

## u5-s1-processi

### Sistemi Operativi

#### Unità 5: I processi

## Aspetti Teorici

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

---

## Argomenti

1. Concetto di processo
  2. Stati di un processo
  3. Process Control Block
  4. Scheduling
  5. Algoritmi di Scheduling
- 

## Concetto di processo

### Definizione. (*Processo*)

- Un elaboratore svolge uno o più *compiti*
  - Esempio: Controllare la temperatura di una stanza
- Un *compito* si svolge tramite un procedimento formale detto *algoritmo*
- Un programma implementa un algoritmo *tramite istruzioni* in linguaggio macchina (passaggio astratto→concreto)
  - Può essere scritto in un linguaggio di programmazione e *compilato*
- Un *processo* è un *programma in esecuzione*

### Nota storica. (*Pre-S.O. e con S.O.*)

Inizialmente, ogni elaboratore *eseguiva un programma per volta*, senza *sistema operativo*.

- Caricato all'*avvio del sistema*

- Oppure eseguiti *sequenzialmente* (batch processing)  
In ogni caso, *mai* venivano eseguiti in *contemporanea*.  
Oggi i sistemi moderni hanno un SO che permette di *eseguire più processi in contemporanea*
- Le risorse del sistema sono gestite *dal SO* che le mette a disposizione tramite le System Call
- Il SO gestisce l'esecuzione dei processi: *scheduling*
- I processi vengono eseguiti a turno

## Struttura del Processo

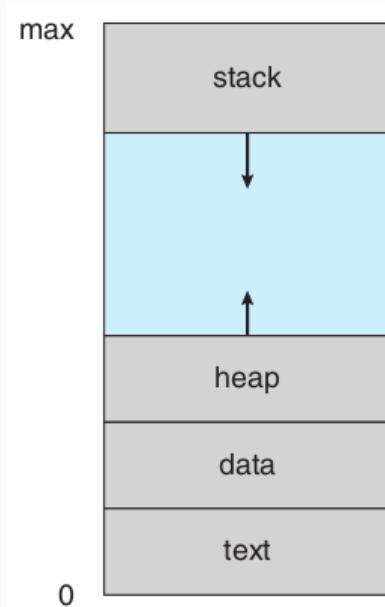
### Struttura del processo in memoria.

Un processo risiede *in memoria*, da allocare.

La struttura di un processo in memoria è generalmente suddivisa in *più sezioni*.

- **Sezione di testo:** contiene il *codice eseguibile*
- **Sezione dati:** contiene le *variabili globali*
- **Stack:** memoria *temporaneamente utilizzata* durante le chiamate di funzioni
- **Heap:** memoria allocata *dinamicamente* durante l'esecuzione del programma (ξ)

**FIGURA.** (*Struttura di un processo da allocare*)



## Tipologie di Processi

Principalmente abbiamo due tipi di processi eseguiti.

### 1. Definizione (*Processo INIT*)

Nei moderni SO, all'avvio del sistema viene avviato un processo fondamentale detto *init*

- Per *Linux* o sistemi *POSIX-like*
- Eseguito fino allo shutdown (+∞)
- Eseguito automaticamente dal *kernel*

- Ha PID 1 (vedremo dopo)
- Il processo `init` avvia altri processi (tramite svariati file di configurazione) in background:
- Per gestire *periferiche*: rete, antivirus
  - Per creare l'*interfaccia grafica* della GUI o del terminale

### **Definizione.** (*Servizio*)

Un processo avviato in background da `init` è detto *Servizio*

- Formati e comandi diversi tra distribuzioni Linux per gestirli
- Comandi: `service` o `systemctl`

### **2. Definizione.** (*Processo utente*)

L'utente può *creare dei processi* - mediante terminale o interfacce grafiche, che a loro volta usando delle System Call - per svolgere i propri compiti

- Browser
- Editor
- Programmi server: server Web, server DNS
- Eccetera...

## Ruolo del Sistema Operativo nei Processi

### **Ruolo del S.O. per i processi.**

Il SO mette a disposizione delle *System Call* per:

1. *Creare* nuovi processi
2. *Sincronizzazione*: Attendere il completamento di altri processi per coordinare un compito complesso
3. *Uccidere* processi

### **Definizione.** (*PID*)

I processi sono identificati dal un *PID*

- Il processo `INIT` ha PID 1 per definizione

## La Vita di un Processo

### Stati di un Processo

Un processo si può trovare in diversi stati.

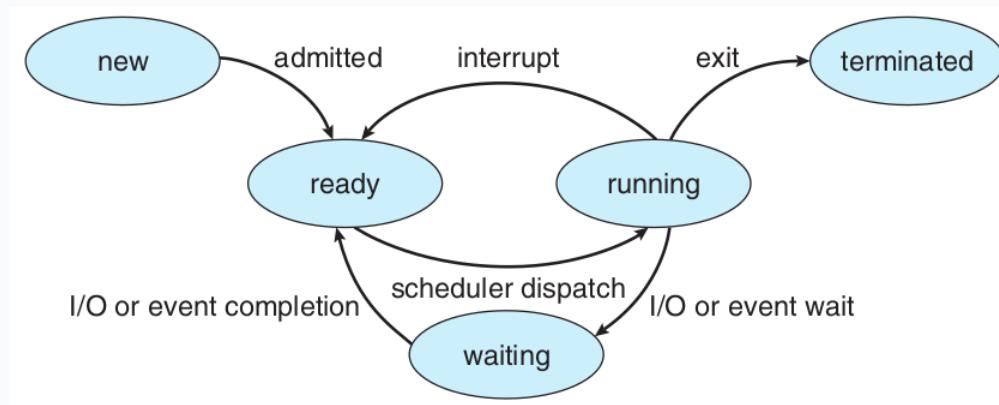
1. *Nascita* (new): è stata richiesta la creazione di un processo mediante un *system call*. Il sistema operativo vede se la richiesta sia *"fattibile o meno"* e ben posta: in tal caso, viene ammessa (admitted).
2. *Pronto* (ready): il processo è pronto per l'esecuzione, ma non è ancora in *esecuzione*.

Aspetta che lo *scheduler* del S.O. dia il turno al processo.

3. **In esecuzione** (running): Il processo fa le sue cose. Da questo punto possiamo avere tre esiti:

1. Il processo ha *finito il suo lavoro* (o qualcosa gli costringe di farlo) ed esce. In tal caso è in stato di "*morte*" (terminated). Ho messo le virgolette, perché il processo starebbe aspettando che la sua morte venga "*confermata*" dal suo padre; il processo non è completamente morto.
2. Lo scheduler del sistema operativo lo mette in pausa e lo *interrompe* (interrupt), facendolo ritornare allo stato di *ready*.
3. Ho una *system call* lenta e complessa (come quella di leggere un disco magnetico), e il processo viene bloccato. Allora il processo viene messo in *attesa* (waiting), finché ciò che lo blocca viene completato. Quando viene completato, viene rimesso in *ready*.

**FIGURA.** (*Schema della vita di un processo*)



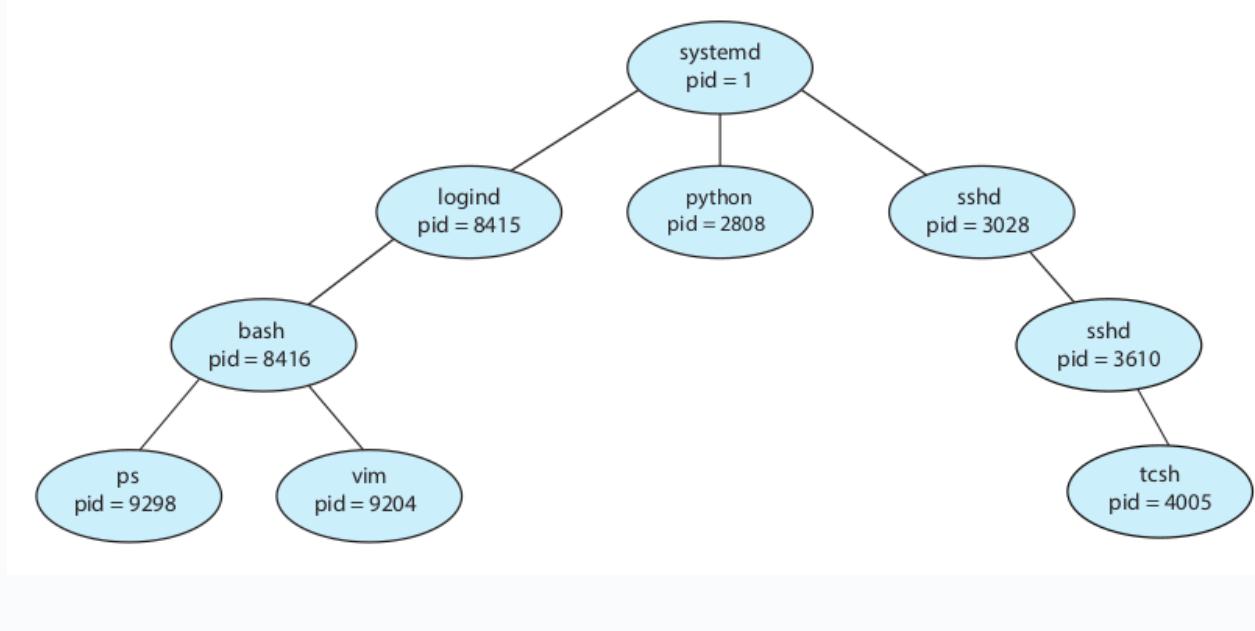
## L'albero dei processi

Usando le System Call, un processo può creare un altro processo. Infatti, tutti i processi sono figli di qualcuno (tranne *INIT*)

- Il processo generato è *figlio* del processo generante
- Si crea un *albero dei processi*
- Se il processo padre termina, i figli *NON* vengono terminati
- I processi *senza padre* diventano figli del processo *init*

### Esempio di albero dei processi:

Nota: su alcuni sistemi **init** è rinominato **systemd**



## Process Control Block

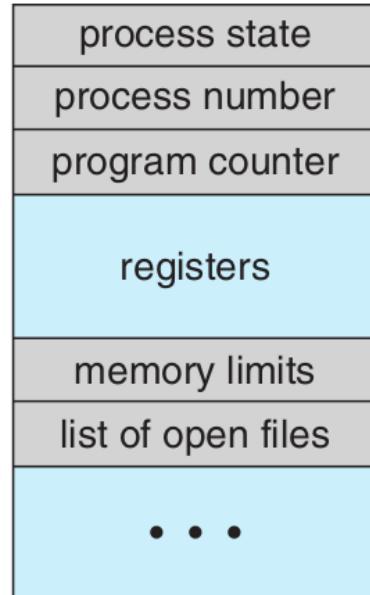
Ogni processo è rappresentato nel sistema operativo da un ***blocco di controllo*** (process control block, PCB) contenente le informazioni connesse. Diventa un'equivalente dell ***inode*** per i file (1).

### Definizione. (*Process Control Block*)

Un **PCB** (Process Control Block) è una struttura dati, contenente dati rappresentativi per un **processo**. Le informazioni registrate sono le seguenti.

- **Stato del processo:** nuovo, pronto, esecuzione, attesa, arresto
- **Program counter:** indirizzo della successiva istruzione da eseguire
- **Registri della CPU:** permettono di interrompere il processo
  - Fondamentale per *interrompere processi*, ovvero *bloccare stato del processo*.  
Stesso discorso per il PC
- **Informazioni di scheduling:** priorità, risorse consumate. Specificate dall'utente
- **Informazioni sulla gestione della memoria:** puntatori alle varie zone di memoria.
- **Informazioni di I/O:** file aperti, operazioni in attesa, ecc...
- Eccetera...

