

Lezione 11

Processi

Sistemi Operativi (9 CFU), CdL Informatica, A. A. 2023/2024
Dipartimento di Scienze Fisiche, Informatiche e Matematiche
Università di Modena e Reggio Emilia

Quote of the day

(Meditate, gente, meditate...)

“This is the UNIX philosophy: write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.”

Douglas McIlroy (1932-)

*Matematico, Ingegnere, Programmatore
Inventore delle pipe e delle utility UNIX*

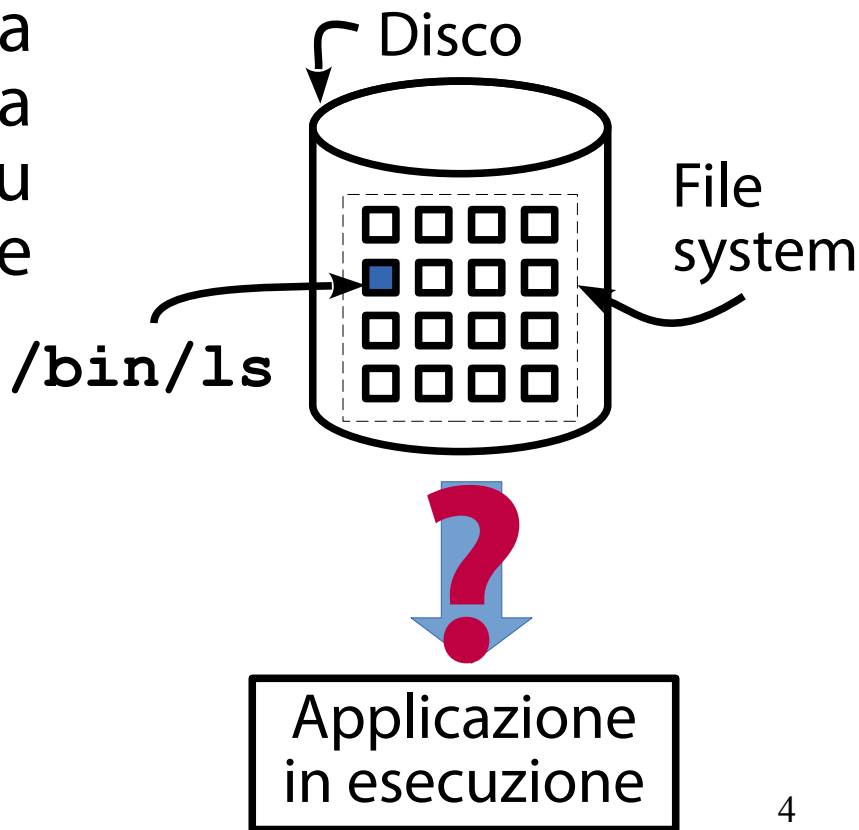


INTRODUZIONE

Lo scenario

(Lo studente vuole capire come file su disco diventano applicazioni in esecuzione)

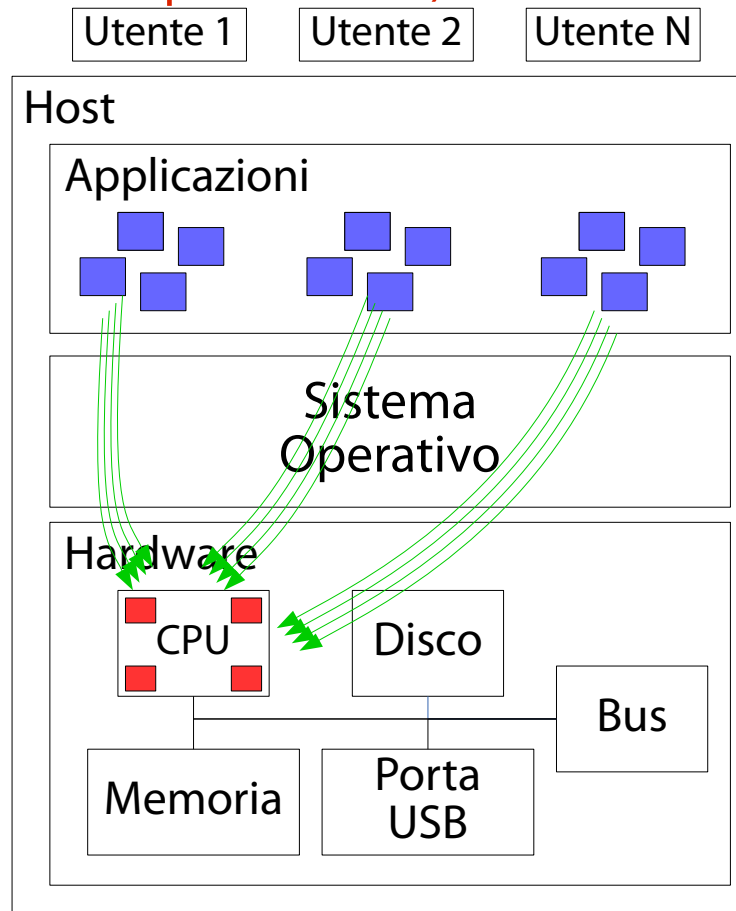
Uno studente in grado di usare la linea di comando vuole capire la magia con cui un programma su disco diventa una applicazione in esecuzione.



Interrogativi 1/3

(Come eseguire tante applicazioni su poche CPU?)

Le applicazioni lanciate dagli utenti sono tante. Le CPU a disposizione sono poche. Come è possibile eseguire in maniera efficiente e veloce una moltitudine di applicazioni su poche unità di calcolo?

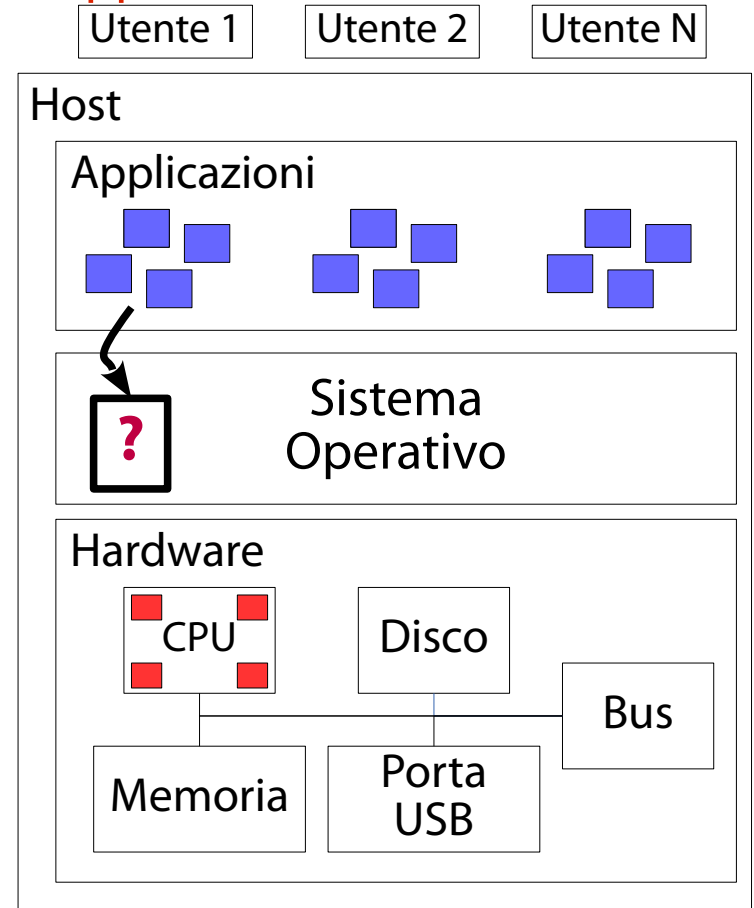


Interrogativi 2/3

(Come si identificano e si visualizzano le applicazioni in esecuzione?)

Come si identificano
applicazioni in esecuzione?
Come si visualizzano
applicazioni in esecuzione?

le
le

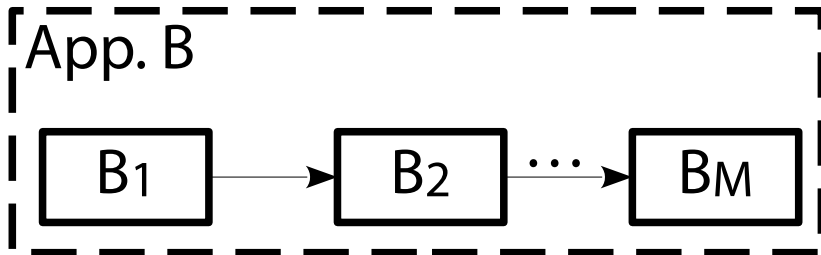
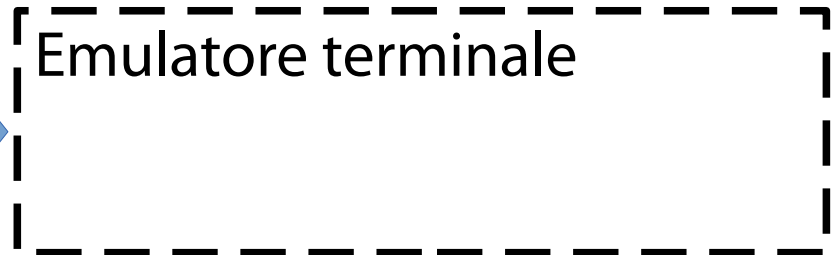
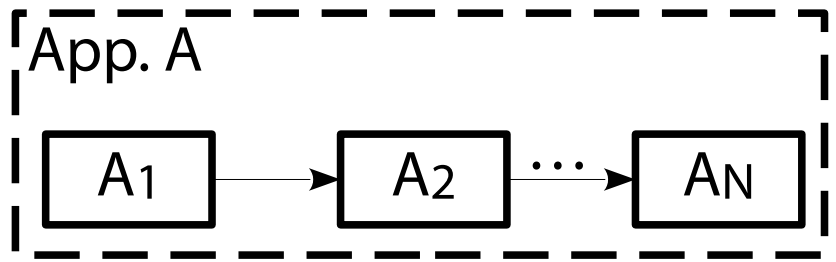


Interrogativi 3/3

(Come costruire applicazioni più complesse e coordinarle su un terminale?)

È possibile combinare più comandi per costruire applicazioni più complesse?

È possibile alternare più applicazioni su un singolo terminale?



Processo

(Astrazione di un programma in esecuzione)

Il **processo** è l'astrazione di un programma in esecuzione.

Il nucleo rappresenta un processo tramite opportune strutture dati e lo gestisce tramite funzioni.

Processo

(Astrazione di un programma in esecuzione)

Strutture dati.

Rappresentazione di uno “stato interno” (Bloccato? In esecuzione? Terminato?).

Puntatori alle risorse in uso (aree di memoria, file aperti, altro).

Funzioni di gestione.

Creazione e terminazione.

Esecuzione di un programma eseguibile.

Comunicazione, sincronizzazione.

Monitoraggio.

Processo vs. programma eseguibile

(Una differenza importante)

Programma eseguibile. È un file memorizzato su supporto secondario. Contiene il codice macchina da eseguire, alcune aree dati, una tabella di simboli utile per il debugging. È un'entità statica. Non si modifica da solo.

Processo. È la rappresentazione del nucleo (in termini di strutture dati e funzioni di gestione) di un programma eseguibile in esecuzione. È un'entità dinamica; evolve il suo stato durante l'esecuzione.

Astrazione delle risorse

(Non esiste solo la CPU...)

Il processo non è l'unica astrazione presente nel SO.
Per usare la CPU è necessario creare una astrazione (il processo) di cui l'utente possa fruire.

Ciò è vero in generale: **per usare una risorsa hw il SO deve fornire** ai suoi utenti **una astrazione** della stessa.

Ad esempio:

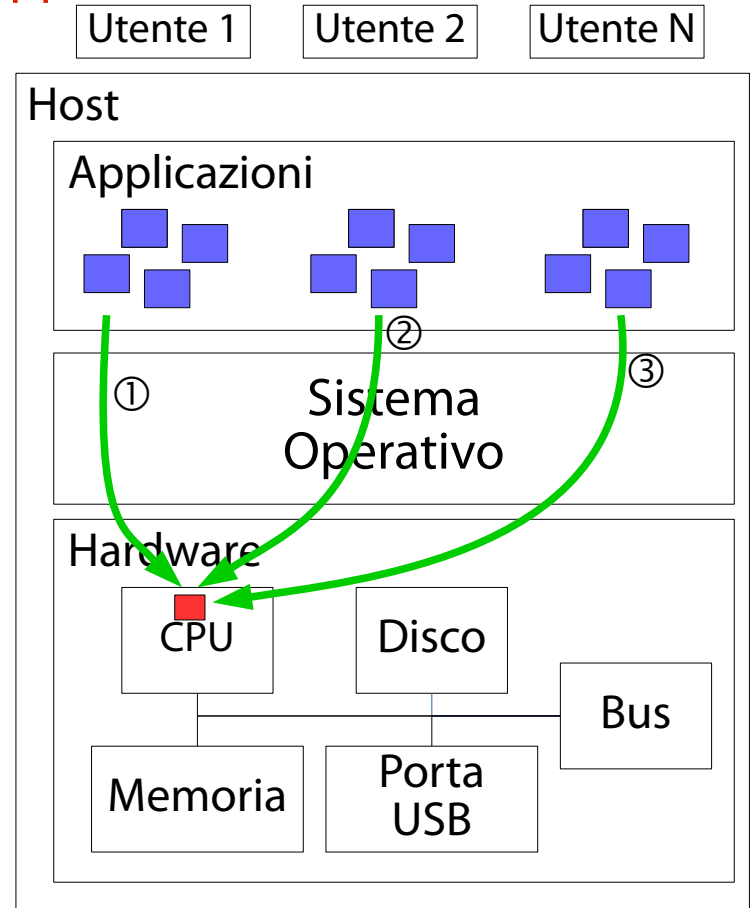
Risorsa "disco" → astrazioni "file" e "directory".

Risorsa "memoria" → astrazione "area di memoria".

Esecuzione in time sharing

(Permette l'esecuzione di tante applicazioni su una CPU)

L'unico modo per eseguire in maniera efficiente e veloce una moltitudine di applicazioni su poche unità di calcolo è l'impiego della tecnica di **time sharing**.



Esecuzione in time sharing

(Permette l'esecuzione di tante applicazioni su una CPU)

Caratteristiche del time sharing.

Esecuzione sequenziale su una CPU di piccole porzioni di una applicazione.

Limite superiore di esecuzione: un quanto di tempo per applicazione.

Se l'applicazione chiede un servizio bloccante (I/O) il SO ne esegue un'altra.

