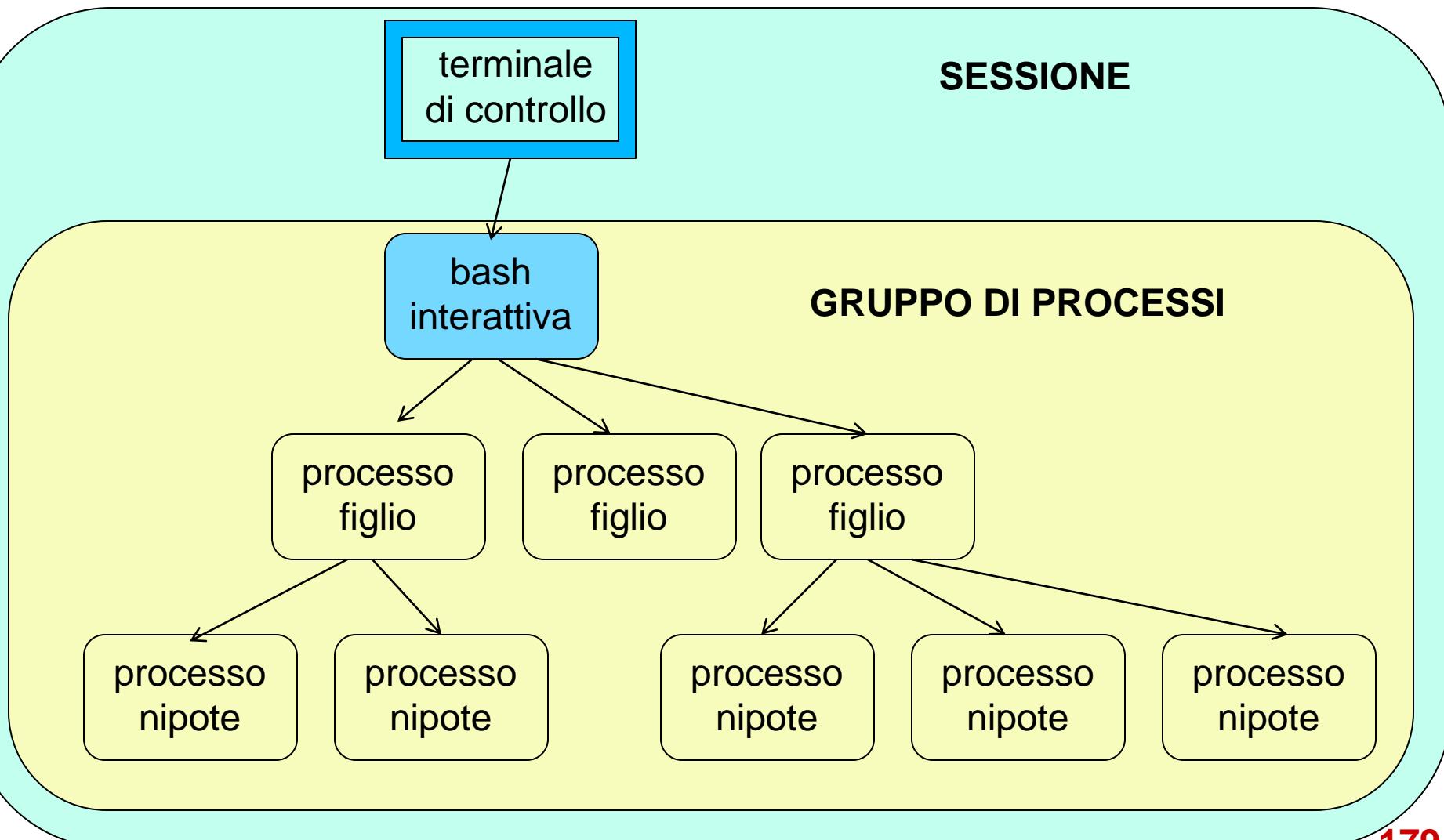


Concetti di Processo, Sessione, Gruppo di Processi, Terminale di controllo del gruppo di processi (0)

