La programmazione di Shell

Shell

- E’ lo strato più esterno del sistema operativo
- Offre due vie di comunicazione con il SO
  - interattivo
  - shell script
- Script di shell
  - è un file (di testo) costituito da una sequenza di comandi
- La shell non è parte del kernel del SO, ma è un normale processo utente
  - Ciò permette di poter modificare agevolmente l’interfaccia verso il sistema operativo

Shell – Caratteristiche

- Espansione/completamento dei nomi dei file
- Ri-direzione dell'I/O (stdin, stdout, stderr)
- Pipeline dei comandi
- Editing e history dei comandi
- Aliasing
- Gestione dei processi (foreground, background sospensione e continuazione)
- Linguaggio di comandi
- Sostituzione delle variabili di shell

Le shell disponibili

- Bourne shell (sh)
  - La shell originaria, preferita nella programmazione sistemistica
- C-shell (csh)
  - La shell di Berkeley, ottima per l'uso interattivo e per gli script non di sistema
- Korn shell (ksh)
  - La Bourne sh riscrita dall'AT&T per assomigliare alla C-shell
- Tahoe (tcsh)
  - Dal progetto Tahoe, una C-shell migliorata

Le shell disponibili

- All’interno del corso useremo la *bash*
  - Bourne again shell (bash)
    - Tipica shell di Linux

http://www.linuxdoc.org/HOWTO/Bash-Prompt-HOWTO/index.html
http://www.linuxdoc.org/HOWTO/Bash-Prog-Intro-HOWTO.html

man bash

Le shell a confronto

Shell	Chi	Complessità relativa (in linee di codice)
sh	S.R. Bourne	1.00
csh	UCB	1.73
bash	GNU, LINUX	2.87
ksh	David Korn (AT&T)	3.19
tcsh	Tahoe	4.54

Esecuzione della shell

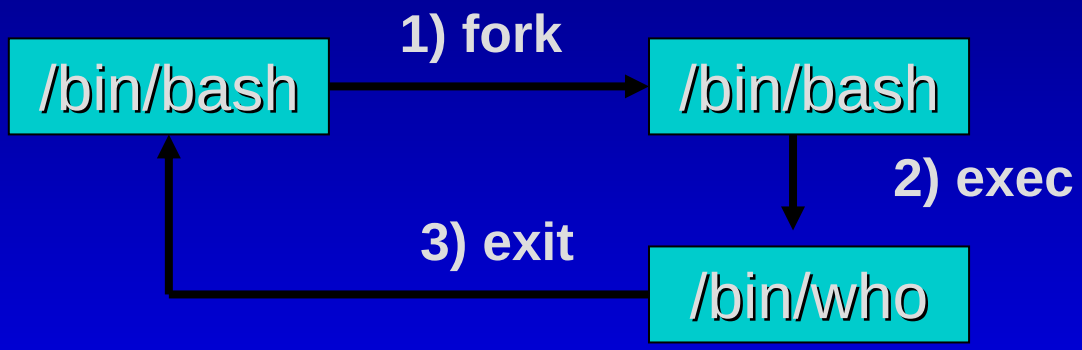
- **`/etc/passwd`** contiene info relative al login
  - tra cui quale programma viene automaticamente eseguito al login (in genere sempre una shell)
- Durante l'esecuzione, la shell cerca nella directory corrente, nell'ordine, i seguenti file di configurazione
  - **`.bash_profile`**
  - **`.bash_login`**
  - **`.profile`**
    - contengono i comandi che vengono eseguiti al login

Esecuzione della shell

- Se la shell non è di tipo “login” viene eseguito il file **`.bashrc`**
- Se non li trova, vengono usati quelli di sistema nella directory **`/etc`**
- E’ previsto anche un file **`.bash_logout`** che viene eseguito alla sconnessione

