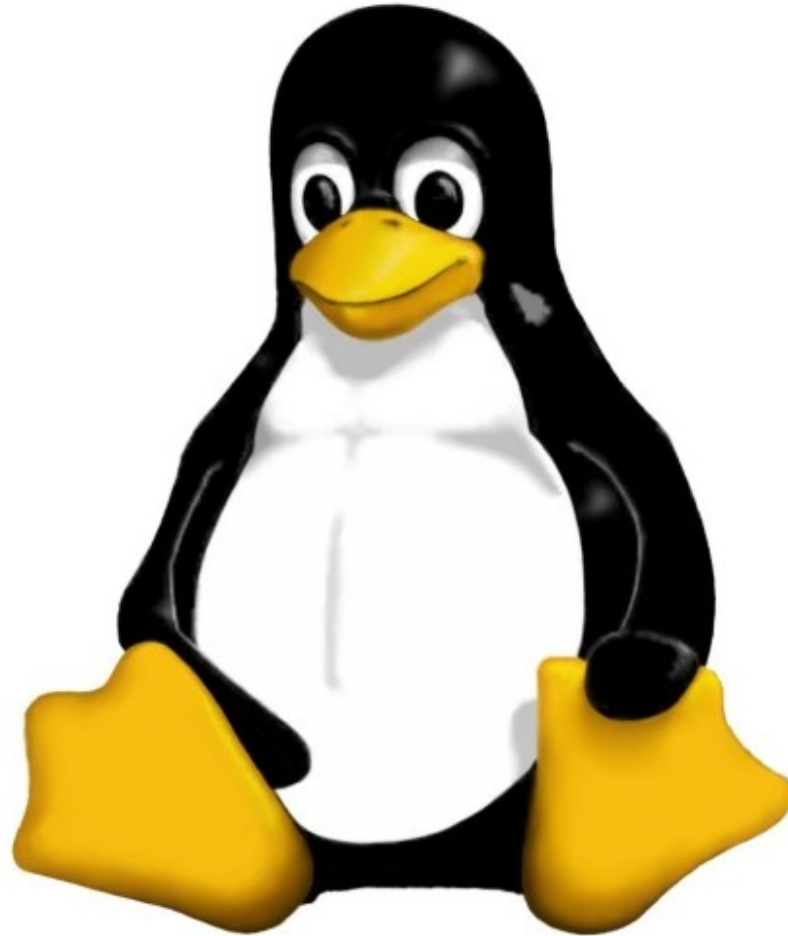


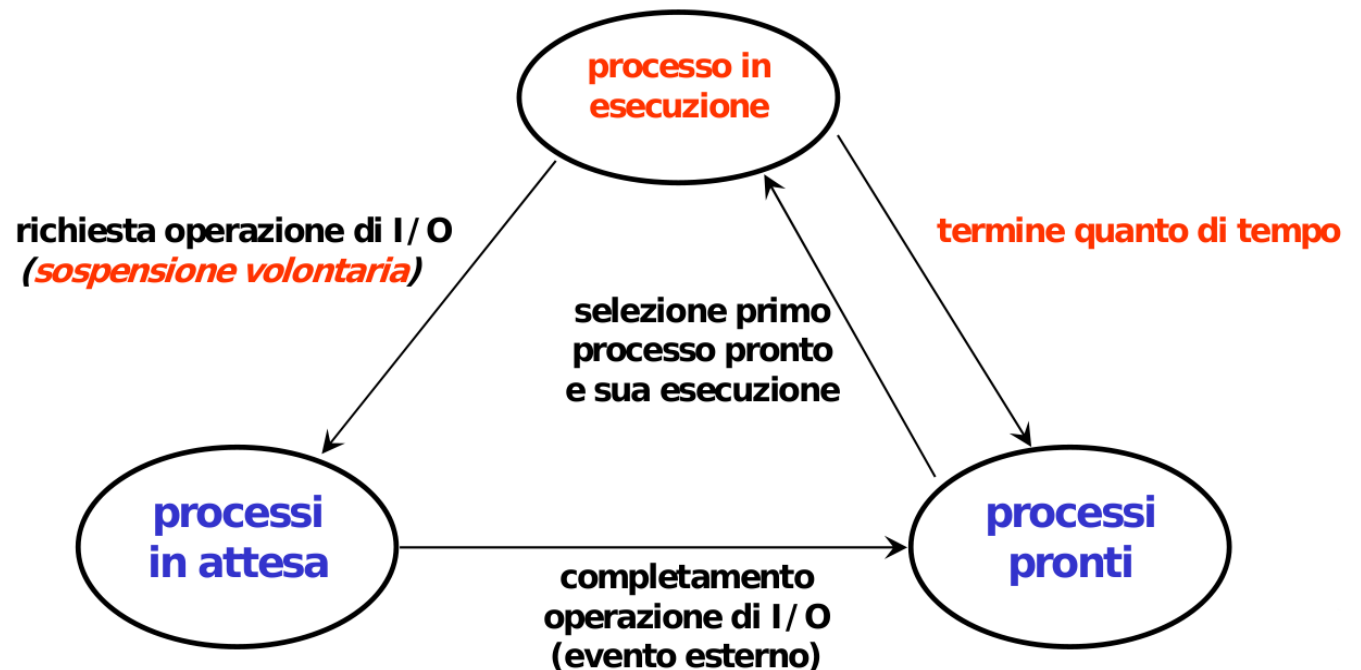
Gestione dei processi

Daniele Iamartino (aka Otacon22)
<otacon22@poul.org>



I processi su Linux

- Il sistema operativo deve gestire più programmi contemporaneamente.
- Abbiamo una o magari anche più CPU, però i processi possono essere anche migliaia, il sistema operativo deve passare da uno all'altro



I processi su Linux

- Ogni processo viene generato come figlio di un altro processo.
- L'unico processo senza padre è init.
- Ogni processo può utilizzare memoria che viene occupata a "blocchetti" chiamati "pagine di memoria".
- Ogni processo che esegue operazioni utili fa uso di "chiamate di sistema" o "syscalls" per aprire files (in Linux e Unix tutto è un file: socket di rete, file del disco, porzioni di memoria, periferiche..)

Riguardo al codice di uscita

- Quando un processo termina restituisce un “codice di uscita”. Se è diverso da 0 qualcosa è andato storto.
- Solitamente il processo padre aspetta la terminazione del processo figlio recuperandone il codice di uscita.
- Nel terminale, ogni volta che lanciamo un programma, la shell (bash) genera un processo figlio a cui far eseguire il nostro programma e dunque attende che la sua esecuzione sia completata e ne raccoglie il codice di uscita.
- Possiamo vedere il codice di uscita di un processo appena terminato in bash digitando
 `echo $?`
- L'utilizzo di `&&` e `||` in bash effettua il controllo proprio di questo codice di uscita
 - Esempio: `cat /file_inesistente && echo “Tutto OK”`

Attributi dei processi

- Ad ogni processo in esecuzione sul sistema è associato un numero intero che lo identifica in modo univoco: il **PID** (**P**rocess **I**dentifier).
 - Il primo processo del sistema, **init**, ha sempre PID 1.
