

Processi

Cenni, vedi seconda parte del corso

Processi

- Ma cos'è un processo?
 - è un *programma in esecuzione completo del suo stato*
 - dati
 - heap
 - descrittori dei file
 - stack
 - etc ...
 - Lo definiremo più in dettaglio quando parleremo di SC relative ai processi

Processi (2)

- Ci sono comandi che permettono di avere informazioni sui processi attivi
 - centinaia di processi attivi su un sistema Unix/Linux

*-- ps permette di avere informazioni sui
-- processi attualmente in esecuzione*

```
bash:~$ ps
```

PID	TTY	TIME	CMD
2692	pts/3	00:00:00	bash
2699	pts/3	00:00:00	ps

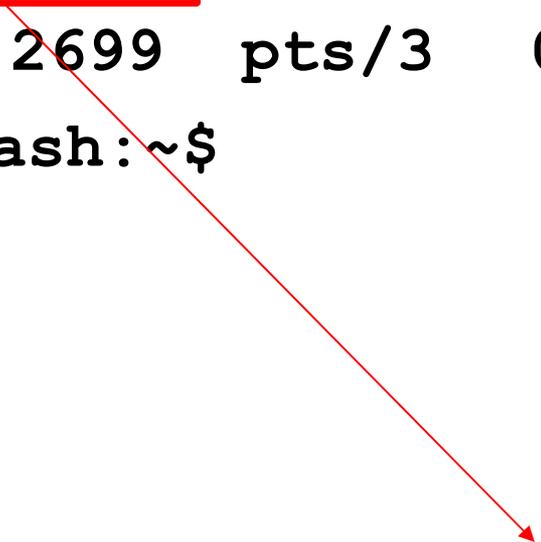
```
bash:~$
```

Processi (3)

```
bash:~$ ps
```

PID	TTY	TIME	CMD
2692	pts/3	00:00:00	bash
2699	pts/3	00:00:00	ps

```
bash:~$
```



*PID --Process identifier
intero che identifica univocamente il processo*

Processi (4)

```
bash:~$ ps
```

PID	TTY	TIME	CMD
2692	pts/3	00:00:00	bash
2699	pts/3	00:00:00	ps

```
bash:~$ ls -l /dev/pts/3
```

```
crw--w---- 1 susanna tty 136,3 ..... /dev/pts/3
```

```
bash:~$
```

*Dispositivo
a caratteri*

Terminale di controllo

*Major, minor number
(Driver, device)*

Processi (5)

```
bash:~$ ps
```

PID	TTY	TIME	CMD
2692	pts/3	00:00:00	bash
2699	pts/3	00:00:00	ps

```
bash:~$
```

*Tempo di CPU accumulato
(dd):hh:mm:ss*

*Nome del file
eseguibile*

Processi: più informazioni ...

```
bash:~$ ps -l
```

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
0	S	1002	2692	8760	0	75	0	-	1079	wait	pts/3	...	bash
0	R	1002	2699	2692	0	76	0	-	619	-	pts/3	...	ps

```
bash:~$
```

Status:

R -- running or runnable

S -- interruptable sleep

(wait for event to complete)

... molti di più

Processi: più informazioni ...(2)

```
bash:~$ ps -l
```

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
0	S	1002	2692	8760	0	75	0	-	1079	wait	pts/3	...	bash
0	R	1002	2699	2692	0	76	0	-	619	-	pts/3	...	ps

```
bash:~$
```

Status:

R -- running or runnable

S -- interruptable sleep

(wait for event to complete)

... molti di più

System call dove il processo è bloccato

Processi: più informazioni ... (3)

```
bash:~$ ps -l
```

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
0	S	1002	2692	8760	0	75	0	-	1079	wait	pts/3	...	bash
0	R	1002	2699	2692	0	76	0	-	619	-	pts/3	...	ps

```
bash:~$
```

Pid del padre

*Virtual size of process
text+data +stack*

Processi: più informazioni ... (4)

```
bash:~$ ps -l
```

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
0	S	1002	2692	8760	0	75	0	-	1079	wait	pts/3	...	bash
0	R	1002	2699	2692	0	76	0	-	619	-	pts/3	...	ps

```
bash:~$
```

Effective user id

%cpu time usato nell'ultimo minuto

Scheduling: Priorità, nice

Processi: ancora esempi ...

```
bash:~$ ps -ef
```

-- lista tutti i processi del sistema

Job control ...

