# Lezione 4 [30/09/22]

## Capitolo 3: Processi

- Concetto di processo
- Scheduling dei processi
- Operazioni sui processi
- Processi cooperanti
- Comunicazione tra processi
- Esempi di sistemi per la IPC
- Comunicazione nei sistemi client/server

## Processo

Un **processo d'elaborazione**, o più brevemente **processo**, si può definire come un programma in esecuzione e costituisce l'unità di lavoro dei moderni sistemi time sharing



Un sistema è costituito da un insieme di processi:

- Quelli del S.O. eseguono il codice di sistema
- Quelli utente eseguono il codice utente

Tutti questi processi possono essere eseguiti in modo concorrente e l'uso della/e CPU è commutato tra i vari processi

## Concetto di processo

Un S.O. esegue una varietà di "programmi"

- I sistemi a lotti (batch) eseguivano lavori **(job)**
- Un sistema a partizione di tempo esegue programmi utenti **(task)**

Spesso i termini job e processo sono utilizzati in modo intercambiabile

Informalmente un processo può essere considerato come un programma in esecuzione

Un programma di per sé non è un processo

Il programma, anche detto **sezione testo**, è un'entità passiva, mentre il processo è un'entità attiva

Un processo include:

- Contatore di programma **(program counter)**
- Contenuto dei registri della CPU
- Un proprio **stack**
- Un **heap**
- Una sezione di dati **(data section)** contente variabili globali

Un programma diventa un processo quando il suo file eseguibile viene caricato in memoria

## Stato del processo

L'esecuzione di un processo deve progredire in maniera sequenziale

Mentre un processo è in esecuzione è soggetto a cambiamenti di stato

Ogni processo può trovarsi in uno tra i seguenti stati:

- **Nuovo (new):** il processo viene creato
- **In esecuzione (running):** quando è in memoria ed ha il controllo della CPU (le sue istruzioni vengono eseguite)
- **In attesa (waiting):** quando è temporaneamente sospeso in attesa di un evento, quale la terminazione di I/O, lo scadere di un timer, la ricezione di un messaggio etc.
- **Pronto (ready):** quando è in memoria ed è pronto per l'esecuzione, ma non ha il controllo della CPU e attende di essere assegnato ad un'unità di elaborazione
- **Terminato (terminated):** quando termina l'esecuzione e abbandona il sistema

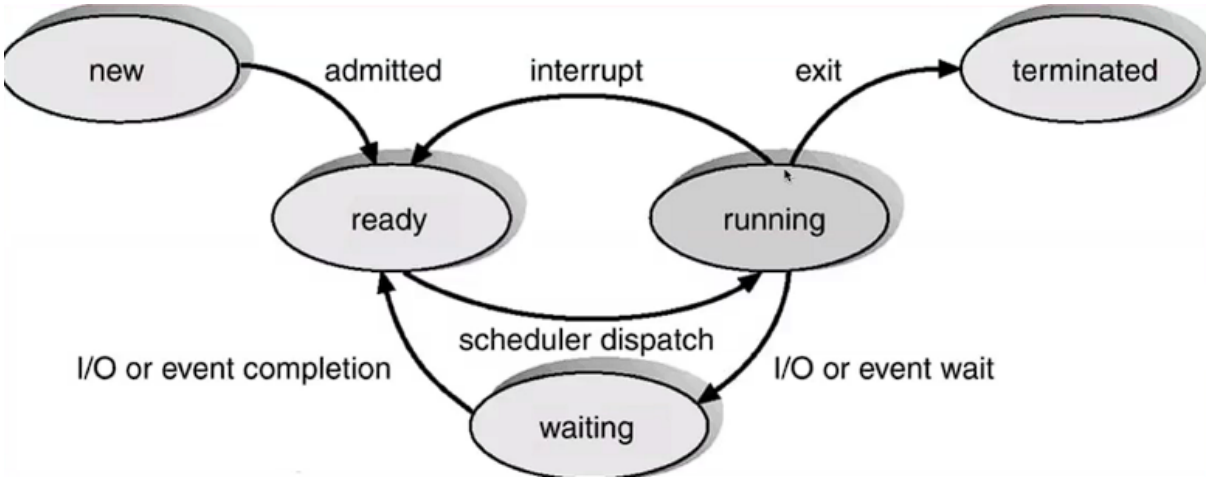
In ciascuna CPU può essere in *esecuzione* un solo processo per volta, anche se molti processi possono essere *pronti* o in *stato d'attesa*