- Ogni processo in Linux è generato come “figlio” di un altro processo “padre” (a parte init). Dunque ad ogni processo è associato il **PPID**, cioè il **PID** del proprio processo padre.
 - È un attributo spesso utile per capire da dove arriva un processo che stiamo controllando
 - Se il padre originale termina la propria esecuzione, il figlio viene “adottato” dal processo init.

Attributi dei processi

- **UID**
 - È l'identificatore dell'utente che ha creato il processo ("il proprietario")
- **GID**
 - È l'identificatore del gruppo associato al processo in esecuzione
- Ogni volta che un processo vuole scrivere/leggere un file, il sistema operativo controlla quali sono i suoi UID e GID e stabilisce quali operazioni può fare (vedi talk sui permessi)
- **COMMAND**
 - È la riga di comando del processo lanciato, comprendente i parametri che gli sono stati passati in fase di lancio.

Stato di un processo

- **Runnable (R)**
 - Il processo può essere eseguito
- **Sleeping (S)**
 - Il processo è in attesa di una risorsa
- **Zombie (D)**
 - Il processo sta tentando di terminare ma il codice di uscita non è ancora stato raccolto
- **Stopped (Z)**
 - Il processo è stato sospeso e non è consentita l'esecuzione (vediamo in seguito)

Vedere i processi in esecuzione: **ps**

- Il comando principale per sapere quali sono tutti i processi in esecuzione sul sistema è **ps** (process snapshot)
- Lanciato senza parametri, ps ci mostra solo i processi in esecuzione nel terminale dove lo lanciamo. Ad esempio:

```
% ps
```

```
PID  TTY    TIME  CMD
```

```
2944 pts/2  00:00:00 bash
```

```
2963 pts/2  00:00:00 ps
```

- Lanciando **ps -e -o pid,ppid,state,command** specifichiamo che vogliamo invece vedere tutti i processi del sistema (opzione **e**) e che per ogni processo vogliamo avere le informazioni su pid, ppid, stato e comando.