**FIGURA. (PCB)**



## Scheduling

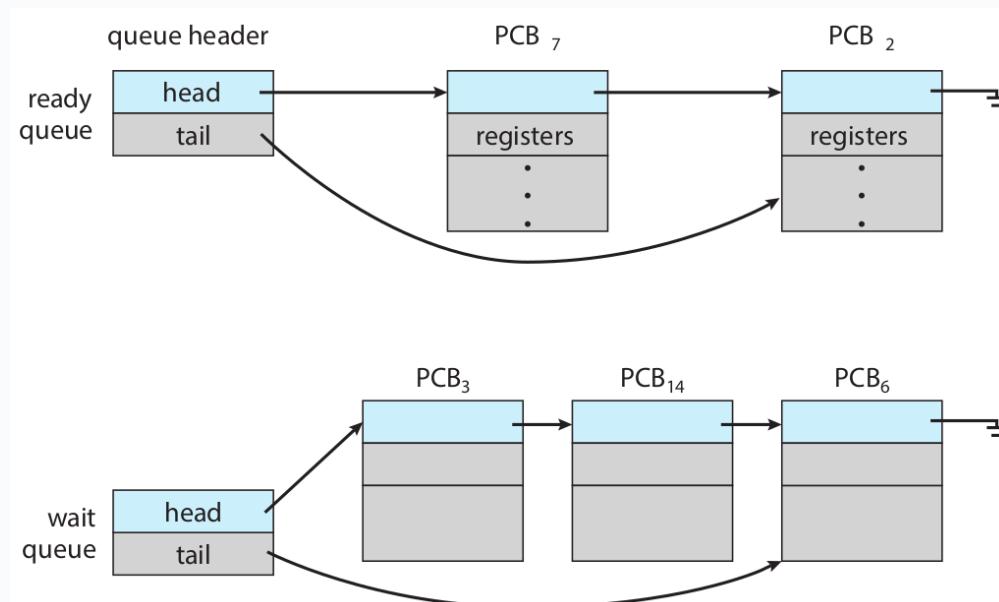
### Scheduler dei Processi

Lo scheduler dei processi ha più ruoli.

#### Ruolo 1.

Lo *scheduler dei processi* seleziona un processo da eseguire dall'insieme di quelli disponibili

- Mantiene una coda dei *processi pronti*
- Mantiene una coda dei *processi in attesa di evento*. Esempio: completare un'azione di I/O



## Idea. (*Classificazione dei processi*)

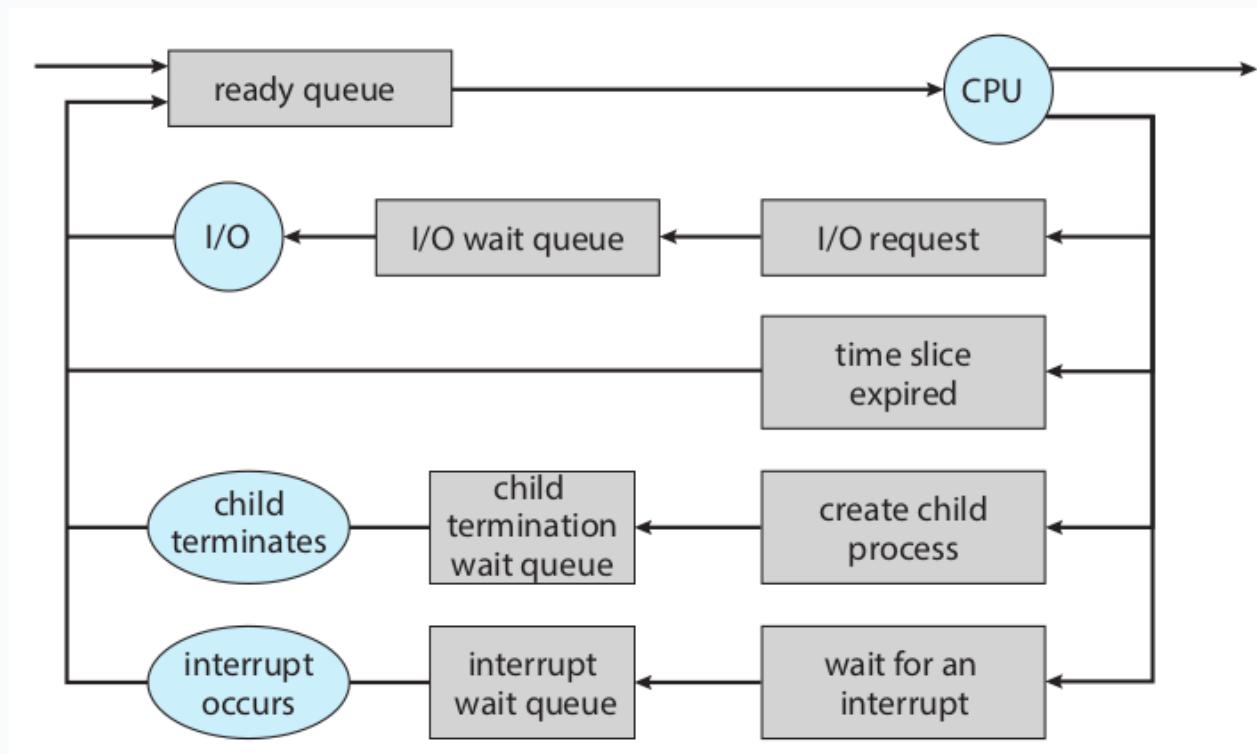
I processi possono essere classificati *in base al tipo di carico* che generano e il collo di bottiglia che li limita.

- **Processo I/O bound:** impiega la maggior parte del proprio tempo nell'*esecuzione di operazioni di I/O*
- **Processo CPU bound:** impiega la maggior parte del proprio tempo nelle *elaborazioni*

Il compito di uno *scheduler* è di *ottimizzare* l'esecuzione dei processi per farli eseguire nel minor tempo possibile

- Interviene più volte al secondo
- Gestisce l'esecuzione col meccanismo del *time sharing*

Ogni processo inizia dalla *Ready Queue* e segue il *diagramma di accodamento* finché non termina



## Ruolo 2. + Definizione. (*Context Switching*)

Lo scheduler decide a quale processa assegnare la/le CPU

Quando decide che (una) CPU deve essere assegnata a un altro processo, esso deve:

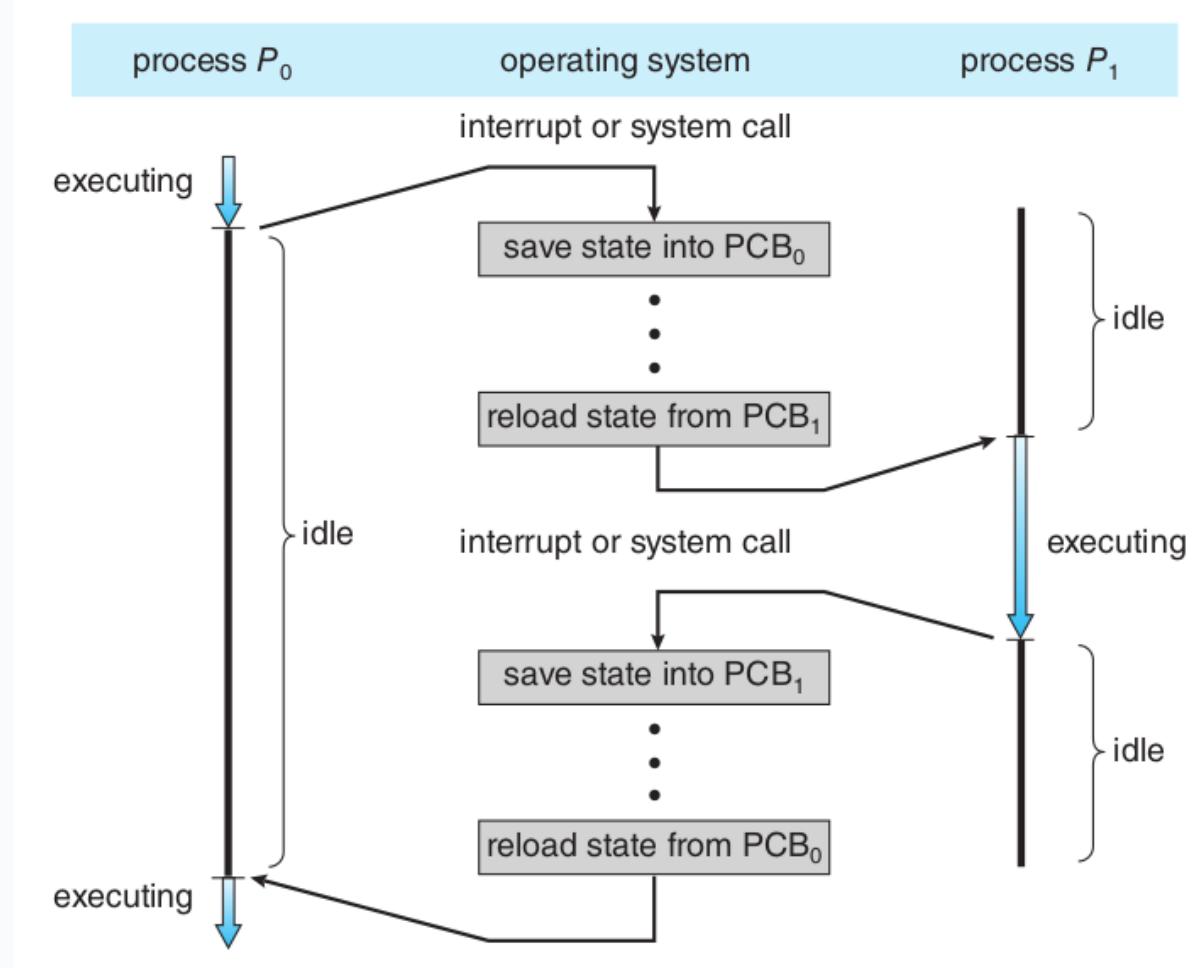
- *Salvare lo stato del processo corrente*
  - Per poterlo poi ripristinare quando il processo stesso potrà ritornare in esecuzione
- *Caricare un nuovo processo* ripristinandone lo stato salvato precedentemente
- Fa molte operazioni sui registri della **CPU**  
Si esegue un salvataggio dello stato e, in seguito, un corrispondente ripristino dello stato, detto **Context Switching**

### **Nota.** (Context Switching e l'efficienza)

Il Context Switching deve essere rapido, siccome è **tempo sprecato**, che non svolge nessun compito utile

- Il SO è ottimizzato per compiere questa azione **velocemente**
- Attualmente nell'ordine di pochi **micro secondi**
- Dipende da hardware e dalle caratteristiche del processo, specialmente la quantità di memoria usata
- Di solito facciamo **context switching** circa una migliaia di volte al secondo.

### Idea Grafica.



- Gli scarti temporali "**idle**" a destra sono **tempo perso!**

## Operazione Yield

**Operazione di Yield** (dall'inglese "dare precedenza  $\nabla$ ") : un processo dice al kernel, che per il momento **non ha operazioni da fare**.

Il kernel rimuove dalla CPU il processo e lo **riaccoda nella lista dei processi pronti** (stato **ready**)

- E' un modo per rilasciare la CPU prima che scada il quanto di tempo assegnato

- Specialmente usato per processi *real time*, per avere maggior *prestazioni*

In Linux:

```
#include <sched.h>
int sched_yield(void);
```

## Algoritmi di Scheduling

### Problema Preliminare

**Richiamo.** (*L'obiettivo dello scheduler*)

L'obiettivo di un SO è di ridurre il *tempo di esecuzione dei processi*. Tuttavia abbiamo dei problemi con questi obiettivi.

- Obiettivo *ambiguo*: conviene eseguire prima un lavoro lungo o uno corto?
- Obiettivo *complesso*: il SO non sa se un processo è lungo/corto, *CPU/I/O intensive*. Praticamente, neanche l'uomo potrebbe delineare un confine netto tra questi due tipi di processi.