Scenario



Concetti di Processo, Gruppo di Processi, Terminale di controllo del gruppo di processi (1)

- **Processi:** Un processo è la più piccola unità di elaborazione completa ed autonoma che può essere eseguita in un computer.
- Un **processo** è un insieme di thread di esecuzione operanti all'interno di un contesto, il quale comprende uno spazio di indirizzamento in memoria ed una tabella dei descrittori di file aperti per quel processo.
- **Gruppo di processi** (process group): Un processo può lanciare l'esecuzione di altri processi: questi altri processi appartengono allo stesso gruppo di processi del processo padre, a meno che non si svolgano azioni che modificano il gruppo di appartenenza.
- **Terminale di controllo:** Un processo lanciato in esecuzione può avere un "controlling terminal" (un terminale da cui è controllato, che è l'astrazione di terminale (console=video+tastiera) da cui prende gli standard input output ed error).
- **Terminale di controllo del gruppo di processi** Un processo lanciato in esecuzione eredita lo stesso "controlling terminal" dal padre che lo ha lanciato, a meno che non si svolgano azioni che sganciano il processo dal terminale di controllo. Quindi tutti i processi di uno stesso gruppo di processi condividono lo stesso terminale di controllo.

Creazione della Sessione

- Per gestire l'insieme di processi, il kernel necessita di raggruppare i processi in modo più complesso che non la semplice relazione padre-figlio.
- Perciò viene usata l'astrazione di "**sessione**" e di "**gruppo di processi**".
- Un utente normalmente apre una **shell interattiva** (di login o no) che viene inserita dal sistema operativo in un terminale (la window che rappresenta la console monitorvideo+tastiera) e da questa shell eseguirà operazioni varie, tra cui l'esecuzione di comandi, eseguibili binari e script.
- Il terminale in cui la shell interattiva esegue è detto "**terminale di controllo**" (controlling terminal).
- La shell interattiva all'inizio **crea una sessione**, specificando un nuovo sessionId, e **lega la sessione al terminale di controllo**. Questa shell interattiva diventa così il leader della sessione.
- I processi discendenti di quella shell apparterranno alla stessa sessione, a meno che non si eseguano operazioni di distacco.
- La shell interattiva prende come stdin stdout ed stderr quelli che il terminale gli fornisce.
 - I comandi, eseguibili binari e script lanciati dalla shell interattiva ereditano i file aperti del padre, quindi ereditano anche gli stdin, stdout ed stderr del terminale.

Motivazione della Sessione

- Quando un utente si disconnette da un sistema, il kernel deve terminare tutti i processi che l'utente sta ancora eseguendo
 - altrimenti, gli utenti lascerebbero un gran numero di vecchi processi in attesa di input che non potranno mai arrivare.
- Come determinare quali processi terminare?
- Per semplificare questa attività, i processi sono organizzati in sessioni.
- Il leader della sessione è il processo che ha creato la sessione.
- L'ID della sessione è uguale al pid del processo che ha creato la sessione tramite la chiamata di sistema setsid ().
 - La funzione setsid () non accetta argomenti e restituisce il nuovo ID di sessione., cioè il PID del processo leader.
- Tutti i discendenti di quel processo sono quindi membri di quella sessione a meno che non si rimuovano specificamente da essa.

Terminale di controllo (Controlling Terminal) di una sessione

- Ogni sessione è legata a un terminale da cui i processi nella sessione ricevono il loro input e ai quali inviano il loro output.
- Quel terminale può essere la console locale della macchina, un terminale connesso su una linea seriale o uno pseudo terminale che si collega a una finestra del window manager locale o attraverso una rete.
- Il terminale a cui è correlata una sessione è chiamato terminale di controllo (o control tty) della sessione.
- **Un terminale può essere il terminale di controllo per una sola sessione alla volta.**
- Sebbene il terminale di controllo per una sessione possa essere modificato, solitamente l'operazione di stabilire il terminale di controllo di una sessione viene eseguita solo dal processo che gestiscono l'accesso iniziale di un utente a un sistema o ad una finestra del window manager.