Vedere i processi in esecuzione: **ps**

- `% ps -e -o pid,ppid,state,command`

PID	PPID	S	COMMAND
1	0	S	/sbin/init
2089	1	R	xchat
1973	1	S	skype
18271	3079	S	bash
1860	1785	S	gnome-panel
3083	1	R	vlc /media/Anime/K-On!-01.avi
20558	11206	S	ssh 10.0.0.1 -l lello

[...]

- Solitamente la combinazione di opzioni più utilizzata è però **“aux”**

Vedere i processi in esecuzione: **ps**

- **ps aux** ci da informazioni su utente (in nome ricavato dal UID del processo), PID, percentuale di utilizzo di CPU, memoria e diverse altre cose.
- **STAT** indica lo stato del processo:
 - R = Runnable
 - S = Sleeping
 - D = In sospensione non interrompibile
 - T = Tracciato/Fermato
 - Z = Zombie
- **ps lax** ci da informazioni più “tecniche” e in teoria è leggermente più veloce perché non converte UID in nome utente
- Se vogliamo vedere anche l'albero dei processi possiamo aggiungere l'opzione **f** (“ASCII forest”)
- I processi tra parentesi quadre sono dei thread del kernel
- Vediamo subito un esempio da terminale

Memoria del sistema

- Sicuramente ci capiterà di essere interessati a sapere quanta memoria (RAM) è presente su un sistema e quanta è utilizzata dai processi
- **free** è il comando che ci da queste informazioni
- Il parametro **-m** ci mostra i valori in MegaByte
- Facciamo attenzione a leggere, perché vengono considerati anche i buffer di sistema (che non è memoria realmente occupata)
- Lo swap indicato è una partizione (o più di una!) che viene utilizzata come memoria ram quando i processi non ci stanno più in memoria e per l'ibernazione.
 - Per sapere quali sono gli swap abilitati: `swapon -s`
 - Per abilitare/disabilitare le partizioni di swap si usa **swapon** e **swapoff**

Modalità interattiva: **top**

- Lanciare un comando ps ci mostra i programmi in esecuzione nel momento in cui lanciamo il comando. In molti casi è utile però cosa sta succedendo più o meno in tempo reale.
- In quei casi **top** fa al caso nostro. Top fornisce un riepilogo aggiornato di tutte le informazioni dei processi, e può **ordinarle** in base a quello che ci interessa.
- Vengono fornite anche informazioni generali sul sistema (memoria totale utilizzata, carico della cpu...)
- C'è un menu “interattivo” che compare premendo h
- **top -u utente** torna utile per vedere i processi in esecuzione da un certo utente che vogliamo tenere sott'occhio (ex. Server multiutente)
- Vediamo un esempio da terminale

I segnali

- Uno dei metodi più semplici per comunicare con i processi in esecuzione è l'invio di segnali.
- Un segnale è un numero intero che viene mandato da un processo ad un altro in esecuzione.
- Quando un processo riceve un segnale possono accadere due cose:
 - Il processo ha una routine di gestione per quel particolare segnale, dunque la esegue appena lo riceve per effettuare delle operazioni
 - Il processo non sa gestire quel segnale, dunque il kernel decide cosa fare in base al tipo di segnale. Molto spesso questo provoca la terminazione del processo.
- Ogni segnale è identificato da un numero intero tra 0 e 30 circa. In molti contesti è possibile scrivere un nome che identifica il segnale.

I segnali

- Uno dei primi segnali che si incontrano, indirettamente, è il segnale di **INT** o **SIGINT**. Questo è il segnale che viene inviato ad un programma nel terminale quando digitiamo Ctrl+C.
- Se il programma non è predisposto a gestire il routine esso terminerà la propria esecuzione appena premiamo Ctrl+C
- Se il programma è predisposto a gestire la routine potrebbe decidere di ignorare il segnale (non eseguire nessuna operazione quando lo riceve), oppure ad esempio di terminare la propria esecuzione in modo più “pulito” (chiude i file aperti etc..)
- Il segnale di INT ha valore numerico **2**.
- Il segnale **TERM** o **SIGTERM** (valore numerico **15**) indica più o meno la stessa cosa che abbiamo appena detto per **INT**, però non viene generato premendo combinazioni di tasti.