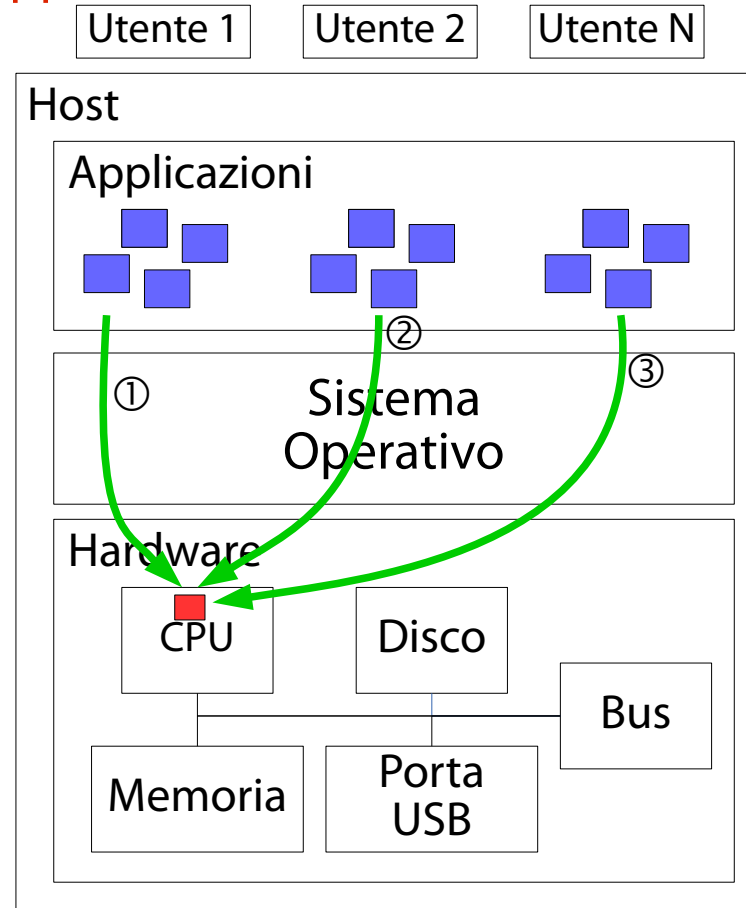
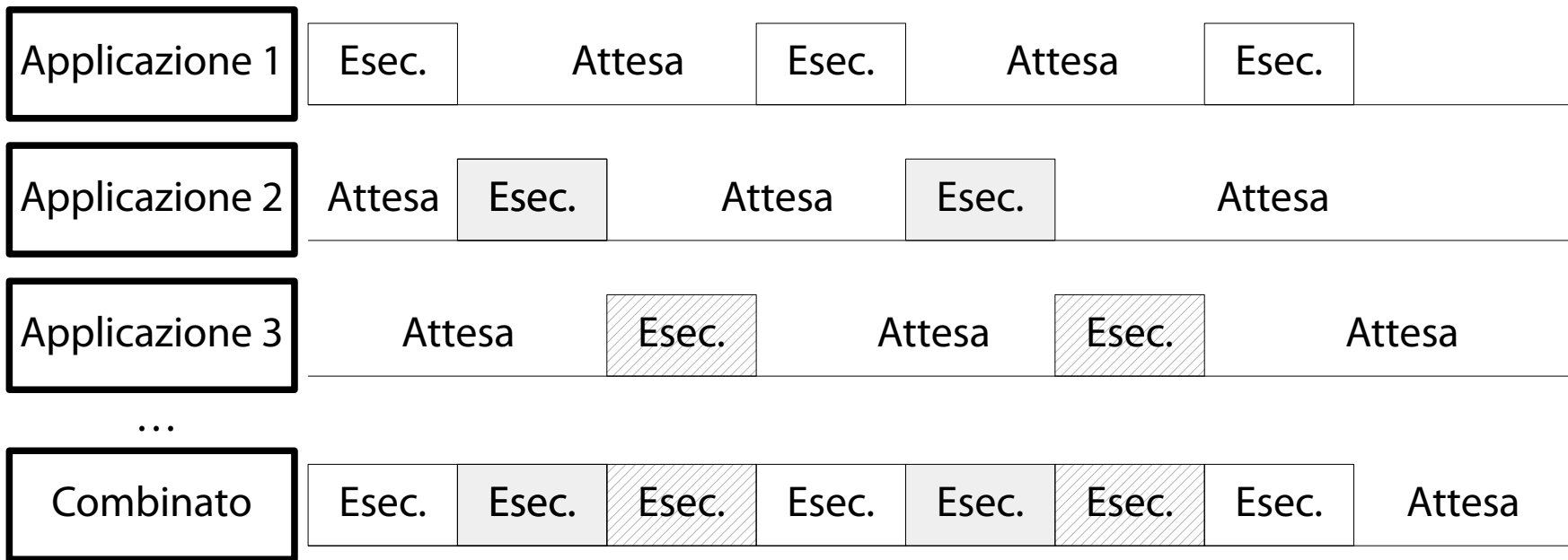


Diagramma temporale

(Dell'esecuzione in time sharing su una CPU)



Tempo

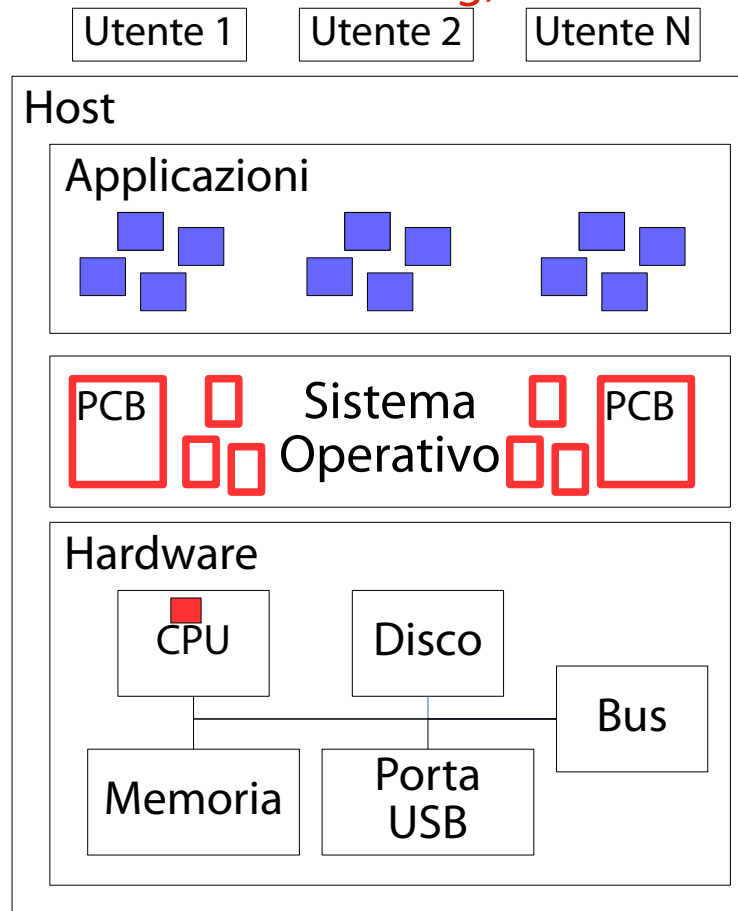
Descrittore di processo

(Fondamentale nell'implementazione del time sharing)

A tal scopo il SO mantiene, per ogni processo, un **descrittore di processo (Process Control Block, PCB)**.

Il PCB è la struttura dati dell'astrazione processo. Esso contiene:

informazioni sullo stato interno del processo.
puntatori alle risorse prenotate dal processo.



Usi del descrittore di processo

(Ecco perché è fondamentale)

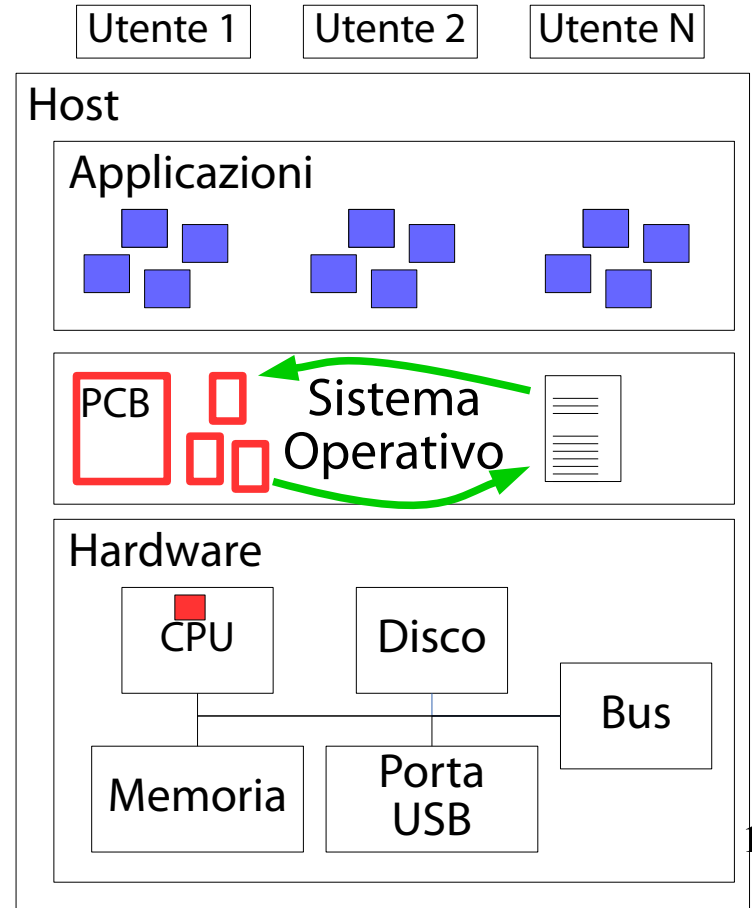
Il SO legge ed aggiorna i PCB tramite funzioni interne. Tipiche operazioni svolte includono:

- identificazione del prossimo processo da eseguire.

- rimozione di un processo dalla CPU.

- ripristino di un processo sulla CPU.
- aggiornamento dei contatori di uso delle risorse.

I valori contenuti nei PCB influenzano le decisioni prese dal SO.



Isolamento dei processi

(Ogni processo “vede” in esclusiva un insieme di risorse: file, memoria, CPU, ...)

I processi sono isolati fra loro.

Ogni processo vede un proprio **spazio degli indirizzi** (l'insieme delle aree di memoria ad esso assegnate).

I descrittori dei file creati da un processo in seguito all'apertura non sono condivisibili con altri processi.

Isolamento dei processi

(Ogni processo “vede” in esclusiva un insieme di risorse: file, memoria, CPU, ...)

Vantaggi.

Sicurezza (un processo non può alterare lo stato di un altro processo).

Svantaggi.

Ridondanza. Le risorse comuni ai diversi processi (in primis, il codice della libreria del C) è in larga parte ripetuto, con conseguente spreco di risorse e rallentamenti nella gestione.

Il kernel Linux implementa tecniche specifiche per mitigare questo problema.

Thread

(Un'astrazione alternativa ai processi)

Un'astrazione alternativa per la rappresentazione di una applicazione in esecuzione è il **thread di esecuzione** (o **thread**).

Un processo può creare diversi thread. Ciascun thread esegue una funzione specifica (definita nel programma associato al processo).

I thread di un processo condividono tutte le risorse (in primis, memoria e file).

Analogamente ai processi, ciascun thread ha un proprio **Thread Control Block (TCB)**.

Thread

(Un'astrazione alternativa ai processi)

Vantaggi.

Prestazioni. Poiché le risorse sono condivise tra processo e thread, non è necessario ricrearle ogni volta. È sufficiente puntarle dal TCB di ogni thread.

Svantaggi.

Il modello di programmazione si complica di parecchio. Bisogna tener conto del fatto che più thread possano accedere alla stessa cella di memoria.

→ Il risultato del calcolo non è più deterministico (dipende dall'ordine con cui i thread accedono alla memoria).

INDIVIDUAZIONE

Scenario e interrogativi

(Come individuare un processo in esecuzione?)

Scenario: un SO in esecuzione sta eseguendo tanti processi per conto di diversi utenti autenticati sull'host. L'utente vuole individuare un processo. I motivi possono essere svariati:

- monitoraggio delle risorse consumate.
- sospensione temporanea.
- debugging.
- uccisione.

Interrogativi:

Come individuare un processo specifico fra tanti?

L'identificatore di processo

(Il "nome" assegnato dal nucleo a ciascun processo)

Uno dei campi contenuti nel descrittore di processo è l'**identificatore di processo (Process Identifier, PID)**.

Uno dei campi contenuti nel descrittore di thread è l'**identificatore di thread (Thread Identifier, TID)**.

Il PID/TID è un numero intero non negativo assegnato dal nucleo all'istante di creazione di un nuovo processo.

I comandi di amministrazione dei processi chiedono spesso il PID come argomento.

→ Per interagire con un processo/thread è spesso necessario conoscere il suo PID/TID.

Identificazione tramite nome esatto

(Si usa il comando **pidof**)

Il comando esterno **pidof** stampa i PID di tutti i processi attivati a partire da un nome esatto.

Si provi a stampare i PID di tutti i processi **bash** attivati.

```
pidof bash
```


Esercizio 1 (1 min.)

Qual è il PID del processo `init` (il gestore dei servizi UNIX)?

Il difetto principale di pidof

(Richiede il nome esatto del processo)

Purtroppo, **pidof** presenta diverse limitazioni. La più evidente è la necessità di usare il nome esatto del comando eseguito.

Se si vogliono cercare i processi aventi nome **bash**, ricordandosi solo che tali processi iniziano con la stringa "ba", si potrebbe provare ad eseguire il comando seguente:

```
pidof ba
```

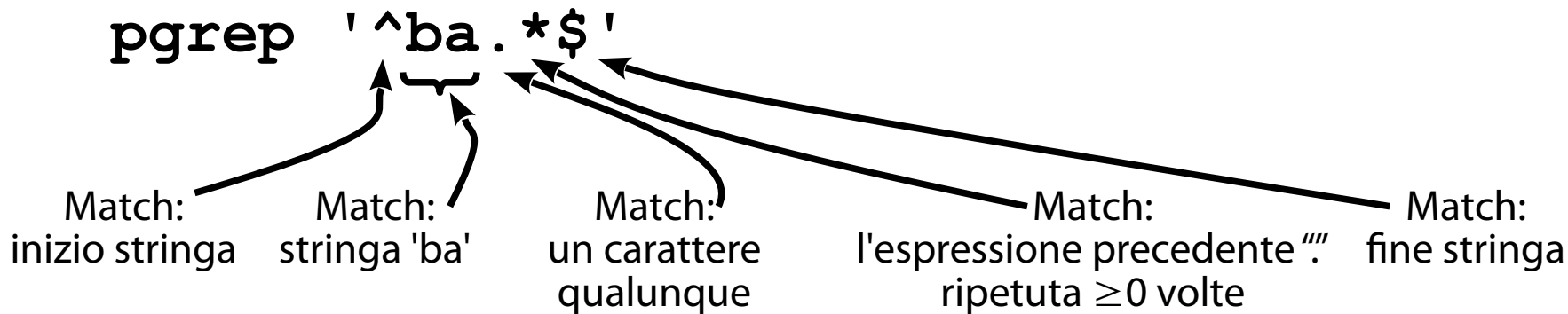
Tale comando, purtroppo, non funziona!

Identificazione tramite nome parziale

(Si usa il comando **pgrep**)

Il comando esterno **pgrep** stampa i PID di tutti i processi attivati a partire da una espressione regolare estesa.

Si provi a stampare i PID di tutti i processi il cui nome inizia con la stringa "ba".



Alcune opzioni notevoli di **pgrep**

(Da ricordare; si incontrano spesso nella carriera di un sysadmin)

man pgrep per tutti i dettagli.

pgrep -n bash: stampa il PID del processo più recente contenente la stringa "bash" nel nome.

pgrep -o bash: stampa il PID del processo meno recente contenente la stringa "bash" nel nome.

pgrep -l bash: stampa il PID e il nome dei processi contenenti la stringa "bash" nel nome.

Esercizio 2 (2 min.)

I processi del desktop “GNOME” iniziano con la stringa “gnome”. Come si possono individuare i processi del desktop in esecuzione?

Esercizio 3 (2 min.)

Elencate tutti i processi (e relativi PID) eseguiti per conto del vostro username.

Leggete la pagina di manuale del comando opportuno per capire come fare.

RIPRODUZIONE

Scenario e interrogativi

(Come fa un'applicazione a creare un processo?)

Scenario: un'applicazione ha bisogno di creare un nuovo processo per sbrigare alcune funzioni.

Interrogativi:

Quali strumenti sono messi a disposizione dell'utente e delle applicazioni per l'avvio di nuovi processi?

Creazione di processi

(Forking)

Durante la propria esecuzione, un processo può creare altri processi. Il meccanismo di creazione è una vera e propria **clonazione** (**forking**).

Processo **clonante**: → processo **padre**.

Processo **clonato**: → processo **figlio**.

Al termine della clonazione, il processo figlio è una copia identica del processo padre.

Entrambi i processi ripartono dall'istruzione successiva alla clonazione.

Caratteristiche della clonazione

(Il processo figlio è una copia identica del processo padre)

Al termine della clonazione, il processo figlio è una copia identica del processo padre.

Punta agli stessi file aperti.

Ottiene una copia privata delle stesse aree di memoria del padre.

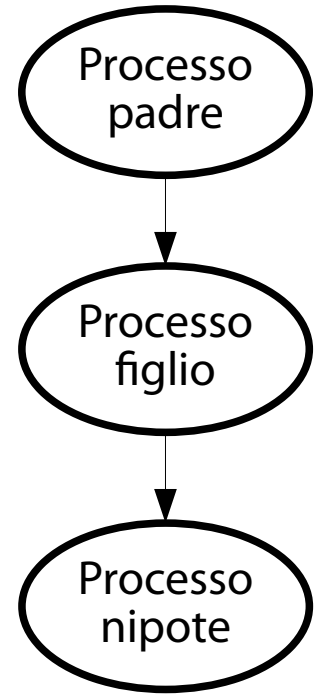
...

Entrambi i processi ripartono dall'istruzione successiva alla clonazione.

