům a editovat je (bash: viz šipka nahoru a dolů a `^R`).

❖ Vylepšená aritmetika:

- příkaz `let`, např. `let "x=2*2"`,
- operace: `+` `-` `*` `/` `%` `!` `<` `>` `<=` `>=` `==` `!=` `=` `++` `--` `...`,
- vyhodnocení bez spouštění dalšího procesu,
- zkrácený zápis, bez expanze cest:

```
(( x=2 ))
(( x=2*x ))
(( x++ ))
echo $x
```

❖ Vylepšené testování:

`[[ ]]`

`(výraz)`

`výraz && výraz`

`výraz || výraz`

Dodatečné operátory `==` a `=~` pro práci s **regulárními výrazy**, zbytek stejně jako `test`.

❖ Substituce příkazů:

`'command'` `$(command)`

❖ Speciální znak “vlnovka”:

~	\$HOME	domovský adresář
~user		domovský adresář daného uživatele
~+	\$PWD	pracovní adresář
~-	\$OLDPWD	předchozí prac. adresář

❖ Primitivní menu:

```
select identifikátor [in seznam slov]
do
    seznam příkazů
done
```

– funguje jako cyklus; nutno ukončit!

❖ Pole:

```
declare -a p          # pole (deklarace nepovinná)
p[1]=a
echo ${p[1]}
p+=(b c)              # přidání prvků
echo ${p[*]}
p=([1]=er [2]=error)  # celé pole
p+=([5]=c [6]=d)      # přidání na pozici
```

```
declare -A q          # asociativní pole (deklarace povinná!)
q[abc]=xyz
q[def]=mno
echo ${q[*]}
echo ${!q[*]}         # použité klíče
```

❖ Příkaz `printf`: formátovaný výpis na standardní výstup.

❖ Zásobník pro práci s adresáři:

- `pushd` – uložení adresáře do zásobníku,
- `popd` – přechod do adresáře z vrcholu zásobníku,
- `dirs` – výpis obsahu zásobníku.

❖ Příkaz set

- bez parametrů vypíše proměnné i funkce,
- jinak nastavuje vlastnosti shellu:

parametr	akce
-n	neprovádí příkazy
-u	chyba pokud proměnná není definována
-v	opisuje čtené příkazy
-x	opisuje prováděné příkazy
--	další znaky jsou argumenty skriptu

- vhodné pro ladění skriptů.

❖ Příklady:

```
set -x -- a b c *  
for i ; do echo $i; done
```

❖ Zpracování přepínačů – getopt:

```
# Handling options a, b with a parameter, c.
```

```
while getopt :ab:c o
do
    case "$o" in
        a)      echo "Option 'a' found.>";;
        b)      echo "Option 'b' found with parameter '$OPTARG'.>";;
        c)      echo "Option 'c' found.>";;
        *)      echo "Use options a, b with a parameter, or c." >&2
                exit 1;;
    esac
done

((OPTIND--))

shift $OPTIND

echo "Remaining arguments: '$*'"
```

Omezení zdrojů

- ❖ **Restricted shell** – zabránění shellu a jeho uživateli (lze i pro skripty) provádět jisté příkazy:
 - použití `cd`, přesměrování, změna `PATH`, spouštění programů zadaných s cestou, použití `exec...`
- ❖ **ulimit** – omezení prostředků dostupných shellu a procesům z něj spuštěným:
 - počet procesů, paměť procesu, počet otevřených souborů, ...
- ❖ **quota** – omezení diskového prostoru pro uživatele.

Funkce

❖ Definice funkce:

```
function ident ()  
{  
    seznam příkazů  
}
```

❖ Volání: `ident argument1 argument2 ...`

❖ Parametry jako u skriptu: `$1 ...`

❖ Ukončení funkce s `exit-code`: `return [exit-code]`.

❖ Definice lokální proměnné: `typeset prom.`

❖ Možnost rekurze.