Funzionamento della shell

- Esempio: esecuzione del comando **who**



- System call coinvolte
  - fork()
    crea un nuovo processo (figlio) che esegue il medesimo codice del padre
  - exec()
    carica un nuovo codice nell'ambito del processo corrente
  - exit()
    termina il processo corrente

Bash – Variabili

- La shell mantiene un insieme di variabili per la personalizzazione dell'ambiente
- Assegnazione: **`variabile=valore`**
- Variabili di shell più importanti
  - PWD la directory corrente
  - PATH elenco di directory in cui cercare comandi
  - HOME directory di login
  - PS1, PS4 stringhe di prompt
    - (vedere PROMPTING su **`man bash`**)
- Le assegnazioni vengono in genere aggiunte all'interno del **`.bash_profile`**

Bash – Variabili

- Per accedere al valore di una variabile, si usa l'operatore **$**
  - Esempio: se **x** vale 123, si può usarne il valore tramite **$x**
- Per visualizzare il valore di una variabile, si usa il comando **echo**
- NOTA
  - I valori delle variabili sono sempre STRINGHE
  - Per valutazioni aritmetiche si può usare l'operatore **$(())**, oppure il comando **let**

Bash – Variabili

- Esempio

```
# x=0
# echo $x+1
0+1
# echo $((x+1))
1
# let “x+=1”
# echo $x
1
```

Bash – Storia dei comandi

- La bash mantiene una storia dei “comandi precedenti” dentro un buffer circolare memorizzato nel file indicato dalle variabili **`HISTFILE`** (default **`.bash_history`**)
- Utile per chiamare comandi o correggerli

Bash – Storia dei comandi

- Per accedere ai comandi
  - !n — esegue il comando n del buffer (potrebbe non esserci)
  - !! — esegue l'ultimo comando
  - !-n — esegue l'n-ultimo comando
  - !$ — l'ultimo parametro del comando precedente
  - !^ — il primo parametro del comando precedente
  - !* — tutti i parametri del comando precedente
  - !stringa — l'ultimo comando che inizia con stringa
  - ^stringa1^stringa2 — sostituisce stringa1 nell'ultimo comando con stringa 2

Bash – Storia dei comandi

- Esempio

```
#cc –g  prog.c
#vi  iop.c
#cc prog.c  iop.c
#a.out
```

- Dopo l'ultimo comando si ha

`#!$`	esegue a.out
`#!-1`	idem
`#!c`	esegue cc prog.c iop.c
`#!v`	esegue vi iop.c
`#rm !*`	esegue rm a.out
`#rm !$`	esegue rm a.out

Bash – Globbing

- Espansione dei nomi dei file (e comandi) con il tasto TAB (o ESC)
  - Per i nomi di file eseguibili la shell cerca nelle directory del PATH
  - Per i file generici, la shell espande i nomi di file nella directory corrente

Bash – Wildcard

- Caratteri speciali
  - **`/`** separa i nomi delle directory in un path
  - **`?`** un qualunque carattere (ma solo uno)
  - **`*`** una qualunque sequenza di caratteri
  - **`~`** la directory di login
  - **`~user`** la directory di login dello **`user`** specificato
  - **`[]`** un carattere tra quelli in parentesi
  - **`{,}`** una parola tra quelle in parentesi (separate da ,)
- Esempio

```
cp ~/.[azX]* ~/rap{1,2,20}.doc ~/man.wk? ~bos
```

Bash – Aliasing

- E’ possibile definire dei comandi con nuovi nomi (alias), tipicamente più semplici

  `alias`

  elenca gli alias definiti

  `alias nome=‘valore’`

  definisce un alias (no spazi prima/dopo =)

  `unalias nome`

  cancella un alias

- Esempio
  - **`alias ll=‘ls-l’`**

Bash – Ambiente

- Le variabili sono di norma locali alla shell
  - Occorre un meccanismo che consenta di passare i valori delle variabili ai processi creati dalla shell (in particolare alle sub-shell)
- L'ambiente della shell è una lista di coppie **`nome=valore`** trasmessa ad ogni processo creato