Attenzione! Padre e figlio sono isolati. Le modifiche di uno non sono viste dall'altro.

Esercizio 4 (2 min.)

Aprirete un terminale. Con gli strumenti a vostra disposizione, individuate un modo per creare una gerarchia di processi come quella mostrata in figura.



Esecuzione di programmi arbitrari

(È previsto un meccanismo di sostituzione di binari)

La procedura di clonazione tramite forking consente di creare un processo figlio, che esegue lo stesso codice del processo padre.

Per eseguire un'applicazione diversa, è previsto un **meccanismo di sostituzione** che:

- carica il contenuto di un file eseguibile in memoria, in maniera efficiente;

- ricostruisce lo spazio di indirizzamento del nuovo processo (codice e dati per file eseguibile e librerie, stack);

- salta ad una locazione di partenza.

ALBERO DEI PROCESSI

Scenario e interrogativi

(È possibile analizzare le relazioni di parentela fra processi?)

Scenario: si considerino diversi processi legati da parentela.

Interrogativi:

Quale struttura dati è usata per memorizzare le relazioni di parentela?

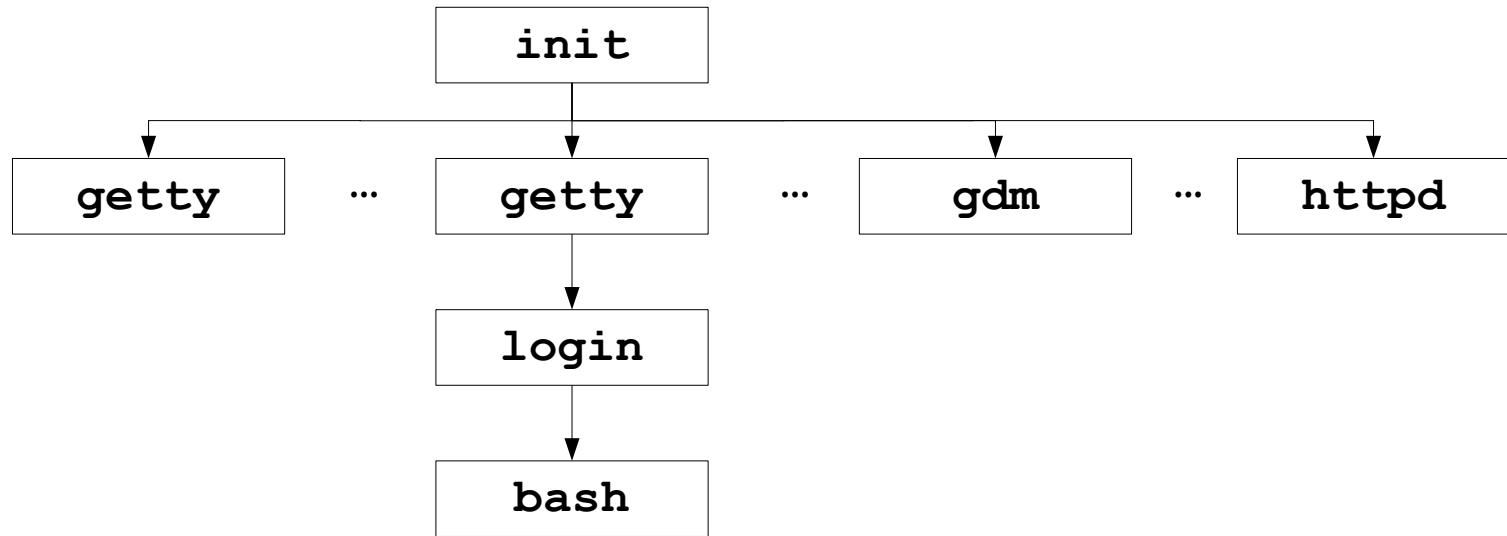
Chi crea il primo processo?

È possibile visualizzare le relazioni di parentela fra processi?

Organizzazione dei processi – SO UNIX

(Un albero)

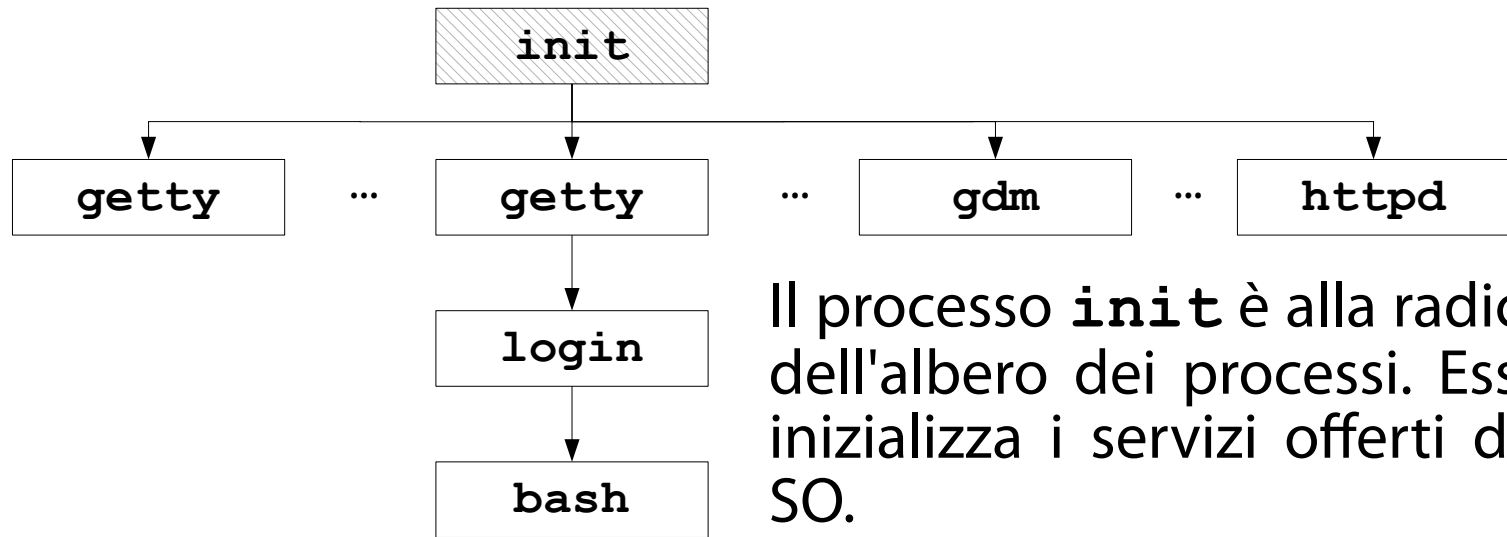
Nei SO UNIX i processi sono organizzati in un albero, detto **albero dei processi (process tree)**.



Organizzazione dei processi – SO UNIX

(Un albero)

Nei SO UNIX i processi sono organizzati in un albero, detto **albero dei processi (process tree)**.

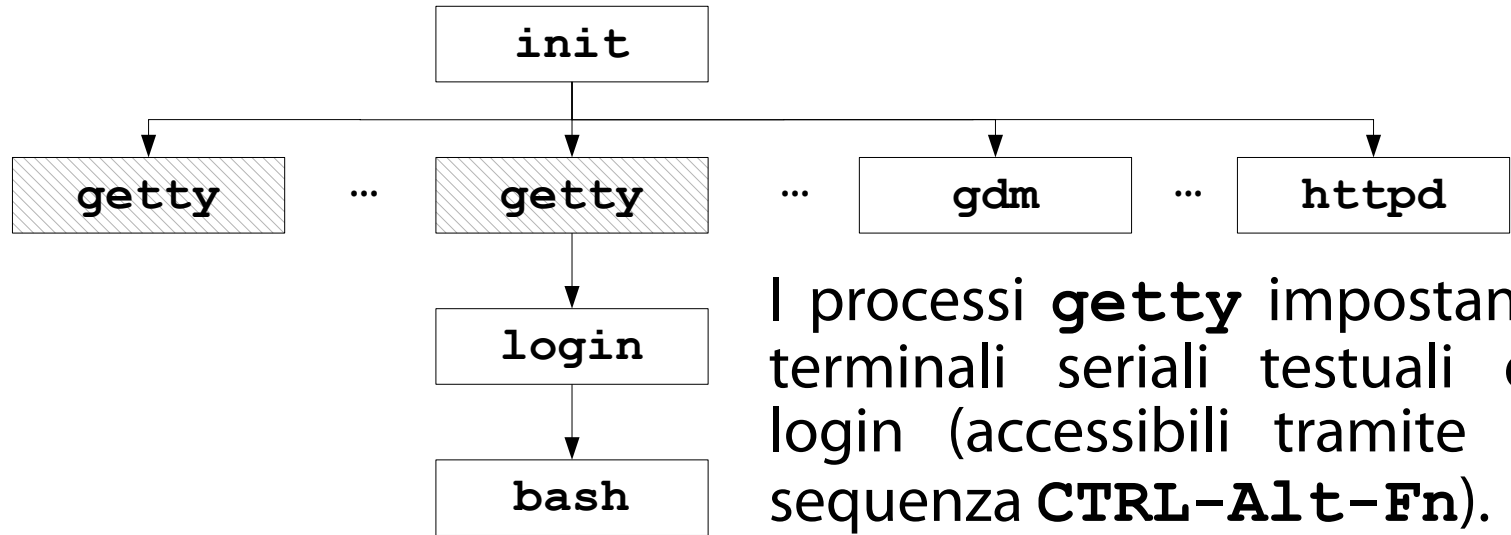


Il processo `init` è alla radice dell'albero dei processi. Esso inizializza i servizi offerti dal SO.

Organizzazione dei processi – SO UNIX

(Un albero)

Nei SO UNIX i processi sono organizzati in un albero, detto **albero dei processi (process tree)**.

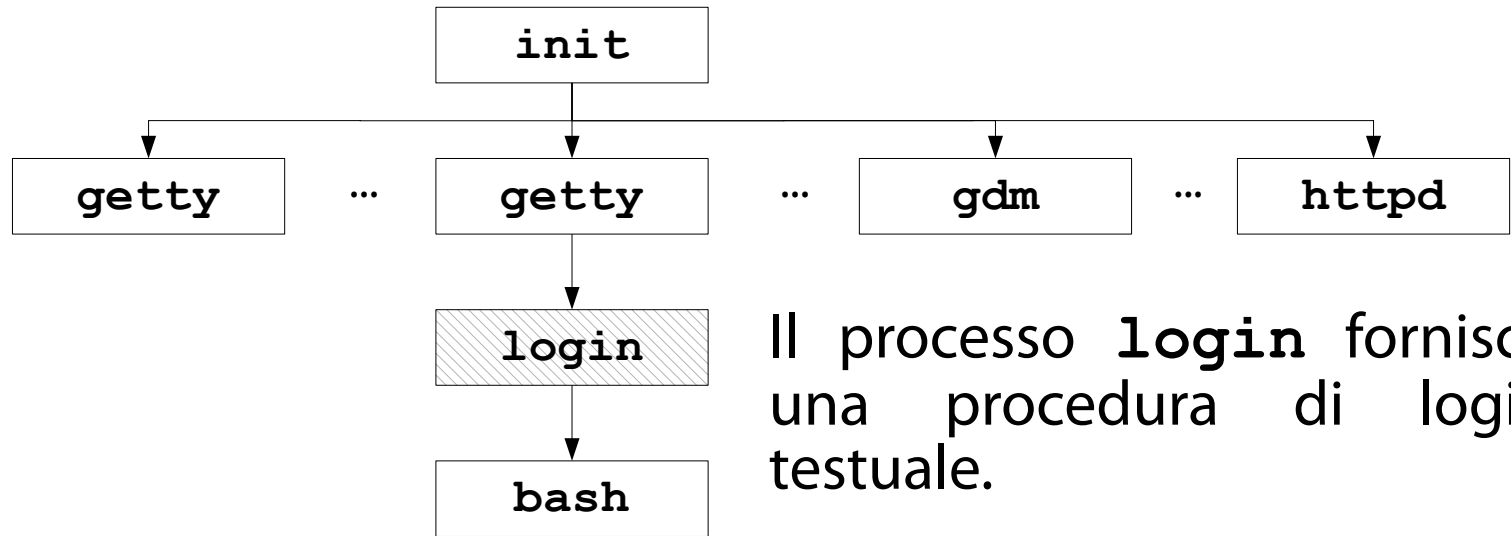


I processi **getty** impostano terminali seriali testuali di login (accessibili tramite la sequenza **CTRL-Alt-Fn**).

Organizzazione dei processi – SO UNIX

(Un albero)

Nei SO UNIX i processi sono organizzati in un albero, detto **albero dei processi (process tree)**.

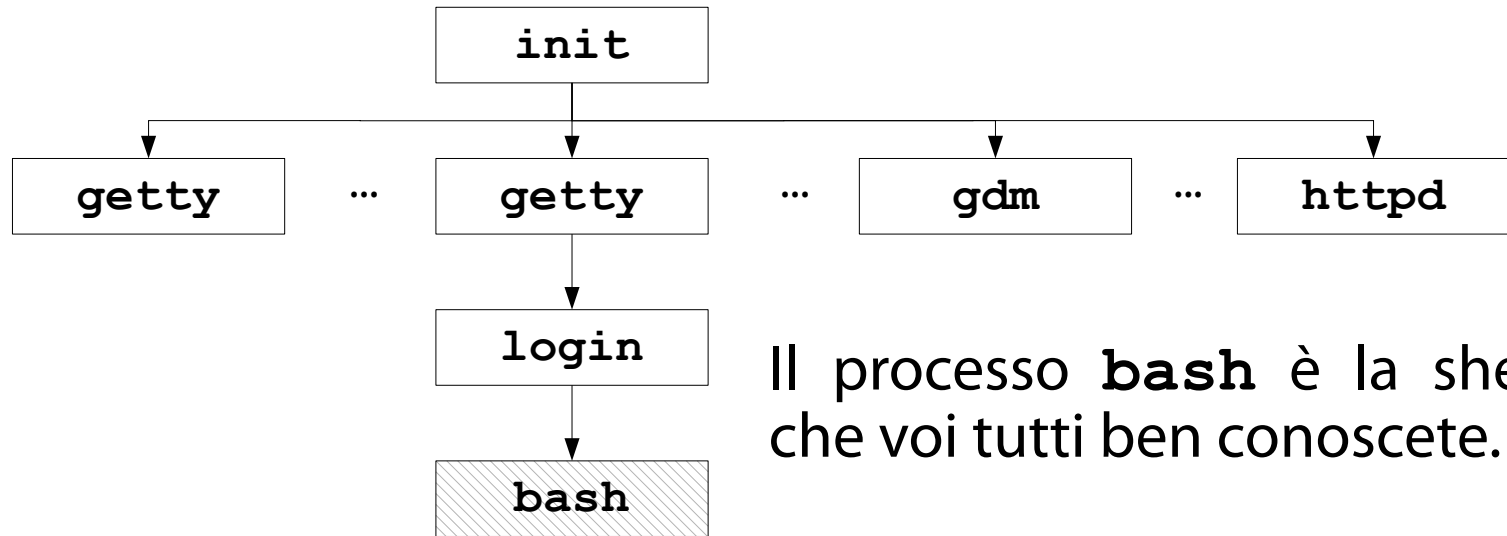


Il processo `login` fornisce una procedura di login testuale.

Organizzazione dei processi – SO UNIX

(Un albero)

Nei SO UNIX i processi sono organizzati in un albero, detto **albero dei processi (process tree)**.