Sospendere, riattivare terminare
processi appartenenti ad un gruppo
(*process groups*)

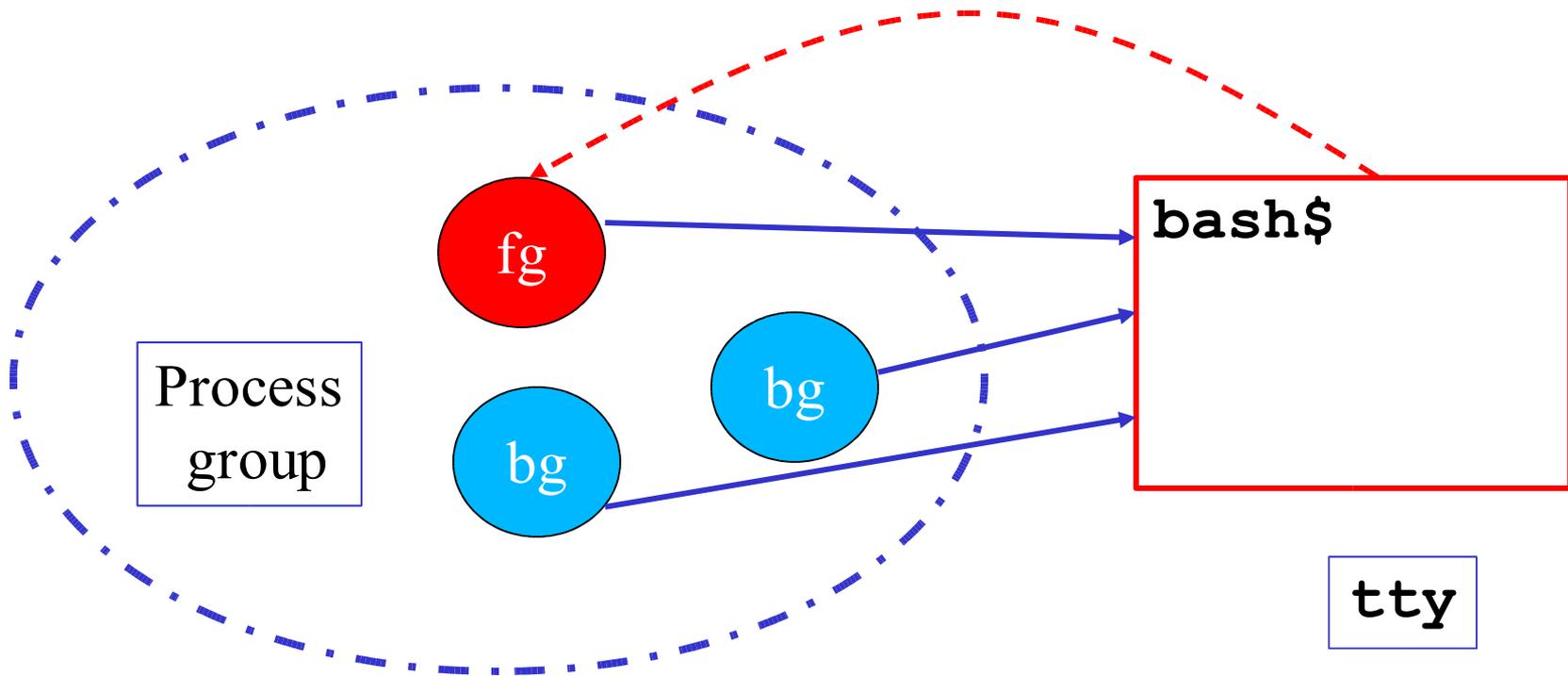
Process group

- Ogni processo Unix appartiene ad un *process group*
 - ogni gruppo è identificato dal *process_group_ID*
 - per ogni gruppo può esistere un *group leader*, quello il cui PID coincide con *process_group_ID*
 - il gruppo si eredita dal padre (dal processo che ci ha attivati)
 - ogni processo può uscire dal gruppo cui appartiene e crearne uno nuovo, *di cui è il leader*, il gruppo verrà popolato dai suoi stessi processi figli

Terminali e job control

- Ogni processo può avere ad un terminale di controllo
 - tipicamente ereditato da chi lo ha attivato
- I processi del gruppo condividono il TC
 - ad ogni istante un solo processo ha accesso al terminale in *input* (il processo in *foreground*)
 - gli altri possono scrivere, e l'output viene combinato
- Il *job control*
 - permette di sospendere, riattivare, terminare processi che appartengono allo stesso gruppo
 - ruotare fra i processi del gruppo l'accesso al terminale

Terminali e job control (2)



Esecuzione in background e foreground

Esecuzione in *background*

- La shell permette di eseguire più di un programma contemporaneamente durante una sessione
- sintassi:

command &

- il comando **command** viene eseguito in background
 - viene eseguito in una sottoshell, di cui la shell non attende la terminazione
 - si passa subito ad eseguire il comando successivo (es. in ambiente interattivo si mostra il prompt)
 - l'exit status è sempre 0
 - *stdin* non viene connesso alla tastiera (un tentativo di input provoca la sospensione del processo)