Appena sono creati, i processi hanno tutte le risorse necessarie alla loro esecuzione, passano da **new** a **ready** e vengono ammessi nella **ready-queue** (quindi sono presenti in memoria centrale ed attendono di utilizzare la CPU)

Il processo di schedulazione **(scheduler)** sceglie dalla ready-queue a quale processo assegnare la CPU **(dispatcher)** e lo promuove allo stato **running**

Da running, posso tornare allo stato **ready** in caso di interruzione o system call

Diagramma di transizione degli stati di un processo:




## Blocco di controllo dei processi

In un sistema operativo ogni processo è rappresentato da un descrittore di processo:

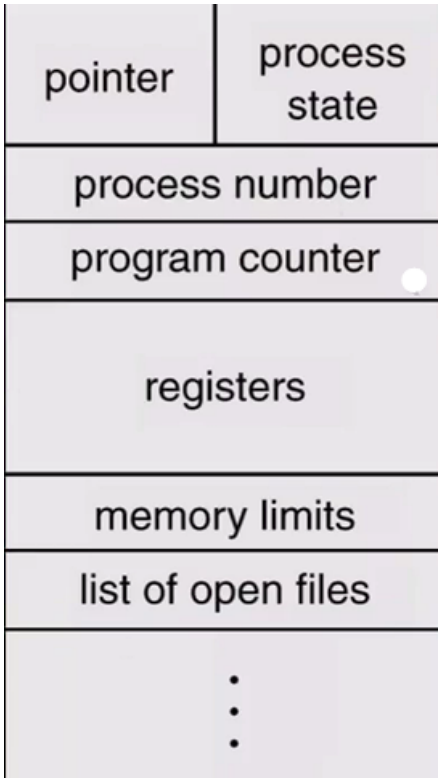
- **Process Descriptor (PD)** - detto anche **blocco di controllo di un processo (Process Control Block - PCB)**

Nel PCB sono contenute informazioni connesse ad uno specifico processo:

- **Stato del processo:** tra new, ready, running, waiting, terminated
- **Contatore di programma:** indica l'indirizzo della successiva istruzione da eseguire
- **Registri di CPU:** accumulatori, registri indice, puntatori alla cima delle strutture a pila **(stack pointer)**, registri d'uso generale e registri contenenti informazioni relative ai codici di condizione
- **Informazioni sullo scheduling di CPU:** priorità del processo, puntatori alle code di scheduling e altri parametri di schedulazione
- **Informazioni sulla gestione della memoria:** registri di base e di limite, tabelle delle pagine in memoria o dei segmenti (a seconda della tecnica usata dal S.O.)
- **Informazioni di contabilizzazione delle risorse:** tempo di CPU, tempo reale di CPU, numero del processo, etc.
- **Informazioni di I/O:** lista dei dispositivi di I/O assegnati al processo, elenco file aperti, etc.

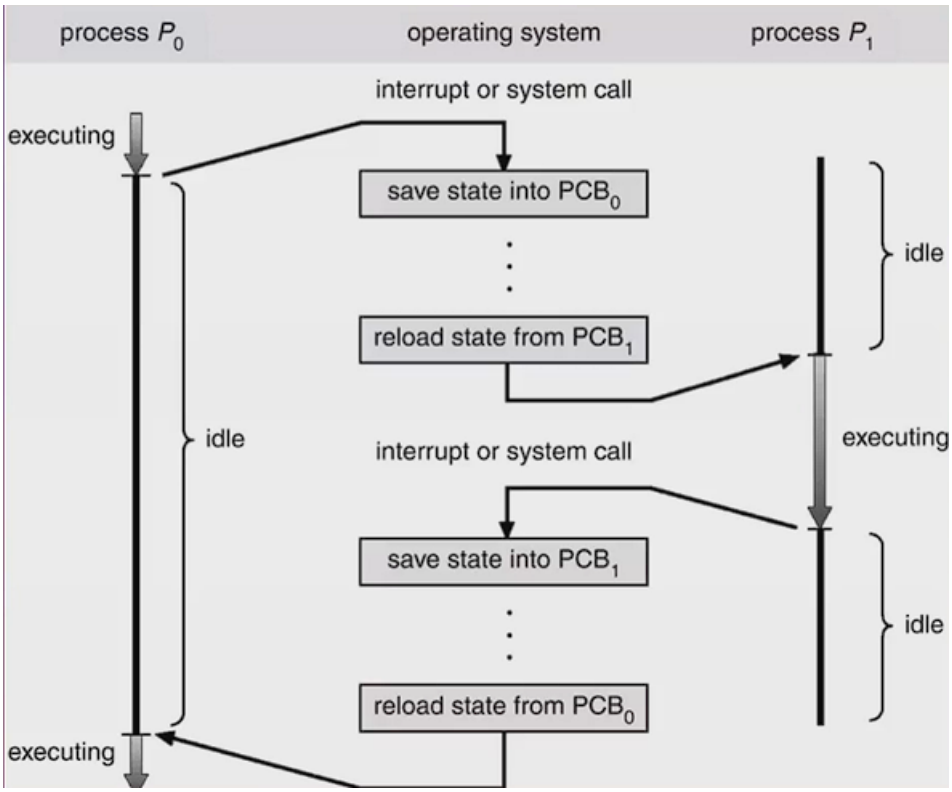
In sintesi il PCB si usa semplicemente come deposito per tutte le informazioni relative ai vari processi