#### Conseguenza.

Esistono diversi *Algoritmi di Scheduling* che si usano per determinare quale processo assegnare a una CPU

### First-Come First Served (FCFS)

**Idea** Il primo processo che richiede la CPU, la ottiene *finché non termina*

**Pro** Semplice!

**Contro:** Inefficiente (!!). Sconveniente eseguire un *processo lungo* per primo

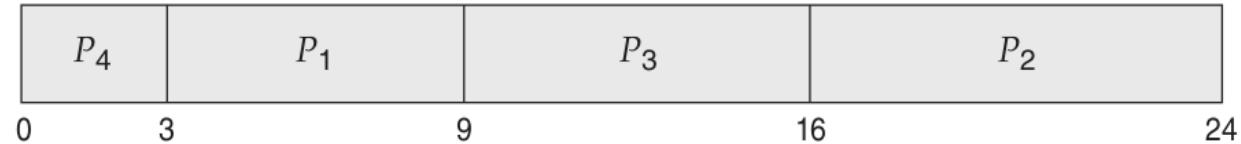


### Shortest-Job First Served (SJFS)

**Idea** Il primo più breve, la ottiene la CPU per primo

**Pro** Efficiente (!) Il tempo medio di completamento si abbassa

**Contro:** Inattuabile (!!!): non si sa quanto dura un processo, impossibile determinarlo a priori.

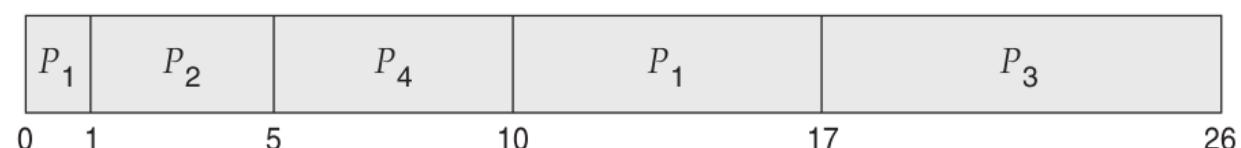


## Round Robin (RR)

**Idea** A turno, ogni processo prende la CPU *per un tempo fissato*

**Pro** Semplice ed equo, il tempo fissato è lo stesso per tutti

**Contro:** Non si possono avere processi ad alta priorità. Forse *troppto* equo



## Priority Scheduling

**Idea** Ogni processo ha una *priorità data dall'utente*. Viene eseguito il processo a priorità più alta

**Pro** Gestisce la priorità

**Contro:** Un processo a bassa priorità *potrebbe non venire mai eseguito*: funziona bene solo se l'utente definisce bene le priorità



## Multi-Level Queue Scheduling (MLQS)

**Idea.** Ci sono *code diverse* per ogni livello di priorità.

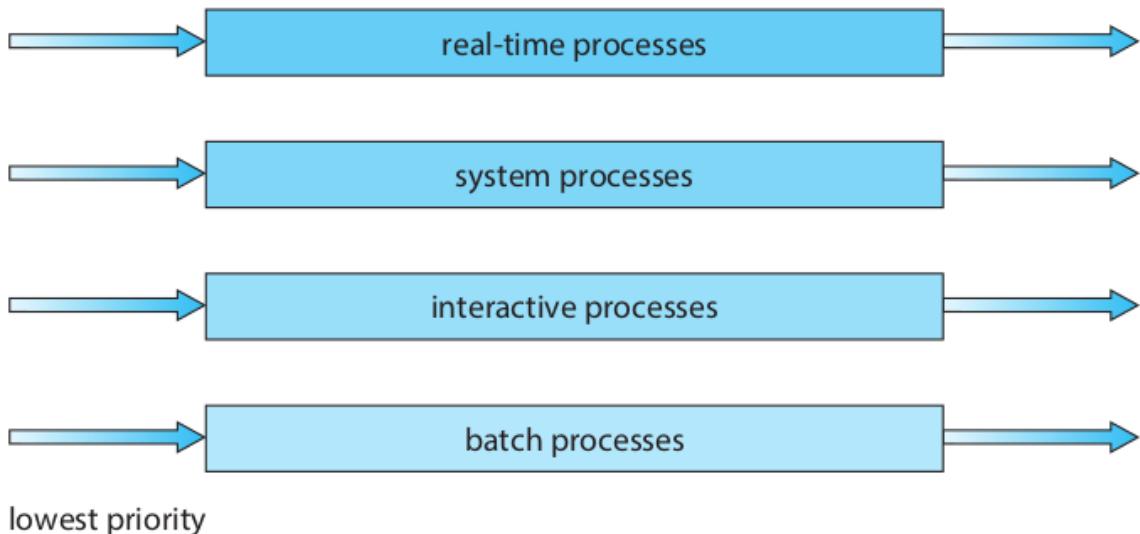
- Ogni coda ha un suo *algoritmo di scheduling*: RR, FCFS
- C'è un *algoritmo di scheduling tra code* (intra-coda)  
In questo modo c'è *flessibilità*: si può avere priorità ma non c'è il rischio che un processo non venga mai eseguito

**Pro** Flessibile, furbo e idealmente semplice

**Contro:** Complesso, un casino da implementare con molti algoritmi e code

Usato in Linux, con varianti

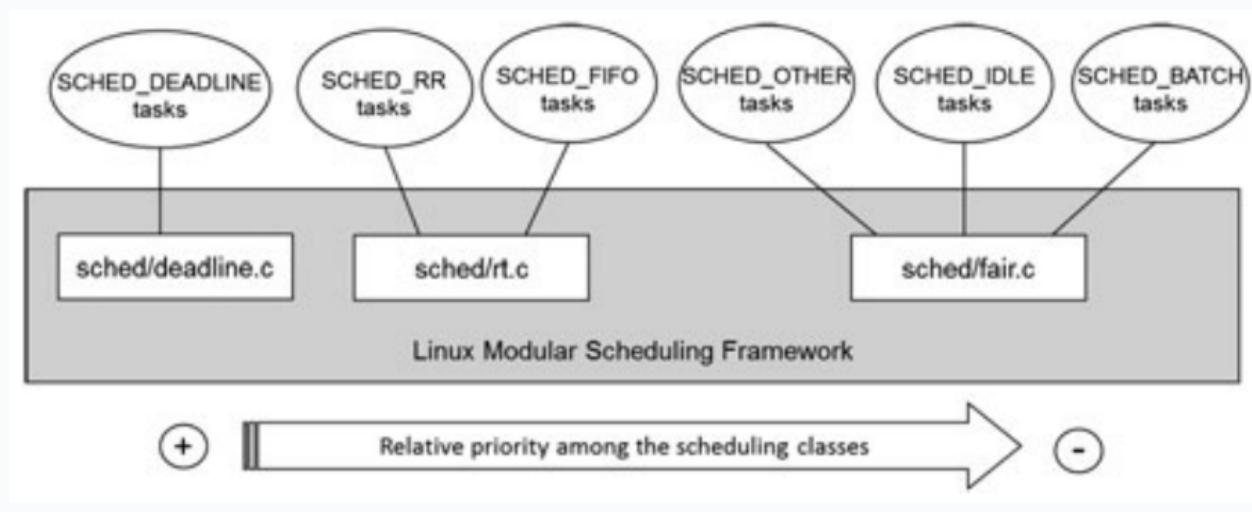
highest priority



## Linux: Completely Fair Scheduler (CFS)

In Linux lo scheduler si chiama *Completely Fair Scheduler*

- I processi sono assegnati a una *Policy di Scheduling* dall'*utente*, ognuna con *meccanismi diversi*
- Il *sistema* provvede a eseguire processi in ogni *policy*, che hanno diversi requisiti
- Le *policy* al loro interno possono gestire *priorità, deadline, ecc...*
  - Ad esempio, **SCHED\_DEADLINE** contiene i processi che hanno la garanzia di *essere eseguiti per un certo intervallo di tempo.*
  - Di default ho **SCHED\_OTHER**, con *Round Robin* + priorità
  - Le system call per definire priorità sono
    - nice(2) getpriority(2) setpriority(2) sched\_setscheduler(2) sched\_getscheduler(2) sched\_setparam(2) sched\_getparam(2) sched\_yield(2)**



## Domande

Un processo è

- **Una componente del sistema operativo**
- **Il codice eseguibile di un programma**
- **Un programma in esecuzione**

*Risposta:* Un programma in esecuzione

I processi sono identificati da:

- **Un nome**
- **Un ID numerico**
- **Non hanno identificativi**

*Risposta:* Un ID numerico

I processi sono in relazione tra loro in una struttura:

- **Ciclica**
- **Ad albero**

*Risposta:* Ad albero

Il Context Switching è:

- **La momentanea sospensione di un processo**
- **La terminazione di un processo**

*Risposta:* La momentanea sospensione di un processo

Nello scheduler di Linux, processi:

- **Hanno tutti la stessa priorità di scheduling**
- **Hanno una priorità assegnabile dall'utente**
- **Hanno una priorità calcolata automaticamente dal SO**

*Risposta:* Hanno una priorità assegnabile dall'utente

## u5-s2-operazioni

### Sistemi Operativi

#### Unità 5: I processi

## Operazioni sui processi

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

---

## Argomenti

Principalmente vedremo le *System Call* per gestire i processi. Ce ne sono poche, ma atomiche.

1. Creazione di un processo

2. Funzione **fork**
  3. Funzione **wait**
  4. Funzione **exec**
  5. Funzione **system**
  6. Funzione **exit**
  7. Altre funzioni
  8. Comandi Bash per Processi
- 

## Manipolazione dei Processi

In un SO, la manipolazione dei processi è effettuata tramite System Call

### In Windows:

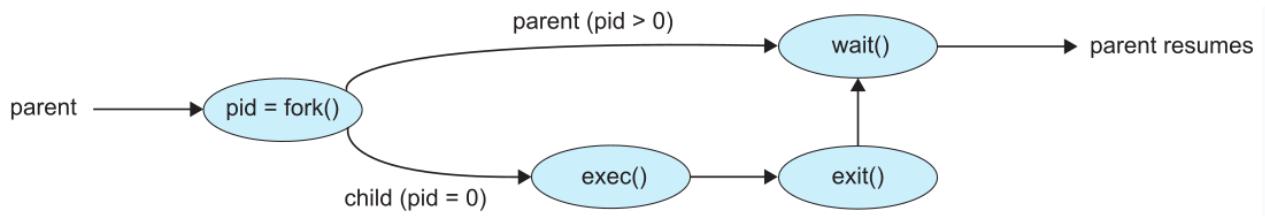
```
BOOL CreateProcessA(  
    LPCSTR      lpApplicationName,  
    LPSTR       lpCommandLine,  
    LPSECURITY_ATTRIBUTES lpProcessAttributes,  
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    BOOL        bInheritHandles,  
    DWORD       dwCreationFlags,  
    LPVOID      lpEnvironment,  
    LPCSTR      lpCurrentDirectory,  
    LPSTARTUPINFOA   lpStartupInfo,  
    LPPROCESS_INFORMATION lpProcessInformation  
) ;
```

Molto complessa!

**In Linux:** esistono 6 System Call principali

- **fork**: crea un *processo duplicato*
- **exec**: carica un *codice eseguibile* (esegue un altro programma)
- **wait**: aspetta la *terminazione di un figlio* (importante per la sincronizzazione!)
- **kill**: invia un segnale (uccide/termina processi)
- **signal**: cattura un segnale
- **exit**: *termina* il processo corrente

**FIGURA.** (*Esempio di chiamate di System Call*)



## Confronto tra Windows e Linux

### Windows vs Linux:

Windows ha una System Call complessa (**CreateProcessA**)

- Molto verboso
- Molti parametri
- Molto tipizzata

Linux preferisce System Call *semplici* e sintetiche:

- **fork** clona un processo
- **exec** permette di eseguire un file eseguibile nel processo corrente

## Funzione **fork**

### Definizione di Fork

#### Definizione.

```
#include <unistd.h>
pid_t fork (void);
```

C

*Crea un nuovo processo figlio, copiando completamente l'immagine di memoria del processo padre (data, heap, stack)*

- I due processi evolvono *indipendentemente*
- La memoria è *completamente indipendente* tra padre e figlio
- Il codice viene generalmente *condiviso tra padre e figlio*
  - Codice copy-on-write (copiato quando viene modificato)

Nota: **pid\_t** è un alias per un **int**, come **size\_t**

#### Note tecniche.

- Tutti i *descrittori dei file aperti* nel processo padre sono *duplicati nel processo figlio*

- Sia il processo child che il processo parent continuano ad eseguire *l'istruzione successiva alla fork*
- Valore di ritorno:
  - Processo *figlio*: 0
  - Processo *padre*: PID del processo figlio
  - Errore della fork: PID negativo (solo padre). Di solito non accade
- Si può sfruttare il valore di ritorno per *differenziare* il processo padre e figlio.

### Osservazioni:

Il valore di ritorno della **fork** è fondamentale!