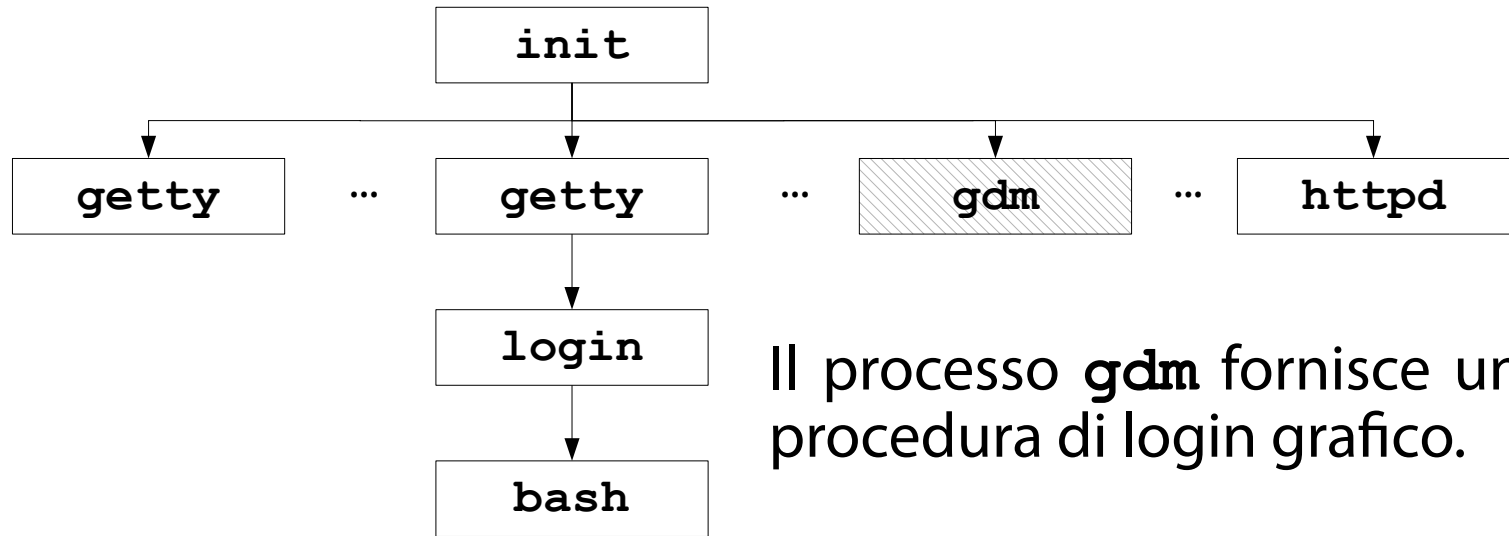


Il processo **bash** è la shell che voi tutti ben conoscete.

Organizzazione dei processi – SO UNIX

(Un albero)

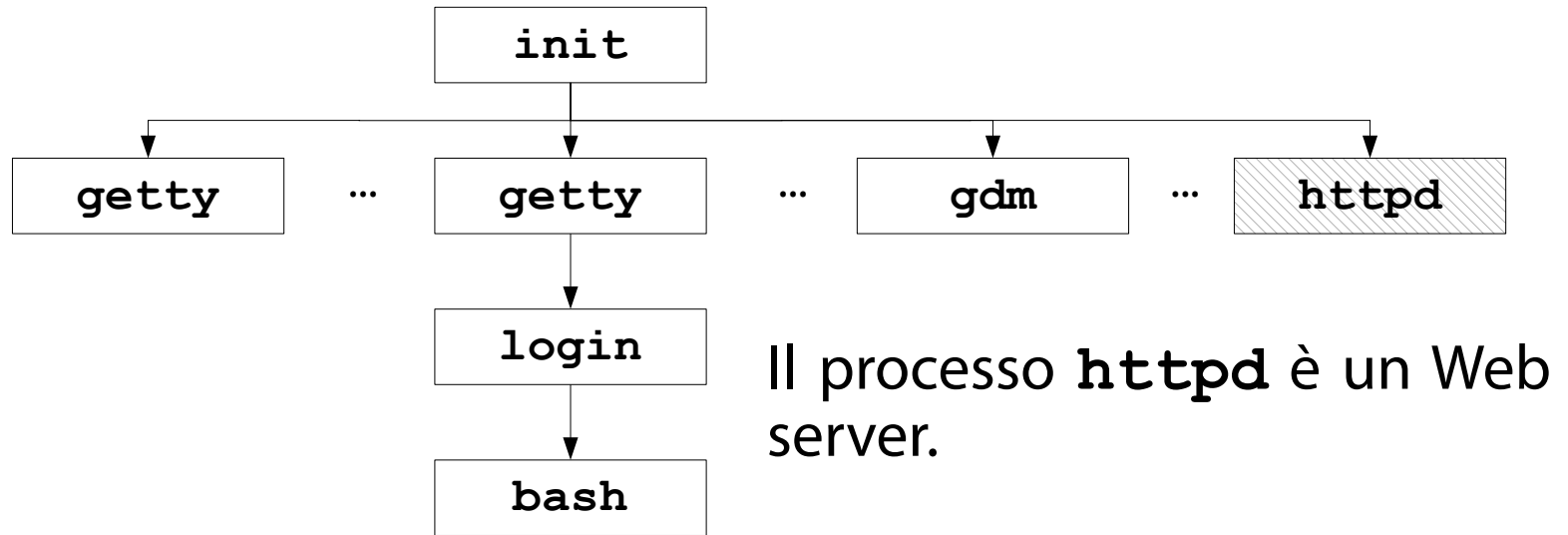
Nei SO UNIX i processi sono organizzati in un albero, detto **albero dei processi (process tree)**.



Organizzazione dei processi – SO UNIX

(Un albero)

Nei SO UNIX i processi sono organizzati in un albero, detto **albero dei processi (process tree)**.



Domanda

(Dovrebbe essere la prima nella vostra testa)

Chi crea il processo **`init`**?

Power On Self Test

(Back to the roots; la nonna di Tutankhamon usava il BIOS in figura...)

All'avvio, il PC esegue il **Power On Self Test (POST)**.

Serie di test cablati nel BIOS.

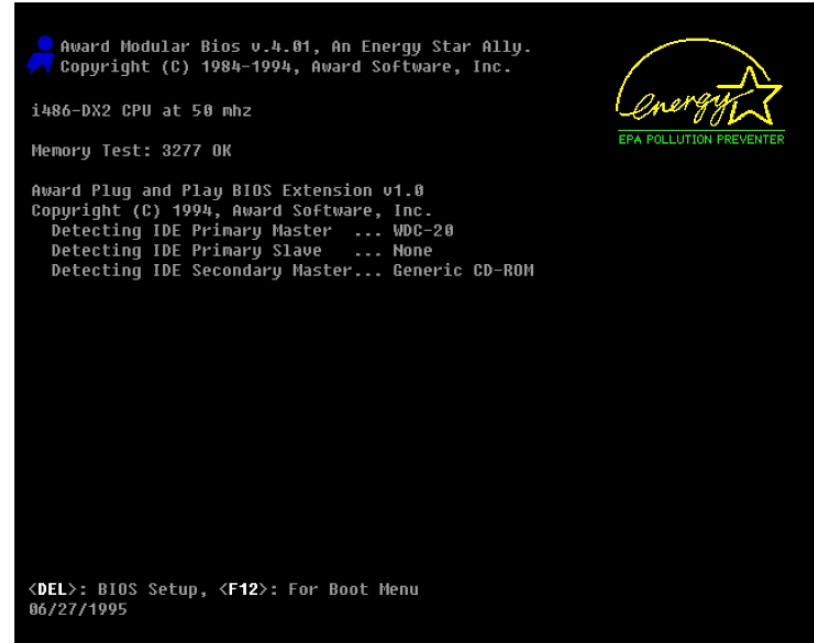
Controllo presenza tastiera.

Controllo RAM.

Inizializzazione scheda video.

Inizializzazione dischi.

Inizializzazione rete.



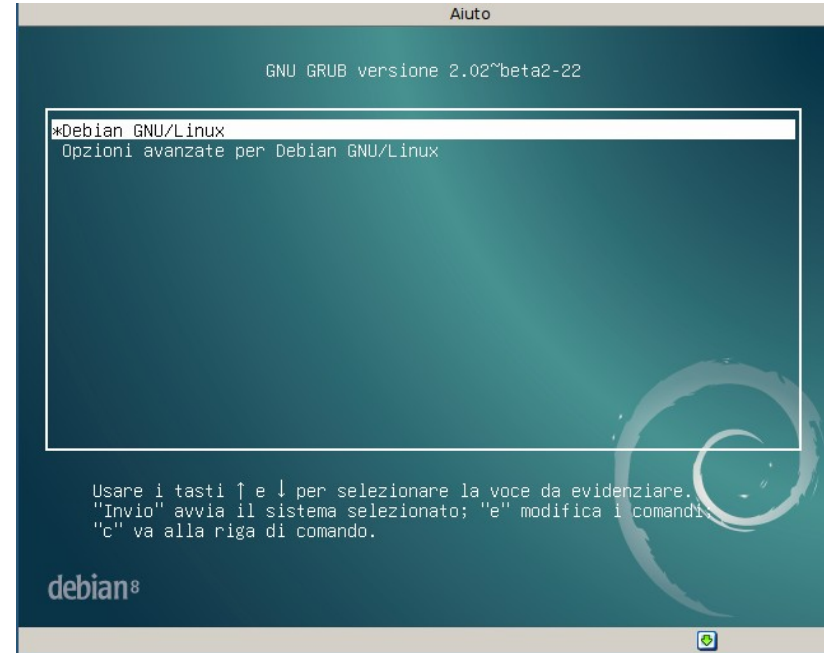
Avvio del boot loader

(Il BIOS carica GRUB da un disco e lo esegue)

Al termine del POST, il BIOS individua una periferica avviabile (ossia, con un boot loader installato).

La prima che trova è usata per caricare il boot loader in memoria centrale.

Il boot loader fa scegliere il SO da avviare all'utente.



Avvio del nucleo

(Il boot loader carica il nucleo ed esegue una funzione di inizializzazione)

Una volta selezionato il SO che si desidera avviare, GRUB carica in memoria centrale il suo nucleo ed esegue una funzione di inizializzazione.

Accensione periferiche.

Configurazione periferiche.

Inizializzazione strutture dati.

```
[ 32.303184] e1000 0000:00:03:0 eth0: Intel(R) PRO/1000 Network Connection
[ 32.304150] ahci 0000:00:1f.2: SSS flag set, parallel bus scan disabled
[ 32.304659] ahci 0000:00:1f.2: AHCI 0001.0100 32 slots 1 ports 3 Gbps 0x1 imp
l SATA mode
[ 32.305263] ahci 0000:00:1f.2: flags: 64bit ncq stag only ccc
[ 32.306254] scsi0 : ahci
[ 32.306619] ata1: SATA max UDMA/133 abar m81920xf0900000 port 0xf0900100 irq
40
[ 32.308147] scsi1 : ata_piix
[ 32.308542] scsi2 : ata_piix
[ 32.308893] ata2: PATA max UDMA/100 cmd 0x1f0 ctl 0x3f6 bmdma 0xe000 irq 14
[ 32.309306] ata3: PATA max UDMA/100 cmd 0x170 ctl 0x376 bmdma 0xe008 irq 15
[ 32.310272] ohci-pci 0000:00:1f.4: OHCI PCI host controller
[ 32.310675] ohci-pci 0000:00:1f.4: new USB bus registered, assigned bus numbe
r 1
[ 32.311330] ohci-pci 0000:00:1f.4: irq 23, io mem 0xf0902000
[ 32.364671] usb usb1: New USB device found, idVendor=1d6b, idProduct=0001
[ 32.365978] usb usb1: New USB device strings: Mfr=3, Product=2, SerialNumber=
1
[ 32.367879] usb usb1: Product: OHCI PCI host controller
[ 32.369129] usb usb1: Manufacturer: Linux 3.16.0-4-amd64 ohci_hcd
[ 32.370387] usb usb1: SerialNumber: 0000:00:1f.4
[ 32.372092] hub 1-0:1.0: USB hub found
[ 32.373744] hub 1-0:1.0: 12 ports detected
```

Avvio di init

(Il nucleo crea il primo processo e ci esegue sopra **/sbin/init**)

Le ultime due operazioni svolte dalla funzione di inizializzazione sono le seguenti.

Creazione del primo processo (con tanto di descrittore).

Uso del primo processo per l'esecuzione del binario **/sbin/init**.

```
[ 36.810886] cfg80211: (2457000 KHz - 2482000 KHz @ 40000 KHz), (N/A, 2000 mBm), (N/A)
[ 36.810887] cfg80211: (2474000 KHz - 2494000 KHz @ 20000 KHz), (N/A, 2000 mBm), (N/A)
[ 36.810888] cfg80211: (5170000 KHz - 5250000 KHz @ 80000 KHz, 160000 KHz AUTD), (N/A, 2000 mBm), (N/A)
[ 36.810889] cfg80211: (5250000 KHz - 5330000 KHz @ 80000 KHz, 160000 KHz AUTD), (N/A, 2000 mBm), (0 s)
[ 36.810890] cfg80211: (5490000 KHz - 5730000 KHz @ 160000 KHz), (N/A, 2000 mBm), (0 s)
[ 36.810891] cfg80211: (5735000 KHz - 5835000 KHz @ 80000 KHz), (N/A, 2000 mBm), (N/A)
[ 36.810891] cfg80211: (57240000 KHz - 63720000 KHz @ 2160000 KHz), (N/A, 0 mBm), (N/A)
[ OK ] Reached target Login Prompts.
[ OK ] Started Authenticate and Authorize Users to Run Privileged Tasks.
[ OK ] Started Accounts Service.
[ OK ] Started Modem Manager.
[ OK ] Started Network Manager.
[ 36.906359] IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready
[ 36.903339] e1000: eth0 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: RX
[ 36.910247] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
[ OK ] Started GNOME Display Manager.
```

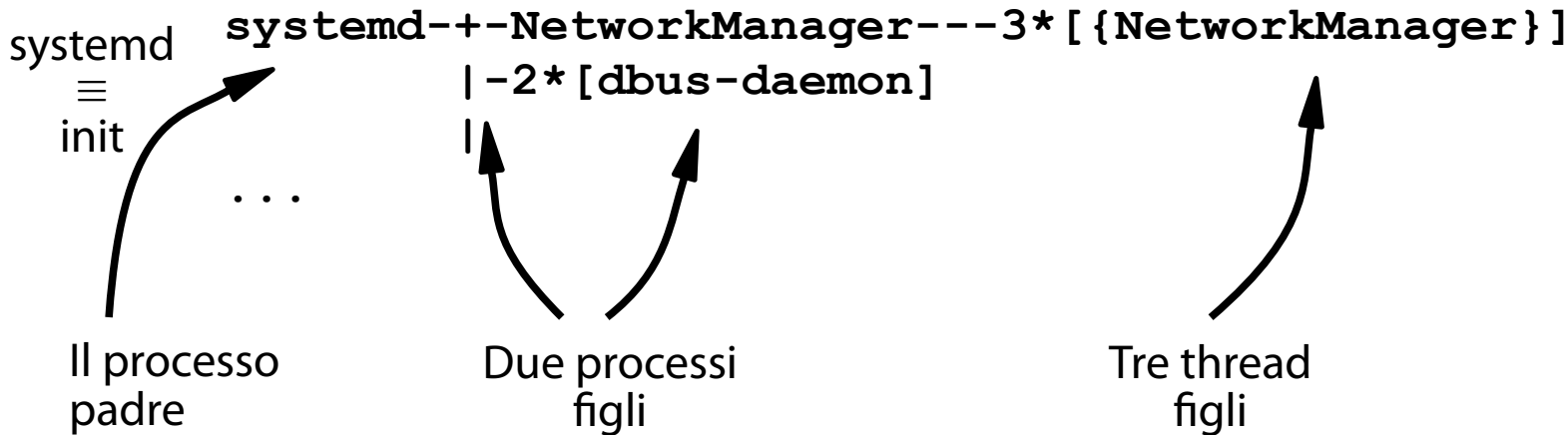
/sbin/init inizializza tutti i servizi offerti dal SO.

Visualizzazione dell'albero dei processi

(Si usa il comando **pstree**)

Il comando esterno **pstree** stampa una rappresentazione compatta dell'albero dei processi.

```
pstree | less -Mr
```



Orrore!

(L'output di `pstree` è letale per il povero studente)



Alcune opzioni notevoli di **ps tree** 1/2

(Da ricordare)

man ps tree per tutti i dettagli.

ps tree -a: stampa i nomi dei processi in esecuzione ed i relativi argomenti.

ps tree -c: stampa una rappresentazione dell'albero più esplicita.

systemd-+

| -2* [dbus-daemon]



systemd-+

| -dbus-daemon

| -dbus-daemon

Alcune opzioni notevoli di **ps tree** 2/2

(Da ricordare)

man ps tree per tutti i dettagli.

ps tree -p: mostra i PID dei processi.

ps tree -H *PID*: evidenzia in neretto il ramo dell'albero dal processo **init** al processo identificato da ***PID***.

Esercizio 5 (2 min.)