Chiusura del Terminale e conseguente Terminazione dei processi della sessione

- Quando viene chiuso il terminale di una sessione, i processi che appartengono a quella sessione, non possono più continuare l'esecuzione e vengono terminati.
- L'operazione di terminazione viene realizzata dal sistema operativo mandando ad ogni processo un segnale di terminazione detto SIGHUP ("signal hang up").
- Ricevendo questo segnale, ogni processo termina.
- In tal modo si liberano le risorse del sistema operativo legate ai processi legati alla sessione terminata a causa della chiusura (voluta o inaspettata) del terminale di controllo della sessione.
- Nota Bene: un processo può essere configurato, da programma, per reagire diversamente all'arrivo di un segnale.

Processi in background e in foreground (1)

• Processi in **foreground**

- Processi che “controllano” il terminale dal quale sono stati lanciati, nel senso che hanno il loro standard input collegato al terminale e impegnano il terminale non permettendo l'esecuzione di altri programmi collegati a quel terminale, fino alla loro terminazione
- In ogni istante, un solo processo è in foreground

• Processi in **background**

- Vengono eseguiti in parallelo rispetto all'esecuzione della bash, nel senso che permettono alla bash che li ha lanciati di leggere ed eseguire altri comandi e di lanciare altri programmi. Usano una copia dei fd della bash che li ha lanciati, quindi condividono con questa stdin/stdout/stderr. Poiché lo standard input rimane collegato al terminale della bash che li ha lanciati, quando terminiamo l'esecuzione della finestra del terminale, il terminale chiude lo stdin ed i processi vengono terminati. Per far proseguire l'esecuzione dei processi in background dopo la terminazione del terminale occorre sganciare i processi dal terminale col comando disown (vedi piu' avanti).

• **Jobs** di una shell

- Processi in background o sospesi **solo figli** di quella shell (vedi jobs).

• Job control

- Permette di portare i processi da background a foreground e viceversa

Segnali - kill (1)

Interruzioni software mandati ai processi per notificare eventi asincroni)

Un segnale è una interruzione software, provocata in un processo destinazione, per notificare un evento asincrono.

Il comando `kill` invia segnali a processi.

Il tipo di segnale viene specificato con un simbolo oppure con il valore numerico.

Il processi destinazione vengono specificati mediante il loro pid.

Supponiamo che 7777 sia il pid di un processo.

I seguenti comandi inviano il segnale SIGTERM al processo destinazione, con ciò ordinandogli di terminare. Il segnale viene accettato solo se proviene dall'utente che esegue il processo (effective user). Il processo può intercettare il segnale e gestirlo diversamente o addirittura ignorarlo. Come si vede, se non si specifica un segnale, per default viene inviato il segnale SIGTERM (valore 15) di terminazione del processo.

<code>kill</code>	7777
<code>kill</code>	-15 7777
<code>kill</code>	-SIGTERM 7777

Esiste un altro segnale di terminazione del processo, SIGKILL o KILL che però il processo non può ignorare.

<code>kill</code>	-9 7777
<code>kill</code>	-SIGKILL 7777
<code>kill</code>	-KILL 7777

E' possibile inviare un segnale di terminazione a tutti i processi che il processo bash corrente può far terminare (cioè tutti i suoi figli diretti, non i nipoti)

<code>kill</code>	-9 -1
-------------------	--------------

Segnali - kill (2)

Interruzioni software mandati ai processi per notificare eventi asincroni)

Altri possibili segnali sono:

kill	-SIGTSTP	pid	(è il segnale lanciato con CTRL Z)
kill	-SIGCONT	pid	(riprende processo sospeso, usato da fg e bg)
kill	-SIGINT	pid	(è il segnale lanciato con CTRL C)

ed altri ancora.