`export variabile [ = valore ]`

assegna un valore a una variabile di ambiente

`printenv [variabile]`

stampa il valore di una o tutte le variabili d'ambiente

`env`

stampa il valore di tutte le variabili d'ambiente

Bash – Variabili di Ambiente

- Le principali variabili d'ambiente

PWD	SHELL
PATH	HOME
HOST	HOSTTYPE
USER	GROUP
MAIL	MAILPATH
OSTYPE	MACHTYPE

- Alcune variabili di ambiente sono legate al valore delle corrispondenti variabili di shell (per es. PATH)

File di comandi (script)

- E’ possibile memorizzare in un file una serie di comandi, eseguibili richiamando il file stesso
- Esecuzione
  - Eseguendo **`bash script argomenti`** sulla linea di comando
  - Eseguendo direttamente **`script`**
    - E’ necessario che il file abbia il permesso di esecuzione, ossia, dopo averlo creato si esegue: **`chmod +rx file`**
    - Per convenzione, la prima riga del file inizia con **`#!`**, seguita dal nome dalla shell entro cui eseguire i comandi **`(#!/bin/bash)`**

Esempio

```
#!/bin/bash
date   #restituisce la data
who    #restituisce chi è connesso
```

Variabili speciali

- La bash memorizza gli argomenti della linea di comando dentro una serie di variabili
  `$1, ... $n`
- Alcune variabili speciali

`$$`	PID del processo shell
`$0`	Il programma corrispondente al processo corrente
`$#`	il numero di argomenti
`$?`	se esistono argomenti (no=0, si=1)
`$*`, **`$@`**	tutti gli argomenti

Variabili vettore

- Definizione
  - enumerando i valori tra parentesi tonde
- Accesso ai campi
  - con la notazione del C usando le parentesi quadre
  - La valutazione dell'espressione richiede gli operatori { }
- NOTA: gli indici partono da 0!

Variabili vettore

- Esempio

```
#v=(1 2 3)
#echo $v
1
#echo $v[1]
1[1]
#echo ${v[2]}
3
```

Bash – Input/Output

- Per stampare un valore su standard output
  `echo espressione`
- Nel caso si tratti di variabili, per stampare il valore, usare **`$`**
- Esempio

```
# X=1
# echo X
X
# echo $X
1
```

Bash – Input/Output

- Per acquisire un valore da standard input

```
read variabile
```

- Esempio

```
# read x
pippo
# echo $x
pippo
```

Bash – Strutture di controllo

- Strutture condizionali

```
if [ condizione ];
  then azioni;
fi
```

```
if [ condizione ];
  then azioni;
elif [condizione ];
  then azioni;
…
else
  azioni;
fi
```

Bash – Strutture di controllo

- Le **parentesi [ ]** che racchiudono la condizione sono in realtà un'abbreviazione del comando test, che può essere usato al loro posto

- Esempio

```
if [ a=0 ]; then # =senza spazi!
  echo $a;
fi

if test a=0; then  # =senza spazi!
  echo $a;
fi
```

Bash – Test e condizioni

- Per specificare condizione in un **`if`** è necessario conoscere il comando **`test`**

```
test operando1 operatore operando2
```

Bash – Test e condizioni

Operatori principali (**`man test`** per altri)

Operatore	Vero se …	# di operandi
-n	operando ha lunghezza ≠ 0	1
-z	operando ha lunghezza = 0	1
-d	esiste una directory con nome = operando	1
-f	esiste un file regolare con nome = operando	1
-e	esiste un file con nome = operando	1
-r, -w, -x	esiste un file leggibile/scrivibile/eseguibile	1
-eq, -neq	gli operandi sono interi e sono uguali/diversi	2
=, !=	gli operandi sono stringhe e sono uguali/diversi	2
-lt, -gt	operando1 <, > operando2	2
-le, -ge	operando1≤,≥ operando2	2