Un programma scritto in termini di **fork** non è immediatamente comprensibile

- Operazione atomica, ma *effetti complessi*
- E' possibile creare *alberi di processi complessi*, con codice complesso

Esempio di utilizzo: *Scrivere un programma che si duplica, ed esegue rami di codici diversi.*

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    pid_t pid;
    pid = fork();
    if (pid == 0){
        printf("Sono il figlio!\n");
    }
    else{
        printf("Sono il Padre!\n");
    }
    return 0;
}
```

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void Figlio(void);
void Padre(void);
int main()
{
    pid_t pid;
    pid = fork();
    if (pid == 0)
        Figlio();
    else
        Padre();
}
void Figlio(void)
{
    int i=0;
    for(i=0;i<10;i++){
        usleep(200);
        printf("\tSono il figlio. i= %d\n",i);
    }
}
void Padre(void)
{
    int i=1;
    for(i=0;i<10;i++){
        usleep(250);
        printf("Sono il padre. i= %d\n",i);
    }
}
```

## Esempi di Fork

**Fork Bomb:** un programma che chiama la **fork** in un ciclo infinito, blocca la macchina a causa dei troppi processi

```
#include <unistd.h>
int main(void)
{
    while(1)
        fork();
}
```

### Fork Bomb in Bash:

SHELL

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

che equivale a:

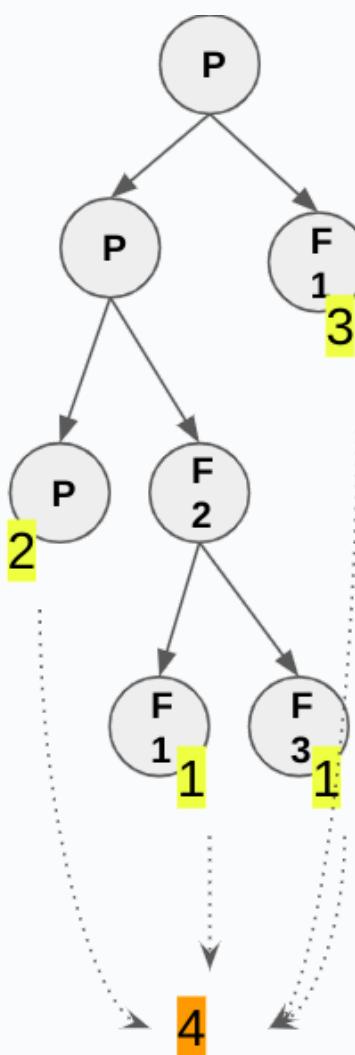
SHELL

```
myfork() {
    myfork | myfork &
}
myfork
```

## Albero di Processi

**Esercizio:** si determini l'albero di processi generato dal seguente codice e l'output generato (*importante per l'esame!*)

```
#include <stdio.h>
#include <unistd.h>
int main(){
    if (fork()){
        if (!fork()){
            fork();
            printf("1 ");
        }
    }  
    else
        printf("2 ");
    else
        printf("3 ");
    printf("4 ");
    return 0;
}
```



**Output:**

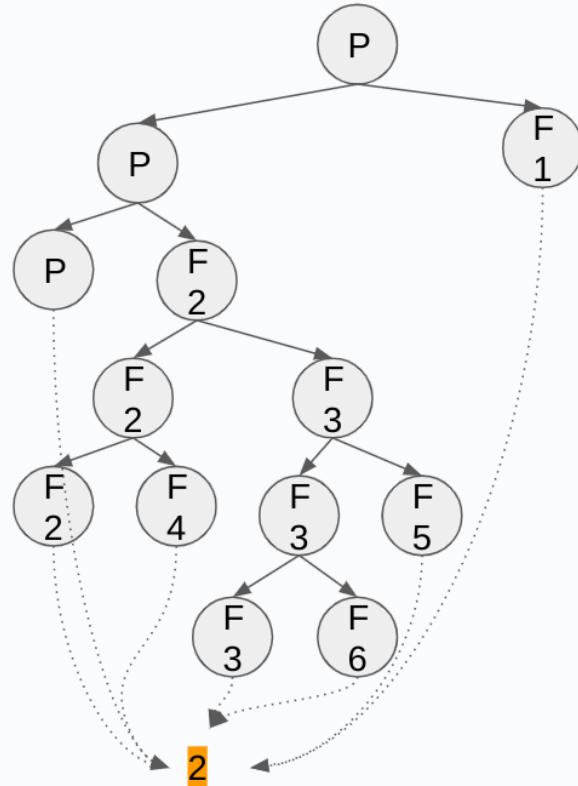
2 4 3 4 1 4 1 4

**Esercizio:** si determini l'albero di processi generato dal seguente codice e l'output generato

```
#include <stdio.h>
#include <unistd.h>

int main(){
    printf("\n");
    if (fork() && (!fork())) {
        if (fork() || fork()) {
            fork();
        }
    }

    printf("2 ");
    return 0;
}
```



**Output:**

2 2 2 2

## Osservazione sulla Bufferizzazione

**Esercizio:** si determini l'output generato dal seguente programma

```
#include <stdio.h>
#include <unistd.h>
int main(int argc,char *argv[]){
    printf("A\n");
    fork();
    printf("B\n");
    fork();
    printf("C\n");
    return 0;
}
```

**Output:**

A  
B  
B  
C  
C  
C  
C

**Esercizio:** si determini l'output generato dal seguente programma

**Nota:** non ci sono i \n nelle printf

```
#include <stdio.h>
#include <unistd.h>
int main(int argc,char *argv[]){
    printf("A ");
    fork();
    printf("B ");
    fork();
    printf("C ");
    return 0;
}
```

## Output:

```
A B C A B C A B C A B C
```

## Perchè?

Dipende dalla duplicazione della memoria dopo la **fork** e dall'I/O bufferizzato della **printf**. Infatti, l'output è **bufferizzato**! Importante per l'esame

---

## Funzione **wait**

### Definizione di Wait

Vediamo la **seconda System Call** fondamentale per i processi.

```
#include <sys/wait.h>
pid_t wait (int *status);
```

Attende la **prima terminazione** di **un** figlio. Entra in stato di **attesa**.

Argomento **status**:

- Puntatore ad un intero;
- Se non è **NULL** specifica lo stato di uscita del processo figlio (**valore restituito dal figlio**) (di solito non ci interessa)

Valore di ritorno:

- Il **PID del figlio terminato**
- 0 in caso di **errore** (esempio: non ci sono figli)

## Casistica:

- Se il processo non ha figli: **Errore**
- Se il processo ha dei figli che sono **già terminati**: ritorna **istantaneamente**
- Se il processo ha dei figli **non ancora terminati**: **blocca il chiamante** finché non termina un figlio

## Note tecniche.

La funzione **wait** **consuma** un figlio per volta.

Dopo che un figlio è stato **ritornato** al padre tramite una **wait**:

- Il SO rilascia **le risorse del processo figlio** (rappresentate sul PCB)

- Il SO mantiene informazioni su processi terminati di cui non è ancora stata effettuata una **wait**
- Traccia che il processo è esistito
- Valore di ritorno e informazioni su esecuzione
- Non verrà ritornato in successive invocazioni

Da questo comportamento possiamo definire altre *classi di processi*.

**Processi Zombie:** processo terminato il cui padre non ha ancora effettuato una **wait**

- Dopo che viene effettuata, il *processo è morto definitivamente* e non ne rimane traccia
- In questo stato, il PCB è ancora presente in memoria

**Processi Orfani:** processi in cui *padre è morto*.

- Se il padre muore, i figli *continuano l'esecuzione*
- Diventano figli del processo *init* ( $PID = 1$ )
- *Periodicamente*, *init* esegue delle **wait** per consumare gli *orfani morti*

## System Call Alternativo

Alternativamente possiamo usare un altro *system call*, più configurabile.

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```

Attende la *prima terminazione* di:

- Un qualsiasi figlio se **pid == -1** (come **wait** classica)
- Un figlio con PID **pid** se **pid>0**
- Un qualsiasi figlio il cui *group ID* è uguale a quello del chiamante se **pid == 0**
- Il figlio il cui *group ID* è uguale a **abs(pid)** se **pid < -1**
  - Esempio: -3 equivale a GID 3.
  - **group ID**: intero positivo associato a un processo. Serve per definire gruppi di processi creati dall'utente. Utile per mantenere ordine.

**Altri argomenti di **waitpid**:**

- **status** come nella **wait**, un "ritorno al valore" secondario
- **options**: controlla *se la funzione è bloccante*. È una *bitmask*.
  - 0 *bloccante*
  - **WNOHANG**: *non blocca* in caso di assenza di figlio già morto. Si usa per "controllare" se un figlio sia morto o meno: utile!
  - Altri flag per intercettare solo figli morti in *condizioni particolari*

# Grafi di Precedenze

Introduciamo una conseguenza di `fork`, `wait` molto importante: i grafi i precedenze.

## Esercizio 1

**Esercizio:** si scriva un programma che implementa il seguente grafo di precedenze con `fork` e `wait`.

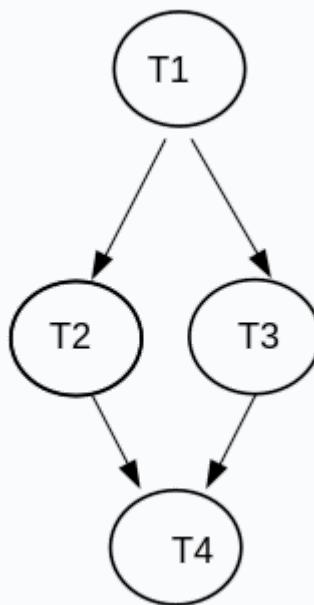
**Nota:**

Ogni biforcazione si implementa tramite una `fork` e ogni ricongiungimento tramite una `wait`

Per risolvere questi tipi di esercizi, è importante *prima fare un diagramma* che pianifica il codice, poi creare il programma effettivo!

**Importante:** questi esercizi permettono di scrivere codice che parallelizza diverse operazioni

Fondamentale per programmazione parallela



**SOLUZIONE.**

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

int main() {
    pid_t pid;
    printf ("T1\n");
    pid = fork();
    if (pid == 0) {
        printf ("T3\n");
        return 0;
    } else {
        printf ("T2\n");
        wait ((int *) 0);
    }
    printf ("T4\n");
    return 0;
}
```

## Esercizio 2

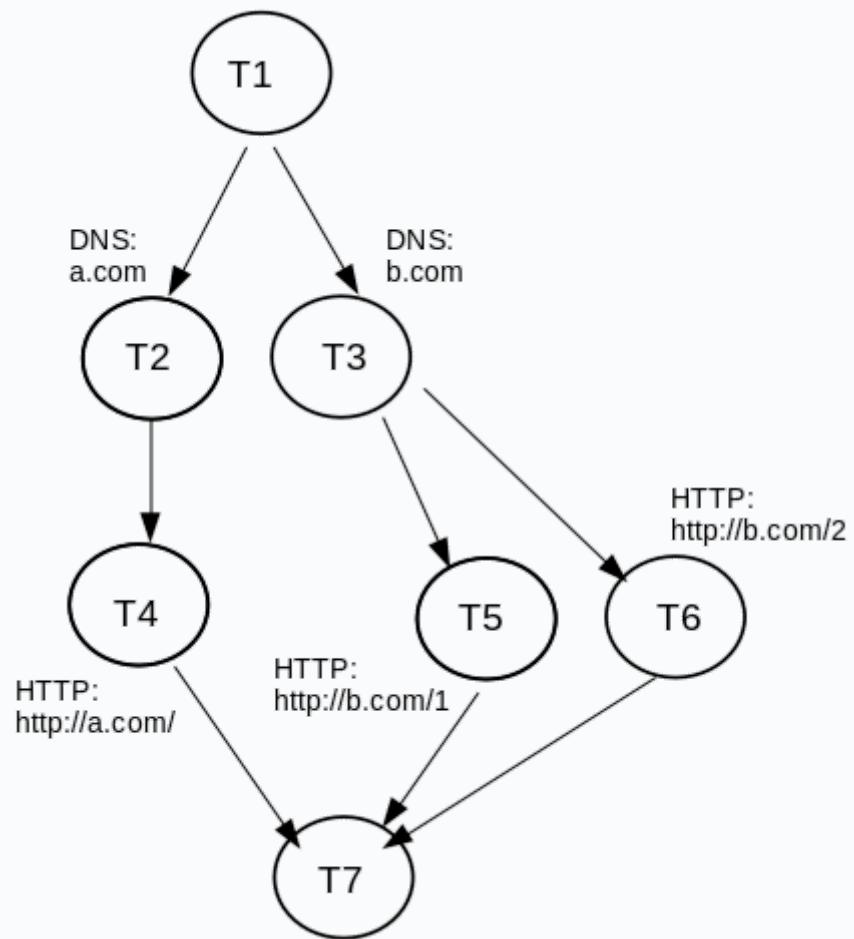
**Esercizio:** si scriva un programma che implementa il seguente grafo di precedenze con **fork** e **wait**.

**Nota:** questo è il grafo per eseguire in maniera efficiente 3 richieste HTTP alle URL.

- **http://a.com/**
- **http://b.com/1**
- **http://b.com/2**

Prima di ogni richiesta, è necessario effettuare la risoluzione DNS. Due URL hanno lo stesso dominio.

**Nota:** nei casi reali, il programmatore deve risolvere il problema efficientemente. Deve costruire da solo il grafo di precedenze.



**Soluzione.**