E' possibile visualizzare l'elenco dei segnali gestiti eseguendo il comando

```
kill      -l
```

NOTA BENE: RISPOSTA DILAZIONATA AI SEGNALI.

Se un processo è bloccato in attesa di input da rete oppure fermo in una sleep (cioè in stato waiting), riceverà e gestirà il segnale solo al termine dell'I/O o della sleep.

Segnali - trap (3)

Ricezione di signal ed esecuzione di azione

Il comando trap definisce l'azione da svolgere al ricevimento di un elenco di segnali, esplicitando il nome di una funzione oppure una stringa di codice bash da eseguire.

trap action lista_di_segnali

ESEMPIO: eseguire così: ./lancia_riceviSIGUSR1e2.sh

riceviSIGUSR1e2.sh

```
#!/bin/bash
trap "echo \"ricevuto SIGUSR1 !!! Termino !!!\"; exit 99" SIGUSR1

ricevutoSIGUSR2() {
    echo "ricevuto SIGUSR2, continuo";
}
trap ricevutoSIGUSR2 SIGUSR2

while true ; do echo -n "."; sleep 1; done
```

lancia_riceviSIGUSR1e2.sh

```
#!/bin/bash
./riceviSIGUSR1e2.sh &
CHILDPID=$!
sleep 5 ; echo "mando SIGUSR2": kill -s SIGUSR2 ${CHILDPID}
sleep 5 ; echo "mando SIGUSR2": kill -s SIGUSR2 ${CHILDPID}
sleep 5; echo "mando SIGUSR1"; kill -s SIGUSR1 ${CHILDPID}
sleep 4;
```

Segnali - trap (4)

La configurazione dell'azione si mantiene-
Annullare l'esecuzione dell'azione

riceviSIGUSR1e2.sh

```
#!/bin/bash
trap "echo \"ricevuto SIGUSR1 !!! Termino !!!\"; exit 99" SIGUSR1

NUMRICEVUTISIGUSR2=0
ricevutoSIGUSR2() {
    ((NUMRICEVUTISIGUSR2=${NUMRICEVUTISIGUSR2}+1))
    if ((${NUMRICEVUTISIGUSR2}==2)) ; then
        echo "ricevuto 2° SIGUSR2, disabilito gestione SIGUSR2";
        trap ":" SIGUSR2                                # non faccio fare piu' niente
    else echo "ricevuto ${NUMRICEVUTISIGUSR2}-esimo SIGUSR2, continuo";
    fi
}
trap ricevutoSIGUSR2 SIGUSR2
while true ; do echo -n ".";
    sleep 1; done
```

- Mando il primo SIGUSR2 per dimostrare che il trap setta che funzione lanciare alla ricezione del signal.
- Mando il secondo SIGUSR2 per dimostrare che il trap mantiene il settaggio anche dopo l'uso.
- Mando il terzo SIGUSR2 per dimostrare che posso cambiare la routine di gestione anche da DENTRO la routine stessa.

lancia_riceviSIGUSR1e2.sh

```
#!/bin/bash
./riceviSIGUSR1e2.sh &
CHILDPID=$!
for (( i=0; ${i}<3; i=${i}+1 )) ; do sleep 3;
    echo "mando SIGUSR2"; kill -s SIGUSR2 ${CHILDPID}; done
sleep 3; echo "mando SIGUSR1"; kill -s SIGUSR1 ${CHILDPID};
wait ${CHILDPID}
```

Attesa terminazione di processo figlio di shell: **wait** (1)

Supponiamo che una shell bash lanci in esecuzione un processo e lo metta in background, poi la shell può fare qualcosa mentre il programma è in esecuzione, e infine la shell vuole aspettare che il programma lanciato termini.

