3 – Shell skripty, překlad a ladění zdrojového kódu

PIN1 cvičení

Matěj Klíma, David Fridrich

FJFI ČVUT v Praze, Katedra fyzikální elektroniky

9. března 2016

Obsah

- Shell - charakterizace, druhy
- Shellové skriptování v TCSH
- Překládání zdrojového kódu - C, Fortran
- Ladění zdrojového kódu
- Řízení překladu pomocí `make`
- Statické/dynamické knihovny

UNIX shell

- Shell – obálka – interpreter příkazové řádky.
- Zprostředkovává komunikaci mezi operačním systémem (jádrem) a uživatelem používajícím terminál.
- Hlavní činnost – spouštění úloh.
- Mimo to lze činnost automatizovat pomocí skriptů – interpretovaný programovací jazyk.

Nejznámější druhy shellu

- `sh` – Bourne Shell – 1977, základní shell, pozdější jsou většinou zpětně kompatibilní, na většině UNIXových systémů je `/bin/sh` alespoň jako link.
- `bash` – Bourne Again SHell – 1989, GNU náhrada za `sh`, v Linuxu asi nejpoužívanější.
- `tcsh` – Turbo/TENEX C Shell – 1983, následník `csh`, některé příkazy inspirovány C syntaxí, má zkratky pro automatické doplňování/editování příkazů (např. `!!` pro předchozí příkaz...).

Příklady budou dále uváděny v BASH. Shell používaný při spuštění můžete nastavit na `nms.fjfi.cvut.cz`.

Konfigurace shellu

- Při spuštění shellu se spouští konfigurační skript (~/.bashrc příp. ~/.tcshrc).
- Zde je možné nastavit parametry příkazové řádky: hlavičku (PS1), délku historie (HISTSIZE, HISTFILESIZE), apod.
- `alias <nazev>='<prikaz>'` vytvoří zkratku pro dlouhý příkaz.
- V tcsh `alias <nazev> <prikaz>`

Proměnné

- Podobně jako v jiných programovacích jazycích lze i v shellu používat proměnné.
- Nejčastěji označujeme velkými písmeny.
- Pokud před názvem proměnné použijeme $, nahradí se při provádění příkazu obsahem proměnné.
- Hodnotu vypíšeme příkazem `echo`.
- Nerozlišujeme zde datové typy.

Proměnné prostředí

- Proměnné prostředí se nastavují už při spuštění systému.
- Tyto proměnné se nastavují pomocí příkazu: `export <promenna>=<hodnota>`.
- V tcsh: `setenv <promenna> <hodnota>`.
- Proměnná `PATH` obsahuje cesty ke všem adresářům obsahujícím používané příkazy oddělené `:`
- Proměnná `HOME` obsahuje cestu k vašemu domovskému adresáři.

Skriptování

- Skript je soubor, ve kterém je umístěna posloupnost příkazů ke spuštění.
- Spouští se příkazy po jednotlivých řádkách.
- Je možné používat různé operátory (>,<,| atd.).
- Rozhodovací struktury `if ...then ...else ...endif`
- Cykly `foreach, for`.
- Příklady `~liska/vyuka/pin1/scripts`.

skript

- textový soubor
- na prvním řádku definice interpretru – např. `#!/bin/bash`
- případně s použitím env: `#!/bin/env python`
- následuje seznam příkazů oddělených středníkem nebo novým řádkem
- volá se buď přímo jako příkaz, nebo `bash a.sh`
- aby se spustil, musí být spustitelný: `chmod +x a.sh`
- komentáře začínají znakem #

Příklad:

```
#!/bin/bash
echo "Ahoj"
#kometar
date; pwd
```

proměnné, argumenty, read

- proměnné deklarujeme přiřazením: `P="ahoj"`
- na proměnnou se odkazujeme pomocí $: `echo $P`
- argumenty skriptu lze získat pomocí $číslo `$1, ${10}`
- $0 název skriptu
- $# počet argumentů
- $@ všechny argumenty oddělené mezerou
- k načtení obsahu proměnné ze standatdního vstupu slouží `read <proměnná>`
- výstup příkazu lze uložit do proměnné pomocí `VYSTUP=$(prikaz)`

Příklad

```
#!/bin/bash
echo "Zadano $# argumentu, skript byl zavolan jako $0"
echo "Argumenty: $@"
echo "treti argument byl $3"
DATUM=$(date)
echo "aktualni cas: $DATUM"
echo "ted napis slovo:"
read A
echo "napsal jsi $A"
```

Podmínky

```
#!/bin/bash
VOLNE=$(df -P -B 1M / |tail -n1 |awk '{print $4}')
if [ "$VOLNE" -lt "512" ]; then
    echo "Kriticky malo mista na disku"
elif [ "$VOLNE" -lt "5120" ]; then
    echo "Malo mista na disku"
else
    echo "Mista je dost"
fi
```

for ... in, for

- **foreach cyklus:**

```
#!/bin/bash
for a in *.eps; do
    epstopdf $a
done
```

- **emulace klasického for cyklu:**

```
#!/bin/bash
F=1
for I in $(seq 2 5); do
    let F=F*I
done
echo $F
```

while

```
#!/bin/bash
F=1
N=$1
I=2
while [ "$I" -le "$N" ]; do
    let F=F*I
    let I=I+1
done
echo $F
```

Návratové hodnoty

- Každý program po ukončení vrací shellu návratovou hodnotu.
- Při úspěšném ukončení je vrácena 0, jinak hodnota chyby (hodnoty $1 - 255$).
- Návratová hodnota je uložena v proměnné $?

