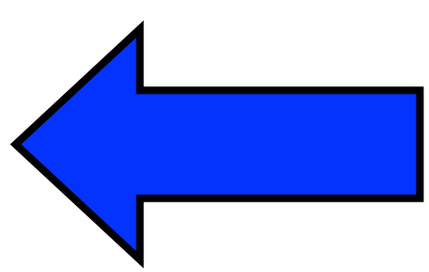
laboratorio di sistemi operativi

bash scripting

Marco Botta
Materiale preparato da Daniele Radicioni

argomenti del laboratorio UNIX

1. introduzione a UNIX;
2. integrazione C: operatori bitwise, precedenze, preprocessore, pacchettizzazione del codice, compilazione condizionale e utility make;
3. controllo dei processi;
4. segnali;
5. pipe e fifo;
6. code di messaggi;
7. semafori;
8. memoria condivisa;
9. introduzione alla programmazione bash.



- il materiale di questa lezione è tratto:
  - dai lucidi della Prof. Baroglio degli anni scorsi;
  - Randal K. Michael, ***Mastering UNIX®Shell Scripting, Bash, Bourne, and Korn Shell Scripting for Programmers, System Administrators, and UNIX Gurus***, Second Edition, Wiley Publishing, Inc., 2008.

```
$ ipcs -s
IPC status from <running system> as of Tue Nov 21 12:04:58 CET 2021
T     ID      KEY         MODE        OWNER    GROUP
Semaphores:
s 196608 0x7402a210 --ra-------   radicion   wheel

$
```

```
#!/bin/bash

for i in `ipcs -s | grep radicion | awk '{print $2}'`; do
  ipcrm -s $i;
done
```

bash

- Bash è la shell di default per Linux. È una shell Unix, cioè:
  - un interprete di comandi che costituisce l'interfaccia utente verso una ricca offerta di utility e linguaggi di programmazione
  - un linguaggio di programmazione tramite il quale combinare utility
- È possibile creare file di comandi (script di shell) che diventano a loro volta nuovi comandi.

caratteristiche principali

- è possibile effettuare redirezioni di input ed output :
  - *$ cmd > outfile*
- è possibile sequenzializzare comandi sullo stesso prompt:
  - *$ cmd1 < dati ; cmd2 ; cmd3 >> outfile*
- è possibile combinare programmi mediante pipe:
  - *$ cmd1 | cmd2 | cmd3*

shell scripting

- Un linguaggio di scripting è un linguaggio di programmazione interpretato destinato a compiti di automazione di sistema, piccole applicazioni.
  - generalmente si tratta di semplici programmi destinati a interagire con sistemi più complessi.
- I primi linguaggi di scripting nacquero dall'esigenza di automatizzare operazioni ripetitive come l'esecuzione di particolari programmi;
  - attualmente sono molto diffusi anche nello sviluppo per il web.

script

- uno ***script*** è un programma interpretato; è un file di testo contenente dichiarazioni di variabili, comandi e strutture di controllo;
  - il file ha i diritti di esecuzione impostati
  - i contenuti di uno script vengono letti ed eseguiti direttamente dalla shell
- di norma lo script ha una prima linea speciale:

  `#!/bin/nomeshell`

  - dove ***nomeshell*** indica il tipo di shell in grado di interpretare lo script (***bash***, ***tcsh***, ***csh***, ***ksh***, ...), es.

`#!/bin/bash`

```
#!/bin/bash
# questo è un commento!!!
echo Hello World
```

mio_script

```
-rwxr-xr-x  1 radicion  staff   16 Dec 4 03:54 mio_script
```

- La prima riga indica al sistema quale software usare per avviare lo script.
- La terza riga è l'azione eseguita dallo script che stamperà a video la scritta ***Hello World***.
  - Fornendo diritti di esecuzione allo script esso potrà essere eseguito tramite il comando

./mio_script

esercizio

```
#!/bin/bash

# questo è un commento!!!
echo Hello World
```

- creare un file di testo;
- copiarvi il codice riportato nell'esempio, salvare e uscire;
- rendere eseguibile il file tramite ***chmod***;
- eseguire il programma così prodotto.

variabili

- Una variabile viene dichiarata nel momento in cui le viene assegnato un valore (stringa o numero);

- Dichiarazione

```
nomevar=valore
```

- Si accede al valore di una variabile tramite ***$nomevar***,

```
echo $nomevar
```

argomenti degli script

- Gli script possono avere parametri, identificati dalla loro posizione. per esempio

```
echo $0 $1
```

stampa i primi due parametri del programma

- Attenzione al valore di ***$0*** ... vi ricordate di ***argv***?

