Kurs administrowania systemem Linux

Zajęcia nr 2: Powłoka systemowa

Instytut Informatyki Uniwersytetu Wrocławskiego

6 marca 2025

Polecenia basha w pliku — skrypty

- Domyślnie rozwijanie historii jest wyłączone.
- Bash kompiluje i wykonuje tylko jedną instrukcję na raz.
- Wniosek: w skrypcie mogą być błędy składniowe, które nie zostaną wykryte podczas wykonania!
- Wykonywanie skryptu:

  `$ bash` *`plik-z-programem`*
  `$ bash -c '`*`tekst programu`*`'`
- Można nadać plikowi z programem prawa do wykonania:

  `$ chmod a+x` *`plik-z-programem`*

  i uruchamiać poleceniem

  `$` *`plik-z-programem`*

  jak zwykły program.

#! (*hash-bang*, *she-bang*)

Uruchomienie skryptu z prawami do wykonania jako programu

- Pierwszy wiersz skryptu postaci

  `#!`*`nazwa-interpretera argumenty`*

  powoduje wykonanie instrukcji

  `nazwa-interpretera argumenty plik-z-programem`
- Np. jeśli plik wykonywalny `myprog` zawiera wiersz

  `#!/usr/bin/gawk -f`

  to polecenie

  `$ myprog`

  spowoduje wykonanie programu

  `/usr/bin/gawk -f myprog`
- *Konwencja:* jeśli *plik-z-programem* nie zawiera *hash-bang*, to nie powinien mieć prawa do wykonania, a jego nazwa powinna mieć rozszerzenie .sh. Jeśli zawiera *hash-bang*, to powinien mieć prawa do wykonania, a nazwa nie powinna zawierać żadnego rozszerzenia.

Struktura leksykalna basha

- Komentarze zaczynają się znakiem `#` i kończą znakiem nowego wiersza (w trybie interaktywnym można wyłączyć).
- Ciąg `\<newline>` jest usuwany.
- *Biały znak*: spacja lub znak tabulacji.
- *Metaznak*: znak nowego wiersza, `|`, `&`, `;`, `(`, `)`, `<`, `>`.
- *Token*: *słowo* (wymaga oddzielenia białymi znakami) lub *operator* (nie wymaga).
- *Słowo*: dowolny ciąg znaków różnych niż białe i metaznaki.
- *Operator*: token zbudowany z jednego lub więcej metaznaków.

Operatory i słowa kluczowe

- Operatory dzielą się na *sterujące* i *przekierowania*.
- *Operator sterujący*: znak nowego wiersza, `||`, `&&`, `&`, `;`, `;;`, `;&`, `;;&`, `|`, `|&`, `(`, `)`.
- *Operator przekierowania*: `<`, `>`, `<<`, `>>`, `<<<`, `<>`, `&>`, `>&`, `<&`, `>&`, `&>>`.
- *Słowo kluczowe*: `!`, `case`, `coproc`, `do`, `done`, `elif`, `else`, `esac`, `fi`, `for`, `function`, `if`, `in`, `select`, `then`, `until`, `while`, `{`, `}`, `time`, `[[`, `]]`. Słowo kluczowe jest zarezerwowane tylko wtedy, gdy nie jest ujęte w cudzysłowy i jest pierwszym słowem instrukcji prostej lub trzecim słowem instrukcji `case` lub `for`.

Quoting

Quoting pozwala tworzyć pojedyncze tokeny zawierające białe znaki i metaznaki:

- *Backslash* `\`: odbiera specjalne znaczenie następnemu znakowi z wyjątkiem `<newline>`.
- Apostrofy `'...'`: odbierają specjalne znaczenie ciągowi znaków.
  Ciąg nie może zawierać `'`.
- Cudzysłowy `"..."`: odbierają specjalne znaczenie ciągowi znaków z wyjątkiem `` ` ``, `$` i `\` (tylko jeśli następuje po nim `` ` ``, `$`, `\`, `"` lub `<newline>`). Zmieniają znaczenie `$*` i `$@`.
- Znaki sterujące w stylu C: `$'...'`. Specjalne znaczenie: `\a`, `\b`, `\e`, `\E`, `\f`, `\n`, `\r`, `\t`, `\v`, `\\`, `\'`, `\"`, `\`*nnn*, `\x`*HH*, `\u`*HHHH*, `\U`*HHHHHHHH*, `\c`*x*.
- Teksty zależne od wersji językowej: `$"..."`.