I segnali

- Nel caso particolare di **SIGINT** il segnale veniva inviato da tastiera, però come inviamo in generale i segnali?
- Il comando **kill** serve a questo scopo. Il nome fa intuire il suo utilizzo principale..
 - `% kill 9587`
Invia al processo con PID 9587 il segnale **SIGTERM** per la terminazione “soft”. In questo caso vengono svuotati e sincronizzati i buffer dei file etc...
 - Possiamo specificare il segnale da inviare numericamente come parametro al comando kill:
`% kill -2 9587`
Invia il segnale 2 (segnale di INT) al processo con pid 9587
- Spesso capita che non ci ricordiamo tutti i numeri, quindi è possibile anche specificare il segnale dal suo nome (inizia sempre con “SIG”)
`% kill -SIGINT 9587`

I segnali

- Altro segnale molto importante è **SIGKILL** (valore 9)
- Questo segnale non può mai essere bloccato o gestito da nessuna applicazione ed indica al sistema operativo di bloccare tutte le risorse per il processo e forzare la sua terminazione immediatamente senza preavviso.
- L'unico caso di processo che “sopravvive” è quando è in corso una operazione di I/O a basso livello tipo una scrittura su disco (DMA).
 - `% kill -9 5534`
 - `% kill -SIGKILL 5534`
- Init è in ogni caso immortale.

I segnali: caso tipico

- Vogliamo cercare tutti i processi con nome “firefox” e terminarli

- `% ps aux | grep firefox`

```
otacon22 4960 0.0 0.0 7640 924 pts/22 S+
22:01 0:00 grep --color=auto vlc
otacon22 25618 1.6 1.1 835460 96668 ? S1
Jun01 23:44 firefox
```

- `% kill 25618`
- Non termina..
- `% kill -9 25618`
- Addio!

I segnali

- Un comando che può rivelarsi molto utile per inviare segnali è **killall**
- **Killall** invia un determinato segnale a tutti i processi con un certo nome.

- `% killall skype`

Invia il segnale SIGTERM (il predefinito come per kill) a tutti i processi del sistema che hanno come nome "skype" (viene effettuata una ricerca)

- `% killall firefox -9`

Invia a tutti i processi con nome "firefox" il segnale 9 (SIGKILL)

- `% killall firefox -SIGKILL`

Come sopra.

I segnali

- Un altro segnale molto utile è **SIGSTOP** (che come SIGKILL non può essere mai bloccato). Esso ferma l'esecuzione del programma e lo lascia fermo in sospeso fino a quando non viene inviato un segnale **SIGCONT**, che lo riattiva.
 - Torna molto utile se c'è qualche programma che sta mangiando memoria o tempo CPU e si vuole fermarlo senza ucciderlo con un SIGKILL (perché potrebbe generare danni magari)
- Quando in un terminale premiamo Ctrl+Z viene inviato al programma il segnale **SIGTSTP**, che è una versione "soft" di **SIGSTOP**. Per riprendere la sospensione dopo aver digitato Ctrl+Z basta lanciare il comando "fg"
- Infine vediamo **SIGHUP** che è un segnale che viene interpretato in due modi:
 - Alcuni demoni lo interpretano come richiesta di reset e riletture della configurazione
 - Molte shell (tra cui shell) inviano un segnale **SIGHUP** lo inviano ai programmi collegati in background ("hup" da "hangup") prima di terminare.
- I segnali possono essere inviati anche da top premendo "k" (esempio?)

Gestire la priorità dei processi

- Il kernel Linux tenta di eseguire in modo alternato ciascun processo che lo necessita. Per scegliere quale processo va eseguito prima degli altri Linux ha uno **scheduler**, cioè un algoritmo che decide **qual'è** il prossimo processo ad andare in esecuzione.
- Solitamente questo lavoro viene fatto molto bene (algoritmo LRU), però in certi casi può essere utile dire allo scheduler che un certo processo ha bisogno di essere eseguito di più rispetto agli altri.
- Questo viene fatto tramite l'attributo **nice** del processo.
- Il valore di nice di un processo indica allo scheduler “quanto vuole essere gentile il processo”, cioè:
 - **Nice alto:** Il processo è **gentile e lascia tempo** per eseguire prima gli altri, più che a se stesso.
 - **Nice basso:** Il processo **non è gentile e vuole eseguire prima degli altri**.