```

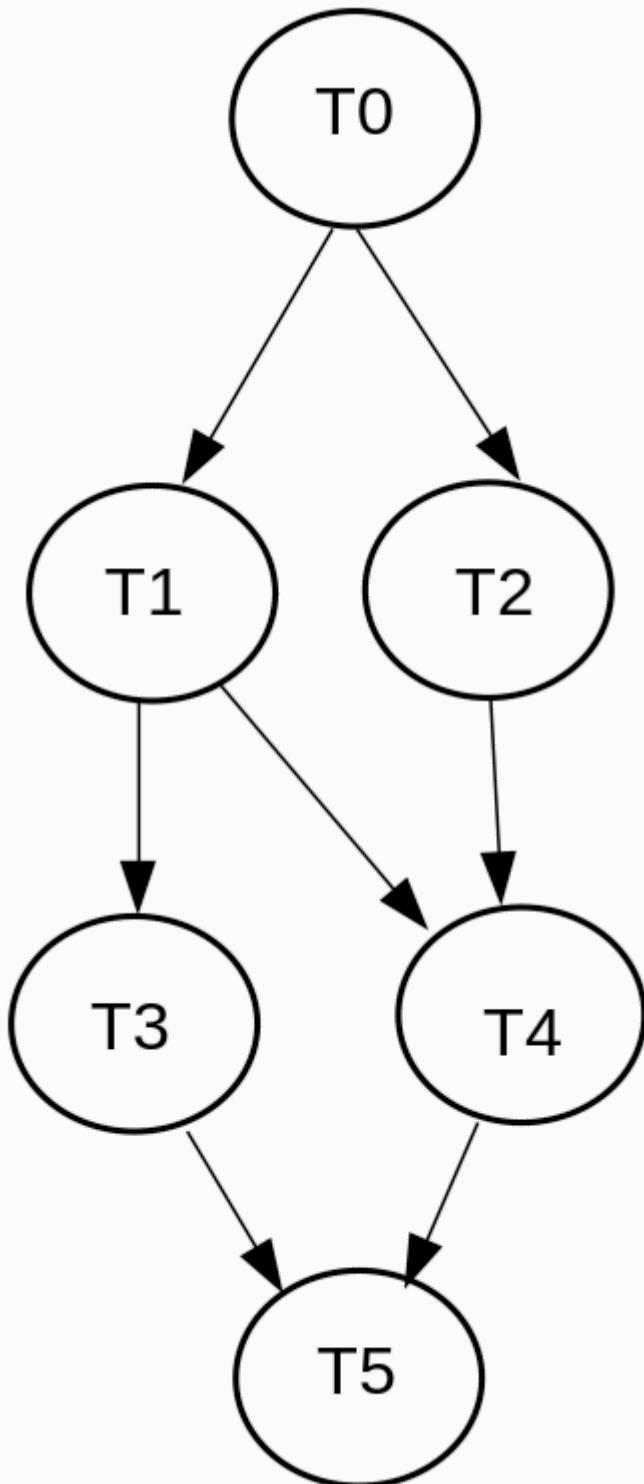
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
    pid_t pid;
    printf ("T1 - Start\n");
    pid = fork();
    if (pid == 0) {
        printf ("T3 - DNS b.com\n");
        pid_t pid2;
        pid2=fork();
        if (pid2==0){
            printf ("T6 - HTTP http://b.com/1\n");
            return 0;
        }
        else{
            printf ("T5 - HTTP http://b.com/2\n");
            wait ((int *) 0); /* Attende T6 */
            return 0;
        }
    } else {
        printf ("T2 - DNS a.com\n");
        printf ("T4 - HTTP http://a.com\n");
        wait ((int *) 0); /* Attende T3 - T5 */
    }
    printf ("T7 - Utilizzo i risultati\n");
    return 0;
}

```

**Nota:** T5 aspetta T6 che è suo figlio. T7 non può **aspettare** T6, in quanto non è suo figlio  
La **wait** **aspetta** solo sui figli, **NON** sui nipoti!

## Esercizio 3

**Esercizio:** si scriva un programma che implementa il seguente grafo di precedenze con **fork** e **wait**.



Questo grafo è **molto difficile** da realizzare mediante sole **fork** e **wait**

T4 non può **attendere** T1. Non è suo figlio! Come faccio a gestire la freccia diagonale?  
In generale:

- Si possono **attendere** solo i figli
- Ogni figlio può essere atteso una volta sola

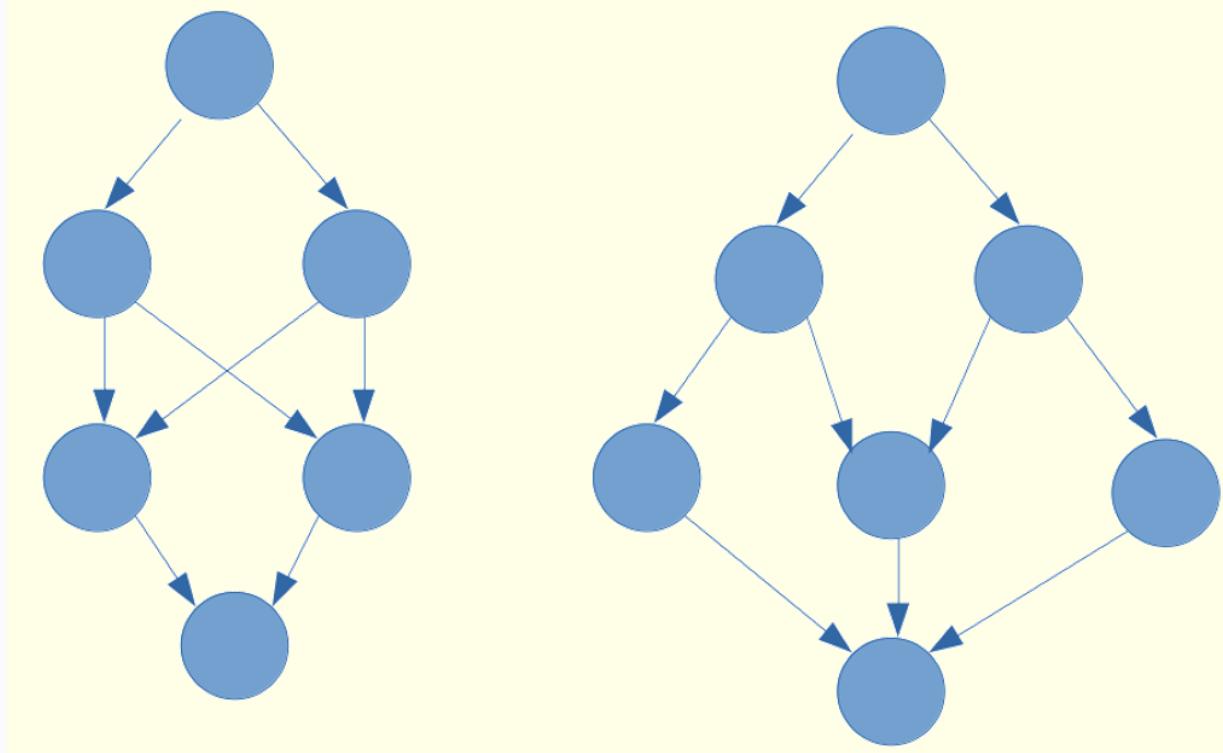
### **PROBLEMA.**

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
    pid_t pid;
    printf ("T0\n");
    pid = fork();
    if (pid == 0) {
        printf ("T2\n");
        wait ( ??? ) /* ----- IMPOSSIBILE! */
        printf ("T4\n");
    } else {
        printf ("T1 -\n");
        printf ("T3 -\n");
        wait ((int *) 0); /* Attende T4 */
    }
    printf ("T7 - Utilizzo i risultati\n");
    return 0;
}
```

Come si può implementare questo grafo? (Da fare per esercizio!)

## Grafi Impossibili

Grafi impossibili (matematicamente dimostrabili con la teoria dei grafi):



## Funzione `exec`

### Differenza tra Fork e Exec

#### FORK.

La `fork` permette di *duplicare un processo*

- Usata quando il figlio deve eseguire lo stesso programma del padre
- In programmi paralleli: web server, database

#### EXEC.

La `exec` permette di *cambiare la natura di un processo* corrente

- Caricando ed eseguendo un programma diverso
- Usata ognqualvolta bisogna avviare un nuovo programma

Quando un processo chiama una `exec`:

- Il processo viene rimpiazzato *completamente* dal codice contenuto nel file specificato (text, data, heap, stack vengono *sostituiti*)
- Il nuovo programma inizia a partire *dalla sua funzione main*, come se fosse un nuovo processo
- Il PID non cambia

#### Conseguenza.

Cosa *eredita* il processo dopo una `exec`:

- *Variabili d'ambiente*: le variabili definite nel terminale
  - Accessibili tramite la `char *getenv(const char *name)`

- PID e PPID (PID del padre)
- Privilegi, current working directory, root e home directory

Cosa *non viene ereditato*:

- File aperti se hanno il flag **close-on-exec**
- Altrimenti lasciati aperti

## Funzioni Exec

Esistono 7 versioni della **exec**.

Hanno la *stessa funzione*, varia *il modo in cui ricevono gli argomenti*. Sono le seguenti

```
#include <unistd.h>

int execl(const char *pathname, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *pathname, const char *arg, ..., char *const envp[]);
int execv(const char *pathname, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(char *pathname, char *argv[], char* envp[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

## Path

Le funzioni con **p** ricevono il *nome dell'eseguibile* e non il path.

- Il SO rintraccia l'eseguibile nelle cartelle dei programmi installati nel sistema
- Che sono definite nella variabile d'ambiente **PATH**

**Esempio:** si equivalgono

```
execlp("cp", ...);
execl("/usr/bin/cp", ...);
```

Perchè, sul mio PC:

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin
```

# Passaggio di Argomenti

- Le funzioni con **l** ("list") specificano gli *argomenti* del *nuovo programma* tramite una lista di argomenti. Simile a **printf**.
  - Esempio:** `execlp("cp", "cp", "file1", "file2");`
  - Nota:** Non dimenticare **argv[0]**!
- Le funzioni con **v** specificano gli *argomenti* del nuovo programma tramite un unico vettore di puntatori a **char**. Equivalente a **argv** nel **main**
  - Il primo argomento deve contenere il nome del file associato all'eseguibile che viene caricato (**argv[0]**)
  - L'array di puntatori deve essere terminato da un puntatore **NULL**
    - Mancando **argc**, questo serve a comunicare la lunghezza del vettore

Esempio:

```
// Semplice
execlp("cp", "cp", "file1", "file2");

// Generico
const char *args[4];
args[0] = "cp";
args[1] = "file1";
args[2] = "file2";
args[3] = NULL;
execvp("cp", args);
```

**Nota:** Non c'è un modo per specificare la lunghezza nel vettore, per convenzione si pone l'ultimo argomento come **NULL**.

## Variabili d'Ambiente

Le funzioni con **e** ricevono un vettore di variabili d'ambiente. Quindi esse *non* vengono ereditate dal processo esistente.

- Le variabili d'ambiente sono specificate nell'ultimo argomento tramite un vettore di puntatori a **char** (come con **execv-**)
  - Terminato da puntatore **NULL**
  - Ogni elemento è una stringa nella forma **nome=valore**

```
char *const args[] = {"ls", "/tmp", NULL};  
execv("/usr/bin/ls", args);  
  
char *const envs[] = {"a=1", "b=2", NULL};  
execve("/usr/bin/ls", args, envs);
```

## Funzione **execve**

### Osservazione:

- La funzione **execve** è una System Call.
- Le altre funzioni sono di libreria, e invocano la **execve** dopo aver correttamente gestito e aggiustato i parametri

## Esercizi su Exec

1. **Esercizio:** si scriva una *simple shell* usando le funzioni **fork**, **wait** e **exec**.  
Importante per l'esame!

### Soluzione.

```

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#define MAXLINE 128
int main() {
    char buf[MAXLINE];
    pid_t pid;
    int status;
    printf("%% "); /* prompt */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        if (buf[strlen(buf) - 1] == '\n')
            buf[strlen(buf) - 1] = 0;
        if ((pid = fork()) < 0) {
            printf("errore di fork "); exit(1);
        } else if (pid == 0) { /* figlio */
            execlp(buf, buf, NULL);
            printf("non posso eseguire: %s\n", buf);
            exit(127);
        } else
            if ((pid = waitpid(pid, &status, 0)) < 0) /* padre */
                {printf("errore di waitpid"); exit(1);}
            printf("%% ");
    }
    exit(0);
}

```

**Nota:** per gestire gli *argomenti* dei comandi invocati, bisognerebbe manipolare le stringhe

2. **Esercizio:** si consideri il seguente programma.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main (int argc, char ** argv) {
    char str[10];
    int n;
    n = atoi(argv[1]) - 1;
    printf ("%d\n", n);
    if (n>0) {
        sprintf (str, "%d", n);
        execl (argv[0], argv[0], str, NULL);
    }
    printf ("End!\n");
    return 1;
}
```

Cosa viene stampato eseguendo **./prog 5** ?

### Soluzione.