Esecuzione in *background* (2)

- Esempio:
 - processi pesanti con scarsa interazione con l'utente

```
bash:~$ sort <file_enorme >file_enorme.ord &
```

```
bash:~$ echo Eccomi!
```

```
Eccomi!
```

```
bash:~$
```

Controllo dei job

- Il builtin **jobs** fornisce la lista dei job in bg nella shell corrente

- un *job* è un insieme di processi correlati che vengono controllati come una singola unità per quanto riguarda l'accesso al terminale di controllo

- es.

```
bash:~$ ( sleep 40; echo done ) &
```

```
bash:~$ jobs
```

```
[1]  Running      emacs Lez3.tex &
```

```
[2]-  Running      emacs Lez3.tex &
```

```
[3]+  Running      ( sleep 40; echo done ) &
```

```
bash:~$
```

Controllo dei job (2)

- Il builtin `jobs`...

— es.

```
bash:~$ ( sleep 40; echo done ) &
```

```
bash:~$ jobs
```

```
[1] Running      emacs Lez3.tex &
```

```
[2]- Running      emacs Lez3.tex &
```

```
[3]+ Running      ( sleep 40; echo done ) &
```

```
bash:~$
```

*1 numero del job
diverso dal pid!!! Vedi ps*

+ job corrente

(spostato per ultimo da foreground a background)

Controllo dei job (3)

- Il builtin `jobs` ...

– es.

```
bash:~$ ( sleep 40; echo done ) &
```

```
bash:~$ jobs
```

```
[1]  Running      emacs Lez3.tex &
```

```
[2] -  Running      emacs Lez3.tex &
```

```
[3]+ Running      ( sleep 40; echo done ) &
```

```
bash:~$
```

*- penultimo job corrente
(penultimo job spostato da foreground a background)*

Controllo dei job (4)

```
bash:~$ ( sleep 40; echo done ) &
```

```
bash:~$ jobs
```

```
[1]  Running      emacs Lez3.tex &
```

```
[2]-  Running      emacs Lez3.tex &
```

```
[3]+  Running      ( sleep 40; echo done ) &
```

```
bash:~$
```

Stato:

Running -- in esecuzione

*Stopped -- sospeso in attesa di essere riportato
in azione*

Terminated -- ucciso da un segnale

Done -- Terminato con exit status 0

Exit -- Terminato con exit status diverso da 0

Controllo dei job (5)

```
bash:~$ ( sleep 40; echo done ) &
```

```
bash:~$ jobs -l
```

```
[1] 20647 Running      emacs Lez3.tex &
```

```
[2]- 20650 Running      emacs Lez3.tex &
```

```
[3]+ 20662 Running      (sleep 40; echo done) &
```

```
bash:~$
```



PID della corrispondente sottoshell

Terminare i job: **kill**

- Il builtin **kill**

```
kill [-l] [-signal] <lista processi o jobs>
```

- i processi sono indicati con il PID,
- i job da %numjob oppure altri modi (vedi man)
- consente di inviare un segnale a un job o un processo
- es.

-- lista dei segnali ammessi

```
bash:~$ kill -l
```

```
1) SIGHUP  2) SIGINT  ...
```

```
9) SIGKILL .....
```

```
bash:~$
```

Terminare i job: **kill** (2)

- i processi possono proteggersi da tutti i segnali eccetto SIGKILL (9)

```
bash:~$ jobs
```

```
[1]  Running      emacs Lez3.tex &
```

```
[2]-  Running      emacs Lez3.tex &
```

```
[3]+  Running      ( sleep 40; echo done ) &
```

```
bash:~$ kill -9 %3
```

```
[3]+  Killed ( sleep 40; echo done )
```

```
bash:~$
```

Sospendere e riattivare un job ...

- CTRL-Z sospende il job in foreground inviando un segnale SIGSTOP

```
bash:~$ sleep 40
```

```
^Z
```

```
bash:~$ jobs
```

```
[1]+  Stopped      sleep 40
```

-- riattiva il job corrente in background

```
bash:~$ bg
```

```
bash:~$ jobs
```

```
[1]+  Running      sleep 40
```

```
bash:~$
```

Sospendere e riattivare un job ... (2)

- CTRL-Z sospende il job in foreground inviando un segnale SIGSTOP

```
bash:~$ sleep 40
```

```
^Z
```

```
bash:~$ jobs
```

```
[1]+  Stopped      sleep 40
```

-- riattiva il job corrente in foreground

```
bash:~$ fg
```

..... *-- aspetta 40 sec in foreground*

```
bash:~$
```

Interrompere un job in foreground

- CTRL-C interrompe il job in foreground inviando un segnale SIGINT

```
bash:~$ sleep 40
```

```
^C
```

```
bash:~$ jobs      -- nessun job attivo
```

```
bash:~$
```

Operatori su stringhe

Minimale...