Descrittore di processo (PCB):




La CPU può essere commutata tra i processi (viene usata da più processi concorrentemente):





## Scheduling e code di scheduling

Quando i processi entrano nel sistema vengono inseriti in una **coda dei processi** composta da tutti i processi del sistema

I processi che risiedono in memoria centrale e che sono pronti e in attesa di essere eseguito, si trovano in una lista detta **coda dei processi pronti (ready queue)**

Generalmente la coda dei processi pronti è memorizzata come **lista concatenata**

Un intestazione della coda dei processi pronti contiene i puntatori al primo e all'ultimo PCB; ogni PCB punta al prossimo PCB della coda

Quando un processo fa una richiesta di I/O il suo identificativo viene inserito nella **coda del dispositivo**:

- Non è detto che il dispositivo richiesto sia libero

I processi quindi si spostano tra varie code

Un processo si colloca inizialmente nella ready queue, dove deve attendere finché non è selezionato per essere eseguito

Una volta che il processo è stato assegnato alla CPU ed è in esecuzione possono verificarsi i seguenti eventi:

- Richiesta di I/O: il processo viene inserito in una coda di I/O
- Creare un processo figlio
- Il processo può essere rimosso forzatamente dalla CPU a causa di un'interruzione ed essere reinserito nella ready queue

Alla terminazione, il processo viene rimosso da tutte le code e vengono deallocati il suo PCB e le varie risorse

Coda dei processi pronti e diverse code di dispositivi di I/O:

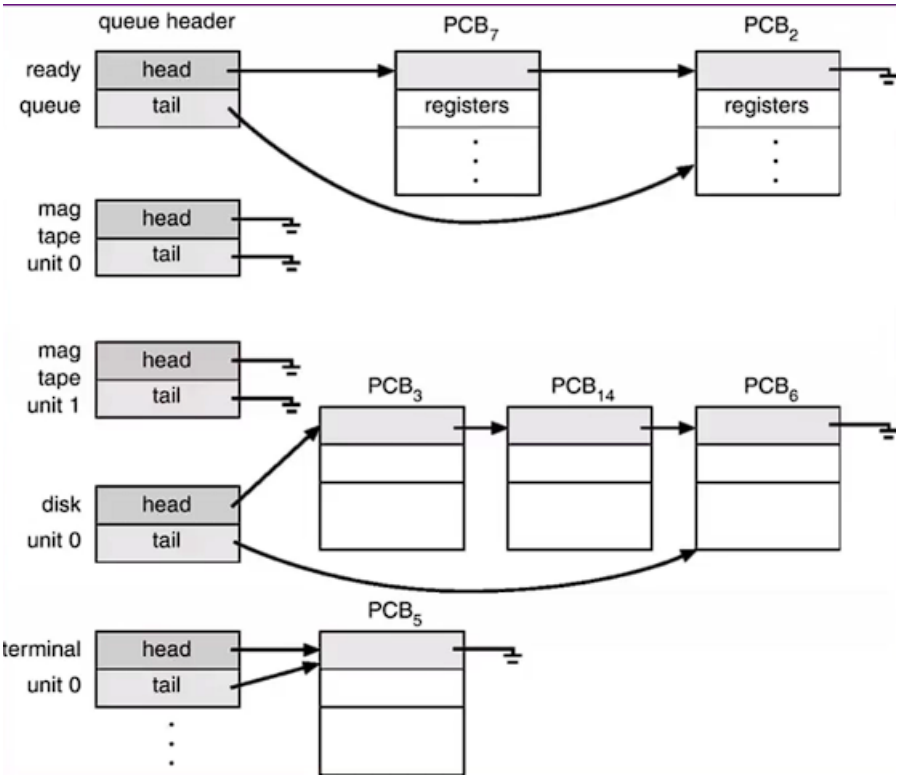
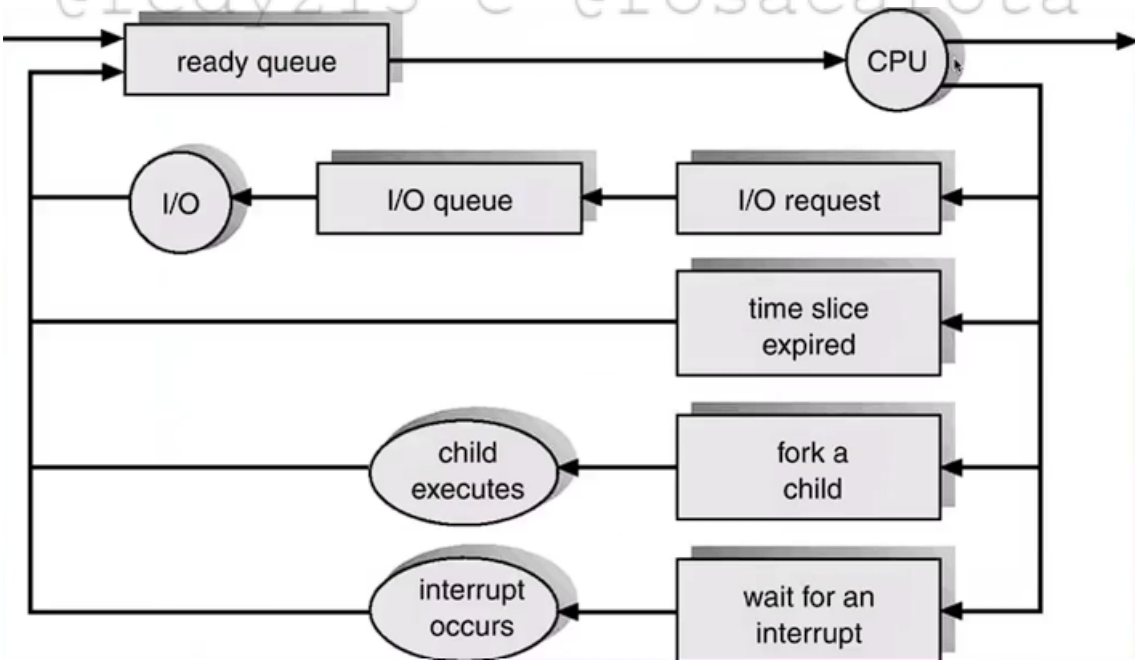



Diagramma di accodamento per lo scheduling dei processi:






## Scheduler di CPU

Lo scheduler seleziona i processi e li carica in memoria assegnando la CPU in modo da ottimizzarne l'utilizzo

**Scheduler a lungo termine** o di job **(long-term scheduler)**:

- Seleziona i processi da inserire nella coda dei processi pronti da memorie secondarie

**Scheduler a breve termine** o di CPU **(short-term scheduler** o **CPU scheduler)**

- Seleziona un processo dalla coda dei processi pronti e gli assegna la CPU

Lo scheduler di job è uno scheduler che viene eseguito con una frequenza molto bassa

- Posso infatti passare dei minuti affinché vengano creati dei nuovi processi

Lo scheduler di job controlla il grado di multiprogrammazione

- Cioè il numero di processi in memoria

In un sistema stabile, la velocità media della creazione dei processi deve essere uguale alla velocità con cui i processi lasciano il sistema

Lo scheduler di CPU viene eseguito molto frequentemente:

- Il processo potrebbe rimanere in esecuzione pochi millisecondi prima di fare una richiesta di I/O