```
4
3
2
1
0
End!
```

## Funzione **system**

### Definizione di System

E' una funzione di libreria che invoca un comando **Bash** e ne attende la conclusione

- Utile per usare programmi esterni in un programma
- Internamente usa: **fork**, **exec** e **wait**

C

```
#include <stdlib.h>

int system(const char *command);
```

Equivale a una **fork** il cui figlio esegue:

C

```
exec("/bin/sh", "sh", "-c", command, (char *) NULL);
```

**Valore di ritorno:** il valore di ritorno *del comando eseguito*

## Implementazione di System

**Implementazione semplice:** (*Importante!*)

C

```
int system(const char *cmd)
{
    int stat;
    pid_t pid;
    if (cmd == NULL)
        return(1);
    if ((pid = fork()) == 0) /* Son */
        exec("/bin/sh", "sh", "-c", cmd, (char *)0);
        _exit(127);
    if (pid == -1) {
        stat = -1; /* Error */
    } else { /* Father */
        while (waitpid(pid, &stat, 0) == -1) {
            if (errno != EINTR){
                stat = -1;
                break;
            }
        }
    }
    return(stat);
}
```

## Esercizi su System

**Esercizio:** Si scriva un programma che fa il listing dettagliato di una cartella.

- La cartella è passata come argomento
  - Se non ci sono argomenti, lista la directory corrente
- Usando **ls -lh cartella**

**Soluzione.**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main (int argc, char * argv[1]) {
    char command[50] = "ls -lh ";
    if (argc == 2)
        strcat(command, argv[1]);
    system(command);
    return(0);
}
```

## Terminazione di un Processo

Ci sono diversi modi per terminare un processo:

### Funzione Exit

#### 1. Modo Standard (valore di ritorno, funzione exit)

- Dal **main** avviene una **return**

```
return status;
```

- Viene chiamata la funzione **exit**  
**c #include <stdlib.h> void exit(int status);**

Tutti i buffer (di console e file) vengono *flushed*

L'argomento **status** è ritornato al SO

Per permettere queste operazioni di pulizia, vengono chiamate tutte le *funzioni di chiusura*:

1. Della *libreria standard*

2. *Definite dall'utente* tramite la funzione **int atexit(void (\*function)(void));**

```
C

void fun(void) { printf("Exiting\n"); }

int main()
{
    atexit(fun);
    exit(10);
}
```

Viene stampato **Exiting**

## System Call \_exit

### 2. System Call **\_exit**

```
C

#include <unistd.h>
void _exit(int status);
```

Termina immediatamente *senza controllare i buffer*.

Invocata nei processi *figli* che potrebbero leggere *buffer* in stato intermedio dei padri

Usata specialmente dopo **exec** fallite

- Il figlio *non dovrebbe eseguire nessuna istruzione* dopo la **exec**!
- I buffer possono contenere *dati del padre che non devono essere scritti dai figli*

#### Nota:

- La **\_exit** è una System Call
- La **exit** è una funzione di libreria. Fa pulizia e poi invoca la **\_exit**

## Funzione Abort

### 3. Terminazione Anomala:

- Viene ricevuto un **segnale** non gestito (vedremo)
- Il programma chiama la **abort**.

```
#include <stdlib.h>
void abort(void);
```

## Riassunto

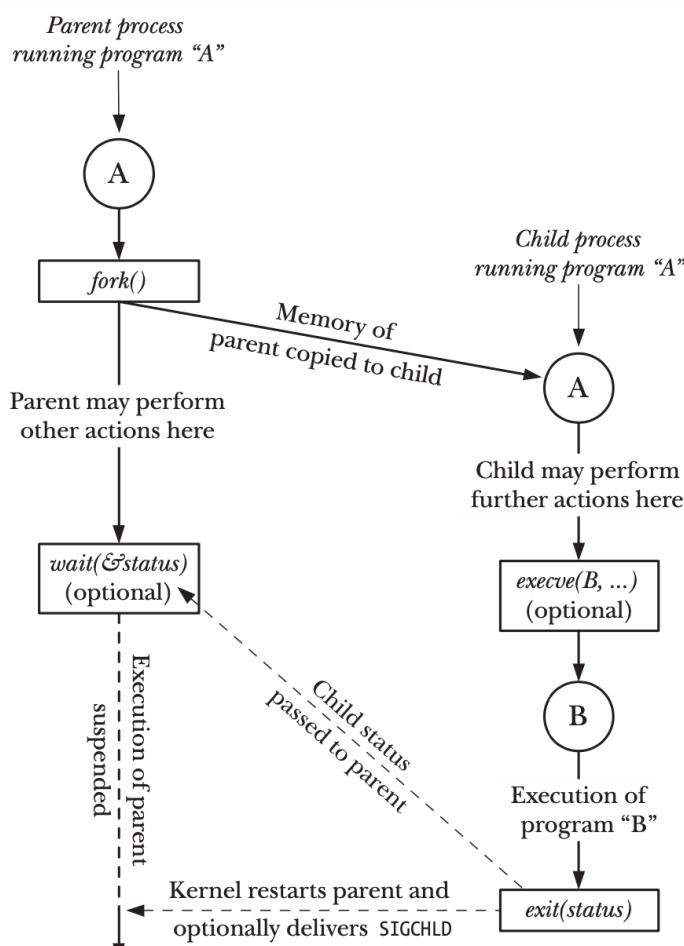
In qualunque modo termini il processo, il kernel compie le seguenti azioni:

- *Rimozione della memoria* utilizzata dal processo
- Chiusura dei *descrittori aperti*

Stato di un processo: raccolto con `wait()`, `waitpid()`. Diventa un *processo-zombie*.

## Riassunto delle Operazioni con Processi

- **fork** duplica il processo corrente
- **execve** tramuta il processo corrente in un altro programma
- **exit** termina il processo corrente (uguale a **return** dal **main**)
- **wait** blocca finchè un processo figlio non termina



## Ottenere PID

Vediamo altre due system call (funzioni) per ottenere *PID* di processi.

```
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

- La **getpid()** ritorna il PID del processo chiamante
- La **getppid()** ritorna il PID del *padre* del processo chiamante

## Comandi Bash per Processi

Vediamo alcuni *comandi Bash* per i processi. Iniziamo riassumendo quelli già noti

- **ps**: lista i processi del sistema
  - Di default mostra solo processi figli del terminale corrente.
    - Con l'opzione **a** mostra tutto
    - Uso comune: **ps fax**
  - Di default, mostra solo processi che sono in foreground (hanno una shell)
  - Con opzione **x** mostra anche quelli in background
  - Opzioni utili: **u** mostra utente proprietario. **f** rende graficamente gerarchia padre-figlio
- **top**: mostra i processi in maniera interattiva
- **htop**: come **top** ma grafica migliorata
  - **top**, **htop** sono l'equivalente del *Task Manager* per il *Linux* sul terminale.
- **which**: fornisce il path assoluto di un programma di sistema. *Molto utile!*

SHELL

```
$ which ls
/usr/bin/ls
```

- I comandi di sistema vengono cercati nelle cartelle indicate nella variabile d'ambiente **\$PATH**
  - Solitamente: **/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin**
- **pgrep**: stampa il PID di tutti i processi di un programma. Letteralmente è **ps + grep**.

```
$ pgrep chrome
480492
480498
480505
```

## Sintassi per Processi in Bash

### Note tecniche. (*importanti!*)

Esecuzione di processi figli da *script bash*:

- Un comando che termina con `&` viene *eseguito in background* (si dice "detached")
  - Viene eseguita una `fork` e una `exec` per eseguire il comando
  - Non esegue la `wait`. Lo script e il programma eseguono *in parallelo*
- Il PID del processo appena creato può essere ottenuto con `$!`
  - Sovrascritto a ogni processo creato! Usare *attentamente*
- Si può usare il comando `wait [PID]` per aspettare
  - Attende il figlio `PID` se specificato, altrimenti un figlio qualsiasi

**Esempio:** Quanto ci mette a eseguire questo codice? (*classica domanda quiz per l'esame!*)

```
sleep 4 & # Sleep viene eseguito in background
PID=$! # Lo script recupera il PID
sleep 2
wait $PID # Lo script attende che sleep termini
```

**Soluzione.** Ci mette 4 secondi, non 6! I programmi eseguono in parallelo

## Informazioni sui Processi

### Il `/proc` file system.

I sistemi Linux/POSIX espongono informazioni sui processi correnti tramite uno *Pseudo File Virtuale* detto `procfs`

- File System Virtuale
- Montato in `/proc` in automatico dal `kernel`
- Permette a chiunque di conoscere lo stato dei processi in esecuzione
- *Tramite normali letture da file*
- Buona alternativa a System Call complicate

- Non *corrisponde* fisicamente sul disco, è un modo per *mappare* le info sul *file system*.

### Sottocartelle del **/proc** file system.

1. Le informazioni su un processo *PID* si trovano nella *directory* **/proc/PID**
- Il *file* **/proc/PID/status** contiene varie informazioni:

SHELL

```
$ cat /proc/1566/status
```

Name: grep

State: R (running)

Tgid: 5452

Pid: 5452

PPid: 743

...

VmPeak: 5004 kB

VmSize: 5004 kB

VmLck: 0 kB

VmHWM: 476 kB

VmRSS: 476 kB

2. La **subdirectory** **/proc/PID/fd** contiene un link per ogni file aperto dal processo

- Il nome di questi link è il *numero del descrittore* usato nel processo
- Ricordare: ogni file aperto *identificato da un numero*

### Esempio:

SHELL

```
/proc/1968/fd/1
```

Rappresenta lo **stdout** del processo 1968.

- Ricorda: 0 è **stdin**, 1 è **stdout**, 3 è **stderr**

Altri *file/subdirectory* del processo *PID* sotto **/proc/PID**

File	Description (process attribute)
cmdline	Command-line arguments delimited by \0
cwd	Symbolic link to current working directory
environ	Environment list <i>NAME=value</i> pairs, delimited by \0
exe	Symbolic link to file being executed
fd	Directory containing symbolic links to files opened by this process
maps	Memory mappings
mem	Process virtual memory (must <i>lseek()</i> to valid offset before I/O)
mounts	Mount points for this process
root	Symbolic link to root directory
status	Various information (e.g., process IDs, credentials, memory usage, signals)
task	Contains one subdirectory for each thread in process (Linux 2.6)

3. Il file system **/proc** fornisce anche molte informazioni sul sistema e possibilità di configurazione
- **/proc/cpuinfo**: informazioni su CPU
  - **/proc/meminfo**: informazioni su memoria

Directory	Information exposed by files in this directory
/proc	Various system information
/proc/net	Status information about networking and sockets
/proc/sys/fs	Settings related to file systems
/proc/sys/kernel	Various general kernel settings
/proc/sys/net	Networking and sockets settings
/proc/sys/vm	Memory-management settings
/proc/sysvipc	Information about System V IPC objects

4. **sysfs**: montato in **/sys**, contiene informazioni sullo *stato del kernel* e sulle periferiche
- Complementare a **/proc**
5. **/dev**: contiene i file speciali che rappresentano le *periferiche*
- Dispositivi a blocchi:
    - Dischi: **/dev/sda1**, **/dev/hda2**
    - CDRom: **/dev/cdrom**; Floppy: **/dev/fd0**
  - Dispositivi a carattere: tastiera, mouse
  - Quando si legge/scrive a questi file speciali, viene invocato il *driver* della periferica corrispondente
  - Volendo (non consigliato!) si potrebbe fare delle operazioni su questi file virtuali.

## Domande

L'esecuzione del seguente codice quanti processi genera (incluso il processo che esegue il **main**) ?

```
#include <stdio.h>
#include <unistd.h>
int main(){
    int N = 2;
    for (i=0; i<N; i++)
        fork();
}
```

- 2
- 3
- 4
- 6

**Risposta:** 4 ( $2^2$ )

Un processo il cui processo padre muore:

- Viene terminato dal SO
- Riceve un segnale dal SO
- Viene ereditato (diventa figlio) dal processo init

**Risposta:** Viene ereditato (diventa figlio) dal processo init

Cosa stampa il seguente codice?