Accesso alle variabili

- Operatori disponibili:

`$<var>` o `${<var>}`

ritorna il valore di `<var>`

`${<var>:-<val>}`

se `<var>` esiste ed ha valore non vuoto, ritorna il suo valore,
altrimenti ritorna `<val>`

`${<var>:=<val>}`

se `<var>` esiste ed ha valore non vuoto, ritorna il suo valore,
altrimenti assegna `<val>` a `<var>` e ritorna `<val>`

`${<var>:?<message>}`

se `<var>` esiste ed ha valore non vuoto, ritorna il suo valore,
altrimenti stampa il nome e `<message>` su `stderr`

Accesso alle variabili (2)

- Operatori disponibili (cont.):

```
${<var>:+<val>}
```

se `<var>` esiste ed ha valore non vuoto, ritorna il valore `<val>`

- Esempi:

```
bash:~$ echo $PIPP0
```

-- variabile non definita

```
bash:~$ echo ${PIPP0:-ii}
```

```
ii
```

```
bash:~$ echo $PIPP0
```

-- ancora non definita

```
bash:~$
```

Accesso alle variabili (3)

- Esempi:

```
bash:~$ echo $PIPP0
```

```
iiii -- variabile definita
```

```
bash:~$ echo ${PIPP0:+kk}
```

```
kk
```

```
bash:~$ echo $PIPP0
```

```
iiii -- ancora stesso valore
```

```
bash:~$
```

Accesso alle variabili (4)

- Esempi:

```
bash:~$ echo $PIPP0
```

```
iiii -- variabile definita
```

```
bash:~$ echo ${PIPP0:+kk}
```

```
kk
```

```
bash:~$ echo $PIPP0
```

```
iiii -- stesso valore
```

```
bash:~$ echo ${PIPP0:=kkll}
```

```
iiii -- non modificata
```

```
bash:~$
```

Sottostringhe

```
${<var>:<offset>}
```

```
${<var>:<offset>:<length>}
```

ritorna la sottostringa di **<var>** che inizia in posizione **<offset>** (NOTA: il primo carattere è in posizione 0)

Nella seconda forma la sottostringa è lunga **<length>** caratteri. Esempio:

```
bash:~$ A=armadillo
```

```
bash:~$ echo ${A:5}
```

```
illo
```

```
bash:~$ echo ${A:5:2}
```

```
il
```

```
bash:~$
```

Lunghezza

```
${#<var>}
```

consente di ottenere la lunghezza (in caratteri) del valore della variabile `<var>` (NOTA: la lunghezza è comunque una stringa)

– Esempio:

```
bash:~$ A=armadillo
```

```
bash:~$ echo ${#A}
```

```
9
```

```
bash:~$ echo ${A:${#A}-4)}
```

```
illo
```

```
bash:~$ B=${A:3:3}
```

```
bash:~$ echo ${#B}          -- $B=adi
```

```
3
```

```
bash:~$
```

Pattern matching

- È possibile selezionare parti del valore di una variabile sulla base di un pattern (modello)
- I pattern possono contenere *,?, e [] e sono analoghi a quelli visti per l'espansione di percorso
- occorrenze iniziali

`${<var>#<pattern>}`

`${<var>##<pattern>}`

se `<pattern>` occorre all'inizio di `${<var>}` ritorna la stringa ottenuta eliminando da `${<var>}` la più corta / la più lunga occorrenza *iniziale* di `<pattern>`

Pattern matching (2)

- Occorrenze finali

`${<var>%<pattern>}`

`${<var>%%<pattern>}`

se `<pattern>` occorre alla fine di `${<var>}` ritorna la stringa ottenuta eliminando da `${<var>}` la più corta / la più lunga occorrenza *finale* di `<pattern>`

- esempi:

- `outfile=${infile%.pcx}.gif`

- rimuove l'eventuale estensione `.pcx` dal nome del file e ci aggiunge `.gif` (`pippo.pcx` → `pippo.gif`)

Pattern matching (3)

- Esempi (cont):

- `basename=${fullpath##*/}`

- rimuove dal `fullpath` il prefisso più lungo che termina con `'/'` (cioè estrae il nome del file dal path completo)

- `dirname=${fullpath%/*}`

- rimuove dal `fullpath` il suffisso più corto che inizia per `'/'` (cioè estrae il nome della directory dal path completo)

```
bash:~$ fullpath=/home/s/susanna/myfile.c
```

```
bash:~$ echo ${fullpath##*/}
```

```
myfile.c
```

```
bash:~$ echo ${fullpath%/*}
```

```
/home/s/susanna
```