Bash – Strutture di controllo

```
case selettore
case1):         azioni;;
case2):         azioni;;
…
):              azioni;;
esac
```

Bash – Strutture di controllo

- Esempio

```
if [ -e “$HOME/.bash_profile” ]; then
    echo “you have a .bash_profile file”;
else
    echo “you have no .bash_profile file”;
fi
```

Bash – Strutture di controllo

- **Esempio**

```
echo “Hit a key, then hit return.”
read Keypress
case “$Keypress” in
    [a-z]) echo “Lowercase letter”;;
    [A-Z]) echo “Upper letter”;;
    [0-9]) echo “Digit”;;
    * ) echo “other”;;
esac
```

Bash – Strutture di controllo

- Ciclo for

```
for arg in [lista]
do
    comandi
done
```

- **`lista`** può essere
  - un elenco di valori
  - una variabile (corrispondente ad una lista di valori)
  - un meta-carattere che può espandersi in una lista di valori
- In assenza della clausola **`in`**, il **`for`** opera su **`$@`**, cioè la lista degli argomenti
- E' previsto anche un ciclo **`for`** che utilizza la stessa sintassi del **`for`** C/Java

Bash – Strutture di controllo

- Esempi

```
for file in *.c
do
 ls –l  “$file”
done
```

****

```
LIMIT=10
for ((a=1;a <= LIMIT; a++))
# Doppie parentesi e “LIMIT”senza “$”
do
 echo –n “$a “
done
```

Bash – Strutture di controllo

- Ciclo while

```
while [ condizione ]
do
  comandi
done
```

- La parte tra **[]** indica l'utilizzo del comando test (come per **if**)

- E' previsto anche un ciclo **while** che utilizza la stessa sintassi C/Java

Bash – Strutture di controllo

- Esempio

```
LIMIT=10
a=1
while [ $a –le $LIMIT ]
# oppure
while ((a <= LIMIT))
do
  echo –n “$a ”
  let a+=1
done
```

Bash – Strutture di controllo

- Ciclo until

```
until [ condizione vera ]
do
  comandi
done
```

- La parte tra **[]** indica l'utilizzo del comando test (come per **if**)

Bash – Strutture di controllo

- Esempio

```
LIMIT=10
a=1
until [ $a –gt $LIMIT ]
do
   echo –n “$a ”
   let a+=1 #oppure a=$(( a+1 ))
done
```

Bash – Funzioni

- E’ possibile usare sottoprogrammi (funzioni)
- Sintassi della definizione

```
function nome {
  comandi
}
```

- La funzione vede quali parametri $1, …$n, come fosse uno script indipendentemente dal resto
- Valore di ritorno tramite il comando **`return valore`**
- Codice di uscita tramite il comando **`exit(valore)`**

Bash – Funzioni

- Esempio

```
function quit {
exit
}

function e {
echo $1
}

e “Hello World”  #“main” dello script
quit
```

Bash – Funzioni

```
function func2 {
  if [ -z “$1” ] ; then
     echo “Parametro 1 ha lunghezza 0”;
  else
     echo “Parametro 1 e’ $1”;
  fi
     return 0
}
func2 “$1”
```

Bash – Uso output di un comando

- E' possibile utilizzare l'output di un comando come "dati" all'interno di un altro comando
- Tramite l'operatore "**‘ ‘**"
- Sintassi
  - **‘comando‘** (‘ = ALT+96 su tastiera italiana)
  - **$(comando)**
- Esempio
  - Cancellazione di tutti i file con il nome **test.log** contenuti nell'albero delle directory **/home/joe**

```
rm ‘find /home/joe –name test.log‘
```