Příklad: Předpokládejme, že soubor a neexistuje:

```
ls a
echo $?
touch a
ls a
echo $?
```

Návratové hodnoty 2

- Na základě návratové hodnoty lze spouštět další příkazy.
- `prikaz && prikaz2` – pokud `prikaz` proběhne úspěšně (jinými slovy vrátí nulu), spustí se `prikaz2`
- `prikaz || prikaz2` – pokud `prikaz` neproběhne úspěšně, spustí se `prikaz2`
- operátory `&&`, `||` lze zapsat zároveň.

Příklad:

```
ls a && echo "soubor existuje" || echo "soubor neexistuje"
```

Přesměrování

- Kromě standardního výstupu existuje i chybový výstup.
- Motivací je např. logování chyb.
- Kromě již známých operátorů pro přesměrování standardního vstupu a výstupu `<`, `<<`, `>`, `|` existují ještě 4 další pro práci s chybovým výstupem.
- `2>`, resp. `2>>` přesměruje chybový výstup do souboru.
- `&>`, resp. `&>>` přesměruje oba výstupy do souboru.
- Pokud nás některý výstup nezajímá, můžeme ho přesměrovat do speciálního zařízení `/dev/null`

Příklad pokusme se vylepšit předchozí příklad

```
ls a &> /dev/null && echo "existuje"|| echo "neexistuje"
```

Curly brackets

- `${A%retezec}` Odstraní retezec na konci proměnné
- `${A#retezec}` Odstraní retezec na začátku proměnné
- `${A/retezec1/retezec2}` Nahradí v proměnné retezec1 za retezec2
- `${#a}` vrátí délku proměnné (počet znaků)
- `${a:2:10}` vrátí 2-10 znak z proměnné

Příklad:

```
for a in *_lala.txt; do echo mv $a ${a/_lala/_};done
```

Pozor: `echo` je použito záměrně, nejprve ověřím, že přejmenovávání funguje správně a pak ho lze spustit přesměrováním do sh pomocí `|sh`

Překladače

GNU/Linux:

- `gcc` – pro C a C++
- `cc` – standardní systémový překladač C (pro kompatibilitu parametrů)
- `g++` – `gcc` přizpůsobené pro překlad pouze C++.
- `gfortran, g77` – překlad Fortranu.

další:

- `MinGW` – Minimalist GNU for Windows
- `icc, icl` – Intel C++ Compiler – dobrá optimalizace pro Intel hardware
- `ifort` – Intel Fortran

Důležité parametry GNU překladačů

-c	Pouze přeložit, nelinkovat, vytvoří objektový `soubor.o`.
-o	Následuje název výstupního souboru, jinak `a.out`.
-g	Vloží do souboru symboly pro debugger (`gdb`).
-pg	Umožňuje program spustit v profileru `gprof`.
-O[0-3]	Úroveň optimalizace překladačem.
-d...	Nastavení co vše se uloží do *core dump* souboru.
-W...	Nastavení varování překladače.
-Isoubor	Include soubory.
-Dmakro	Definice maker.
-Lknihovny	Cesty k používaným knihovnám.

Často používané přípony souborů

`.c`	C – zdrojový kód.
`.cpp`	C++ – zdrojov kód.
`.f`	Fortran – zdrojový kód s pevným formátováním.
`.f90`	Fortran 90 – zdrojový kód s volným formátováním.
`.S`	Assembler – zdrojový kód.
`.o`	Binární objekt.
`.a`	Statická knihovna.
`.so`	Dynamická knihovna (Shared Object).

Příkazy pro debugger gdb

`gdb <program>` spustíme nástroj na ladění programů.

`r <parametry>`	Spustit program v debuggeru (se vstupními parametry).
`p <promenna>`	Při pozastavení běhu programu vypíše hodnotu proměnné.
`break <soubor>:<radek>`	Vloží breakpoint na dané místo.
`ignore <i> <j>`	Ignorovat *j* průchodů breakpointem *i*.
`c`	Pokračovat v běhu programu.
`q`	Ukončit gdb (a náš program).

Alternativně použijeme grafické rozhraní `ddd`.

Soubor vygenerovaný při pádu programu se nazývá **core dump**.

Využití paměti

- Pro analýzu využití paměti je vhodný program `valgrind`.
- Obsahuje více nástrojů (např. pro analýzu paralelního kódu), nejčastěji používáme memcheck.
- `valgrind` odhalí úniky paměti (memory leaks) způsobené nejčastěji tím, že zapomeneme dealokovat dynamickou proměnnou.
- Přehled o celkovém využití paměti.
- Je schopen vyhledat špatné použití ukazatelů.

Profiler

- Nástroj pro analýzu využití systémových prostředků programem.
- Nejprve program zkompilujeme s volbou -pg.
- Poté náš program spustíme, ten vygeneruje soubor `gmon.out`.
- Profiler spustíme příkazem `gprof`.
- Výstupem profileru je statistika využití proc. času, počtů volání jednotlivých funkcí apod.

Řízení překladu pomocí make

- V souboru Makefile můžeme vytvořit skript pro překlad většího balíku kódu.
- Definujeme jednotlivé úkoly pomocí názvu a dvojtečky.
- `make all` přeloží všechny pozměněné soubory.
- `make clean` smaže soubory vytvořené programem make.
- Make umožňuje vytvářet a používat proměnné podobně jako v shellovém skriptu.