Sostituzione di sottostringhe

- È possibile sostituire le occorrenze di un pattern nel valore di una variabile

`${<var>/<pattern>/<string>}`

`${<var>//<pattern>/<string>}`

- l'occorrenza più lunga di **pattern** in **var** è sostituita con **string**.
- La prima forma sostituisce solo la prima occorrenza, la seconda le sostituisce tutte
- se **string** è vuota le occorrenze incontrate sono eliminate
- se il primo carattere è **#** o **%** l'occorrenza deve trovarsi all'inizio o alla fine della variabile
- se **var** è ***** o **@** l'operazione è applicata ad ogni parametro posizionale, e viene ritornata la lista risultante

Sostituzione di sottostringhe (2)

- Esempi:

```
bash:~$ echo $A
```

```
unEsempioDiSostituzione
```

```
bash:~$ echo ${A/e/eee}
```

```
unEseeeeempioDiSostituzione
```

```
bash:~$ echo ${A//e/eee}
```

```
unEseeeeempioDiSostituzioneeee
```

```
bash:~$ echo ${A/%e/eee}
```

```
unEsempioDiSostituzioneeee
```

```
bash:~$ ${A/#*n/eee}
```

```
eeeEsempioDiSostituzione
```

```
bash:~$
```

Esempio: **pushd popd**

- **pushd e popd**

- sono dei builtin che implementano uno stack di directory

- pushd <dir>** cambia la working directory in **<dir>** e mette **<dir>** sullo stack

- popd** elimina la directory sul top dello stack e si sposta nella nuova directory top

```
bash:~$ pushd didattica
```

```
/home/s/susanna/didattica /home/s/susanna
```

```
bash:~/didattica$ popd
```

```
/home/s/susanna
```

```
bash:~$
```

Esempio: pushd popd (2)

- Discutiamo come implementarli come funzioni bash:

– una prima soluzione

#DIRSTACK variabile che implementa lo stack

```
function pushd ()
```

```
{
```

```
    DIRNAME=${1:? "missing directory name" }
```

```
    cd $DIRNAME &&
```

```
    DIRSTACK="$PWD ${DIRSTACK:-$OLDPWD} "
```

```
    echo $DIRSTACK
```

```
}
```

Esempio: pushd popd (3)

#DIRSTACK variabile che implementa lo stack

```
function popd ()
```

```
{
```

```
    DIRSTACK=${DIRSTACK#* }
```

```
    cd ${DIRSTACK%% *} 
```

```
    echo $PWD
```

```
}
```

Esempio: `pushd popd` (4)

- Problemi:
 - non gestisce o gestisce male errori come
 - ‘directory non esistente’
 - stack vuoto (nella `popd`)
 - non permette di trattare directory che hanno uno spazio nel nome
 - l’implementazione dei builtin veri è molto più complessa

Controllo del flusso

If, while etc...

Strutture di controllo

- Permettono di
 - condizionare l'esecuzione di porzioni di codice al verificarsi di certi eventi
 - eseguire ripetutamente alcune parti etc.
- Bash fornisce tutte le strutture di controllo tipiche dei programmi imperativi
 - vengono usate soprattutto negli script ma si possono usare anche nella linea di comando

Strutture di controllo (2)

– **if-then-else**

- esegue una lista di comandi se una condizione è / non è vera

– **for**

- ripete una lista di comandi un numero prefissato di volte

– **while, until**

- ripete una lista di comandi finchè una certa condizione è vera / falsa

– **case**

- esegue una lista di comandi scelta in base al valore di una variabile

– **select**

- permette all'utente di scegliere fra una lista di opzioni

Costrutto **if**

- esegue liste di comandi differenti, in funzione di condizioni espresse anch'esse da liste di comandi
- sintassi (usando ‘;’ come terminatore di lista di comandi)

```
if <condition>; then  
    <command-list>  
[elif <condition>; then  
    <command-list>] ...  
[else  
    <command-list>]  
fi
```

Costrutto **if** (2)

– sintassi (usando ‘newline’ come terminatore di lista)

```
if <condition>  
then  
    <command-list>  
[elif <condition>  
then  
    <command-list>] ...  
[else  
    <command-list>]  
fi
```

Costrutto **if** (3)

- Semantica:
 - esegue la lista di comandi `<condition>` che segue `if`
 - se l'*exit status* è 0 (*vero*) esegue `<command-list>` che segue `then` e termina
 - altrimenti esegue le condizioni degli `elif` in sequenza fino a trovarne una verificata
 - se nessuna condizione è verificata esegue la `<command-list>` che segue `else`, se esiste, e termina
 - l'*exit status* è quello dell'ultimo comando eseguito (0 se non ha eseguito niente)

Costrutto **if** (4)

