Corso di Laboratorio di Sistemi Operativi

Lezione 4

Alessandro Dal Palù

email: alessandro.dalpalu@unipr.it

web: www.unipr.it/~dalpalu

Script in Bash

- Gli shell *script* sono programmi interpretati dalla shell, scritti con un linguaggio i cui costrutti atomici sono i comandi di shell

- Lo shell script viene scritto mediante un editor testuale (vi, emacs, ..)

- Lo script può essere eseguito passando il file all'interprete come argomento: `/bin/bash script-name`

- E' possibile lanciare direttamente il nome dello script scrivendo il nome dell'interprete come prima riga dello script stesso:
  `#!/bin/bash`
  ....

- Un file script deve possedere l'attributo *x* (eseguibile)

- Se la directory che contiene lo script non è inclusa in `PATH`, lanciare con `./nome_script` (consiglio: aggiungere . a `PATH`)

Script: Input e Output da terminale

- Output: `echo stringa` (opzione *-n* non va a capo dopo la stampa)
- Input: `read varname`: la stringa letta viene assegnata alla variabile `varname`.
- `read var1 var2 ...`: legge le parole che compongono input e le in ordine a var1, var2...
- `read -a nomearray`: le parole della stringa entrano in un array
- Esempio:

```
#!/bin/bash
echo -n "Inserire una stringa:"
read -a vett << EOF
zero uno due tre
EOF
echo ${vett[2]}
```

Array

- Sono dinamici ad una sola dimensione con indice numerico a partire da 0

- Esempio: `a=(0 1 2 3 4) ; echo ${a[2]}`

- Per ottenere la dimensione: `${#a[*]}` oppure `${#a[@]}`

Parametri di shell e Liste

- Una lista è una sequenza di caratteri separata da spazio (di default)
- La lista più utilizzata è quella dei parametri di shell.
- `$0` rappresenta il nome dello script (o shell)
- `$1 $2` ecc rappresentano i parametri posizionali
- `$*` o `$@` rappresentano la lista intera dei parametri a partire dal primo
- Le liste possono essere scandite con il comando `in` (vedi ciclo for)

Command substitution

Il meccanismo di **command substitution** permette di sostituire ad un comando o pipeline quanto stampato sullo standard output da quest'ultimo. Esempi:

```
> date
Tue Nov 19 17:50:10  2002
> vardata=‘date‘
> echo $vardata
Tue Nov 19 17:51:28  2002
```

Un comando molto usato con le command susbstitution è `basename` (restituisce il nome di un file, senza il path):

```
> basefile=‘basename /usr/bin/man‘
> echo $basefile
man
```

Importante: per operare una command substitution si devono usare gli "apici rovesciati" o backquote (‘), non gli apici normali (’) che si usano come meccanismo di quoting.

if-then-else

- Il comando condizionale

```
if condition_command
then
   true_commands
else
   false_commands
fi
```

- esegue il comando `condition_command`, utilizza il suo **exit status** per decidere se eseguire i comandi `true_commands` (exit status 0) o i comandi `false_commands` (exit status diverso da zero).
- Nota: per collassare più righe in una, è necessario usare il separatore di comandi `;`

Condizioni: exit status e comando test (I)

Se la condizione che si vuole specificare non è esprimibile tramite l'exit status di un "normale" comando, si può utilizzare l'apposito comando `test`:

`test expression` oppure `[expression]`

che restituisce un exit status pari a `0` se `expression` è vera, pari a `1` altrimenti.

Si possono costruire vari tipi di espressioni:

- espressioni che controllano se un file possiede certi attributi:
  `-e` *f* restituisce vero se *f* esiste;
  `-f` *f* restituisce vero se *f* è un file ordinario;
  `-d` *f* restituisce vero se *f* è una directory;
  `-r` *f* restituisce vero se *f* è leggibile dall'utente;
  `-w` *f* restituisce vero se *f* è scrivibile dall'utente;
  `-x` *f* restituisce vero se *f* è eseguibile dall'utente;

- espressioni su stringhe:
  `-z` *str* restituisce vero se *str* è di lunghezza zero;
  `-n` *str* restituisce vero se *str* non è di lunghezza zero;
  str1 `=` *str2* restituisce vero se *str1* è uguale a *str2*;
  str1 `!=` *str2* restituisce vero se *str1* è diversa da *str2*;

Condizioni: exit status e comando test (II)