Bash – Filtri

- Programmi che ricevono dati di ingresso da stdin e generano risultati su stdout
- Molto utili assieme alla ri-direzione dell'I/O
- Alcuni dei filtri più usati sono

```
more
sort
grep, fgrep, egrep
cut
head, tail
uniq
wc
awk (sed)
```

Bash – grep

- Per cercare se una stringa compare all'interno di un file

```
grep [-opzioni] pattern file
```

Opzioni

- **-c** conta le righe che contengono il pattern
- **-i** ignora la differenza maiuscolo/minuscolo
- **-l** elenca solo i nomi dei file contenenti il pattern
- **-n** indica il numero d'ordine delle righe
- **-v** considera solo righe che non contengono il pattern

Bash – Espressioni regolari

- I pattern di ricerca in grep possono essere normali stringhe di caratteri o espressioni regolari. In questo caso, alcuni caratteri hanno un significato speciale (a meno che siano preceduti da \)

`.`	un carattere qualunque
`^`	inizio riga
`$`	fine riga
`*`	ripetizione (zero o più volte)
`+`	ripetizione (una o più volte)
`[ ]`	un carattere tra quelli in parentesi
`[^]`	un carattere esclusi quelli in parentesi
`\<`	inizio parola
`\>`	fine parola

Bash – Varianti di `grep`

`fgrep [option] [string] [file] …`

- I pattern di ricerca sono stringhe
- E’ veloce e compatto

`egrep [option] [string] [file] …`

- I pattern di ricerca sono delle espressioni regolari estese
- E’ potente ma lento
- Richiede molta memoria

Bash – Ordinamento di dati

`sort [-opzioni] file …`

Opzioni

`-b`	ignora gli spazi iniziali
`-d`	(modo alfabetico) confronta solo lettere, cifre e spazi
`-f`	ignora la differenza maiuscolo/minuscolo
`-n`	(modo numerico) confronta le stringhe di cifre in modo numerico
`-o file`	scrive i dati da ordinare in **`file`**
`-r`	ordinamento inverso
`-t carattere`	usa **`carattere`** come separatore per i campi
`-k S1,S2`	usa i campi dalla posizione S1 alla S2

Bash – Selezione di Campi

cut –c*lista* file
cut –f*lista* [-dchar] [-s] file

- ***lista*** specifica un intervallo del tipo
  - **a,b** significa ‘**a**’ e ‘**b**’
  - **a-b** significa da ‘**a**’ a ‘**b**’

Bash – Selezione di Campi

- Opzioni
  - **`-c`** seleziona per caratteri
  - **`-f`** seleziona per campi
    Il campo è definito dal separatore
    (default carattere TAB)
  - **`-d`*`char`*** **`char`** è usato come separatore
  - **`-s`** considera solo le linee che contengono il separatore
- Esempi

```
cut –c1-12 file
```

prende i primi 12 caratteri di ogni riga del file

```
cut –c1, 4 file
```

prende il campo 1 e 4 di ogni riga del file

```
cut –f1-4 file
```

prende i primi 4 campi di ogni riga del file

Bash – Selezione di Campi

- Altri esempi

  `cut –d: -f1,5 /etc/passwd`

  Estrae user e nome completo degli utenti

  `ps –x | cut –d” “ –f1`

  Elenca i PID dei processi nel sistema

Bash – wc

`wc [-c] [-l] [-w] file`

Legge i file nell'ordine e conta il numero di caratteri, linee e parole

Opzioni

- **`-c`** conta solo i caratteri
- **`-l`** conta solo le righe
- **`-w`** conta solo le parole

- Esempio

  `ps -x | tail +2 | wc –l`

  Conta il numero di processi attivi (tail +2 per togliere l'intestazione)

Bash – uniq

uniq [-u][-c] file

- Trasferisce l'input sull'output sopprimendo duplicazioni contigue di righe
- Assume che l'input sia ordinato
- Opzioni
  - **-u** visualizza solo righe non ripetute
  - **-c** visualizza anche il contatore del numero di righe