Shell

Shell

- la shell è un programma che si frappone fra l'utente ed il sitema operativo permettendo, mediante una interfaccia testuale, di eseguire *comandi*
- esistono molti tipi di shell diverse, ma tutte offrono funzionalità comuni:
  - esecuzione di eseguibili e cmd. interni
  - redirezioni
  - wildcard
  - scripting
  - variabili
  - job control
  - gestione del path
  - raggruppamento di cmd.
  - pipeline
  - sostituzione di cmd.
- utilizzeremo la sintassi della shell più diffusa su sistemi GNU/Linux: la Bourne Again shell (bash)

Variabili

- una shell in esecuzione mantiene un insieme di variabili imperative di tipo stringa, è possibile:
  - assegnare valori a tali variabili (non è necessario dichiararle) ...

    ```
    nome="valore della variabile"
    ```
  - ... ed accedere al loro valore

    ```
    echo $nome
    ```
- esistono due tipi di variabili: *locali* e *d'ambiente*
  - le variabili d'ambiente vengono ereditate dai processi figli
  - le nuove variabili sono locali, ma possono divenire d'ambiente

    ```
    export nome
    ```

Eseguibili e comandi interni

- la shell permette di eseguire file eseguibili e comandi interni
  - file eseguibili
    - risiedono sul filesystem
      - vengono specificati con path assoluti o relativi

        `/sbin/ifconfig`

        `../ifconfig`
      - o cercati nei percorsi specificati nella variabile di ambiente $PATH

        `PATH=/usr/bin:/sbin`

        `ifconfig`
    - viene creato un nuovo processo per ognuno di essi, la shell attende che questo termini (o che liberi la console)

Eseguibili e comandi interni

- comandi interni
  - sono comandi implementati nella shell stessa
  - non risiedono sul filesystem
  - vengono eseguiti dallo stesso processo della shell
  - e.g. echo, cd, pwd
  - la loro documentazione è accedibile con il comando interno help
- sia per eseguibili che per comandi interni, la shell interpreta la linea di comando fornita dall'utente, creando l'array dei *parametri posizionali* (associato ad ogni processo)
  - e.g.
    - ls /etc /dev par.0 = “ls”, par.1 = “/etc”, par.2 = “/dev”
    - echo pippo par.0 = “echo”, par.1 = “pippo”

Comandi interni “notevoli”

- exec
  - per eseguire un file eseguibile, la shell crea un nuovo processo
  - il comando exec permette di eseguire un file eseguibile sostituendolo al processo corrente
  - e.g. confrontate i risultati di “ls” ed “exec ls”
- shift
  - essendo un processo, anche la shell dispone di un array di parametri posizionali con i quali è stata invocata
  - sono accedibili grazie alle variabili $0, $1, $2, ...
    - $* è l'indice dell'ultimo parametro
    - $@ corrisponde alla lista dei parametri
  - il comando shift effettua lo shift a sinistra di tutti i parametri il cui indice è ≥ 1, $1 viene perso

Metacaratteri

- l'interpretazione della riga di comando tratta in maniera particolare alcuni caratteri, detti metacaratteri:

  > < * ? | ( ) ; \ spazio tab cr lf

  - e.g. il carattere spazio separa gli argomenti a linea di comando per creare l'array corrispondente
- per inbire il comportamento particolare dei metecaratteri è necessario precederli con un \
  - questo procedimento è una delle forme possibili di *quoting*
- e.g.

  ls foo bar (1 argomento)

  vs

  ls foo\ bar (2 argomenti)

Redirezione

- ogni processo in esecuzione è associato ad un insieme di file aperti, 3 di essi sono predefiniti:
  - *standard input* (stdin, file descriptor 0)
    - è un file sola lettura, utilizzato per leggere input da tastiera
  - *standard output* (stdout, file descriptor 1)
  - *standard error* (stderr, file descriptor 2)
    - sono file sola scrittura, utilizzati per emettere output a video

- la shell permette di redirezionare sia i file di input che i file di output di un processo all'atto della sua esecuzione