- Uso tipico

- siccome 0 significa esecuzione non anomala:

```
if <esecuzione regolare del comando>; then
    <elaborazione normale>
else
    <gestione errore>
fi
```

Costrutto `if` : esempi

- Esempio: miglioriamo la `pushd`

```
function pushd ()
{
    DIRNAME=${1:? "missing directory name"}
    if cd $DIRNAME; then
        DIRSTACK="$PWD ${DIRSTACK:-$OLDPWD}"
        echo $DIRSTACK
    else
        echo "Error still in $PWD"
    fi
}
```

Costrutto **if** : esempi (2)

- Esempio: ridefiniamo **cd**

```
function cd ()
{
    builtin cd "$@"
    es=$?
    echo "cd: $OLDPWD --> $PWD"
    return $es
}
```

Costrutto **if** : esempi (3)

- Esempio: ridefiniamo `cd`

```
function cd ()  
{  
    builtin cd "$@"  
    es=$?  
    echo "cd: $OLDPWD --> $PWD"  
    return $es  
}
```

Richiede l'esecuzione di un builtin specifico (altrimenti chiamerebbe la funz. che stiamo definendo)

Da esplicitamente l'*exit status* del `cd`
Altrimenti è quello dell'ultimo comando eseguito (`echo`)

Condizione: combinare *exit status*

- **&&**, **||**, **!** (and, or, negazione) si possono usare per combinare gli exit status nelle condizioni
- Es: verifichiamo che un file contenga una di due parole date:

```
file=$1; wrd1=$2; wrd2=$3;  
if grep $wrd1 $file || grep $wrd2 $file; then  
    echo "$wrd1 o $wrd2 sono in $file"  
fi
```

- analogamente se ci sono entrambe ...

Test

- La condizione dell'**if** è un comando (possibilmente composto) ma questo non significa che si può testare solo se un comando completa la sua esecuzione
- con la seguente sintassi

```
test <condition> oppure [ <condition> ]
```

- si può controllare:
 - proprietà dei file (presenza, assenza, permessi...)
 - confronti tra stringhe e interi
 - combinazioni logiche di condizioni

Test - stringhe

- Alcuni confronti fra stringhe:
 - con la condizione di verità

<code>str1 = str2</code>	<i>se str1 e str2 sono uguali</i>
<code>str1 != str2</code>	<i>se str1 e str2 sono diverse</i>
<code>-n str1</code>	<i>se str1 non è nulla</i>
<code>-z str1</code>	<i>se str1 è nulla</i>

Test - stringhe: esempio

- Miglioriamo la `popd ()` gestendo lo stack vuoto

```
function popd ()
{
    DIRSTACK=${DIRSTACK#* }
    if [-n "$DIRSTACK" ]; then
        cd =${DIRSTACK%% *}
        echo $PWD
    else
        echo "Stack empty still in $PWD"
    fi
}
```

Test - attributi file

- `-e file` *se file esiste*
- `-d file` *se file esiste ed è directory*
- `-f file` *se file esiste e non è speciale
(dir, dev)*
- `-s file` *se file esiste e non è vuoto*
- `-x -r -w file` *controlla diritti
esecuzione, lettura e scrittura*
- `-O file` *se sei l'owner del file*
- `-G file` *se un tuo gruppo è gruppo di file*
- `file1 -nt file2`
- `file1 -ot file2`
*se file1 è più nuovo (vecchio) di file2
(data ultima modifica)*

Esempio: script `mygunzip`

```
#!/bin/bash
file=$1
if [ -z "$file" ] || ! [ -e "$file" ] ; then
    echo "Usage: mygunzip filename"
    exit 1
else
    ext=${filename##*.} #determina il suffisso
    if ! [ $ext = gz ] ; then
        mv $file $file.gz
        $file=$file.gz #se non è gz lo aggiunge
    fi
    gunzip $file
fi
```

Test - Operatori logici

- Diverse condizioni su stringhe e file possono essere combinate all'interno di un test tramite gli *operatori logici* :

`-a (and) -o (or) ! (not)`

- All'interno di una condizione (`test` o `[...]`) la sintassi è

`\(expr1 \) -a \(expr2 \)`

`\(expr1 \) -o \(expr2 \)`

`! expr1`

Operatori logici : esempi

- Miglioriamo ancora la pushd

```
function pushd ()
{ DIRNAME=$1
if [ -n "$DIRNAME" ] &&
  [ \( -d "$DIRNAME" \) -a
    \( -x "$DIRNAME" \) ]; then
  cd DIRNAME;
  DIRSTACK="$PWD ${DIRSTACK:-$OLDPWD}"
  echo $DIRSTACK
else
  echo "Error still in $PWD"; return 1
fi }
```