- espressioni su valori numerici:
  num1 `-eq` *num2* restituisce vero se *num1* è uguale a *num2*;
  num1 `-ne` *num2* restituisce vero se *num1* non è uguale a *num2*;
  num1 `-lt` *num2* restituisce vero se *num1* è minore di *num2*;
  num1 `-gt` *num2* restituisce vero se *num1* è maggiore di *num2*;
  num1 `-le` *num2* restituisce vero se *num1* è minore o uguale a *num2*;
  num1 `-ge` *num2* restituisce vero se *num1* è maggiore o uguale a *num2*

- operatori booleani:
  exp1 `-a` *exp2* restituisce vero se sono vere sia *exp1* che *exp2*
  exp1 `-o` *exp2* restituisce vero se è vera *exp1* o *exp2*
  `!`*exp* restituisce vero se non è vera *exp*

Per costruire espressioni numeriche complesse: `$[expression]`
Ad esempio:

```
> num1=2
> num1=$[$num1*3+1]
> echo $num1
7
```

Cicli while

Sintassi:

```
while condition_command
do
    commands
done
```

L'effetto risultante è che vengono eseguiti i comandi `commands` finché la condizione `condition_command` è vera. Esempio:

```
while test -e $1
do
    sleep 2
done
```

Lo script precedente esegue un ciclo che dura finché il file fornito come argomento non viene cancellato. Il comando che viene eseguito come corpo del `while` è una pausa di 2 secondi.

Cicli for

Sintassi:

```
for var in wordlist
do
    commands
done
```

L'effetto risultante è che vengono eseguiti i comandi `commands` per tutti gli elementi contenuti in `wordlist` (l'elemento corrente è memorizzato nella variabile `var`). Esempio:

```
for i in 1 2 3 4 5
do
    echo the value of i is $i
done
exit 0
```

```
for i in *
do
    echo Listing file: $i
done
exit 0
```

Select

Permette all'utente di scegliere tra le opzioni elencate dopo `in`

```
#!/bin/bash
select param in uno due tre
 do
  echo Hai selezionato: $param
  break
 done
```

Cosa succede senza `break`?

Case

Permette di gestire una scelta multipla:

```
case parola in
modello1) lista comandi;;
modello2) lista comandi;;
*) comandi;;
esac
```

Esempio:

```
case $1 in
-a) echo scelta a ;;
-b) echo scelta b ;;
-c) echo scelta c ;;
*) echo opz. sconosciuta ;;
esac
```

Esempio (I)

Progettare uno script, chiamato `listfiles`, che prende due parametri, una directory e la dimensione di un file in byte. Lo script deve fornire il nome di tutti i file regolari contenuti nella directory parametro ai quali avete accesso e che sono più piccoli della dimensione data. Si controlli che i parametri passati sulla linea di comando siano due e che il primo sia una directory.

Esempio di soluzione (prima parte: controllo dei parametri):

```
if test $# -ne 2
then
   echo 'usage: listfiles <dirpath> <dimensione>'
   exit 1
fi
if ! test -d $1; then
   echo 'usage: listfiles <dirpath> <dimensione>'
   exit 1
fi
```

Esempio (II)

Esempio di soluzione (seconda parte: esecuzione del compito stabilito nell'esercizio):

```
 for i in $1/* do
    if test -r $i -a -f $i
    then
        size=‘wc -c <$i‘
        if test $size -lt $2
        then
            echo ‘basename $i‘ has size $size bytes
        fi
    fi
 done

exit 0
```

Esercizi 1^a parte

- Progettare uno script `draw_triangle` che prende in input un parametro intero con valore da 3 a 15 e disegna sullo standard output un triangolo (utilizzando i caratteri -, / e \) come nel seguente esempio:

```
> draw_triangle 4
   /\
  /  \
 /    \
--------
```

- Progettare uno script che prende in input come parametro il nome di una directory e conta quanti file hanno dimensione $\leq$ 1KB, $\leq$ 1MB, $\leq$ 1GB e $>$ 1GB.

Esercizi 2^a parte

- Progettare uno utility-script `processi` per la gestione user-friendly dei processi in memoria. Le operazioni di base sono gestite tramite un semplice menu testuale con input da tastiera.

- Visualizzazione dei processi dell' utente corrente (PID e riga comando che lo ha generato)

- Eseguire kill su un proprio processo (in input il PID)

- Eseguire kill -9 su un proprio processo (in input il PID)

- Mostrare l'elenco degli utenti che hanno almeno un processo attivo nel sistema

- Ogni funzione, una volta completata, riporta al menu principale di scelta

Make (per la prossima lezione)

- `https://www.gnu.org/software/make/manual/html_node/Introduction.html`
- Leggere le informazioni nel sito (Rule introduction, Simple Makefile e How Make works)
- Creare un riassunto in latex (in italiano) da allegare alla relazione