I processi possono essere descritti come:

- **Con prevalenza di I/O (I/O bound):**
  - Impiega la maggior parte del tempo in operazioni di I/O, inframezzando piccoli tempi di CPU
- **Con prevalenza d'elaborazione (CPU bound):**
  - Poche operazioni di I/O, prevalenza di elaborazione



Uno scheduler per rendere efficiente il suo lavoro è tenuto a selezionare una buona combinazione di processi I/O bound e CPU bound, altrimenti il sistema risulterebbe sbilanciato

**Scheduler a medio termine:**

In alcuni S.O: come quelli time sharing si può introdurre un livello di scheduling intermedio

- Vantaggio derivato dalla rimozione dei processi dalla memoria; il processo può essere reintrodotto in memoria successivamente, riprendendo l'esecuzione da dove era stata abbandonata (questo sistema si chiama **avvicendamento dei processi in memoria**)

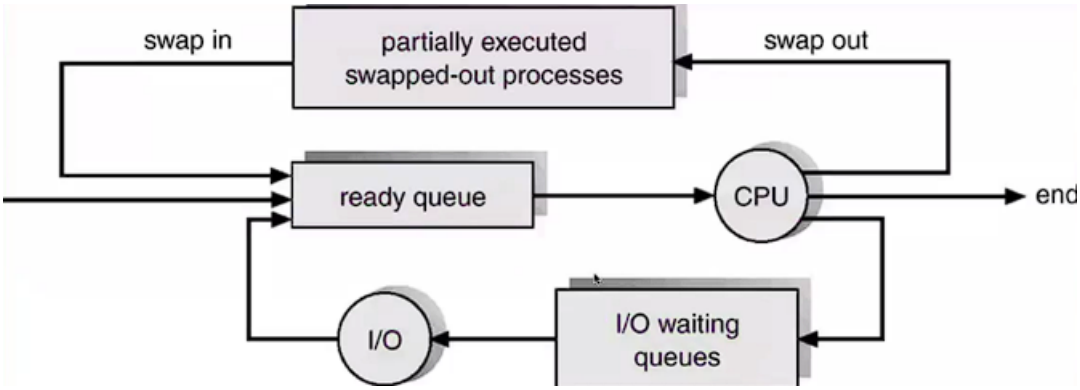
> 💡 Un modulo del S.O., il **dispatcher** passa effettivamente il controllo della CPU ai processi scelti dallo scheduler a breve termine, poiché si attiva a ogni commutazione di contesto, il dispatcher dovrebbe essere quanto più rapido possibile e il tempo richiesto dal dispatcher per fermare un processo ed avviarne l'esecuzione di un altro è noto come **latenza di dispatch**

**Avvicendamento (swapping)** - richiesto per lo scheduler a medio termine, questa operazione è molto lenta

Tramite un operazione di **swap-in** prendo i processi dalla memoria di massa e li metto in memoria centrale

Tramite un operazione di **swap-out** prendo i processi dalla memoria centrale e li metto in memoria di massa

Aggiunta di scheduling a medio termine:






## Cambio di contesto

I **cambio di contesto (context switch)** solitamente avvengono in seguito a interruzioni: il sistema deve salvare il contesto del processo corrente per poterlo poi ripristinare quando il processo stesso potrà ritornare in esecuzione

Il contesto è contenuto all'interno del PCB

In termini generali si esegue quindi un **salvataggio dello stato** corrente della CPU; in seguito si attuerà un **ripristino dello stato** per poter riprendere l'elaborazione dal punto in cui era stata interrotta

- Il passaggio della CPU ad un nuovo processo implica quindi:
  - La registrazione dello stato del processo vecchio
  - Il caricamento dello stato precedentemente registrato del nuovo processo

Il tempo necessario al cambio di contesto è overhead:

- Il sistema non compie nessun lavoro direttamente utile alla computazione
- Varia in genere tra 1 e 1000 microsecondi

La durata del cambio di contesto dipende molto dall'architettura

## Operazioni: creazione di un processo

Durante la sua esecuzione un processo può creare altri processi

Il processo creante è chiamato **padre**, mentre il processo creato è chiamato **figlio**

La gerarchia che esiste in questa struttura è ricorsiva, per cui un processo figlio può essere, a sua volta, padre di altri processi