Redirezione

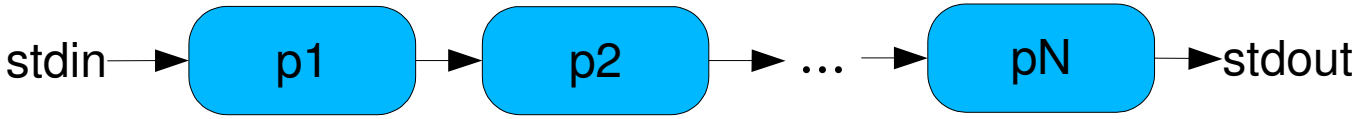
- salvataggio dell'output di processo su file
  - salvataggio di stdout: *comando > nome_file*
    - e.g. ls /etc > etc.txt
  - salvataggio di stderr: *comando 2> nome_file*
  - salvataggio di entrambi: *comando &> nome_file*
  - utilizzando “>>” al posto di “>” l'output del processo viene aggiunto al file destinazione anziché sostituito
  - oss: per buttare l'output di un processo possiamo redirezionarlo su /dev/null (e.g. startx &> /dev/null)
- redirezione dell'output su un altro file descriptor
  - forma generale: *comando src_fd>&dst_fd*
    - e.g. ls asflkgjhaslkfjg 2>&1

Redirezione

- lettura dell'input da file
  - *comando < nome_file*
    - e.g. mail zack@cs.unibo.it < testo.txt
    - e.g. sort < studenti.txt > studenti_ord.txt
- forme generali
  - apertura e ridirezione del file descriptor n, in sola scrittura
    - *comando n> target*
  - apertura e ridirezione del file descriptor n, in sola lettura
    - *comando n< target*
  - apertura e ridirezione del file descriptor n, in lettura/scrittura
    - *comando n<> target*

Pipeline

- è possibile far comunicare tra loro sequenze di processi, collegando stdout di uno di essi a stdin del successivo in una struttura chiamata *pipeline*



- sintassi: *comando1 | comando2 | .... | comandoN*
  - e.g. ls | wc
  - e.g. cat /etc/passwd | cut -f 1 -d : | sort
  - e.g. sort < studenti.txt | sort | uniq > studenti_ord_nodup.txt
  - e.g. ls -R | less

Raggruppamento di comandi

- sequenze di comandi:
  - sintassi: *comando1 ; comando 2 ; ... ; comandoN*
    - e.g. *date; ls; pwd*
  - i comandi vengono eseguiti sequenzialmente, attendendo la terminazione
- raggruppamento di comandi:
  - sintassi: *(sequenza_di_comandi)*
    - e.g. *(date; ls; pwd)*
  - per ogni gruppo viene eseguita una nuova shell che esegue la sequenza di comandi e termina

    date; ls; pwd > out.txt

    vs

    (date; ls; pwd) > out.txt

Sequenze condizionali

- return code
  - ogni processo, al momento della sua terminazione, restituisce un valore numerico (*return code*) intero
    - il valore 0 viene solitamente interpretato come successo, tutti gli altri come insuccesso
  - la shell ha accesso al valore di ritorno dell'ultimo processo eseguito (variabile $?)
- sequenze condizionali
  - *comando1 && comando2*
    - esegue *comando1*, se return code = 0 esegue *comando2*
  - *comando1 || comando2*
    - esegue *comando1*, se return code ≠ 0 esegue *comando2*
  - e.g. gcc myprog.c && ./a.out

Espansione delle wildcard

- la shell permette diverse forme di *espansione* nei comandi forniti dall'utente: utilizzandole è possibile scrivere comandi abbreviati
- la forma più semplice di espansione è l'*espansione delle wildcard* che permette di definire abbreviazioni che vengono espanse con l'aiuto del filesystem
  - il carattere * viene espanso con zero o più caratteri
  - il carattere ? viene espanso con esattamente un carattere
  - stringhe che contengono questi metacaratteri vengono espanse in una lista (separata da spazi) di file esistenti sul filesystem che rispettano le regole di * e ?
  - e.g. ls /etc/*.conf

Espansione dei comandi

- l'*espansione dei comandi* permette di sostituire una stringa (che rappresenti un comando) con il risultato dell'esecuzione della stessa
- sintassi
  - sintassi sh: *`comando`*
  - sintassi bash: *$(comando)* più comoda per l'annidamento
- e.g.
  - echo la data di oggi e' `date`
  - data=$(date)
  - echo nel sistema ci sono al momento `who | wc -l ` utenti
  - risultato=`expr 1 + 2 + 3 \* 7` (notate il quoting di *)

