Wstęp do informatyki

Polecenia (cz.2)

Lista procesów

top

Pokaż listę procesów – polecenie interaktywne

Procesy

Uruchamianie w tle.
shell nie czeka na zakończenie procesu, można wydawać kolejne polecenia.

```
> ls &
[1] 11101
...
[1]+ Done
```

Poprawne zakończenie - `Done`

Procesy

Czasem wymagane jest przekierowanie strumieni, gdy proces działający w tle intensywnie pisze na wyjściu.

```
> emacs &

> grep  -v  Ala *.txt  >wynik.txt 2>/dev/null &

> false &
[1] 11102
[1]+ Exit 1
```

Kod zakończenia – porażka (<>0)

Procesy

Przypomnienie – niektóre polecenia (wbudowane, patrz `help`) nie uruchamiają nowego procesu. Wyjątek: program uruchamiany w tle.

Polecenie wbudowane, nie ma nowego procesu:

```
> echo A
```

Powstanie proces potomny (bash), który wykona to polecenie.

```
> echo A &
```

Programowanie w shellu

Skrypt = plik tekstowy zawierający polecenia dla intepretera (tu: bash).

```
> cat skrypt.sh
ls
pwd
> ./skrypt.sh
a.txt b.txt skrypt.sh
/home/gorecki/wi
```

Pamiętajmy o nadaniu praw dostępu:

```
> chmod u+x skrypt.sh
```

Słowo – ciąg liter i cyfr
Fraza – ciąg słów oddzielonych białymi znakami
Potok – ciąg fraz oddzielony |
Lista – ciąg potoków oddzielony ; & && ||

Lista w nawiasach ( ) jest frazą.

Proste polecenie to fraza, w której pierwsze słowo jest interpretowane jako nazwa programu do wykonania bądź instrukcja.

Shell programowanie

Kod stanu – wykonanie każdej frazy dostarcza wynik liczbowy. Jeśli wykonanie zakończyło się sukcesem wynikiem powinno być 0. Jest to rozszerzenie pojęcia *kodu zakończenia procesu.*

Wynikiem potoku jest wynik ostatniej frazy.

Negacja potoku: dodaj ! przed.

Shell programowanie

Wynik listy:

& - potok jest wykonywany asynchronicznie (w tle)

```
> emacs &
```

; - wykonanie sekwencyjne; wykonanie następnego potoku po zakończeniu poprzedniego

```
> ls /tmp; echo ”WI”; cat skrypt.sh
```

Shell programowanie

&& - następny potok będzie wykonany gdy poprzedni zakończył działanie z wynikiem 0 (tak jak operator koniunkcji AND)

```
> true && true && echo ”AAA”
```

|| - następny potok będzie wykonany gdy poprzedni zakończył działanie z wynikiem różnym od 0 (tak jak operator alternatywy OR)

```
> false || echo ”AAA”
```

Środowisko

Środowisko przechowuje zmienne ich wartości. Dostęp do zmiennej ma postać $nazwa.
Każdy proces posiada własny zestaw zmiennych.

```
> echo $HOME
/home/gorecki

> cnt=1; echo $cnt
1

> set
...
```

Środowisko

Środowisko przechowuje zmienne ich wartości. Dostęp do zmiennej ma postać $nazwa.
Każdy proces posiada własny zestaw zmiennych.

```
> echo $HOME
/home/gorecki

> cnt=1; echo $cnt
1

> set
...
```

Środowisko

Zmienne mogą być eksportowane do procesów potomnych. Przykład (wykonać kolejne polecenia):

```
> mz=0
> bash
> echo $mz

> exit
> export mz=JestDobrze
> bash
> echo $mz
JestDobrze
> mz=JestesPewien
> exit
> echo $mz
???
```

Środowisko

Warto zapamiętać:

```
> export PATH=$PATH:.
```

Praca domowa – do czego można tego użyć?

Wskazówka:

```
> ./skrypt.sh
> skrypt.sh
```

Środowisko

Użyteczne zmienne specjalne (często używane w skryptach):

$0 – nazwa bieżącego polecenia
$1,$2,$3,... - parametry
$* - wszystkie parametry
$# - liczba parametrów
$? - kod ostatnio wykonanego polecenia

shift [n] – przesuń parametry $1,$2,... o n pozycji

Testy

`test opcje`

Sprawdzanie warunków wszelakich. Używane często w warunkach instrukcji if, while, etc.
Forma skrócona: `[ opcje ]`

```
> test -r plik.txt # czy plik dający się czytać
> [ $VAL -gt 0 ] # prawda gdy $VAL > 0
> [ -f a.txt ] # prawda gdy a.txt jest plikiem
> [ -d k ] # prawda gdy k jest katalogiem
```

Więcej: man test

Wyrażenia arytmetyczne

`let wyrażenie`

Obliczanie wyrażeń.
Forma skrócona: `(( wyrażenie ))`

```
> cnt=0; let cnt=cnt+1; echo $cnt
> (( cnt++ ))
> (( cnt=10 ))
```

Więcej w man bash (OBLICZANIE WYRAŻEŃ)

Oblicznie wyrażenia

`expr wyrażenie`

Obliczanie wyrażeń (arytmetycznych i tekstowych) i wynik na stdout – polecenie expr.

```
> napis=”Hello!”
> expr length $napis
> expr 1 + 3
```

Więcej w man expr.