```
#include <stdio.h>
#include <unistd.h>
int main(){
    if ( fork() ){
        printf("A\n");
    }else{
        fork();
        printf("B\n");
    }
}
```

- |     |     |     |     |
|-----|-----|-----|-----|
| • A | • A | • A | • A |
| B   | A   | B   | A   |
| B   | A   |     | B   |

**Risposta:**

A

B

B

La System Call `execve` crea un nuovo processo?

- Sempre
- Mai
- Dipende da come viene invocata

**Risposta:** Mai (cambia la natura del processo)

La funzione `system` crea un nuovo processo?

- Sempre
- Mai
- Dipende da come viene invocata

**Risposta:** Sempre (è stata implementata mediante fork)

## u5-s3-segnali

### Sistemi Operativi

#### Unità 5: I processi

## I Segnali

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

---

## Argomenti

1. Concetto di segnale
  2. Segnali in Linux
  3. System Call `sigaction`
  4. System Call `kill`
  5. System Call `raise`
  6. System Call `pause`
  7. System Call `alarm`
  8. Considerazioni
  9. Segnali nella shell
-

# Concetto di segnale

## Richiamo agli Interrupt

### RICHIAMO. (*Interrupt*)

In quasi tutti i sistemi ad elaboratore, esistono gli *interrupt*:

Un *interrupt* informa la CPU che deve interrompere il compito corrente per eseguire un'azione impellente (1).

Un interrupt viene generato da:

- Un *dispositivo hardware* che vuole notificare al sistema un evento (di solito una linea elettrica, circa 5V)
- Particolari *istruzioni nel codice* (e.g., istruzione **INT**)
  - Quando un processo chiama una System Call genera un interrupt software
  - Sono delle "*finte interrupt*"

## Definizione di Segnale

Un *segnale* permette la *gestione di eventi asincroni* che interrompono il normale funzionamento di un *processo*

- E' un interrupt software
  - In particolare mirate sui *processi*
- Notifica un evento a un processo specifico

Possono essere *generati* da

- *Kernel* per comunicare eventi eccezionali:
  - Condizioni di errore
  - Azioni dell'utente (e.g., **CTRL+C** su tastiera)
- Un *altro processo* (se ne ha i permessi):
  - Permettono una primitiva comunicazione tra processi
  - Usando la system call **kill**

## Notizie Storiche

Esistono dalle *prime versioni di Unix* (siamo negli anni '90; 1)

- Formalizzati in Unix 4
- Rappresentava un *primo meccanismo di Inter-process communication*

In principio erano inaffidabile e gestiti in modo *best-effort*

- *Potevano andare perduti*
- La gestione era *complicata*
- *Poca configurazione* possibile

I segnali esistono anche in *Windows*, sebbene abbiano un funzionamento leggermente diverso (in realtà molto!). Tuttavia ci concentreremo sui *segnali in Linux*.

---

## Segnali in Linux

### Molteplicità dei Segnali

Esistono *diversi tipi di segnali* in Linux

- Dipende dalle versioni di Linux
- Comando **kill -l** lista i segnali
  - 64 in Ubuntu 20
  - Circa 10 saranno importanti per noi

Ogni segnale ha un *identificatore mnemonico e numerico*

- Identificatori di segnali iniziano con i tre caratteri SIG (che sta per "SI~~G~~nal")
  - Es. **SIGINT** è il segnale di interruzione e ha numero 2
- I nomi simbolici corrispondono ad un intero positivo (in C queste costanti vengono definite su **signal.h**)

Come detto prima, ogni segnale viene generato da un *evento specifico nel SO*, o *manualmente* da un processo.

Un segnale può avere i seguenti effetti su un processo:

### Effetti dei Segnali

#### Importanti

1. Viene *ignorato*
2. *Termina il processo*
3. Interrompe momentaneamente il processo. Esegue una *funzione handler*.  
Dopodiché il processo riprende

#### Secondari

4. Crea un *core dump*: un *file* che contiene lo stato del programma per poter essere debuggato
5. *Stoppa* il processo
6. *Fa ripartire* il processo

Adesso vediamo *in specifico* quali segnali hanno i comportamenti elencati sopra

#### Segnali ignorati di default:

- **SIGCHLD**: inviato al padre quando un figlio termina

## Segnali che di default terminano un processo:

Sono tanti!

- **SIGINT**: viene inviato al processo in esecuzione quando si preme **CTRL+C**; vedremo che si può modificare il comportamento
- **SIGABRT**: inviato da system call **abort()** (ha un'utilità abbastanza discutibile)
- **SIGFPE**: inviato da eccezione aritmetica (ad esempio l'operazione  $\frac{1}{0}$ )
- **SIGHUP**: Inviato ad un processo se il terminale *viene disconnesso* (sta per "Signal HangUP")
- **SIGKILL**: Maniera sicura per uccidere un processo.  
**Nota:** Non si può creare un handler per **SIGKILL**! Ovvero il suo comportamento non può essere modificato!
- **SIGSEGV**: Accesso di memoria non valido
- **SIGTERM**: Segnale di *terminazione* normalmente usato. Generato dal comando **kill** di default
- **SIGUSR1** e **SIGUSR2**: generati solo da processi utente, mai dal SO. Servono per comunicazione tra processi
  - Di conseguenza sono da *modificare!*

---

## Lista completa degli segnali su Linux

Lista più completa.

Il comportamento di default può essere modificato:

- Per *ignorare* un segnale
- Per *gestirlo* tramite un *handler*
- *NON* per indurre Terminazione o Core Dump
- Tranne **SIGKILL** e **SIGSTOP** (non vanno mai modificati!)

Name	Description	Default
SIGABRT	Abort process	Core
SIGALRM	Real-time timer expiration	Term
SIGBUS	Memory access error	Core
SIGCHLD	Child stopped or terminated	Ignore
SIGCONT	Continue if stopped	Cont
SIGFPE	Arithmetic exception	Core
SIGHUP	Hangup	Term
SIGILL	Illegal Instruction	Core
SIGINT	Interrupt from keyboard	Term
SIGIO	I/O Possible	Term
SIGKILL	Sure kill	Term
SIGPIPE	Broken pipe	Term
SIGPROF	Profiling timer expired	Term
SIGPWR	Power about to fail	Term
SIGQUIT	Terminal quit	Core
SIGSEGV	Invalid memory reference	Core
SIGSTKFLT	Stack fault on coprocessor	Term
SIGSTOP	Sure stop	Stop
SIGSYS	Invalid system call	Core
SIGTERM	Terminate process	Term
SIGTRAP	Trace/breakpoint trap	Core
SIGTSTP	Terminal stop	Stop
SIGTTIN	Terminal input from background	Stop
SIGTTOU	Terminal output from background	Stop
SIGURG	Urgent data on socket	Ignore
SIGUSR1	User-defined signal 1	Term
SIGUSR2	User-defined signal 2	Term
SIGVTALRM	Virtual timer expired	Term
SIGWINCH	Terminal window size changed	Ignore
SIGXCPU	CPU time limit exceeded	Core
SIGXFSZ	File size limit exceeded	Core

## Signal Handler

Un processo può definire un *signal handler*.

- Una *funzione* che viene eseguita quando il processo riceve il segnale
- Se non lo fa, c'è il *comportamento di default*

L'idea si esprime con la seguente frase: "*Se e quando ricevi un certo segnale, esegui questa funzione*"

## Stato di un Segnale

Fasi di vita di un segnale:

1. **Generazione:** da parte del *kernel o di un processo*
2. **Consegna:** nel più breve tempo possibile consegna il segnale al processo.
  - Finché un segnale non è consegnato è *pending*

- Questa è una fase importante!
- Vengono gestiti mediante una **bitmask** (lo terremo in mente per l'osservazione seguente)

### 3. Gestione:

- Il kernel avvia la **funzione handler** del processo nel caso ce ne sia una
- Altrimenti compie l'**azione di default** per quel segnale (termina o ignora)

### Osservazione:

I segnali non vengono accodati.

I segnali pendenti per un processo sono gestite da una **mask**.

- Se lo **stesso segnale** è generato **più volte** prime che sia consegnato, esso lo sarà **una sola volta**

## Manipolazione dei Segnali

Adesso vediamo una serie di **system call** su C per manipolare le System Call. Si userà la libreria **<signal.h>**.

### System Call **sigaction**

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

*Modifica il comportamento* del processo corrente a un segnale particolare.

#### Argomenti:

- **signum**: segnale da trattare
- **act**: puntatore a struttura che definisce trattamento
- **oldact**: puntatore a comportamento precedente. Può servire per ristabilire il comportamento precedente. Ci permette di poter *sapere il vecchio comportamento*.

**Ritorna** -1 se c'è stato errore

Adesso vediamo la struct **sigaciton**.

```

struct sigaction {
    void (* sa_handler )( int );
    sigset_t sa_mask ;
    int sa_flags ;
    void (* sa_restorer )( void );
};

```

- **sa\_handler** specifica il comportamento
  - Se **funzione**, specifica un *handler* (un puntatore a funzione che ritorna nulla)
  - Se **SIG\_IGN** ignora
  - Se **SIG\_DFL** ripristina comportamento di default
- **sa\_mask**: *segnali da bloccare* mentre l'handler è in esecuzione (gestita con **bitmask**)
  - Si inizializza con la funzione **int sigemptyset(sigset\_t \*set);**
  - Nel corso non faremo altre operazioni su questo campo
- **sa\_flags**: flag (non vediamo). Settiamo sempre a 0
- **sa\_restorer**: per uso interno

**Esempio:** si crei una funzione per ignorare un segnale definito dal chiamante

```

int ignoreSignal ( int sig )
{
    struct sigaction sa ;
    sa.sa_handler = SIG_IGN ;
    sa.sa_flags = 0;
    sigemptyset ( &sa.sa_mask );
    return sigaction ( sig , &sa , NULL );
}

```

### Funzione Handler.

La funzione handler deve prendere un argomento **int**

- Quando *invocata dal SO*, contiene il numero del segnale
- Deve ritornare **void**

```
void myHandler ( int sig )
{
    /* Actions to be performed when signal
       is delivered */
}
```

**Nota:** L'handler è una funzione del programma.

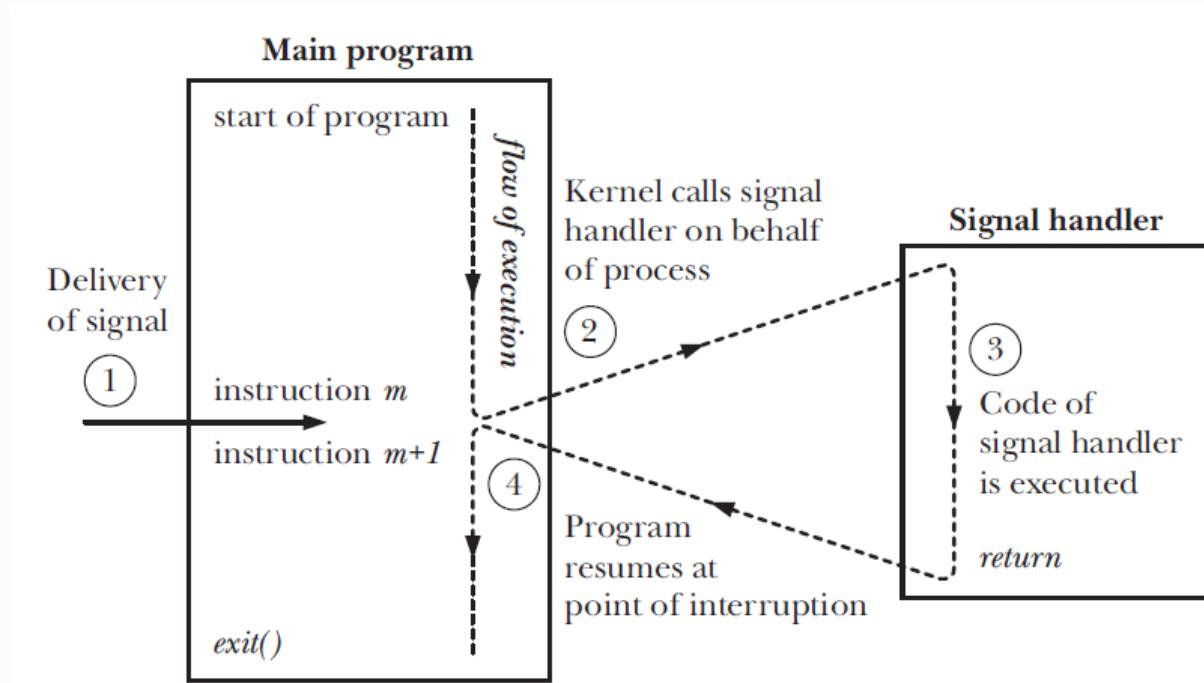
- Che viene invocata **automaticamente** dal SO alla ricezione del segnale
- E **non dal programmatore**. Potrebbe, ma non ha senso

## Funzionamento Interno di **sigaction**

Viene invocata **automaticamente dal kernel** alla consegna del segnale

Il programma si **interrompe**, esegue l'handler

Infine, **continua l'esecuzione** dal punto di interruzione



## Esercizio con **sigaction**

**Esempio:** si crei un programma che gestisce i segnali **SIGINT**, **SIGHUP** e **SIGTERM**