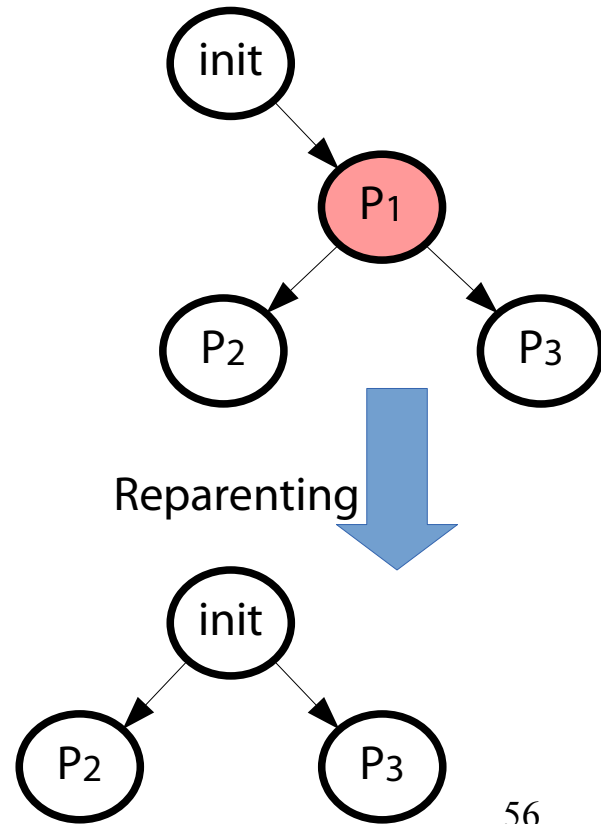
Mostrate la catena di processi da **init** ad uno qualunque dei processi **bash**.

Processi orfani e child reaping

(Se muore il padre, a chi s'attacca il figlio?)

Se un processo genitore termina, tutti i suoi figli diventano **orfani**. In tali condizioni, il nucleo opera una procedura detta **reparenting**.

Reparenting: Il processo orfano diventa figlio del processo **init**. Il reparenting serve per mantenere connesso l'albero dei processi.



COMUNICAZIONE TRAMITE SEGNALI

Scenario e interrogativi

(È possibile analizzare le relazioni di parentela fra processi?)

Scenario: si considerino diversi processi in esecuzione.

Interrogativi:

Esistono meccanismi per effettuare notifiche ad un processo?

È possibile associare azioni arbitrarie a queste notifiche?

Definizione

(Che cosa sono i segnali)

I **segnali** sono uno dei meccanismi primordiali di comunicazione fra processi.

Il processo sorgente genera (**raise**) un segnale e lo fa consegnare al processo destinatario attraverso il nucleo.

Il processo destinatario esegue una funzione di notifica, detta **gestore di segnale (signal handler)**.

Origine dei segnali

(**Asincrona** o sincrona)

Un segnale è generato in seguito ad eventi asincroni provocati dall'utente.

L'utente invia sequenze di controllo al terminale.

<CTRL>-c: interruzione definitiva di un comando.

<CTRL>-z: interruzione temporanea di un comando.

L'utente ridimensiona una finestra di terminale.

...

Origine dei segnali

(Asincrona o **sincrona**)

Alcuni eventi sono sincroni a condizioni anomale della macchina.

La CPU esegue un'istruzione malformata o illegale (divisione per zero).

La CPU prova ad accedere ad una locazione di memoria non assegnata al processo.

Identificazione dei segnali

(Identificatore intero (nucleo), stringa **SIGNOME** (utente))

Il nucleo identifica un segnale tramite un numero intero a partire da 1.

Ad ogni identificatore è associata una stringa nel formato **SIGNOME**, dove **NOME** è un nome sintetico in inglese che rappresenta il segnale.

SIGTERM, SIGKILL, SIGINT.

I primi 31 segnali sono considerati standard in tutte le piattaforme POSIX.

man 7 signal per tutti i dettagli.

Reazioni alla consegna del segnale

(Viene eseguita una funzione impostabile dall'utente)

Il processo destinatario reagisce alla ricezione di un segnale con l'esecuzione di una funzione, detta **gestore del segnale (signal handler)**.

Quando il gestore è invocato in corrispondenza della consegna, si dice che il segnale è stato gestito o acciuffato (**caught**).

Gestori standard o personalizzate

(Default → li fornisce il kernel; Personalizzati → li fornisce l'utente)

Il nucleo fornisce diversi gestori standard, che possono essere impostati per eseguire azioni comuni.

In alternativa, l'utente può impostare una propria funzione come gestore di uno specifico segnale.

Alcuni segnali non prevedono l'impostazione di un gestore personalizzato.

Gestori standard

(IGNORE: il kernel non consegna il segnale; TERMINATE: il processo termina)

IGNORE.

Il nucleo non consegna al processo alcun segnale ignorato.

Il processo non si accorge dei tentativi di consegna.

TERMINATE.

Il processo è terminato (killed).

Si parla di terminazione anomala (contrapposta alla normale che avviene tramite **exit()**).

Gestori standard

(STOPPED: il processo è bloccato; CONTINUED: il processo è ripristinato)

STOPPED.

L'esecuzione del processo è sospesa temporaneamente.

CONTINUED.

L'esecuzione del processo è ripristinata.

Gestori standard

(CORE DUMP: si salva la memoria in un file)

CORE DUMP.

Si genera una immagine della memoria virtuale del processo, e la si salva in un file. L'immagine può essere usata con un debugger per investigare un crash. Il processo viene successivamente terminato.

“Core dump”? Eh?

(Confused german shepherd is confused)

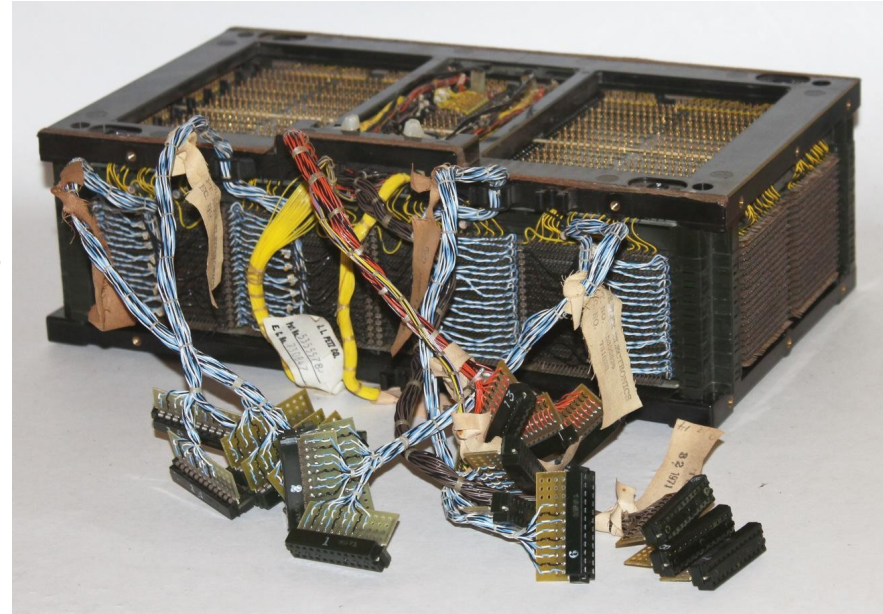


L'origine del termine "Core dump"

(Risale ai primi mainframe)

La RAM dei mainframe degli anni '60 e '70 era basata su tecnologia a **nuclei magnetici di ferrite (magnetic core memory)**.

Core dump: stampa (su carta o su file) dei valori dei registri e della memoria di un processo, in seguito ad un crash.



Memoria a nucleo magnetico dell'IBM 360 (1970 ca.)

Gestori di default

(Quelli attivi se non si specifica un gestore di segnale)

Se non si specifica un gestore di segnale, un processo esegue il suo gestore di default.

SIGINT, SIGTERM: TERMINATE.

SIGSEGV: CORE DUMP.

...

man 7 signal per tutti i dettagli.

Alcuni dei segnali più comuni

(man 7 signal per tutti i dettagli)

SIGTERM (uscita anomala con possibilità di cleanup).

ID: 15.

Azione di default: **TERMINATE**.

Gestibile con handler: sì.

SIGKILL (uscita anomala non gestibile).

ID: 9.

Azione di default: **TERMINATE**.

Gestibile con handler: no.

Alcuni dei segnali più comuni

(`man 7 signal` per tutti i dettagli)

SIGINT (interruzione definitiva).

ID: 2.

Azione di default: **TERMINATE**.

Gestibile con handler: sì.

SIGCHLD (un processo figlio ha cambiato stato).

ID: 17.

Azione di default: **IGNORE**.

Gestibile con handler: sì.

Alcuni dei segnali più comuni

(man 7 signal per tutti i dettagli)

SIGSTOP (sospensione temporanea).

ID: 19.

Azione di default: **STOPPED**.

Gestibile con handler: no.

SIGCONT (ripristino).

ID: 18.

Azione di default: **CONTINUED**.

Gestibile con handler: sì.

Alcuni dei segnali più comuni

(`man 7 signal` per tutti i dettagli)

SIGQUIT (terminazione con core dump, CTRL-\\).

ID: 3.

Azione di default: **CORE DUMP**.

Gestibile con handler: sì.

SIGSEGV (accesso illegale alla memoria).

ID: 11.

Azione di default: **CORE DUMP**.

Gestibile con handler: sì.

Invio di un segnale ad un processo

(Si usa il comando esterno `/bin/kill`)

Il comando esterno `/bin/kill` invia segnali a processi.

`/bin/kill [options] PID...`

I processi sono identificati tramite il loro PID. I segnali possono essere specificati tramite identificatore oppure tramite il loro nome. Se non si specifica un segnale, si assume **SIGTERM**.

Un esempio concreto

(Avvio e terminazione del comando `sleep 1000`)

Si esegua il comando seguente, che si blocca per mille secondi:

```
sleep 1000
```

Su un altro terminale, si identifichi il PID del processo:

```
pgrep sleep  
1234
```

Si invii il segnale di default (SIGTERM) al processo, che lo termina:

```
/bin/kill 1234
```

Elenco dei segnali

(Sai usare l'opzione `-l` o `-L` del comando `/bin/kill`)

L'opzione `-l` del comando `/bin/kill` stampa l'elenco dei segnali disponibili:

```
/bin/kill -l
```

L'elenco è alquanto spartano e mostra solo i nomi brevi dei segnali senza il prefisso **SIG**.

Un elenco migliore (contenente identificatori e nomi brevi dei segnali) è ottenibile con l'opzione `-L`:

```
/bin/kill -L
```

Una osservazione importante

Esiste anche il builtin BASH `kill`)

Si esegua il comando seguente per identificare tutte le varianti di `kill`:

```
type -a kill
```

Si dovrebbe ottenere l'output seguente:

```
kill è un comando interno di shell
```

```
kill è /usr/bin/kill
```

```
kill è /bin/kill
```

Il comando `kill` è anche un builtin di shell! Occhio a non confondersi!

Esercizio 6 (1 min.)

Avviate il comando **top**.

Terminate il processo appena creato, inviandogli un segnale opportuno.

Il comando **pkill**

(Introduce diverse migliorie rispetto a **/bin/kill**)

/bin/kill seleziona i processi solamente in base al PID.

Il comando esterno **/usr/bin/pkill** può selezionare il processo in base al suo nome o a diversi filtri di ricerca.

- Appartenenza ad uno specifico utente o gruppo.

- Scelta del processo più recente o più anziano (secondo la data di creazione).

- Uso di uno specifico terminale.

Invio di un segnale specifico

(Si usa l'opzione `--signal` di `pkill`)

Per inviare il segnale **SIGNAL** ad un processo identificato da **NOME**, si può usare l'opzione `--signal` di `pkill`:

```
pkill --signal SIGNAL NOME
```

Una variante piuttosto comune consiste nell'uso dell'opzione `-SIGNAL`:

```
pkill -SIGNAL NOME
```

Un esempio concreto

(Avvio e terminazione di `sleep 1000`)

Su un terminale si esegua il comando seguente:

```
sleep 1000
```

Si invii il segnale di terminazione al processo con uno dei due comandi seguenti equivalenti:

```
pkill --signal TERM sleep
```

```
pkill -TERM sleep
```

Esercizio 7 (3 min.)

Eseguite un emulatore di terminale.

Leggete la pagina di manuale di **pskill** e individuate un modo per terminare l'istanza più recente del processo **bash**.

COMPOSIZIONE

Scenario e interrogativi

(È possibile combinare applicazioni di base per creare applicazioni complesse?)

Scenario: un utente vuole costruire applicazioni più complesse a partire da applicazioni di base.

Interrogativi:

Esistono meccanismi per combinare più applicazioni?

Pipe

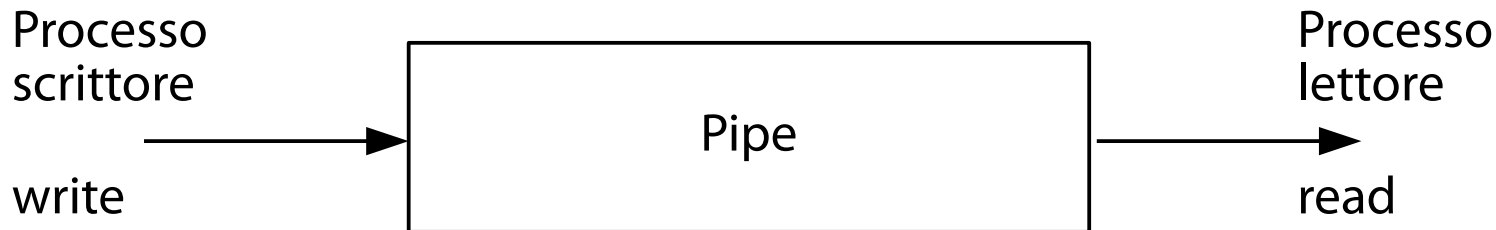
(Un processo scrittore invia dati ad un processo lettore)

La **pipe (tubazione)** è un meccanismo di comunicazione unidirezionale (half-duplex) e sequenziale fra un processo scrittore e un processo lettore.

Comunicazione stream: lo scrittore ed il lettore possono scrivere e leggere, indipendentemente, messaggi di lunghezza arbitraria.

→ Blocco in caso di scrittura su pipe “piena”.

→ Blocco in caso di lettura da pipe “vuota”.



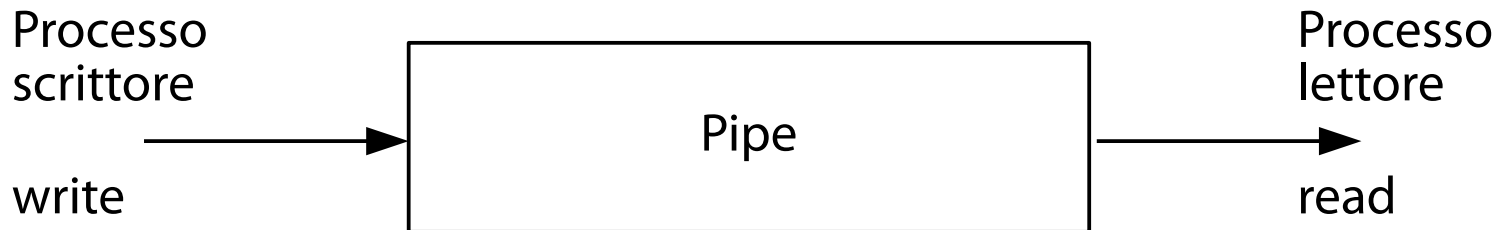
Pipe

(Un processo scrittore invia dati ad un processo lettore)

La **pipe (tubazione)** è un meccanismo di comunicazione unidirezionale (half-duplex) e sequenziale fra un processo scrittore e un processo lettore.

Comunicazione sequenziale: le informazioni sono lette nell'ordine in cui vengono scritte.

→ una volta lette, le informazioni spariscono dalla pipe e non possono più ripresentarsi (a meno che non vengano reinserite all'interno del “tubo”).



Dimensione del buffer

(64 KB)

Il buffer di una pipe ha dimensione pari a 64KB.

Le scritture piccole ($\leq 4\text{KB}$) sono eseguite in un colpo solo.

Una scrittura di dimensione maggiore di 4KB viene eseguita a porzioni di 4KB alla volta.

Semantica di terminazione

(Necessariamente complicata)

Quando un processo chiude STDIN o STDOUT, il lato corrispondente della pipe si rompe.

STDIN → ingresso.

STDOUT → uscita.

Un processo che prova a leggere da/scrivere su una pipe rotta riceve il segnale SIGPIPE, che ha come azione di default la terminazione del processo.