Altre espansioni

- la sintassi che abbiamo già visto per accedere ai valori della variabili non è altro che una forma di espansione: l'*espansione delle variabili*
  - e.g. echo $USER $HOME
- la shell bash offre molti altri tipi di espansione, tra i quali:
  - espansione aritmetica e.g. $((1+2*3*))*
  - brace expansion e.g. echo {a,b}{c,d}
  - tilde expansion e.g. cd ~szacchir/

Quoting

- a volte è necessario inibire una o più forme di espansione
- è possibile farlo utilizzando due nuove forme di quoting
  - single quote
    - utilizza virgolette singole '
    - tutto ciò che è racchiuso all'interno di ' ... ' non è suscettibile ad espansioni
    - e.g. echo '* `whoami` $HOME'
  - double quote
    - utilizza virgolette doppie “
    - tutto ciò che è racchiuso all'interno di “ ... “ è suscettibile ad espansioni di variabili e di comandi
    - e.g. echo “* `whoami` $HOME”

Riferimenti

- l'enciclopedica man page di bash: man bash
- Unix Power Tools, Jerry Peek et al., O'Reilly
- Unix Shell Programming, 3rd edition, Stephen Kochan et al.

Shell scripting

Shell scripting

- la shell è uno strumento potente per l'esecuzione di comandi arbitrariamente complessi
- ciò nonostante, per automatizzare l'esecuzione di compiti lunghi e ripetitivi, ogni shell fornisce un proprio linguaggio di programmazione *(shellscript)* che permette l'esecuzione di programmi *(script)* memorizzati in file eseguibili testuali
- caratteristiche comuni di questi linguaggi:
  - turing completi
  - approccio procedurale
  - non tipati (l'unico tipo disponibile è la stringa)
  - interpretati
- studieremo il linguaggio della shell Bourne Again (bash)

Script

- uno *script* (non necessariamente di shell) è un programma intelleggibile ad un interprete
  - quando viene richiesta l'esecuzione di un file eseguibile, la shell legge la prima riga del file, il file è considerato uno script se questa ha la forma **`#!`*`interpreter_path [arg ...]`***
  - per eseguire uno script la shell invoca l'interprete appendendo alla sua lista di parametri posizionali il nome dello script
  - e.g. “cat-script”:

```
#!/bin/cat
contenuto dello script
```

- un *bash script* è un programma interpretabile da bash, e.g.:

```
#!/bin/bash
echo “Hello, world!”
```

Input da stdin

- è possibile leggere da standard input utilizzando il comando interno “read”
- sintassi: *read <nome1> <nome2> ... <nomeN>*
- legge una riga da standard input, la spezza ai blank, ed associa ogni componente ad una delle variabili specificate
  - nel caso vi siano più componenti che variabili, l'ultima variabile contiene anche le rimanenti
- e.g., script inverti

```
#!/bin/bash
read nome cognome
echo “Dott. $cognome, $nome”
```

Variabili predefinite

- bash offre un vasto insieme di variabili locali e d'ambiente predefinite, ne riportiamo alcune:
- locali
  - $@, $*, $*n*, $# accesso ai parametri a riga di comando
  - $$ PID della shell
  - $- flag associati alla shell corrente
- di ambiente
  - $IFS Internal Field Separator
  - $PS1, $PS2 prompt

Matematica

- è possibile effettuare computazioni aritmetiche in due modi
  - utilizzando il comando esterno expr
    - facendo attenzione al quoting
  - utilizzando l'espansione aritmetica $((...)) della shell
- per effettuare computazioni matematiche in virgola mobile o reali è necessario fare ricorso a programmi esterni (e.g. bc)

Verifica di predicati booleani

- il comando interno test permette di valutare predicati e varia il suo valore di ritorno in base alla loro veridicità
- alcuni predicati
  - su file
    - esistenza di un file regolare — test -f *nome*
    - esistenza di una directory — test -d *nome*
    - leggibilità di un file — test -r *nome*
    - “più nuovo di” — test *file1* -nt *file2*
  - su stringhe
    - è la stringa vuota — test -z *stringa*
    - uguaglianza di stringhe — test *stringa1* = *stringa2*