Così facendo si viene a creare un **albero dei processi**

La maggior parte dei sistemi operativi identifica un processo per mezzo di un numero univoco, solitamente un intero, detto **identificatore del processo (PID)**

**Risorse:**

- Padre e figlio condividono tutte le risorse
- Il figlio condivide un sottoinsieme delle risorse del padre
- Padre e figlio non condividono risorse

**Esecuzione:**

- Il padre continua l'esecuzione, concorrentemente al processo figlio
- Il padre attende che i processi figli terminino la loro esecuzione

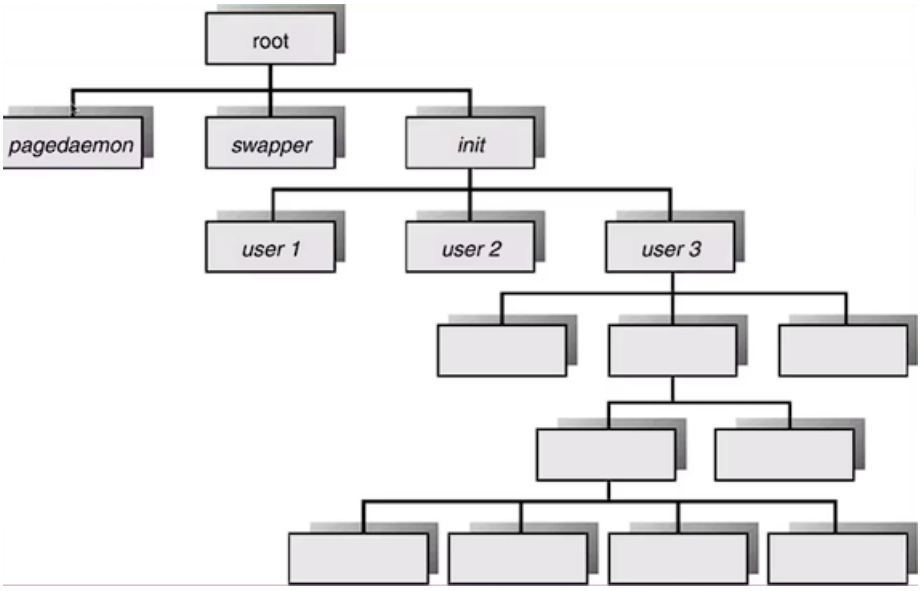
**Spazio di indirizzi:**

- Il processo figlio è duplicato dal padre
- Nel processo figlio si carica un programma

**Unix:** 

- `fork`
- `exec` dopo `fork` carica un programma nel processo figlio

Albero di processi in un sistema UNIX:



Creazione di un processo in Unix:

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    pid_t pid;
    pid = fork(); // Genera un nuovo processo

    if (pid < 0) { // Errore
        fprintf(stderr, "Errore nella creazione del processo\n");
        exit(-1);
    }
    else if (pid == 0) { // Processo figlio
        execlp("/bin/ls", "ls", NULL);
    }
    else { // Processo padre
        wait(NULL);
        printf("Il processo figlio ha terminato\n");
        exit(0);
    }
}
```

System call utilizzate: `fork`, `exec`, `wait`, `exit`

La `fork` chiede al sistema operativo (si passa in modo di sistema) di creare un processo figlio, esso avrà un nome differente (PID differente), generalmente più piccolo



Il processo figlio avrà una zona di memoria interamente sua, duplicata da quella del padre (copia dello spazio di indirizzi) e verrà inserito nella coda dei processi pronti

`fork()` restituisce valori diversi al padre e al figlio, al padre restituirà il PID del figlio (es. 45), al figlio invece restituirà 0 come PID

Tramite l'intero restituito dalla `fork()`, il sistema identificherà univocamente quel processo

Quando i due processi saranno schedulati inizieranno ad eseguire le loro istruzioni

Tramite la system call `wait()` il processo passa allo stato di waiting, il sistema andrà quindi a selezione dalla ready queue un nuovo processo (assumiamo il figlio nel nostro caso)

La system call `exec()` sostituisce lo spazio di indirizzi del processo con un nuovo programma ("ls" nel nostro caso), il figlio, quindi, da questo momento in poi, eseguirà il nuovo programma

Una volta che il figlio ha terminato, il padre riceve un'interruzione, fa una stampa, ed esegue la system call `exit()`, terminando la sua esecuzione