Składnia basha

- Parser dzieli ciągi słów na zdania zakończone `;` lub `<newline>`.
- Zdania zaczynające się słowem kluczowym są fragmentami instrukcji złożonych.
- Instrukcje proste: [*przypisanie zmiennej* ...] *nazwa-programu* [*argument* ...]
- Potoki: [`time` [`-p`]] [`!`] *instrukcja*$_1$ [[`|`|`&`] *instrukcja*$_2$ ...]
- Listy instrukcji: *potok*$_1$ [[`;`|`&`|`&&`|`||`] *potok*$_2$ ...]
- Instrukcje złożone (zawierają listy instrukcji, zmienne, wzorce, wyrażenia arytmetyczne i wyrażenia logiczne).
- W instrucji prostej lub za instrukcją złożoną mogą wystąpić przekierowania (uwaga na jednoznaczność!).

Wyrażenia, funkcje i zmienne

Wyrażenia

- Wyrażenie arytmetyczne: `(( wyr ))`, por. instrukcję `let`
- Wyrażenie logiczne: `[[ log ]]`, por. instrukcje `test` i `[`

Funkcje

- Składnia: [`function`] *zm* [`()`] *instrukcja złożona*
- Musi wystąpić co najmniej jeden z tokenów `function` i `()`
- Funkcje są rekurencyjne
- Wywołanie funkcji jest instrukcją prostą

Zmienne

- Składnia: ciąg liter, cyfr i znaku `_` nie zaczynający się cyfrą lub zmienna specjalna.
- Zmienne specjalne: `*`, `&`, `#`, `?`, `-`, `$`, `!`, `0`, n ($n > 0$), `_`
- Odwołanie do zmiennej („dereferencja”): `$`[`{`]*zmienna*[`}`]

Rozwinięcia w Bashu

Na tekście instrukcji prostej wybranej do wykonania wykonuje się w kolejności ciąg rozwinięć:

1. rozwinięcia nawiasów wąsatych, np. `file{1,2,3}`, `file{1..10}`,
2. rozwinięcia tyldy, np. `~/Downloads/`,
   rozwinięcia zmiennych, np. `$HOME`,
   podstawienia instrukcji, np. `$(cat file.txt)`,
   podstawienia procesów, np. `<(pdftops file.pdf -)`,
   rozwinięcia arytmetyczne, np. `$((N+1))`,
3. (powtórny) podział słów,
4. rozwinięcia nazw plików (*globy*), np. `file?-*.txt`.

Rozwinięcia są wykonywane od lewej do prawej.

Zasady rozwijania

Rozwinięcia tylko w jednej instrukcji prostej

- Podstawienia są wykonywane *tylko* w bieżąco *wykonywanej* instrukcji *prostej*:

```
$ cd /usr/share; (cd ..; echo $PWD); echo $PWD
/usr
/usr/share
```

- Podobnie, w ciągach instrukcji zagnieżdżonych wewnątrz instrukcji prostej, tj. w konstrukcjach `$(...)` i `<(...)` oraz w wyrażeniach arytmetycznych, tj. w konstrukcji `$((...))`, podstawienia są wykonywane dopiero *podczas wykonywania* każdej z tych instrukcji osobno:

```
$ cd /usr/share; echo "$PWD $(cd ..; echo $PWD)"
/usr/share /usr
$ N=0; echo $((N++, N))
1
```

Zasady rozwijania, cd.

Powtórny podział słów i jego wyłączanie

- Powtórny podział słów jest niezbędny, gdyż inne podstawienia (z wyjątkiem rozwinięć *globów*) wstawiają pojedyncze tokeny:

```
$ DIRS="/usr /var /etc"; ls -d $DIRS; ls -d "$DIRS"
/etc/ /usr/ /var/
/bin/ls: cannot access /usr /var /etc: No such file or directory
```

- Podstawowym zadaniem cudzysłowów "..." jest wyłączenie powtórnego podziału na słowa.
- *Globy* wstawiają wiele tokenów (po jednym dla każdej nazwy pliku, nawet jeśli zawiera spacje) i jako jedyne są rozwijane *po* powtórnym podziale na słowa. Dzięki temu nazwy plików zawierające spacje nie rozpadną się.

```
$ touch 'a b'; ls a*
a b
$ eval ls a*
/bin/ls: cannot access a: No such file or directory
/bin/ls: cannot access b: No such file or directory
```

Rozwinięcia wyliczeniowych nawiasów wąsatych

Wygenerowanie ciągu słów według wzoru:

```
$ echo Klaud{ia,io,yna}
Klaudia Klaudio Klaudyna
```

Polecenie

```
$ mkdir -p /usr/local/share/texmf/{fonts/{{truetype,tfm}/myfont,{map,enc}/pdftex},tex/latex/myfont}
```

utworzy katalogi (co ciekawe, w podanej niżej kolejności):

```
/usr/local/share/texmf/fonts/enc/pdftex/
/usr/local/share/texmf/fonts/map/pdftex/
/usr/local/share/texmf/fonts/tfm/myfont/
/usr/local/share/texmf/fonts/truetype/myfont/
/usr/local/share/texmf/tex/latex/myfont/
```