Test - Interi

- Si possono effettuare test su stringhe interpretate come valori interi
 - lt (minore)
 - gt (maggiore)
 - le (min uguale)
 - ge (maggiore uguale)
 - eq (uguale)
 - ne (diverso)
- Attenzione, questi test sono utili solo per mescolare test su stringhe e interi, altrimenti `$ ((<cond>))` è più efficiente ed espressivo
 - include =, <, >, >=, <=, ==, !=, &&, ||

Costrutto **for**

- Permette di eseguire un blocco di istruzioni un numero prefissato di volte. Una variabile, detta *variabile di loop*, assume un valore diverso ad ogni iterazione
- diversamente dai costrutti **for** dei linguaggi convenzionali non permette di specificare *quante* iterazioni fare, ma una *lista di valori assunti dalla variabile di loop*. Sintassi

```
for <var> [ in <list> ]; do  
    <command-list>  
done
```

- se <list> è omessa si assume la lista degli argomenti dello script (\$@)

Costrutto `for` (2)

- Semantica:
 - Espande l'elenco `<list>` generando la lista degli elementi
 - Esegue una scansione degli elementi nella lista (separatore il primo carattere in `$IFS`)
 - Alla variabile `<var>` ad ogni iterazione viene assegnato un nuovo elemento della lista e quindi si esegue il blocco `<command-list>` (che tipicamente riferisce la variabile di loop)
 - L'*exit status* è quello dell'ultimo comando eseguito all'interno della lista `do` oppure 0 se nessun comando è stato eseguito

Costrutto **for** : esempi

- Es.

```
change <old> <new>
```

- ridenomina ogni file con suffisso ``.<new>`` nella directory corrente sostituendo il suffisso con ``.<old>``

```
bash:~$ ls
```

```
h.txt g.fig r.txt
```

```
bash:~$ change txt txtnew
```

```
bash:~$ ls
```

```
h.txtnew g.fig r.txtnew
```

```
bash:~$
```

Costrutto **for** : esempi (2)

```
#!/bin/bash
OLD=$1
NEW=$1
for FILE in *.$OLD ; do
    mv $FILE ${FILE%$OLD}.$NEW
done
```

Costrutto **for** : esempi (3)

- Es. funzione **mylsR**

- si comporta come **ls -R**
- discende ricorsivamente le directory fornite come argomento evidenziandone la struttura

```
bash:~$ mylsR
```

```
dir1
```

```
  subdir1
```

```
    file1
```

```
    ...
```

```
  subdir2
```

```
    subdir3
```

```
      file2
```

```
...
```

Costrutto **for** : esempi (4)

- Vediamo prima una funzione analoga a `ls -R` senza strutturazione

```
function tracedir () {  
  for file in "$@" ; do  
    echo $file  
    if [ -d "$file" ]; then  
      cd $file  
      tracedir $(command ls)  
      cd ..  
    fi  
  done
```

Costrutto **for** : esempi (5)

È ricorsiva!!!!

```
function tracedir () {  
  for file in "$@" ; do  
    echo $file  
    if [ -d "$file" ]; then  
      cd $file  
      tracedir $(command ls)  
      cd ..  
    fi  
  done
```

Esegue il comando ls e non
eventuali funzioni

Costrutto **for** : esempi (6)

```
function mylsR () {  
    singletab= "\t"  
    for file in "$@" ; do  
        echo -e $tab$file  
        if [ -d "$file" ]; then  
            cd $file  
            tab=$tab$singletab  
            mylsR $(command ls)  
            cd ..  
            tab=${tab%"\t"}  
        fi  
    done
```

Costrutto **for** : esempi (7)

```
function mylsR () {  
  singletab= "\t"  
  for file in "$@" ; do  
    echo -e $tab$file  
    if [ -d "$file" ]; then  
      cd $file  
      tab=$tab$singletab  
      mylsR $(command ls)  
      cd ..  
      tab=${tab%"\t"}  
    fi  
  done
```

Con **-e** echo interpreta **\t** come un carattere di escape TAB

Costrutto case

- Permette di confrontare una stringa con una lista di pattern, e di eseguire di conseguenza diversi blocchi di istruzioni (simile a switch in C, Java)
- Sintassi:

```
case <expr> in
    <pattern> )
        <command-list> ;;
    <pattern> )
        <command-list> ;;
...
esac
```

Costrutto case (2)

- Semantica:
 - L'espressione **<expr>** in genere una variabile viene espansa e poi confrontata con ognuno dei **<pattern>**
 - stesse regole dell'espansione di percorso
 - il confronto avviene in sequenza
 - Se un pattern viene verificato si esegue la lista di comandi corrispondente e si esce
 - Ogni pattern può in realtà essere l'or di più pattern
<pattern1> | ... | <patternN>
 - L'exit status è quello dell'ultimo comando eseguito oppure 0 se nessun comando è stato eseguito