Verifica di predicati booleani

- alcuni predicati
  - su numeri interi
    - sintassi generica test *num1 op num2*
    - op è uno tra: -lt (less than), -le (less or equal), -eq (equal), -ge (greater or equal), -gt (greater than)
  - connettivi logici:
    - and test *pred1* -a *pred2*
    - or test *pred1* -o *pred2*
    - not test ! *pred*
- sintassi concisa di test: [*predicato*]
  - e.g. test $x -ge 10 = [$x -ge 10]

Controllo condizionale

- la bash dispone dell'usuale costrutto if ... then ... else
  - la guardia è un comando, il ramo then viene eseguito se il suo return code è 0, altrimenti viene eseguito il ramo else
- la guardia è tipicamente (ma non necessariamente!) il comando test, nella sua sintassi concisa, e.g.:

```
echo “inserisci numero”
read number
if [ $number -lt 0 ]; then
  echo negative
elif [ $number -eq 0 ]; then
  echo zero
else
  echo positive
fi
```

Pattern matching

- è disponibile un costrutto di pattern matching su stringhe, i pattern sono simili a quelli disponibili per l'espansione delle wildcard, e.g.:

```
case "$1" in
start)
  start-stop-daemon --start ...
  ;;
reload | force-reload)
  ;;
*)
  exit 1
  ;;
esac
```

Cicli limitati

- è disponibile un costrutto di ciclo limitato che permette di ciclare su un insieme finito di stringhe, e.g.:

```
for dirname in /tmp /var/tmp; do
  rm -f $dirname/*~
done
```

- la semantica usuale del ciclo for delimitato da interi si ottiene utilizzando for in congiunzione con il programma seq e l'espansione dei comandi, e.g.:

```
test -d tests/ && exit 1
mkdir tests/
for i in `seq 1 100`; do
  echo "test_$i" > tests/$i.txt
done
```

Cicli illimitati

- è disponibile un costrutto di ciclo non limitato che permette di ciclare fintanto che l'esecuzione di un comando ha return code 0:

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

```
x=76
while [ $x -gt 57 ]; do
   x=`expr $x - 1`
done
```

- è disponibile anche la versione until che permette di ciclare fintanto che l'esecuzione di un comando ha return code diverso da 0:

```
until false; do
  echo "guru meditation"
done
```

Gestione dei segnali

- è possibile installare presso la shell in esecuzione dei comandi di callback che verranno eseguiti al verificarsi di eventi
  - sono eventi: le ricezioni di segnali e la terminazione della shell
- per installare callback si utilizza il comando interno trap
- e.g.:

```
set -e
tmp=`tempfile`
trap "rm $tmp" EXIT
...
```

Shell function

- è possibile definire funzioni (*shell functions*) utilizzando la sintassi: *nome-funzione* () { *comandi* }
- una volta definite, le funzioni possono essere invocate come normali comandi di shell, passando loro parametri posizionali
  - il loro corpo viene eseguito con un nuovo insieme di parametri posizionali

```
hello ()
{
  echo "Hello, $1 World!"
}
hello "Functions"
```

Shell scripting engineering

- per aumentare la mantenibilità di shell script di dimensioni considerevoli è possibile dividerli in più file
- utilizzando il comando . (punto) o source, è possibile interpretare il contenuto di script esterni
  - è quindi possibile organizzare gli script in librerie di funzioni invocabili esternamente

Debugging

- ad ogni shell è associato un insieme di flag, identificati da una singola lettera, modificabili attraverso il comando interno set con la sintassi
  - set +*flag* attiva un flag
  - set -*flag* disattiva un flag
- due flag sono di particolare utilità per il debugging di script
  - flag “e”, se settato impone alla shell di uscire non appena un comando ritorna un return code diverso da 0
  - flag “x”, se settato la shell stampa su stdout i comandi e i loro argomenti prima di eseguirli

Esercizio

- implementare, utilizzando bash script, una utility **junk**
- synopsis:
  - junk filename **...**
  - junk -l
  - junk -p
- quando invocata su una lista di file, sposta questi file in una directory “.junk”, nella home dell'utente (se la directory non esiste, la crea automaticamente)
- quando invocata con -l, mostra il contenuto di .junk/ (segnalando eventualmente che non esiste)
- quando invocata con -p, svuota la directory .junk (chiedendo conferma interattivamente all'utente)