Per fare questo devo utilizzare il comando built-in **wait** a cui passo come argomento il process identifier del processo di cui attendere la terminazione. Il comando wait termina quando termina il processo di cui è stato passato il PID.

```
program.exe arg1 arg2 arg3 .... argN &  
PIDsalvato=$!  
echo "il PID di program.exe e\' ${PIDsalvato}"  
... faccio qualcosa d'altro ...  
wait ${PIDsalvato}           <-- attende terminazione di program.exe
```

NB: il comando wait può essere invocato SOLO dalla shell che ha lanciato il processo di cui vogliamo attendere la terminazione. Se l'attesa per la terminazione viene invocata da una shell diversa, il comando wait termina subito e restituisce 127.

NB: il comando wait restituisce come exit status l'exit status restituito dal processo figlio specificato dal PID. Se il PID specificato non corrisponde ad un processo figlio in esecuzione, il comando wait termina subito e restituisce 127. esempio:

```
sleep 20 &  
wait $!    <-- attende fine di sleep 20 e restituisce risultato di sleep
```

Attesa terminazione di processo figlio di shell: **wait** (2)

Il comando wait può prendere diversi argomenti a riga di comando:

- **un elenco di PID** : in tal caso wait aspetta la terminazione di tutti i processi indicati e restituisce come risultato il risultato dell'ultimo processo nella lista di PID, indipendentemente dall'ordine in cui terminano
- **un elenco di jobs** : in tal caso wait aspetta la terminazione di tutti i jobs indicati
- **nessun argomento** : in tal caso wait aspetta la terminazione di tutti i processi figli della shell in cui il comando wait è stato lanciato e restituisce exit status 0 non essendo indicato uno specifico figlio di cui restituire l'exit status.

```
(sleep 20 ; exit 3 ) &
```

```
wait $!
```

```
echo "il valore restituito da wait è: " $?
```

<-- attende terminazione di exit 3
<-- visualizza 3

```
(sleep 20 ; false ) &
```

```
wait $!
```

```
echo "il valore restituito da false è: " $?
```

<-- attende terminazione di false
<-- visualizza 1

```
progr &
```

```
pid1=$!
```

```
prog2 &
```

```
pid2=$!
```

```
prog3 &
```

```
pid3=$!
```

```
wait ${pid1} ${pid3} ${pid2} <-- attende terminazione dei 3 prog e restituisce l' exit status  
di prog2 cioè del programma il cui pid compare per ultimo nell'elenco 200
```

wait, processi Zombie, processi Orfani, processo init (1)

Un processo (padre) genera un processo figlio e lo esegue in background.

Il processo figlio prima o poi **termina** e restituisce un valore intero.

Il processo padre vuole sapere quale è il valore restituito dal figlio per sapere se tutto è andato bene o no. A questo scopo, il processo padre esegue il comando **wait pidfiglio** che attende la terminazione del processo figlio avente pid pidfiglio e restituisce il risultato del processo figlio.

1.Se il processo figlio termina prima del processo padre, il sistema operativo rilascia le risorse occupate dal processo figlio **ma mantiene nelle sue tabelle una struttura pcb (process control block) con una descrizione del processo terminato**. Questa struttura conserva anche il **pid** e il **risultato** del processo figlio. In questo momento il processo figlio è uno **zombie** poiché è terminato ma la sua struttura descrittiva è ancora presente nelle tabelle del sistema operativo.

- **La struttura pcb viene eliminata solo quando il padre invoca il comando wait** (o una system call waitpid che fa la stessa cosa) per attendere la terminazione del figlio. A quel punto il processo figlio sparisce dalle tabelle e il suo pid può essere riutilizzato per altri processi.