Správa prací – job control

- ❖ **Job** (úloha) v shellu odpovídá prováděné koloně procesů (pipeline).
- ❖ Při spuštění kolony se vypíše `[jid] pid`, kde `jid` je identifikace úlohy a `pid` identifikace posledního procesu v koloně.
- ❖ Příkaz `jobs` vypíše aktuálně prováděné úlohy.
- ❖ Úloha může být spuštěna **na popředí**, nebo pomocí `&` **na pozadí**.
- ❖ Úloha běžící na popředí může být pozastavena pomocí `^Z` a přesunuta na pozadí pomocí `bg` (a zpět pomocí `fg`).
- ❖ Explicitní identifikace úlohy v rámci `fg`, `bg`, `kill`,...: `%jid`

Interaktivní a log-in shell

❖ Shell může být spuštěn v různých režimech – pro bash máme dva významné módy, které se mohou kombinovat:

- **interaktivní bash** (parametr `-i`, typicky vstup/výstup z terminálu) a
- **log-in shell** (parametr `-l` či `-login`).

❖ **Start, běh a ukončení interpretu příkazů** závisí na režimu v němž shell běží. Např. pro interaktivní log-in bash platí:

1. úvodní sekvence: `/etc/profile` (existuje-li) a dále `~/.bash_profile`, `~/.bash_login`, nebo `~/.profile`,
2. tisk `$PS1`, zadávání příkazů,
3. `exit`, `^D`, `logout` – ukončení interpretu s provedením `~/.bash_logout`.

❖ Výběr **implicitního interpretu příkazů**:

- `/etc/passwd`
- `chsh` – change shell

Shrnutí expanzí v shellu

❖ Při provádění příkazu **shell** provádí následující expanze:

1. Zleva doprava rozvoj
 - složených závorek (např. `a{b,c,d}e` na `abe ace ade`) – není ve standardu,
 - vlnovek,
 - proměnných,
 - vložených příkazů a
 - aritmetických výrazů `$((...))`.
2. Rozčlenění na argumenty dle IFS.
3. Rozvoj jmen souborů.
4. Odstranění kvotování.

Utility UNIXu

❖ Utiliy UNIXu:

- užitečné programy (asi 150),
- součást normy SUSv3/v4,
- různé nástroje na zpracování textu atd.

❖ Přehled základních programů:

awk	jazyk pro zpracování textu, výpočty atd.
cmp	porovnání obsahu souborů po bajtech
cut	výběr sloupců textu
dd	kopie (a konverze) části souboru
bc	kalkulátor s neomezenou přesností

Pokračování na dalším slajdu...

❖ Přehled základních programů – pokračování...

<code>df</code>	volné místo na disku
<code>diff</code>	rozdíl textových souborů (viz i <code>tkdiff</code>)
<code>du</code>	zabrané místo na disku
<code>file</code>	informace o typu souboru
<code>find</code>	hledání souborů
<code>grep</code>	výběr řádků textového souboru
<code>iconv</code>	překódování znakových sad
<code>nl</code>	očíslování řádků
<code>od</code>	výpis obsahu binárního souboru
<code>patch</code>	oprava textu podle výstupu <code>diff</code>
<code>sed</code>	neinteraktivní editor textu
<code>sort</code>	řazení řádků
<code>split</code>	rozdělení souboru na menší
<code>tr</code>	záměna znaků v souboru
<code>uniq</code>	vynechání opakujících se řádků
<code>xargs</code>	zpracování argumentů (např. po <code>find</code>)

Program grep

❖ Umožňuje **výběr řádků** podle regulárního výrazu.

❖ Existují **tři varianty**:

- fgrep – rychlejší, ale neumí regulární výrazy
- grep – základní regulární výrazy
- egrep – rozšířené regulární výrazy

❖ **Příklady použití:**

```
fgrep -f seznam soubor
grep '^*[A-Z]' soubor
egrep '(Jan|Honza) +Novák' soubor
```

❖ **Rozšířené (extended) regulární výrazy:**

znak	význam
+	1 – n výskytů předchozího podvýrazu
?	0 – 1 výskyt předchozího podvýrazu
{ m }	m výskytů předchozího podvýrazu
{ m, n }	$m – n$ výskytů předchozího podvýrazu
(r)	specifikuje podvýraz, např: (ab*c)*
	odděluje dvě varianty, např: (ano ne)?