→ Una volta che un processo della pipe esce, tutti gli altri escono per via della propagazione dei SIGPIPE.

L'operatore |

(Implementa la pipe)

In BASH, l'operatore | implementa il meccanismo delle pipe nella forma seguente:

COMANDO₁ | COMANDO₂ | . . . | COMANDO_N

Il comando composto ora visto ha la semantica seguente.

COMANDO₁ legge lo STDIN da terminale o da file.

L'operatore |

(Implementa la pipe)

In BASH, l'operatore | implementa il meccanismo delle pipe nella forma seguente:

COMANDO₁ | COMANDO₂ | . . . | COMANDO_N

Il comando composto ora visto ha la semantica seguente.

Lo **STDOUT** di **COMANDO_i** punta allo **STDIN** di **COMANDO_{i+1}**. Pertanto, **COMANDO_{i+1}** legge in input l'output di **COMANDO_i**. [$i=1, \dots, N-1$].

L'operatore |

(Implementa la pipe)

In BASH, l'operatore | implementa il meccanismo delle pipe nella forma seguente:

COMANDO₁ | COMANDO₂ | . . . | COMANDO_N

Il comando composto ora visto ha la semantica seguente.

COMANDO_N legge STDIN da **COMANDO_{N-1}** e produce un output su terminale o file.

Una osservazione importante

(L'operatore `|` lascia passare `STDOUT`, non `STDERR`!)

L'operatore `|` lascia passare solo lo `STDOUT`, non lo `STDERR`!

Si esegua il comando seguente (**nonesistente** è il nome di un file che non esiste):

```
ls . noneistente
```

Si ripeta il comando, paginando l'output di `ls`:

```
ls . noneistente | less -Mr
```

Si preme `<CTRL>-l` per rigenerare l'output.

Si dovrebbe vedere solo lo `STDOUT` di `ls`, non lo `STDERR`.

Passaggio di STDOUT e STDERR

(Si copia STDERR in STDOUT prima di usare la pipe)

Per far passare il contenuto dei canali STDOUT e STDERR, si può copiare STDERR in STDOUT prima di usare la pipe.

```
ls . nonesistente 2>&1 | less -Mr
```

In alternativa si può usare la sintassi breve | &, sinonimo di 2>&1 |:

```
ls . nonesistente |& less -Mr
```

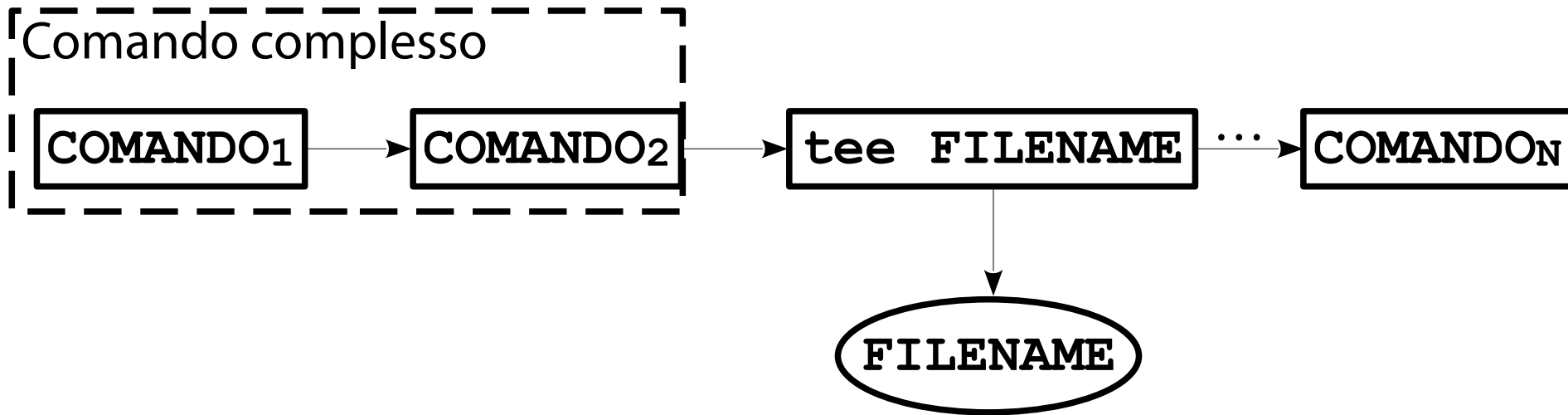
Esercizio 8 (2 min.)

Visualizzate i trenta file più grandi nell'intero file system.

Tubazione a T

(Si usa il comando **tee**)

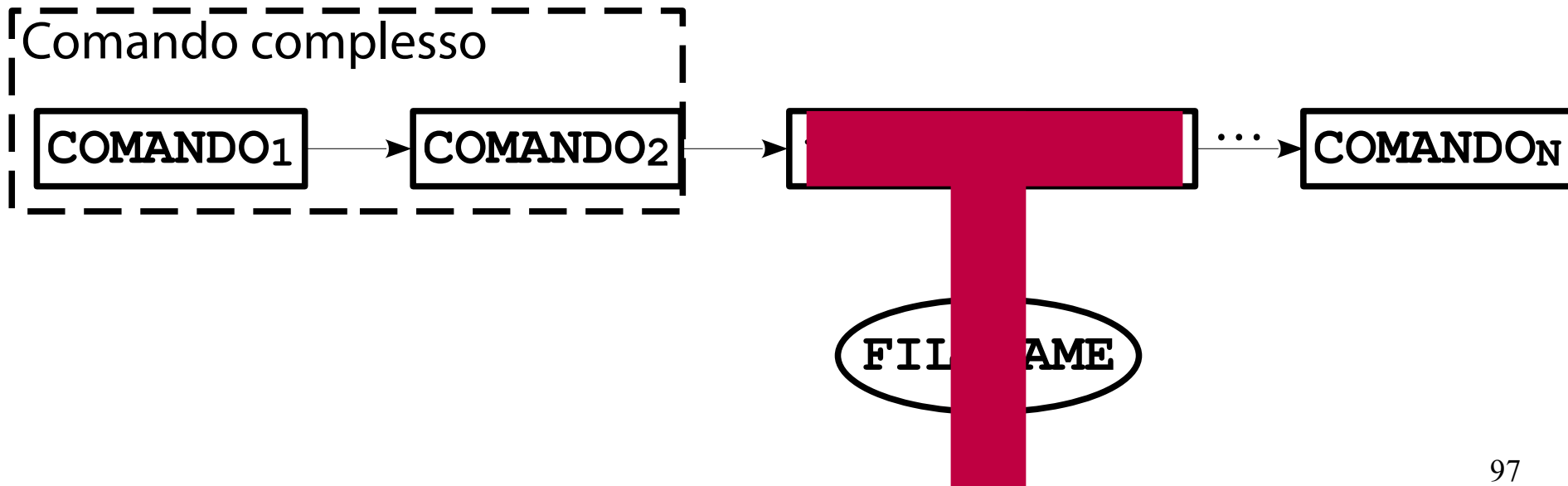
Il comando **tee** introduce un meccanismo di tubazione a T. Lo STDIN è scritto su STDOUT e su un file.



Tubazione a T

(Per chi se lo stesse chiedendo)

Il comando **tee** introduce un meccanismo di tubazione a T. Lo STDIN è scritto su STDOUT e su un file.



Memorizzazione risultati intermedi pipe

(Uno degli usi più comuni di **tee**)

Il comando **tee** può essere usato per salvare su file gli STDOUT di tutti i comandi di una pipe:

```
C1 | tee out1.txt | C2 | tee out2.txt | ...
```

Ciò può risultare utile per verificare la correttezza degli output intermedi.

Un esempio concreto

(Individuazione dei dieci file più grandi in un file system)

Si esegua il comando seguente:

```
find / -printf "%s %p\n" | tee find.txt  
| sort -nrk1 | tee sort.txt | head
```

Il comando scrive nei file **find.txt** e **sort.txt** l'output dei comandi **find** e **sort**, rispettivamente. I file possono essere ispezionati per attività di debugging.

Una osservazione importante

(I file di output intermedi potrebbero essere troncati)

Non appena un processo termina (correttamente o in maniera anomala), gli altri processi lo seguono a ruota (per via della propagazione dei SIGPIPE).

A seconda di come è progettata la pipe, alcuni file di output intermedi potrebbero essere non completi.

Nel caso di studio appena visto:

find deve terminare, altrimenti **sort** non può ordinare;
non appena **head** riceve dieci righe, termina (l'output di **sort** sarà verosimilmente troncato).

Esercizio 9 (3 min.)

Il comando seguente estrae le colonne “username” e “user ID” di `/etc/passwd` e stampa il valore delle ultime dieci righe:

```
cut -f1,4 /etc/passwd | head -10
```

Verificate la correttezza del comando.

Named pipe

(Pipe su un file)

Una **named pipe** (o **FIFO**) è quasi identica ad una pipe. L'unica differenza è che gli STDIN e STDOUT dei processi si riferiscono ad un file speciale su disco.

Tecnicamente, il file speciale ha lunghezza zero ed ha una propria struttura di metadati (inode).

Il file speciale è riconoscibile dal primo carattere **p** nell'output del comando **ls -l**.

Creazione di una named pipe

(Si usa il comando **mkfifo**)

Il comando **mkfifo** crea un file speciale per una named pipe:

```
mkfifo FILENAME
```

Si osservi più da vicino il file creato:

```
ls -l
```

Il file è marcato **p** e ha lunghezza pari a zero.

Uno streaming server dei poveri

(Con semplici strumenti da linea di comando)

Si provi a scrivere uno streaming server spartano usando esclusivamente strumenti da linea di comando.

Cosa serve?

- Uno strumento per scaricare brani da Internet.

- Uno strumento per riprodurre brani.

- Una named pipe che connette lo scaricatore di brani con il riproduttore musicale.

I comandi coinvolti

(**mkfifo**, **wget**, **mpv**)

Strumento di comunicazione:

named pipe creata con il comando **mkfifo**.

Client HTTP (per il download di brani):

wget (pacchetto software **wget**).

Riproduttore multimediale “streaming”:

mpv (pacchetto software **mpv**).

Creazione della named pipe

(Si usa il comando **mkfifo**)

Si crei una named pipe, buffer intermedio fra HTTP client e riproduttore multimediale.

```
mkfifo buffer
```

Download del file multimediale

(Si usa il comando **wget**)

Si scarichi il filmato (o il brano audio) che si intende ascoltare, riversandolo nella named pipe.

```
wget -q -O buffer
```

```
http://weblab.ing.unimo.it/people/andreolini/file-audio.ogg
```

Il comando **wget**:

- prova a scrivere sulla named pipe;

- si blocca se non esiste un processo lettore (o se esiste e non drena il buffer);

- continua a riversare buffer nella named pipe altrimenti.

Riproduzione del file multimediale

(Si usa il comando **mpv**)

Si può riprodurre il file multimediale dalla named pipe con il comando **mpv**:

```
mpv buffer
```

Se tutto va a buon fine, dovrebbe iniziare la riproduzione del file.

Esercizio 10 (2 min.)

Visualizzate i trenta file più grandi nell'intero file system. Usate le named pipe per la comunicazione tra processi. Spiegate il blocco dei vari processi coinvolti.

JOB CONTROL

Scenario e interrogativi

(Un po' più complessi rispetto al solito)

Scenario: in un SO moderno, l'interprete dei comandi legge comandi da un file o, più spesso, da un dispositivo terminale;
crea i processi corrispettivi, caricando in memoria le immagini dei comandi coinvolti.

Spesso il comando immesso è complesso, costituito da una pipe di N comandi.

Esempio classico (che cosa fa?):

```
(cd /src/dir && tar cf - .) | ( cd /dst/dir && tar xf -)
```

Scenario e interrogativi

(Un po' più complessi rispetto al solito)

Scenario: in un SO moderno, l'interprete dei comandi legge comandi da un file o, più spesso, da un dispositivo terminale;
crea i processi corrispettivi, caricando in memoria le immagini dei comandi coinvolti.

Ogni comando (semplice o complesso) blocca il terminale.

Lanciato un comando, non se ne può lanciare un altro fino al termine del precedente.

No, lanciare un nuovo terminale non vale :-)

Scenario e interrogativi

(Un po' più complessi rispetto al solito)

Scenario: in un SO moderno, l'interprete dei comandi legge comandi da un file o, più spesso, da un dispositivo terminale;
crea i processi corrispettivi, caricando in memoria le immagini dei comandi coinvolti.

Se un processo termina, tutti gli altri coinvolti in un comando complesso terminano non appena provano ad interagirci (SIGPIPE).

Scenario e interrogativi

(Un po' più complessi rispetto al solito)

Interrogativi:

È possibile rappresentare in maniera efficiente il gruppo di processi costituente un comando complesso?

In questo modo, si potrebbero terminare tutti subito.

Esistono meccanismi per lanciare applicazioni anche complesse senza “bloccare” il terminale?

In questo modo, il terminale sarebbe riutilizzabile.

Job control

(Permette di gestire più comandi complessi in uno stesso terminale)

Il **job control** è un insieme di comandi e meccanismi che consente di gestire l'esecuzione simultanea di più comandi complessi sullo stesso terminale.

Il job control fornisce strumenti di gestione del gruppo di processi associati ad un comando (semplice o complesso).

- Identificazione.

- Esecuzione "agganciata" ad un terminale.

- Esecuzione "sganciata" dal terminale.

- Sospensione, ripristino.

- Interruzione, terminazione.

Motivazioni del job control

(Gestire più applicazioni interattive su un “unico” schermo nei vecchi sistemi)

L'esigenza del job control nasce nei primi sistemi UNIX, dotati di terminali seriali.

- Il terminale seriale ha un solo schermo testuale.

- Un unico comando alla volta può essere interattivo con il terminale.

- Necessità “alternare” più comandi al terminale.

Pipeline

(Rappresentazione generale di un comando)

Il generico comando lanciabile dall'interprete dei comandi BASH prende il nome di **pipeline** e ha la forma seguente:

$$C_1 \ [\ [\ | \ | \ &] \ C_2 \ \dots \ [\ | \ | \ &] \ C_N \]$$

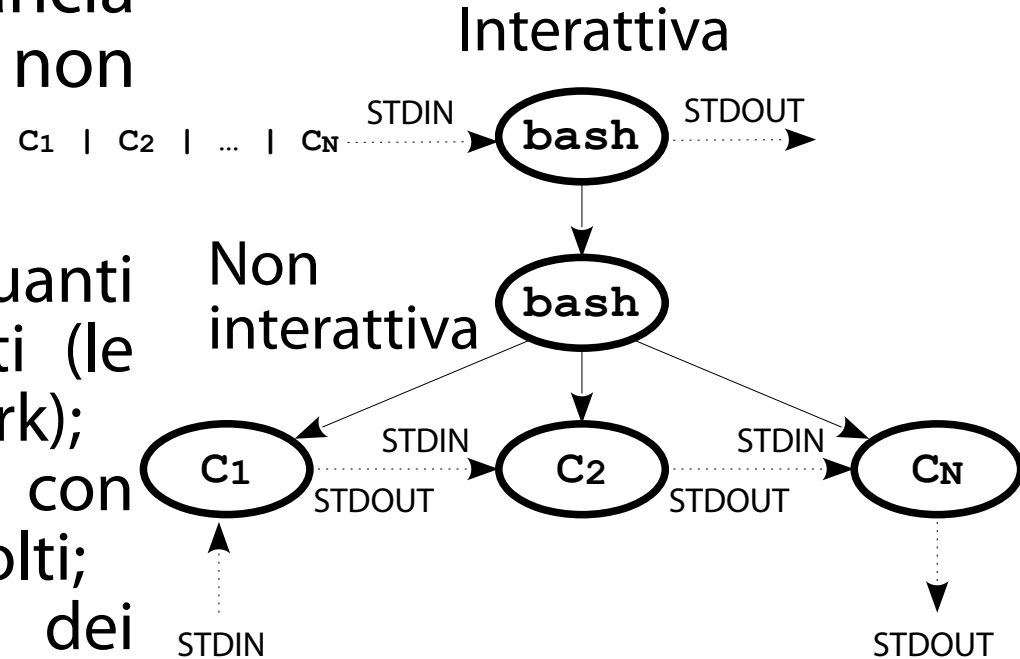
In parole povere, si possono eseguire uno o più comandi separati dagli operatori `|` o `&`.