2.Se invece il processo padre termina senza fare la wait per il figlio, il processo figlio diventa orfano. Quando un processo orfano termina, oppure quando uno zombie diventa orfano perché termina il padre, **l'orfano viene adottato dal processo init**, quello con pid 1 che ha originato tutti i processi al boot. Gli orfani diventano figli di init. **Ogni tanto il processo init chiama la wait sui propri figli**, anche quelli adottati ovviamente, e così fa rilasciare i pcb degli orfani.

- Quindi, **morto il padre, i figli che terminano vengono eliminati (no zombie)** 201

Processi Zombie, Processi Orfani, processo init (2)

Per quale motivo un processo padre non fa una wait per attendere la terminazione di un processo figlio?

1. o per sbaglio, perché il programmatore se ne è dimenticato,
2. oppure appositamente, per impedire che un altro suo nuovo figlio prenda dal sistema operativo proprio il pid del processo figlio terminato.

In entrambi i casi, si rischia di esaurire i pid disponibili nel sistema operativo.

killall - Terminazione di tutti i processi con un dato nome

Il comando **killall** viene usato, solitamente, per terminare l'esecuzione di tutti i processi che hanno lo stesso nome; quel nome viene passato come argomento al comando killall. Il comando killall per default invia a quei processi il segnale SIGTERM, a meno che non si specifichi un diverso segnale.

In realtà, il comando killall può servire più genericamente ad inviare ai processi un segnale specificato mediante un argomento a riga di comando.

Inoltre, alcuni parametri permettono di individuare in modo diverso i processi a cui inviare il segnale.

Esempio:

Supponiamo che, **nella directory corrente** esistano i file binari eseguibili di nome **attendiA.exe** e **attendiC.exe** e che esista uno script di comandi **attendiB.sh**, e che l'esecuzione di ciascuno di questi duri qualche minuto.

Posso lanciare in background più istanze di ciascun eseguibile/script così:

```
for (( i=1; $i < 10 ; i=$i+1 )) ; do ./attendiA.exe & ./attendiB.sh & done
```

Se voglio, posso terminarli tutti i processi attendiA.exe ed attendiB.sh usando il seguente comando:

```
killall -r 'attendi[[:upper:]]*'
```

Il comando ordina di usare l'espressione regolare (che segue l'opzione -r) per selezionare il nome dei processi tra attualmente in esecuzione da eliminare.

Notare che gli apici semplici impediscono che sia la bash a interpretare i metacaratteri. L'interpretazione viene così fatta dalla killall con i nomi dei processi in memoria e non con quelli della directory corrente.

killall - Interpretazione del nome dei processi in memoria

Verifichiamo che la killall interpreta i metacaratteri cercando di **associarli ai nomi dei processi in memoria e non al nome dei file nel filesystem.**

Supponiamo che, **nella directory corrente** esistano i file binari eseguibili di nome **attendiA.exe** e **attendiC.exe** e che esista uno script di comandi **attendiB.sh**, e che l'esecuzione di ciascuno di questi duri qualche minuto.

Posso lanciare in background più istanze di ciascun eseguibile/script così:

```
for (( i=1; $i < 10 ; i=$i+1 )) ; do ./attendiA.exe & ./attendiB.sh & done
```

Ora lanciamo il seguente comando, SI NOTI CHE NON CI SONO APICI SINGOLI:

```
killall -r attendi[[:upper:]]*
```

Senza gli apici semplici, che impedirebbero alla bash di interpretare i metacaratteri, **l'interpretazione viene PRIMA effettuata dalla bash con i nomi dei file della directory corrente e, SOLO IN UN SECONDO MOMENTO, viene effettuata dalla killall con i nomi dei processi in memoria.**

la bash interpreta i metacaratteri e, in pratica lancerei

```
killall -r attendiA.exe attendiB.sh attendiC.exe
```

Allora, la killall mostra un output che fa capire che ha cercato in memoria il processo attendiC.exe ma non l'ha trovato

attendiC.sh: no process found