Manipulace textu

❖ Program `cut` – umožňuje výběr sloupců textu.

```
cut -d: -f1,5 /etc/passwd  
cut -c1,5-10 soubor # znaky na pozici 1 a 5-10
```

❖ Program `sed`:

- Neinteraktivní editor textu (streaming editor).
- Kromě základních editačních operací umožňuje i podmíněné a nepodmíněné skoky (a tedy i cykly) a práci s registrem.

```
sed 's/novák/Novák/g' soubor  
sed 's/^[^:]*-/ /' /etc/passwd  
sed -e "/$xname/p" -e "/|/d" soubor_seznam  
sed '/tel:/y/0123456789/xxxxxxxxxx/' soubor  
sed -n '3,7p' soubor  
sed '1a\  
tento text bude přidán na 2. řádek' soubor  
sed -n '/start/,/stop/p' soubor
```

❖ Program `awk`:

- AWK je programovací jazyk vhodný pro zpracování textu (často strukturovaného do tabulek), výpočty atd.

```
awk '{s+=$1}END{print s}' soubor_cisel  
awk '{if(NF>0){s+=$1;n++}}  
    END{print n " " s/n}' soubor_cisel  
awk -f awk-program soubor
```

❖ Program `paste`:

- Spojení odpovídajících řádků vstupních souborů.

```
paste -d\| sloupec1.txt sloupec2.txt
```

Porovnání souborů a patchování

❖ Program `cmp`:

- Porovná dva soubory nebo jejich části byte po byte.

```
cmp soubor1 soubor2
```

❖ Program `diff`:

- Výpis rozdílů textových souborů (porovnává řádek po řádku).

```
diff old.txt new.txt  
diff -C2 old.c new.c  
diff -u2 old.c new.c  
diff -urN dir1 dir2
```

❖ Program `patch`:

- Změna textu na základě výstupu z programu `diff`.
- Podobný přístup užít v aplikacích pro správu verzí programů (cvs, svn, git, ...).

Hledání souborů

❖ Program `find`:

- Vyhledání souborů podle zadané podmínky a provedení určitých akcí nad nalezenými soubory.

```
find . -name '*.c'
find / -type d
find / -size +1000000c -exec ls -l {} \;
find / -size +1000000c -execdir command {} \;
find / -type f -size -2 -print0 | xargs -0 ls -l
find / -type f -size -2 -exec ls -l {} +
find . -mtime -1
find . -mtime +365 -exec ls -l {} \;
```

Řazení

❖ Program `sort`:

- seřazení řádků.

```
sort  soubor
sort  -u  -n  soubor_čísel
sort  -t:  -k3,3n  /etc/passwd
```

❖ Program `uniq`:

- odstranění duplicitních řádků ze seřazeného souboru.

❖ Program `comm`:

- výpis unikátních/duplicitních řádků seřazených souborů.

```
comm  soubor1  soubor2  # 3 sloupce
comm  -1 -2   s1 s2    # jen duplicity
comm  -2 -3   s1 s2    # pouze v s1
```

Další nástroje programátora

- skriptovací jazyky a interprety (perl, python, tcl, ...)
- překladače (cc/gcc, c++/g++, ...)
- assemblery (nasm, ...)
- linker `ld` (statické knihovny `.a`, dynamické knihovny `.so`)
 - výpis dynamických knihoven používaných programem: `ldd`,
 - knihovny standardně v `/lib` a `/usr/lib`,
 - cesta k případným dalším knihovnám: `LD_LIBRARY_PATH`,
 - run-time sledování funkcí volaných z dynamických knihoven: `ldtrace`.

- Program `make`:

- automatizace (nejen) překlada a linkování,
- příklad souboru `makefile` (pozor na odsazení tabelátory):

```
test: test.o tisk.o
    gcc $(CFLAGS) -o test test.o tisk.o
```

```
test.o: test.c
    gcc $(CFLAGS) -c test.c
```

```
tisk.o: tisk.c
    gcc $(CFLAGS) -c tisk.c
```

```
clean:
    rm -f *.o test
```

- použití: `make`, `make CFLAGS=-g`, `make clean`.

- **automatizovaná konfigurace** – GNU autoconf:
 - Generuje na základě šablony založené na volání předpřipravených maker skripty pro konfiguraci překladu (určení platformy, ověření dostupnosti knihoven a nástrojů, nastavení cest, ...), překlad a instalaci programů šířených ve zdrojové podobě.
 - Používá se mj. spolu s automake (usnadnění tvorby makefile) a autoscan (usnadnění tvorby šablon pro autoconf).
 - Použití vygenerovaných skriptů: `./configure`, `make`, `make install`
- **ladění**: debugger např. ddd postavený na gdb (překlad s ladícími informacemi `gcc -g ...`)
- **sledování volání jádra**: strace
- **profiling**: profiler např. gprof (překlad pomocí `gcc -pg ...`)