Pipeline nell'albero dei processi

(Rappresentazione generale di un comando)

Per ogni pipeline, BASH lancia una nuova istanza di BASH non interattiva che:

- crea le pipe necessarie;
- crea tanti processi quanti sono i comandi coinvolti (le pipe sono ereditate dal fork);
- sostituisce le immagini con quelle dei comandi coinvolti;
- attende la terminazione dei processi coinvolti.



Gruppo di processi

(Raccoglie tutti i processi in una pipeline)

Un **gruppo di processi** (**process group** o **job**) è l'insieme dei processi che partecipano ad una pipeline.

L'ultimo processo della pipeline è detto **process group leader**; il suo stato di uscita è ritornato a BASH ed è rappresentativo dell'intera pipeline.

Il gruppo di processi è identificato da un **identificatore di gruppo dei processi** (**Process Group ID** o **PGID**). Il PGID è il PID del process group leader.

Il gruppo di processi è definito fino a quando almeno un processo della pipeline è ancora attivo.

Ogni processo fa parte di un gruppo di processi.

Sessione

(Raccoglie tutti i gruppi di processi che condividono un terminale)

Il SO GNU/Linux raggruppa processi in **sessioni**.

Sessione: è un insieme di gruppi di processi che condivide lo stesso dispositivo terminale.

Una sessione è creata nelle occasioni seguenti:

- login testuale;

- login grafico;

- avvio di un server;

- su richiesta esplicita di una applicazione (non considerato nella presente introduzione).

Evoluzione di una sessione

(Inizialmente un singolo gruppo di processi; poi, altri gruppi)

Inizialmente, una sessione consta di un unico gruppo di processi contenente un unico processo:

- interprete dei comandi, ad es. BASH (login testuale);
- gestore di login grafico, ad es. GDM (login grafico);
- il processo server di controllo, ad es. SSHD (avvio di un server).

Successivamente possono aggiungersi altri gruppi di processi:

- le pipeline eseguite tramite BASH (login testuale);
- le applicazioni grafiche (login grafico).

Session leader

(È il primo processo creato in una sessione, e la rappresenta)

Il processo lanciato per primo in una sessione è detto **session leader**.

La sessione è identificata da un **identificatore di sessione (session ID, SID)**. Il SID è il PID del session leader.

Il session leader può essere o non essere agganciato ad un dispositivo terminale.

BASH lo è (è interattivo), un server no (non è interattivo).
Se il session leader è agganciato ad un dispositivo terminale, esso viene condiviso con tutti i gruppi di processi nella sessione corrispondente e prende il nome di **terminale di controllo**.

Gruppo di processi in primo piano

(È l'unico che può leggere input dallo STDIN del terminale)

I gruppi di processi all'interno di una sessione sono divisi in due categorie.

La prima categoria è il **gruppo di processi in primo piano (foreground process group)**.

Questo gruppo è l'unico ad essere attaccato al terminale di controllo sia in ingresso (STDIN) sia in uscita (STDOUT).
Ve ne può essere solo uno.

Gruppi di processi sullo sfondo

(Non possono leggere dallo STDIN del terminale)

La seconda categoria è il **gruppo di processi sullo sfondo (background process group)**. A differenza della categoria precedente, vi possono essere più pipeline di questo tipo.

Questo gruppo non può essere attaccato allo STDIN del terminale di controllo.

Se un gruppo di processi in background prova a leggere da STDIN, riceve il segnale SIGTTIN dal nucleo e viene temporaneamente bloccato. Per ripristinarlo, occorre portare il gruppo di processi di nuovo in primo piano.

Gruppi di processi sullo sfondo

(Possono scrivere su STDOUT del terminale di controllo)

I gruppi di processi sullo sfondo possono scrivere sullo STDOUT del terminale di controllo.

Volendo, si può configurare il dispositivo terminale in modo tale da impedire l'accesso anche allo STDOUT.

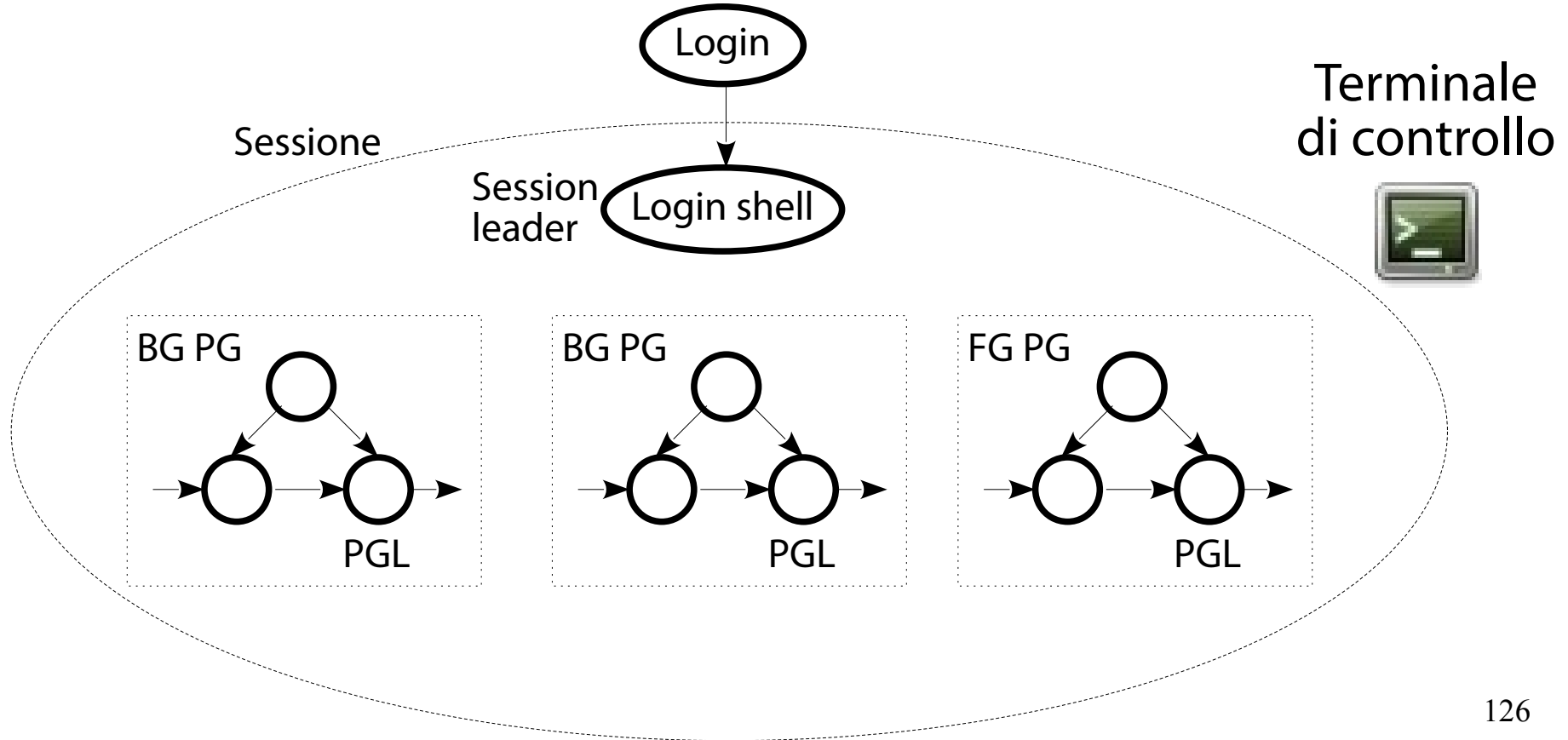
Il comando seguente attiva l'opzione **TOSTOP**:

```
stty tostop
```

A seguito di tale comando, se un gruppo di processi sullo sfondo prova a scrivere su STDOUT, riceve il segnale SIGTTOU dal nucleo e viene temporaneamente bloccato. Per ripristinarlo, occorre portare il gruppo di processi di nuovo in primo piano.

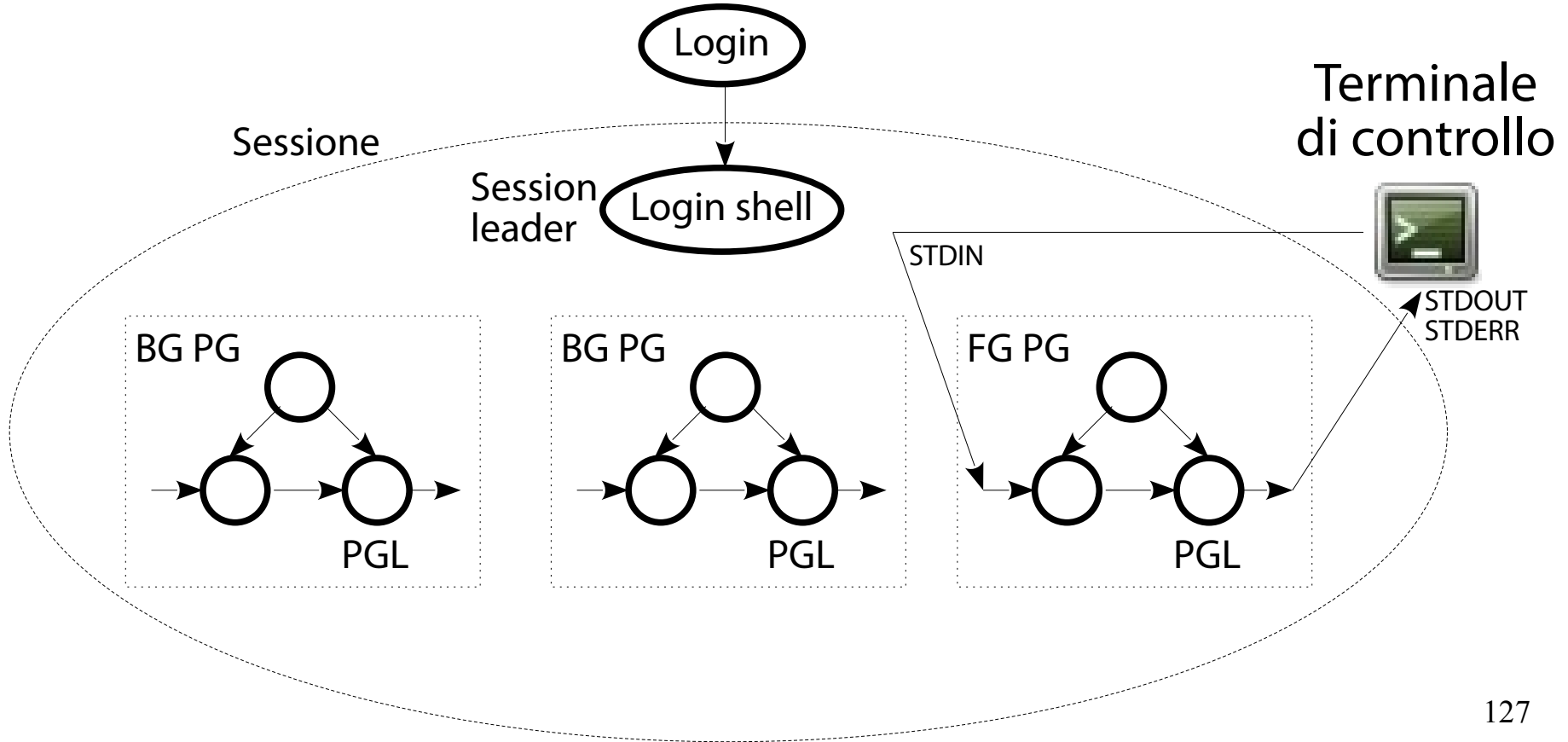
Modello di job control

(Caso di studio "login testuale")



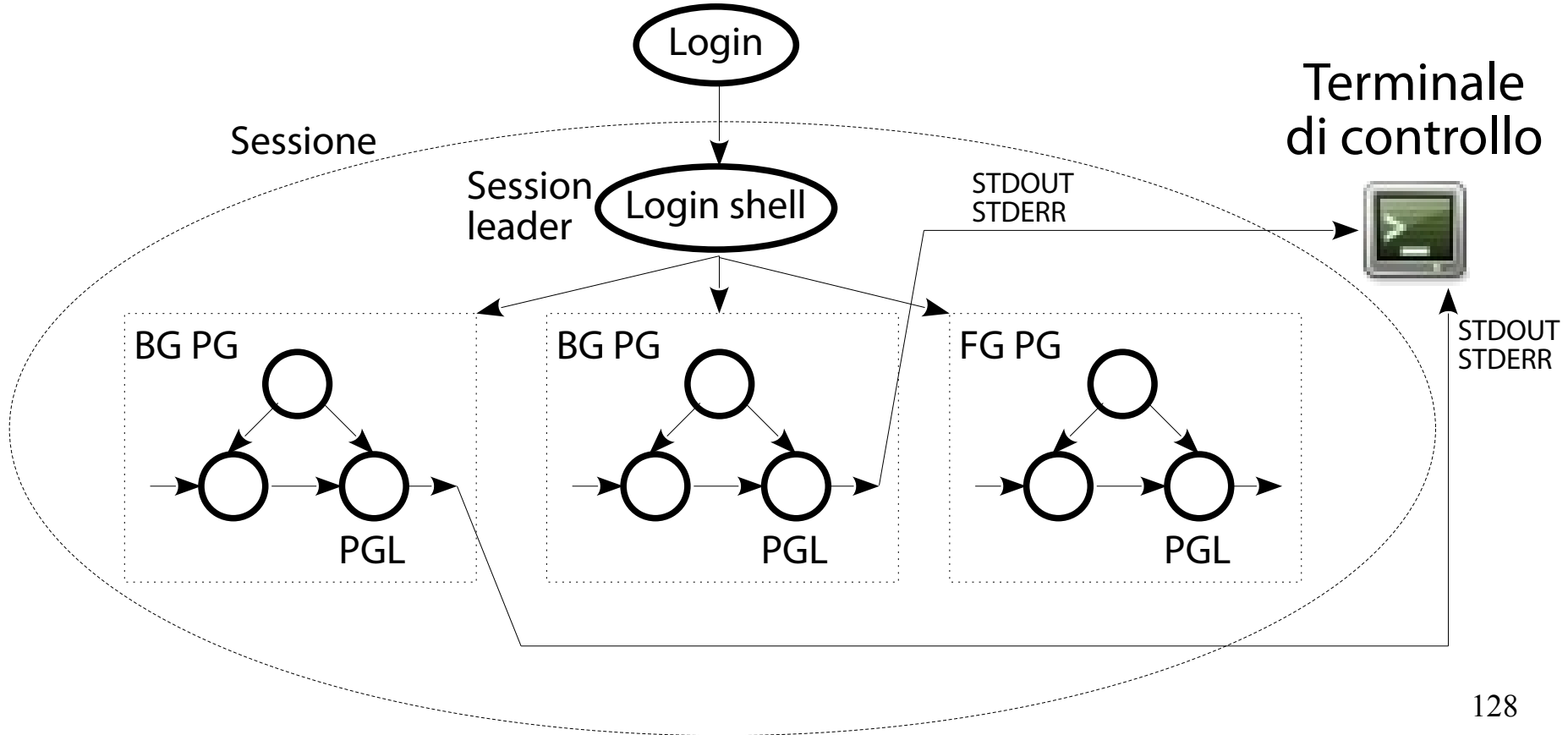
Modello di job control

(Il gruppo in primo piano ha controllo completo del terminale)



Modello di job control

(Un gruppo sullo sfondo ha, nel caso migliore, controllo su STDOUT e STDERR)



Un esempio concreto

(Gestione di diverse pipeline con gli strumenti di job control)

A puro titolo di esempio, si eseguono alcune pipeline e si si eseguono le tipiche operazioni di gestione dei job. Tali operazioni sono implementate tramite comandi interni di BASH e discusse nelle slide seguenti.

Esecuzione job in primo piano

(Questo è facile)

Innanzitutto si esegue la seguente, semplicissima pipeline in primo piano:

```
find / -printf "%s %p\n" | sort -nrk 1 | head
```

Sospensione temporanea

(Si usa la sequenza di tasti <CTRL>-z)

La sequenza di tasti <CTRL>-z sospende temporaneamente l'intero gruppo dei processi.

La sospensione avviene inviando il segnale SIGTSTP a tutti i processi nel gruppo. L'effetto è quello di congelare i processi, i quali non possono continuare ad eseguire.