esercizio

```
#!/bin/bash

vrb1=5
echo $vrb1
```

```
#!/bin/bash

echo $0 $1
```

- Scrivere ed eseguire i due script riportati sopra.

quoting

- Quotare significa interpretare come normali dei caratteri che hanno un significato speciale, come per esempio *$*

```
#!/bin/bash
VAR=prova
VAR1=23
echo $VRB $VRB1
echo $0 $1

echo "il valore è $VAR"
echo `ls -l $VAR`
echo '$VAR'
```

APICI DOPPI

APICI SINGOLI INVERSI

APICI SINGOLI

quoting

```
#!/bin/bash
VAR=prova
VAR1=23
echo $VAR $VAR1
echo $0 $1

echo "il valore è $VAR"
echo `ls -l $VAR`
echo '$VAR'
```

stampa la stringa sostituendo il valore

quoting

```
#!/bin/bash
VAR=prova
VAR1=23
echo $VAR $VAR1
echo $0 $1

echo "il valore è $VAR"
echo `ls -l $VAR`
echo '$VAR'
```

stampa la stringa sostituendo il valore

effettua sostituzione, interpreta come comando e restituisce il risultato dell'esecuzione del comando

quoting

```
#!/bin/bash
VAR=prova
VAR1=23
echo $VAR $VAR1
echo $0 $1

echo "il valore è $VAR"
echo `ls -l $VAR`
echo '$VAR'
```

stampa la stringa sostituendo il valore

effettua sostituzione, interpreta come comando e restituisce il risultato dell'esecuzione del comando

non effettua alcuna sostituzione

assegnamento

```
#!/bin/bash

tmp=`ls -la | wc -l`
tmp1=`ls -l | wc -l`

echo "numero di elementi nella directory: "$tmp
echo "di questi, non hidden: " $tmp1

let "diff=$tmp-$tmp1"
echo "hidden: " $diff
```

let permette di assegnare a variabili il valore di espressioni

sintassi

```
let "var = espressione"
```

operatori

```
+ - * / % ...
```

costrutti di controllo

```
let "ripeti = 1"
while [ $ripeti -lt 3 ]
do
  echo "hip hip"
  let "ripeti = $ripeti + 1"
done
echo "hurra!!"
```

costrutti di controllo

Inizializzazione
(anche: *ripeti=1*)

operatore di confronto

condizione: *[condizione]*
spazi obbligatori

istruzione di ciclo

incremento

```
let "ripeti = 1"
while [ $ripeti -lt 3 ]
do
  echo "hip hip"
  let "ripeti = $ripeti + 1"
done
echo "hurra!!"
```

= uguale	*-gt* maggiore	*-ge* maggiore uguale	*-eq* uguale
!= diverso	*-lt* minore	*-le* minore uguale	*-ne* diverso

lettura da *stdin*

```
finito=go
while [ $finito != quit ]
do
  echo "un altro giro? [go/quit] "
  read finito
done
```

legge un valore da standard input e lo assegna alla variabile indicata

if then else

```
DEFDIR=path_to_dir
if [ $1 ]
  then
  if [ $1 = "-d" ]
    then
    mia_var=`wc -l $2`
    echo $mia_var
  else
    echo "opzione sconosciuta"
  fi
  else
    mia_var=`wc -l $DEFDIR`
    echo $mia_var
fi
```

if then else

```
DEFDIR=path_to_dir
if [ $1 ]
  then
  if [ $1 = "-d" ]
    then
    mia_var=`wc -l $2`
    echo $mia_var
  else
    echo "opzione sconosciuta"
  fi
  else
    mia_var=`wc -l $DEFDIR`
    echo $mia_var
fi
```

se il parametro 1 è definito

se è uguale a “-d”

esegui queste istruzioni

altrimenti

se param. 1 non è definito

if then else

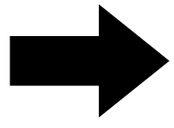
```
DEFDIR=path_to_dir
if [ $1 ]
  then
  if [ $1 = "-d" ]
    then
    mia_var=`wc -l $2`
    echo $mia_var
  else
    echo "opzione sconosciuta"
  fi
  else
    mia_var=`wc -l $DEFDIR`
    echo $mia_var
fi
```

condizioni composte ***[condizione1] && [condizione2]***
[condizione1] || [condizione2]

test su file

- Bash definisce una serie di operatori per effettuare test su file

```
[ -r documento.txt ]
[ -x $1 ]
[ -d $1 ] && [ -w $1 ]
```



- `-f` file esiste
- `-s` file non è vuoto
- `-r` file leggibile
- `-w` file scrivibile
- `-x` file eseguibile
- `-d` è una directory
- `-h` è un link simbolico
- ...

il file indicato è leggibile?

il parametro è un eseguibile?