Gestire la priorità dei processi

- Normalmente il nice è impostato a 0 per quasi tutti i processi del sistema. I valori possibili sono da -20 a +19
- Alcune applicazioni necessitano di nice basso, come ad esempio pulseaudio che gestisce l'audio sul sistema. Altrimenti si sentirebbe “Gracchiare che è un piacere” (cit.) O altre che gestiscono l'orario.
- Per abbassare il valore di nice sotto zero è necessario essere root.
- Il comando utilizzato per cambiare il valore di nice di un programma è **renice**

- Ad esempio per abbassare il nice di un processo a -5:

```
% renice -5 48876
```

- Se vogliamo invece eseguire direttamente un programma con una certa priorità si usa **nice**:

- Ad esempio:

```
% nice -n 8 /usr/bin/programma
```

- Attenzione! Alcune versioni di **nice** considerano diversamente i segni!

Modalità interattiva: **htop**

- Non è installato di default quasi in nessuna distribuzione, ma è facilmente reperibile.
- Ci permette di visualizzare in modo più comodo e chiaro le informazioni sul sistema
- Si vede in modo semplice (e colorato) il carico di ciascuno dei processori e della memoria
- In basso c'è una legenda per le operazioni che semplifica la vita.
- Gestire il nice è facilissimo, interattivo e colorato! (nice diversi da 0 sono in rosso)
- Vediamo un veloce esempio da terminale

Tenere processi in background

- Qualche volta può capitare di dover tenere aperto un processo anche dopo che il terminale viene chiuso.

comando &

Mette in background un processo all'interno della shell. Possiamo continuare a lavorare. Quando chiudiamo il terminale (e quindi la shell), **bash** invia a tutti i processi figli il segnale **SIGHUP**, provocando la terminazione di tutti i processi in background.

nohup comando &

Risolve questo problema, dicendo a bash di non inviare il segnale di hup a questo processo in background quando viene chiusa.

- Esempio

Tenere processi in background

- Altra soluzione è l'utilizzo di **screen**.
- Nasce da quando non esistevano le finestre dei terminali sui sistemi grafici :)
- Screen crea delle intere sessioni di shell che possono essere lasciate in background “distaccandosi” e poi “riagganciandosi” alle varie sessioni quando serve.
- Il modo migliore per capire come funziona è provarlo.
- È utilissimo per controllare sistemi in remoto: stiamo controllando un server remoto e stiamo editando un file. Ci salta la corrente a casa e perdiamo la sessione sul server. Con screen resta aperta e possiamo poi “riagganciarci” e riprendere il lavoro.
- Necessita un po' di pratica per gli shortcut da tastiera (Ctrl+A+“ .. lol)

Diagnosi delle syscall

- Ogni processo del sistema operativo che compie delle operazioni utili fa uso di chiamate di sistema. Le chiamate di sistema sono “funzioni” del sistema operativo (per la gestione dei file, dei socket di rete, dei permessi, per i filesystem, per stampare a schermo e **tantissimo** altro).
- In Linux esistono intorno alle 400 chiamate di sistema.
- Alcune volte ci sono dei programmi che hanno dei problemi ad eseguire o continuano a bloccarsi e non c'è modo di capirne il motivo.
- Molto spesso può essere utile “tracciare” tutte le chiamate di sistema di cui fa uso per capire il problema.
- **Strace** è il comando per questo scopo. Strace si mette a livello del sistema operativo e “intercetta” tutte le chiamate di sistema richieste dal programma al sistema operativo.

Diagnosi delle syscall

- Lanciando il comando:

```
% strace -p 2356
```

Andiamo ad “agganciarci” al processo in esecuzione con pid 2356 e vediamo tutte le chiamate di sistema che sta effettuando (con i relativi parametri).