(*) Długości napisu także: `${#napis}`

Pętla for

```
for NAME [ in WORDS ;]  do COMMANDS; done
```

Zwykła pętla for po słowach z WORDS. Jeśli WORDS nie są podane pętla wykonywana po $@ (czyli $1 $2 $3 ...).

```
> for i in * ; do echo $i ; cp $i /tmp ; done
```

```
> for i do echo $i ; done
```

```
for i do
 echo $i
done
```

Przykłady

```
> for i in ala ma kota ; do echo $i ; done
> for i in $( seq 1 100 ); do echo $i; done
```

Pętla for - arytmetyczna

```
for (( INIT; WAR; POST )) ; do COMMANDS; done
```

Na początku jest obliczane wyrażenie INIT

Wielokrotnie jest obliczane wyrażenie WAR, aż do osiągnięcia wartości 0.

Jeżeli WAR jest niezerowe wykonywane są polecenia i obliczane jest POST

Jeżeli pominięto jakieś wyrażenie, to domyślnie jest tam 1.

Przykład

```
>for (( i=1 ; i <=100; i++)) ; do echo $i ; done

for (( i = 1; i <= 10 ; i++ )) ; do
  j=$i.txt; echo $j
  touch $j
done
```

Pętla while

```
while COMMANDS1; do COMMANDS2; done
```

Pętla while wykonywane COMMANDS2 dopóki COMMANDS1 są spełnione (zerowy kod zakończenia - $?).

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

```
> cnt=0
> while [ $cnt -lt 10 ]
do
   let cnt=cnt+1
   echo $cnt
done
```

read

`read a b c ...` – czytaj jedną linię z wejścia i zapisz na podane zmienne; separatory to białe znaki (spacje, tabulatory); jeśli na wejściu jest więcej napisów niż zmiennych to ostatnia zmienna otrzyma resztę.

```
> read i
WI
> echo $i

> read a b
Ala ma kota
> echo $a
Ala
> echo $b
???
```

read

`read` zwraca kod 0 (sukces) gdy coś zostało przeczytane; gdy błąd lub koniec pliku zwraca <>0.

```
while read linia; do
  echo $linia
done
```

`read` jest często używany z `while`

```
while read a b c <a.txt; do echo $a; done
```

Źle

```
while read a b c ; do echo $a; done <a.txt
```

OK

Przykład z read

```
> ls -al | while read prawa reszta
do
    echo $prawa
    echo $reszta
done
```

Instrukcja if

```
if COMMANDS1; then COMMANDS2 ;
 [ elif COMMANDS2; then COMMANDS4;] ...
 [ else COMMANDS5; ] fi
```

Instrukcja warunkowa.

```
> if true; then echo ”PRAWDA”; fi
```

```
>  if  [  $cnt  -gt  10  ];  then  exit  -1;  else
(( cnt++ )); fi
```

Przykład + break

Instrukcja `break` przerywa pętlę `(for, while)`.

```
while read i; do
  if [ ”$i” == ”ala” ]
    then break
  fi
done
```

Funkcje

```
[function] NAME () { COMMANDS; }
```

Definiowanie funkcji

```
NAME ARG1 ARG2
```

Uruchamianie funkcji; (Parametry w $1 $2 ...)

```
f() { echo $1 ; }
f ”Ala bez kota”
```

Cudzysłowy, apostrofy, etc

```
> echo $HOME     *
/home/gorecki a.txt b.txt
```

Zwykły cudzysłów: ”...” rozwija zmienne ale nie wzorce plików.

```
> echo ”$HOME     *”
/home/gorecki *
```

Apostrof kasuje znaczenie znaków specjalnych

```
> echo '$HOME     *'
$HOME     *
```

Uwaga: dlaczego użyłem spacji w echo?

Cudzysłowy, apostrofy, etc

Odwotny apostrof `...` – wykonuje polecenie i podstawia tekst z stdout.
Forma zalecana: `$(polecenie)`

```
> echo `ls`
> echo `date`
> $(date)
???

> for i in *.jpg
do
  mv $i `basename $i .jpg`.jpeg
done
```

Przykłady dalsze

```
> for i in `seq 1 100`; do echo $i; done
```