$1 è una directory scrivibile?

for … in . . .

```
for variabile in lista-valori
do
   istruzioni
done
```

cursore: non viene incrementato esplicitamente

```
for myf in `ls *.c`
do
   echo $myf
done
```

```
#!/bin/bash

for i in `ipcs -s`; do
   // fai qualcosa
done
```

for … in . . .

```
for variabile in lista-valori
do
   istruzioni
done
```

cursore: non viene incrementato esplicitamente

```
for myf in `ls *.c`
do
   echo $myf
done
```

per tutti i file con estensione `.c` della directory corrente

altro esempio

```
for myf in `ls *.c`
do
   echo `< $myf`
done
```

esportazione di variabili

- ***Scope*** di una variabile: lo script in cui è dichiarata
- … e se da uno script ne richiamiamo un altro?

mio_script

```
dato=`< mioinput`
saluta
```

saluta

```
echo "ciao " $dato
```

- se da ***mio_script*** richiamo ***saluta***, dal secondo ho accesso alle variabili dichiarate nel primo ?

esportazione di variabili

- ***Scope*** di una variabile: lo script in cui è dichiarata
- ... e se da uno script ne richiamiamo un altro?

mio_script

```
dato=`< mioinput`
saluta
```

saluta

```
echo "ciao " $dato
```

- "saluta" viene eseguito da una bash figlia di quella che esegue ***mio_script*** dalla quale non si ha accesso alle variabili di ***mio_script***

NO!

esportazione di variabili

- per rendere una variabile visibile dalle sotto-shell occorre esportarla

mio_script

```
export dato=`< mioinput`
saluta
```

saluta

```
echo "ciao " $dato
```

File di configurazione: *.bashrc*

```
export PATH=.:mybin:$PATH
```

File di configurazione: ***.bashrc***

```
export PATH=.:mybin:$PATH
```

concatena alla lista attuale di directory in cui cercare gli eseguibili le directory '.' (directory di lavoro, qualunque essa sia) e '*mybin*'

File di configurazione: *.bashrc, .zprofile*

```
export PATH=.:mybin:$PATH
```

concatena alla lista attuale di directory in cui cercare gli eseguibili le directory '.' (directory di lavoro, qualunque essa sia) e '*mybin*'

```
alias a=alias
a hi=history
a m=more
a c=clear
a rm="rm -i"
a x=exit
a so=source
a d=date
```

definizione di un po' di alias di comodo, rinomine di comandi

.*bashrc* contiene delle specifiche che vengono eseguite all'avvio di una bash

regex matching with grep

- general syntax is ***grep [options] [pattern] [file]***
  - patterns are defined as regular expressions.
  - anchor characters: '^' and '$' at the beginning and end of the pattern
  - wildcard character: '.' is used to match any character
  - escaped characters: any special character can be matched as a regular character by escaping it with '\'
  - range: a set of characters listed in a '[' and ']' pair specify a range of characters to be matched
  - * the preceding item is matched zero or more times

regex matching with grep

- options
  - ***grep -i***: case insensitive
  - ***grep -A num ...***; ***grep -B num...***; matches ***num*** lines after/before the pattern; ***grep -C num...***; matches num lines before and after the pattern
  - ***grep -v ...:*** to invert match (that is: all lines that do not contain the searched pattern)
  - ***grep -c ...:*** counts the number of matches
  - ***grep -n ...***: displays the matched lines and line numbers
  - ***grep -w ...***: matches the whole word
  - ***grep -E ...***: pattern is interpreted as an extended regular expression

AWK

- da "Aho, Weinberger, & Kernighan"…
- Awk è una utility per semplici preprocessamenti di testo, ma è anche utile per compiti più articolati.