Rozwinięcia arytmetycznych nawiasów wąsatych

```
$ echo {5..12}
5 6 7 8 9 10 11 12
$ echo {8..1..2}
8 6 4 2
```

Można zagnieżdżać w wyliczeniowych nawiasach wąsatych:

```
$ echo {{5..12},{14..18}}
5 6 7 8 9 10 11 12 14 15 16 17 18
```

Przydatne rozszerzenie (zera wiodące):

```
$ echo {01..15}
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
```

Rozwinięcia nazwiasów wąsatych są wykonywane *przed* główną grupą rozwinięć (podstawieniami zmiennych itp.):

```
$ X1=a; X2=b; X3=c; echo $X{1..3}
a b c
```

Rozwinięcia tyldy

Formy:

- `~` — `$HOME` użytkownika, np. `~/.bashrc` → `/home/user/.bashrc`
- `~`*`otheruser`* — `$HOME` użytkownika *`otheruser`*,
  np. `~jan/.bashrc` → `/home/jan/.bashrc`
- `~+` — `$PWD`
- `~-` — `$OLDPWD`, np.

  ```
  $ cd /tmp; touch a; cd ~; echo $OLDPWD $PWD; ls ~-/a
  /tmp /home/user
  /tmp/a
  ```
- `~`*`N`*, `~+`*`N`*, `~-`*`N`* — zob. polecenie wbudowane `dirs`.

Podstawy korzystania z powłoki systemowej

Idiomatyzmy basha

- Środowisko a podprocesy: używanie zmiennych w podprocesach
- Problemy podziału na słowa: spacje w nazwach plików
- Usuwanie pustych słów
- Błędy w trakcie wykonania skryptów
- Globy
- Przekierowania
- *Sourcing*

Wywoływanie programów

- Opcje programów i `getopt`
- Dokumentacja: `man` i `info`

Refleksje

Współpraca lub walka z powłoką

- Działanie skierowane na osiągnięcie celu nie jest dobre!
- Przede wszystkim należy zrozumieć i wyjaśnić!
- Nie „kopcie się”, tylko współpracujcie z powłoką!

Przekombinowane

```
program_1 < <(program_2)
```

Refleksje

Współpraca lub walka z powłoką

- Działanie skierowane na osiągnięcie celu nie jest dobre!
- Przede wszystkim należy zrozumieć i wyjaśnić!
- Nie „kopcie się”, tylko współpracujcie z powłoką!

Przekombinowane

```
program_1 < <(program_2)
program_2 | program_1
```

Jak lepiej?

```
program_1 &| program_2
program_1 2>&1 | program_2
```

Środowisko a podprocesy

script.sh

```
echo "Hello $MSG!"
```

```
$ MSG=World; bash script.sh
???
```

Środowisko a podprocesy

script.sh

```
echo "Hello $MSG!"
```

```
$ MSG=World; bash script.sh
Hello !
```

Środowisko a podprocesy

script.sh

```
echo "Hello $MSG!"
```

```
$ export MSG=World; bash script.sh
???
```

Środowisko a podprocesy

script.sh

```
echo "Hello $MSG!"
```

```
$ export MSG=World; bash script.sh
Hello World!
```

Środowisko a podprocesy

script.sh

```
echo "Hello $MSG!"
```

```
$ MSG=World bash script.sh
???
```

Środowisko a podprocesy

script.sh

```
echo "Hello $MSG!"
```

```
$ MSG=World bash script.sh
Hello World!
```

Środowisko a podprocesy

script.sh

```
cmd there
```

```
$ function cmd { echo "Hello $1!" }
$ bash script.sh
???
```

Środowisko a podprocesy

script.sh

```
cmd there
```

```
$ function cmd { echo "Hello $1!" }
$ bash script.sh
script.sh: line 1: cmd: command not found
```

Środowisko a podprocesy

script.sh

```
cmd there
```

```
$ function cmd { echo "Hello $1!" }
$ export -f cmd; bash script.sh
???
```

Środowisko a podprocesy

script.sh

```
cmd there
```

```
$ function cmd { echo "Hello $1!" }
$ export -f cmd; bash script.sh
Hello there!
```

Środowisko a podprocesy

script.sh

```
cmd there
```

```
$ function cmd { echo "Hello $1!" }
$ export -f cmd; bash script.sh
Hello there!
```

- Zmienne i funkcje są przekazywane **do** podprocesu tylko wtedy, gdy są eksportowane.
- Nie ma możliwości przekazania zmiennych i funkcji **z** podprocesu.