Il job ID

(Un altro identificatore del gruppo di processi, usato da BASH)

Non appena si interrompe la pipeline, BASH stampa il messaggio seguente:




<code>[1]+</code>	<code>Fermato</code>	<code>find / -printf ...</code>
		
Job ID	Stato della pipeline	Pipeline eseguita

Il numero tra parentesi quadre è il **job ID**, un intero usato da BASH per rappresentare il gruppo di processi. Esso è in corrispondenza 1-1 con il PGID.

Sospensione temporanea

(Si usa la sequenza di tasti <CTRL>-z)

Non appena si interrompe la pipeline, BASH stampa il messaggio seguente:

<code>[1]+</code>	<code>Fermato</code>	<code>find / -printf ...</code>
		
Job ID	Stato della pipeline	Pipeline eseguita

Il carattere `+` evidenzia il **job corrente**, ovvero l'ultimo job:

interrotto con <CTRL>-z mentre era in primo piano;

OPPURE

lanciato sullo sfondo (operatore `&` discusso in seguito).

Elenco dei job

(Si usa il comando interno `jobs`)

Il comando interno `jobs` permette di elencare i job lanciati:

`jobs`

L'opzione `-l` stampa anche i PID di tutti i processi della pipeline:

```
jobs -l
[1]+  2079 Fermato      find / -printf "%s %p\n"
      2080             | sort -nrk 1
      2081             | head
```

Esecuzione job sullo sfondo

(Questo è facile)


È possibile eseguire una pipeline sullo sfondo (**background**) appendendo l'operatore **&** al termine della stessa.

La pipeline seguente esegue sullo sfondo:

```
sleep 7200 &
```

BASH stampa il messaggio seguente:

```
[2] 2231
```



Job ID Process group ID

Elenco dei job

(Si usa il comando interno `jobs`)

Si stampa nuovamente l'elenco dei job:

```
jobs -l
[1]+  2079 Fermato      find / -printf "%s %p\n"
      2080              | sort -nrk 1
      2081              | head
[2]-  2231 In esec.     sleep 7200 &
```

Risulta un nuovo job in esecuzione.

Tale job è marcato con il carattere `-`.

Il job precedente

(Insieme al job corrente consente un'alternanza tra job recenti)

Il carattere `-` identifica il **job precedente** seguendo un algoritmo piuttosto cervellotico.

L'idea è quella di fornire due scorciatoie di BASH (`+` e `-`) con le quali l'utente può rapidamente alternare in primo piano i due job più "recenti".

Gli studenti più volenterosi possono studiare la funzione `set_current_job()`, definita nel file `jobs.c` dell'albero sorgente di BASH.

Ripristino job 1 in primo piano

(Si usa il comando interno **fg**)

Per ripristinare un job in primo piano si può eseguire il comando interno **fg**.

Un job può essere rappresentato in modi diversi:

%JOB_ID

%COMANDO (o sottostringa di **COMANDO**)

%+ 0 %

%-

Per ripristinare il job 1 si può eseguire, ad esempio:

fg %1

Sospensione del job 1

(Si usa la sequenza di tasti <CTRL>-z)

Dopo il ripristino, il job 1 si appropria integralmente del terminale di controllo.

- Può usare STDOUT.

- Può usare STDIN.

- Non si possono eseguire altre pipeline.

Si interrompa nuovamente il job con <CTRL>-z.

Ripristino job 1 sullo sfondo

(Si usa il comando interno **bg**)

Per ripristinare un job sullo sfondo si può eseguire il comando interno **bg**.

Un job può essere rappresentato in modi diversi:

%JOB_ID

%COMANDO (o sottostringa di **COMANDO**)

%+ 0 %

%-

Per ripristinare il job 1 si può eseguire, ad esempio:

bg %1

Elenco dei job

(Si usa il comando interno `jobs`)

Si attende un po' e si stampa nuovamente l'elenco dei job:

```
jobs -l
[1]-  2079  Uscita 1      find / -printf "%s %p\n"
      2080  Pipe int.   |  sort -nrk 1
      2081  Complet.   |  head
[2]+  2231  In esec.    sleep 7200 &
```

La prima pipeline è terminata, e lo stato di uscita dei vari processi è dettagliato accanto al PID.

Esercizio 11 (2 min.)

Lanciate sullo sfondo un comando che legge un valore da terminale e lo memorizza nella variabile **a**.

Visualizzate l'elenco dei job.

Notate qualcosa di strano?

Come è possibile portare a termine l'operazione?

BASH ED ESECUZIONE DEI PROCESSI

Scenario e interrogativi

(Quali altri strumenti sono disponibili per l'esecuzione dei processi BASH?)

Scenario: l'utente vuole scoprire altri strumenti a disposizione di BASH per l'esecuzione dei processi.

Interrogativi:

Quali sono tali strumenti? Come funzionano?

Sostituzione di comando

(Si usa l'operatore `$ ()`)

L'operatore `$ (COMMAND)` introduce una espansione di parametro nota con il nome di **sostituzione di comando** (**command substitution**).

Viene eseguito il comando `COMMAND`; il suo output è sostituito all'espressione `$ (COMMAND)`.

Ad esempio, per memorizzare in una variabile intera `a` il numero di linee di `/etc/passwd`:

```
declare -i a
```

```
a=$(wc -l /etc/passwd | cut -f1 -d" ")
```

Esercizio 12 (2 min.)

Cercate un comando che stampa una sequenza di numeri.

Usate tale comando per implementare un ciclo for che stampa i numeri da 1 a 100.

Sostituzione di processo

(Si usano gli operatori `< ()` e `> ()`)

Gli operatori `< (COMMAND)` e `> (COMMAND)` introducono una espansione di parametro nota con il nome di **sostituzione di processo** (**process substitution**).

Per ogni istanza di tale operatore in un comando, BASH crea al volo un file di nome `/dev/fd/N` (`N` intero). Questo file diventa STDIN o STDOUT di `COMMAND`. Viene eseguito il comando `COMMAND`, ed il suo input/output fluisce in `/dev/fd/N`.

Sostituzione di processo < ()

(L'output di un comando è presentato ad un altro comando come file)

Se si usa **< (COMMAND)**, l'output di **COMMAND** viene fatto vedere come un file al comando che invoca **< ()**.

Ad esempio, per eseguire (senza output intermedi!) la seguente trasformazione su **/etc/passwd**:

stampa una colonna di interi crescenti, la prima e l'ultima colonna di **/etc/passwd**;

limita l'output alle prime dieci righe di **/etc/passwd**;

si può eseguire il comando seguente (su una riga sola):

```
paste <(seq 1 10)
      <(cut -f1 -d":" /etc/passwd | head)
      <(cut -f7 -d":" /etc/passwd | head)
```

La (triste) alternativa a <()

(Uso di file di output intermedi)

L'alternativa all'uso di <() consiste nell'uso di file intermedi per la memorizzazione temporanea dell'output dei comandi intermedi.

```
seq 1 10 > out1.txt
```

```
cut -f1 -d":" /etc/passwd > out2.txt
```

```
cut -f7 -d":" /etc/passwd > out3.txt
```

```
paste out1.txt out2.txt out3.txt
```

```
rm out1.txt out2.txt out3.txt
```

Sostituzione di processo > ()

(L'output di un comando è presentato ad un altro comando come file)

Se si usa **>(COMMAND)**, l'output scritto su un file di nome **/dev/fd/N** dal comando che usa **>()** viene fatto vedere come input (STDIN) al comando **COMMAND**.

Ad esempio, per far elaborare al volo da più programmi il file **/etc/passwd** si può eseguire il comando (su una riga):

```
tee < /etc/passwd  
    >(COMANDO1)  
    >(COMANDO2)  
    ...
```

La (triste) alternativa a >()

(Uso di file di output intermedi)

L'alternativa all'uso di >() consiste nell'uso di file intermedi per la memorizzazione temporanea dell'output dei comandi intermedi.

```
tee < /etc/passwd out.txt
```

```
COMANDO1 < out.txt
```

```
COMANDO2 < out.txt
```

```
...
```

```
rm out.txt
```

Esercizio 13 (4 min.)

Il comando **diff** confronta due file e stampa una rappresentazione efficiente delle loro differenze.

A partire da questa informazione, individuate un metodo per trovare le differenze nei contenuti di due pagine Web distinte.

Sostituzione di immagini

(Si usa il comando interno **exec**)

Il comando interno **exec** (già usato in precedenza per gestire i descrittori di file) consente anche di sostituire l'immagine del processo corrente con quella di un nuovo comando.

exec [opzioni] COMANDO

ATTENZIONE! Il comando **exec** non crea una nuova istanza di BASH, bensì sostituisce quella esistente interattiva. Il comando seguente termina anche l'emulatore dopo cinque secondi di attesa:

exec sleep 5

FORK BOMB

Scenario e interrogativi

(È possibile per un utente malizioso piantare un SO UNIX?)

Scenario: un utente malizioso vuole provare a negare il servizio agli altri utenti, lanciando un processo che crea ricorsivamente processi figli per sempre.

La sua ipotesi è la seguente: lanciando processi uno dietro l'altro, prima o dopo il SO esaurisce il numero di descrittori di processo disponibili (che è finito) e non riesce più a far partire un'applicazione.

Scenario e interrogativi

(È possibile per un utente malizioso piantare un SO UNIX?)

Interrogativi:

Riesce l'utente a scrivere un siffatto programma ed a lanciarlo?

È possibile mitigare (se non annullare) gli effetti negativi di una tale negazione di servizio?

(Ken would disapprove that)

Una **fork bomb** è un processo che provoca la creazione di un numero spropositato di figli.



Le conseguenze di una fork bomb

(Letali per il SO)

La fork bomb è un meccanismo rozzo ma efficace di **negazione del servizio** (**Denial of Service, DoS**).

Ogni processo consuma risorse e contribuisce al rallentamento della macchina.

Il nucleo pone un limite superiore al numero di processi eseguibili. Raggiunto quel limite, nessuna applicazione è più in grado di partire.

→ Il SO si pianta; l'utente non è più in grado di interagire con esso!

Un esempio di fork bomb in BASH

(Più che un programma sembra l'output di una manata sulla tastiera)

Che ci crediate o no, il seguente programma in BASH mette in ginocchio il vostro SO. In pochi secondi.

Provare per credere.

BACKUP YOUR DATA FIRST.

```
: ( ) { : | : & } ; :
```



*Denis "Jaromil" Rojo
(1977-)*

*Programmatore
Attivista Free Software
L'autore del comando*

Vince is amazed

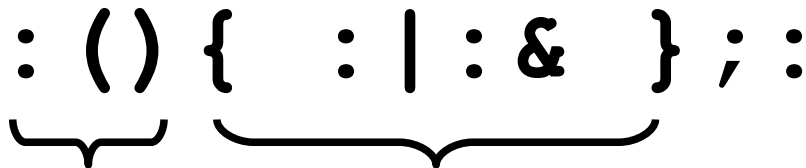
(You should be, too)



Come funziona?

(":" è un nome insolito, ma esteticamente bello a vedersi; dà simmetria)

`:` `()` `{` `:` `|` `:` `&` `}` `;` `:`



BASH

Si definisce una
funzione di BASH
dal nome :.

Questo è il
corpo della
funzione.

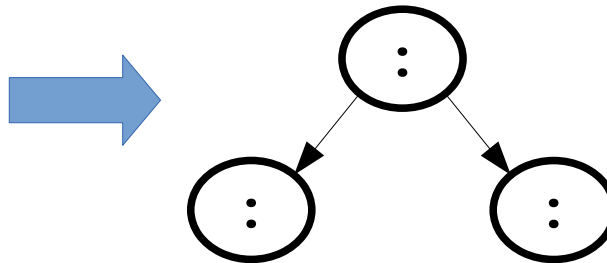
Come funziona?

(OK, quindi un processo ne genera altri due)

: () { : | : & } ; :

BASH

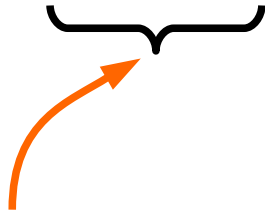
La funzione : richiama
due comandi in pipe.
Essa produce la struttura
di processi illustrata
qui a fianco.



Come funziona?

(la riproduzione avviene all'infinito; la tabella dei processi si attoppa)

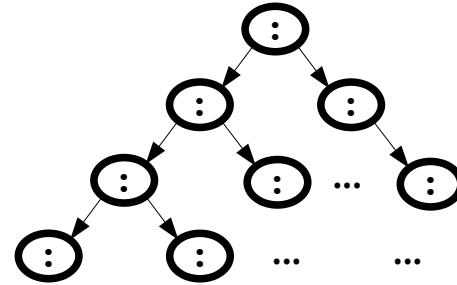
: () { : | : & } ; :



Poiché l'invocazione è ricorsiva, si genera un albero binario completo di processi.

Prima o poi si raggiunge il numero massimo di processi in esecuzione.

BASH



Come funziona?

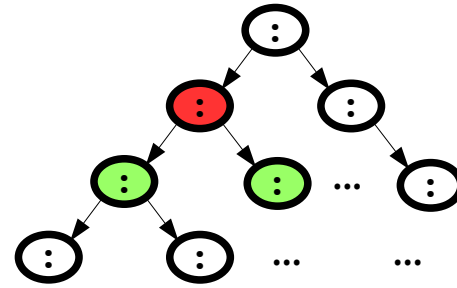
(Ma porca p...)

: () { : | : & } ; :

L'esecuzione in background della pipe ha un effetto collaterale simpaticissimo.

Se un processo padre BASH è terminato, i suoi processi figli lanciati in background non escono!

BASH




➡ Non basta terminare il processo padre per terminare la fork bomb. È necessario terminare tutti i processi. Tuttavia, il nucleo è più veloce a creare processi che a terminarli...

Come funziona?

(Questo era facile)

: () { : | : & } ; :



BASH



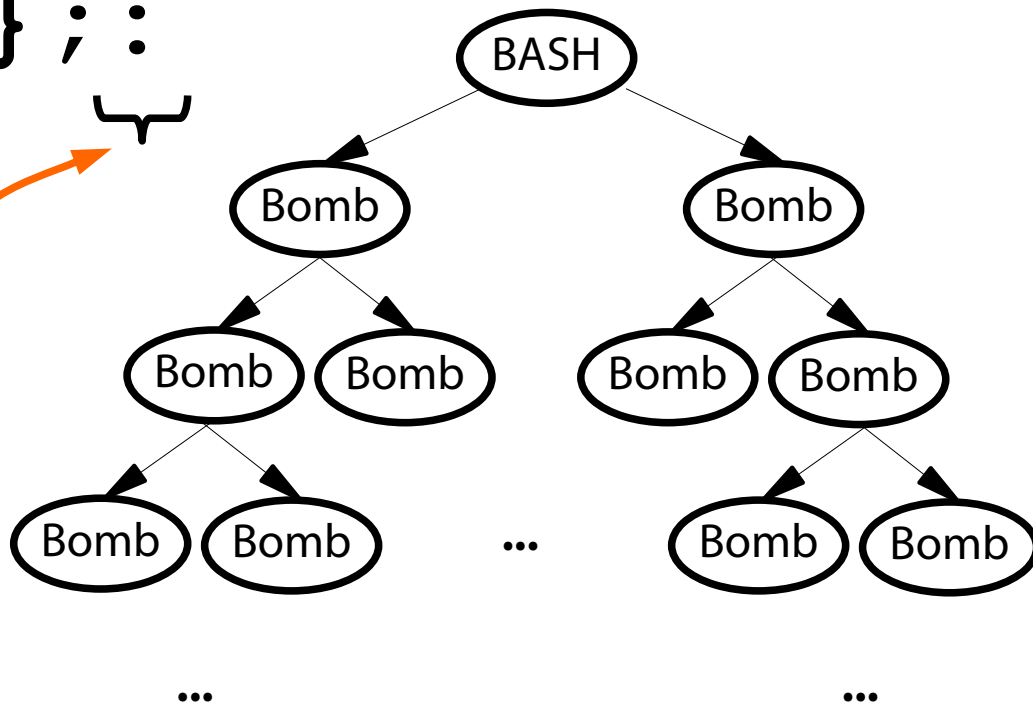
Il carattere ; è il separatore
dei comandi in BASH.

Come funziona?

(Pure questo era facile)

: () { : | : & } ; :

Viene invocata la funzione :.
Si aprono le danze.



Come proteggersi dalla fork bomb?

(Bella domanda!)

È possibile proteggersi dalla fork bomb?

Sì, è possibile.

Come?

Lo scoprirete in una delle prossime ~~puntate~~ lezioni.

つづく