Costrutto **case** : esempio

- La funzione

cd old new

- che con 1 o 0 parametri si comporta come il builtin **cd**
- mentre con 2 parametri cerca nel pathname della directory corrente la stringa **old**, se la trova la sostituisce con **new** e cerca di spostarsi nella directory corrispondente

Costrutto **case** : esempio (2)

```
function cd () {
  case "$#" in
    0|1 ) builtin cd $1;;
    2   ) newdir="{PWD//$1/$2}"
          case "$newdir" in
            $PWD) echo "bash; cd; bad \
                    substitution" 1>&2
                  return 1;;
            *   ) echo "bash; cd; too many arg " 1>&2
                  return 1;;
          esac
  esac
}
```

Costrutto `select`

- Permette di generare un menu e gestire la scelta da tastiera dell'utente

- Sintassi:

```
select <expr> [ in <list> ]; do
```

```
  <command-list>
```

```
done
```

- **Semantica:**

- il comando `<list>` viene espanso generando una lista di elementi (se è assente si usa "\$@")

Costrutto `select` (2)

- Semantica (cont):

- ogni elemento della lista viene proposto sullo standard error (ognuno preceduto da un numero) .
 - Quindi viene mostrato il prompt di `$PS3` (di default `$`) e chiesto il numero all'utente
- la scelta fatta viene memorizzata nella variabile **REPLY** e l'elemento corrispondente della lista in `<var>`
- con una scelta non valida il menu viene riproposto
- se è valida si esegue `<command-list>` e si ripete tutto
- si esce con il builtin **break**
- L'exit status è quello dell'ultimo comando eseguito oppure 0 se nessun comando è stato eseguito

Costrutto **select** : esempio

- La funzione

icd

- che elenca le directory presenti in quella corrente
- e a scelta dell'utente si sposta in una di queste

Costrutto `select`: esempio (2)

```
function icd () {
PS3="Scelta?"
select dest in $(command ls -aF | grep "/"); do
    if [ $dest ]; then
        cd $dest
        echo "bash; icd; Changed to $dest"
        break
    else
        echo "bash; icd; wrong choice"
    fi
done
}
```

Costrutti **while** **until**

- Permettono di ripetere l'esecuzione di un blocco di istruzioni fino al verificarsi (**while**) o al falsificarsi (**until**) di una condizione

- Sintassi:

```
while <condition>; do           until <condition>; do  
    <command-list>                <command-list>  
done                             done
```

- la condizione **<condition>** è analoga a quella dell'**if**
- al solito vera (0), falsa (**!=0**)
- L'exit status è quello dell'ultimo comando di **<command-list>** oppure 0 se non si entra nel ciclo

while until : esempio

- Uno script

`printpath`

- che stampa le directory del path una per riga
- numerandoli con interi a partire da 0
- NB: ricordarsi che il separatore in **PATH** è ‘:’

while until : esempio (2)

```
#!/bin/bash
path=${PATH%:}
declare -i I=0      # var intera
path=${path#:}
echo "Le directory nel path sono:"
while [ $path ]; do
    echo "  $I ) ${path%%:*}"
    path=${path#:}
    I=$((I + 1))
done
```

C'è molto di più....

- Si possono trattare opzioni sulla riga di comando (builtin **shift**, **getopts**)
- si possono modificare gli attributi delle variabili (comando **declare ...**)
- è possibile definire array
- è possibile leggere l'input dell'utente (builtin **read**)
- è possibile costruire comandi all'interno dello script ed eseguirli (comando **eval**)
- e molto altro ...
-

Consigli per il debugging ...

Prima di tutto ...

- **ATTENZIONE:**

- gli script possono essere pericolosi, proteggete file e directory ed eseguiteli in ambienti non danneggiabili finchè non siete ragionevolmente sicuri della loro correttezza!
- Attenzione a lasciare gli spazi dove servono ed agli effetti delle espansioni!

Opzioni per il debugging

- Alcune opzioni utili per il debugging:
 - settabili con comando **set** [-/+o]
 - **noexec -n** : non esegue, verifica solo la correttezza sintattica
 - **verbose -v** : stampa ogni comando prima di eseguirlo

Opzioni per il debugging (2)

- Alcune opzioni utili per il debugging (cont):

- **xtrace -x** : mostra il risultato dell'espansione prima di eseguire il comando

```
bash:~$ ls *.c
```

```
pippo.c pluto.c
```

```
bash:~$ set -x
```

```
bash:~$ ls *.c
```

```
+ ls -F pippo.c pluto.c
```

```
pippo.c pluto.c
```

```
bash:~$
```

Un debugger per bash

- Scaricabile dal sito del corso
 - Dall libro *learning the bash shell* ..
 - Sommario dei comandi principali in linea ...