- Ogni volta che il programma aprirà un file vedremo una `open( )`, ogni volta che scriverà su un file o su un socket vedremo `write( )` etc....
- Per lanciare strace è sempre necessario essere **root**.
- Spesso, in programmi “poco comunicativi”, si riesce a scoprire che continuano a bloccarsi solo perché manca un file (e quindi vediamo la syscall `open( )` fallire in continuazione)
- `man 2 nome_syscall` può essere un buon modo per capire cosa fa una certa syscall che vediamo in strace.
- Possiamo anche tracciare già da quando lanciamo il programma:

```
% strace firefox
```

Vedere i file aperti: lsof

- I processi che sono in esecuzione sul sistema operativo spesso aprono dei file in lettura e/o scrittura.
- In Unix (e quindi anche in Linux) vale però la filosofia

“tutto è un file”

Dunque spessissimo un processo ha dei file aperti. Ad esempio anche i socket di rete sono visti come file (speciali) in cui leggere e scrivere dati.

- Spesso i programmi fanno uso di librerie condivise (file .so), che sono effettivamente lette durante l'esecuzione.
- Il sistema operativo ha una “tabella globale dei file aperti”, dove tiene traccia di tutti i file aperti e quali programmi li stanno utilizzando.
- Il comando **lsof** permette di visualizzare tutti i file aperti sul sistema.
- Poiché come abbiamo detto vengono visualizzati anche i socket, spesso capita che ci siano degli indirizzi ip di cui risolvere l'hostname e questo rallenta la generazione della lista. **lsof -n** specifica di non fare questa operazione e la visualizzazione si velocizza.
- **lsof -p 4243** visualizza tutti i file aperti dal processo con pid 4243

Vedere i file aperti: lsof

- Problema classico è quando siamo da terminale, montiamo una chiavetta/disco e poi vogliamo smontarla mentre però ancora ci sono file aperti su di essa. Il sistema non ci permette di farlo:

```
% umount /dev/sdc1
```

```
umount: /media/DiscoUSB: device is busy.
```

(In some cases useful info about processes that use
the device is found by `lsof(8)` or `fuser(1)`)

- La soluzione classica è dunque:

```
% lsof -n | grep "/media/DiscoUSB"
```

- E subito si scopre quali sono i processi che stanno utilizzando quel disco

Vedere i file aperti: lsof

- Sintassi dell'output:

COMMAND	PID	USER	FD	TYPE	DEVICE	SIZE/OFF	NODE	NAME
vlc	25618	otacon22	23r	REG	8,17	2117519360	118359154	/media/Anime/K-ON!-01.avi

- FD è il File Descriptor. In ogni processo, ad ogni file viene associato un numero in sequenza. 0,1 e 2 sono rispettivamente stdin,stdout,stderr. Ad ogni nuovo file che viene aperto dal programma si aggiunge un file descriptor. 23r significa il file descriptor 23 aperto in lettura (r=read,w=write,u=read e write)
- REG è il tipo di file (Regular), ci sono tantissimi tipi (dispositivo, socket di rete, socket locale, etc, etc...)
- SIZE/OFF è la dimensione del file o l'offset di lettura
- NODE è l'i-node del file, l'identificatore unico

Il filesystem /proc

- È un filesystem completamente dedicato a contenere informazioni e parametri del sistema operativo
- Nella cartella principale /proc troviamo vari files di informazioni “generali”:
 - /proc/cpuinfo contiene informazioni sulla/e CPU
 - /proc/meminfo informazioni sulla memoria
 - /proc/version descrizione della versione del kernel
- Esiste una cartella per ogni processo all'interno di /proc , che ha come nome il PID del processo.
- All'interno di ciascuna troviamo svariate informazioni tipo:
 - /proc/PID_PROCESSO/cwd è un collegamento simbolico alla cartella di “lavoro” del processo
 - /proc/PID_PROCESSO/cmdline è la stringa con cui è stato lanciato il processo in esecuzione (contiene i parametri quindi)
 - ...

/proc/..PID../fd/

- Caratteristica molto interessante è la cartella **fd** all'interno di ogni cartella di processo
- In **fd** troviamo dei file numerati che corrispondono ai *file descriptor* aperti da quel processo
- 0,1 e 2 sono sempre stdin,stderr,stdout
- Se il programma che stiamo considerando apre un altro file, verrà creato un altro collegamento in questa cartella.
- Ad esempio se stiamo guardando un video sul nostro player video preferito e cerchiamo il suo PID, andando nella cartella /proc/PID_PLAYER/fd/ troveremo un qualche file descriptor corrispondente al file video attualmente aperto.

Domande?

Grazie!