input: testo *formattato*

- immaginiamo di avere un file di testo, ***anagrafica.txt***, nel seguente formato:

```
nome,cognome,età,altezza,città,professione,sport
Rossi,Mario,57,174,Torino,architetto,calcio
Bianchi,Beppe,40,185,Milano,ingegnere,sci
Gialli,Alex,32,164,Torino,informatico,tennis
Verdi,Mario,38,179,Treviso,avvocato,squash
...
```

anagrafica.txt

```
id,nome,cognome,età,altezza,città,professione,sport
1,Rossi,Mario,57,174,Torino,architetto,calcio
2,Bianchi,Beppe,40,185,Milano,portinaio,sci
3,Gialli,Alex,32,164,Torino,informatico,calcio
4,Verdi,Mario,38,179,Treviso,avvocato,sci
...
```

anagrafica.txt

```
$ awk '/Mario/' anagrafica.txt

1,Rossi,Mario,57,174,Torino,architetto,calcio
4,Verdi,Mario,38,179,Treviso,avvocato,sci
```

- Awk cerca fra le linee del file quelle che contengono la stringa "Mario" e le stampa.
  - Altri comandi permettono di effettuare lo stesso filtraggio (vedere sul manuale il comando ***grep***)

```
id,nome,cognome,età,altezza,città,professione,sport
1,Rossi,Mario,57,174,Torino,architetto,calcio
2,Bianchi,Beppe,40,185,Milano,portinaio,sci
3,Gialli,Alex                                  rmatic
4,Verdi,Mario                                  ocato,
...                                                  ica.txt
```

Field Separator, di default è lo spazio

campo 1 e campo2

```
$ awk 'BEGIN {FS=","} /Mario/{print $1,$2}' anagrafica.txt

1 Rossi
4 Verdi
```

- Awk stampa i campi 1 e 2 delle linee del file ***anagrafica.txt*** contenenti la stringa "Mario".

awk <search pattern> {<program actions>}

```
$ awk 'BEGIN {FS= ","} /Mario/{print $1,$2}' anagrafica.txt
```

- generalizzando, awk funziona con questa sintassi:

awk <search pattern> {<program actions>}

  - awk ricerca all'interno del file in input le linee che contengono il pattern ricercato: per le linee trovate, esegue le azioni specificate.

```
id,nome,cognome,età,altezza,città,professione,sport
1,Rossi,Mario,57,174,Torino,architetto,calcio
2,Bianchi,Beppe,40,185,Milano,portinaio,sci
3,Gialli,Alex,32,164,Torino,informatico,calcio
4,Verdi,Mario,38,179,Treviso,avvocato,sci
...
```

anagrafica.txt

```
awk 'BEGIN {FS=","}
/Mario/ {eta += $4}
/Mario/ {print $1,$3,$4}
END {print "eta tot dei Mario = " eta}' anagrafica.txt

1 Mario 57
4 Mario 38
eta tot dei Mario = 95
```

- script analogo ai precedenti, in più calcola l'età totale delle persone di nome Mario

elaborazione di vettori di dati

- un altro esempio potrebbe essere quello di prendere in input un file contenente dati strutturati (separati da tabulazioni) in questo formato

```
BabelSynsetId  WikipediaPageTitle  synset1_weight1  synset2_weight2 …
```

  - il significato di questi campi non ha importanza (chi fosse interessato trova una descrizione all'URL https://goo.gl/ecmvHY)

- il primo elemento è un ID, seguito da un numero variabile di campi.
- il file originale contiene 2.8M di righe; quello da noi utilizzato solo 10K. il file in questione è presente fra il materiale della lezione.

task: conto dei campi

- il compito è quello, di riscrivere l'identificatore (il primo elemento di ogni linea) e il numero di campi presenti in ciascun record separati da tabulazione; idealmente l'output dovrebbe avere la forma

```
bn:00000001n    3
bn:00000002n    65
bn:00000003n    102
bn:00000004n    99
bn:00000005n    138
bn:00000006n    7
bn:00000007n    17
bn:00000008n    63
```

task: conto dei campi

- non si fornisce ad awk un criterio di selezione
- è necessario specificare che il separatore è la tabulazione
- il numero di campi presenti è restituito da NF

task: conto dei campi

- non si fornisce ad awk un criterio di selezione
- è necessario specificare che il separatore è la tabulazione
- il numero di campi presenti è restituito da NF

```
$ cat infile.txt | awk -F $'\t' '{ print $1 "\t" NF-1 }' > newfile
```

```
bn:00000001n    3
bn:00000002n    65
bn:00000003n    102
bn:00000004n    99
bn:00000005n    138
bn:00000006n    7
```