```
#include <signal.h>
#include <stdio.h>
void func(int signum){
    printf("ricevo %d\n", signum);
}

int main (){
    struct sigaction new_action, old_action;

    new_action.sa_handler = func;
    sigemptyset (&new_action.sa_mask); /* Si noti l'uso di sigemptyset */
    new_action.sa_flags = 0;

    sigaction (SIGINT, &new_action, NULL);
    sigaction (SIGHUP, &new_action, NULL);
    sigaction (SIGTERM, &new_action, NULL);

    while(1) ;
    return 0;
}
```

Per terminare il programma, bisogna mandargli un segnale **SIGKILL**. Lo si fa digitando  
**pkill -KILL <nome prog>**

---

## System Call **signal**

Esiste la System Call **signal**, più a basso livello.

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

Argomenti:

- **sig**: quale segnale gestire
- **handler** specifica il comportamento. E' *puntatore a funzione*.

**Nota:** consigliato usare **sigaction**

## System Call **kill**

Manda un segnale *ad un processo* oppure a un *gruppo di processi*

```
C  
#include <sys/types.h>  
#include <signal.h>  
  
int kill(pid_t pid, int sig);
```

Argomenti:

- **sig**: segnale da mandare
- **pid**:
  - se **> 0**: spedito al processo identificato da **pid**
  - se **0**: spedito a tutti i processi appartenenti allo *stesso gruppo* del processo che invoca **kill()**
    - se **<0**: spedito al gruppo di processi identificati da **-pid** (ovvero  $|pid|$ )
    - se **-1**: a *tutti i processi*: da *non usare!*

*Privilegi*: un processo può mandare un segnale solo a processi dello *stesso utente*. Tranne *root*, che può mandare a tutti.

## Esercizio con **kill**

**Esercizio**: si crei un programma che genera un processo figlio. Il padre manda al figlio un segnale **SIGUSR1** ogni secondo. Il figlio stampa l'avvenuta ricezione.

C

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void handler(int signum){
    printf("Ricevuto\n");
}

int main (){
    pid_t pid;
    struct sigaction action;

    pid = fork();
    if (pid!=0){ /* Father */
        while(1){
            sleep(1);
            kill (pid, SIGUSR1);
            /* pause(); Would be equivalent */
        }
    } else{ /* Child */
        action.sa_handler = handler;
        sigemptyset (&action.sa_mask);
        action.sa_flags = 0;
        sigaction (SIGUSR1, &action, NULL);
        while (1);
    }
    return 0;
}
```

## System Call **raise**

C

```
#include <signal.h>
int raise (int sig);
```

Permette a un processo di *inviare un segnale a se stesso*.

Di fatto:

C

```
raise (sig);
```

equivale a:

C

```
kill (getpid(), sig);
```

## System Call **pause**

C

```
#include <unistd.h>
int pause (void);
```

*Sospende il processo fino all'arrivo di un segnale*

Serve a implementare l'*attesa passiva* di un segnale

Ritorna dopo che il segnale è stato catturato ed il gestore è stato eseguito, *restituisce sempre* (-1)

## System Call **alarm**

C

```
#include <unistd.h>
unsigned int alarm (unsigned int seconds);
```

Implementa un *timeout*

Il SO manda un segnale **SIGALRM** al processo dopo **seconds** secondi

- Se non vi era già un timeout settato, restituisce subito 0
- Altrimenti, restituisce *i secondi che mancano* allo scadere dell'ultimo allarme settato.  
*Cancella il vecchio timeout* e inserisce il nuovo
- Se **seconds** è 0, si *disattiva* il timeout

### Osservazioni:

Il timeout è gestito dal kernel.

Il tempo effettivo può essere leggermente maggiore a causa del tempo di reazione del kernel

- Quindi **non** è affidabile fino a millisecondi *ms* o *μs*!
  - Ovvero non vanno implementati in sistemi dove i millisecondi contano
- 

## Esercizio con **alarm** e **pause**

**Esempio:** funzione **sleep** implementata con **alarm** e **pause**

```
C  
static void myAlarm (int signo) {  
    return;  
}  
void mySleep (unsigned int nsecs) {  
    signal(SIGALRM, myAlarm)  
    alarm (nsecs);  
    pause ();  
}
```

## Considerazioni sulle System Call per Segnali

Un handler è un flusso di **esecuzione concorrente**

- Può iniziare in *qualsiasi istante*
- Mentre il flusso principale sta compiendo qualsiasi azione
- Operazione molto delicata!

### Importante:

Per questo motivo l'handler **non deve modificare variabili** globali che sono usate anche dal flusso principale

- Potrebbe portare in stato inconsistente
- Potrebbe portare al **problema dell'incremento perso**

## Problema dell'incremento perso

- Immaginiamo che il programma venga interrotto tra la riga 2 e la riga 3 del seguente codice:

```
1 local = global; /* Supponiamo global = 1 */  
2 local++; /* Local = 2 */  
--- Interruzione ---  
3 global = local;
```

- L'handler esegue l'operazione:

```
global++; /* Global = 2 */
```

- La variabile **global** viene *incrementata*
- Il programma riprende dall'istruzione 3.

```
...  
--- Interruzione ---  
3 global = local; /* Global = 2 */
```

- L'incremento dell'handler si è perso. **global** = 2 anzichè 3. Questo è un grande problema!

Problema che vedremo approfonditamente per i programmi *multi-thread*

## Conseguenze

Da questo definiamo le nozioni di *funzione rientrante* e *funzione non rientrante*.

### Definizioni:

1. **Funzione rientrante:** può essere usata con sicurezza in più flussi
2. **Funzione non rientrante:** *NON* può essere usata con sicurezza in più flussi (può portare a stati inconsistenti)

In generale, negli handler, bisogna:

- Chiamare *solo funzioni rientranti*
- Evitare di *manipolare variabili globali* che sono usate dal flusso principale.

**Classificazione.** Classifichiamo alcune funzioni già note

- Molte funzioni di libreria C *NON* sono rientranti
  - **fprintf**, **fscanf**: gestione del *buffer* problematica.

- Non vanno chiamate dentro un handler!
- Alcune funzioni *sono rientranti* e possono essere interrotte senza problemi: **sleep**, etc. Vedi **man**
- Le *System Call* sono rientranti:
  - Se un programma riceve un segnale mentre è eseguita una sua system call (e.g., **read**):
  - Le System Call *bloccanti* terminano e non riprendono (e.g., **read**, **write**, **wait**)
  - Le System Call *non bloccanti* riprendono (e.g., **fork**, **getpid**)

## Segnali nella shell

Vediamo alcuni comandi per *operare* con i segnali.

### 1. Kill

SHELL

```
kill pid
```

Invia un segnale al processo **PID**.

Di default manda **SIGTERM**.

Possibile specificare con opzioni **-KILL** **-INT**

SHELL

```
pkill nome
```

```
killall nome
```

Stesso comportamento, ma manda il segnale a tutti i processi del programma **nome**

## Esercizio Misto

**Esercizio:** si scriva un programma in C che memorizza quanti **SIGTERM** ha ricevuto. Alla pressione di **CTRL+C** stampa tale numero e termina. Si nomini il programma **sample**.

Si scriva anche uno script bash che manda 10 segnali **SIGTERM** al processo.

**Programma Bash:**

```
for i in $( seq 5) ; do
    pkill sample
done
```

### Programma C:

```
C

#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int c;

void handler(int signum){
    if (signum==SIGTERM)
        c+=1;
    else if (signum==SIGINT){
        printf("Ricevuti %d SIGTERM\n", c);
        exit (0);
    }
}

int main (){
    struct sigaction action;
    c=0;
    action.sa_handler = handler;
    sigemptyset (&action.sa_mask);
    action.sa_flags = 0;
    sigaction (SIGTERM, &action, NULL);
    sigaction (SIGINT, &action, NULL);

    while (1);
}
```

Notare l'uso dell' handler! E' in grado di ricevere e maneggiare segnali diversi! (Volendo potevo pure creare due handler diversi)

Quando si preme **CTRL+C**, viene mandato un **SIGINT** al programma, che stampa **c** e termina

**Extra:** si faccia uno script bash che automatizza tutta la sequenza: avvio del programma in C, consegna segnali e chiusura.

SHELL

```
./sample &
PID=$!
for i in $( seq 5) ; do
    kill $PID
done
kill -INT $PID
```

## Comandi per Mandare Segnali

Abbiamo dei *comandi* per mandare *segnali a processi*

1. Se **CTRL+C**, viene inviato **SIGINT**
  - Programma termina se non c'è un handler
2. Se **CTRL+Z** viene inviato **SIGTSTP**
  - Di default l'applicazione viene sospesa
  - E messa in background dalla shell
  - A questo punto possiamo fare due cose:
    - **fg** fa riprendere l'esecuzione in foreground
    - **bg** far riprendere l'esecuzione in background

**Molto utile** se ho lanciato un comando lungo e voglio usare la shell mentre esegue

SHELL

```
$ ./longjob
^Z
[1]+ Stopped      ./longjob
$ bg
[1]+ ./longjob &
$ terminale libero
```

3. Quando eseguo un programma in background (**./job &**) e chiudo il terminale, viene mandato il segnale di Hang Up **SIGHUP**
  - Di default il programma viene terminato
  - Si può modificare comportamento

Oppure uso il comando **nohup** che esegue un comando immune a `SIGHUP`

```
nohup ./job
```

Utile se lancio job su terminale remoto e devo andare a casa!

**Alternativa più pulita:** comando **screen** che genera *terminale virtuale*

## Handler in Bash

Handler di segnali in script bash: si usa il comando **trap**. Ascoltate la trap?

SHELL

```
trap command SIGNAL
```

Esegue il comando o la funzione **command** se lo script riceve il segnale **SIGNAL**  
Esiste lo pseudo-segnale aggiuntivo **EXIT**, chiamato quando lo script termina

SHELL

```
tempfile=/tmp/tmpdata  
trap "rm -f $tempfile" EXIT
```

## Esercizio Bash

**Esercizio:** si crei un programma bash che conta quanti SIGUSR2 riceve, e li stampa quando viene premuto **CTRL+C** e lo si nomini **sample.sh**

```
#!/bin/bash

count=0
function husr(){
    let count++
}
function hint(){
    echo "Ricevuti $count SIGUSR2"
    exit 0
}

trap husr SIGUSR2
trap hint SIGINT

while true; do
    sleep 1
done
```

Si inviano i segnali col comando: **bash pkill -USR2 sample.sh**

**Nota:** dichiarazione di funzione in Bash

## Domande

Quale System Call si usa per generare un segnale?

- **signal**
- **kill**
- **write**
- **send**

**Risposta:** kill

Una funzione handler riceve degli argomenti?

- **No**
- **Riceve una stringa**
- **Riceve un intero**

**Risposta:** Riceve un intero

Quale è il comportamento di default di un processo quando riceve un segnale?

- **Il segnale viene ignorato**
- **Il processo termina**
- **Dipende dal segnale**

**Risposta:** Dipende dal segnale

Un signal handler può modificare le variabili globali del processo?

- **Sì**
- **No**

**Risposta:** Tecnicamente sì, ma fortemente sconsigliato (sì)

Quale segnale viene inviato dal SO quando si preme **CTRL+C** sulla tastiera?

- **SIGKILL**
- **SIGINT**
- **SIGHUP**
- **SIGSTP**

## u5-s4-inter-process-communication

### Sistemi Operativi

#### Unità 5: I processi

# Inter-Process Communication

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

---

## Argomenti

1. Obiettivi
  2. Le *pipe*
  3. Le *FIFO*
  4. Cenni di memoria virtuale
  5. Memoria condivisa con **shmget**
  6. Memoria condivisa con **mmap**
  7. Problematiche
- 

## Obiettivi

**Definizione.** (*Processi indipendenti e cooperanti*)

In un sistema dotato di SO, diversi processi sono in esecuzione contemporaneamente.

Essi possono essere classificati in:

- **Processi indipendenti:** non sono influenzati né influenzano altri processi
- **Processi cooperanti:** *interagiscono con altri processi*. Devono usare meccanismi opportuni per farlo  
Ci concentreremo sui *processi cooperanti*, in particolare i loro meccanismi.

Tutti i SO mettono a disposizione strumenti per la *Inter-Process Communication*

Sono tipicamente basati su:

- *Scambio di messaggi* (esempio: segnali)
- *Scambio di dati* (vedremo le *pipe* o le *FIFO*)
- *Memoria condivisa* (vedremo *mmap* o *shmget*)

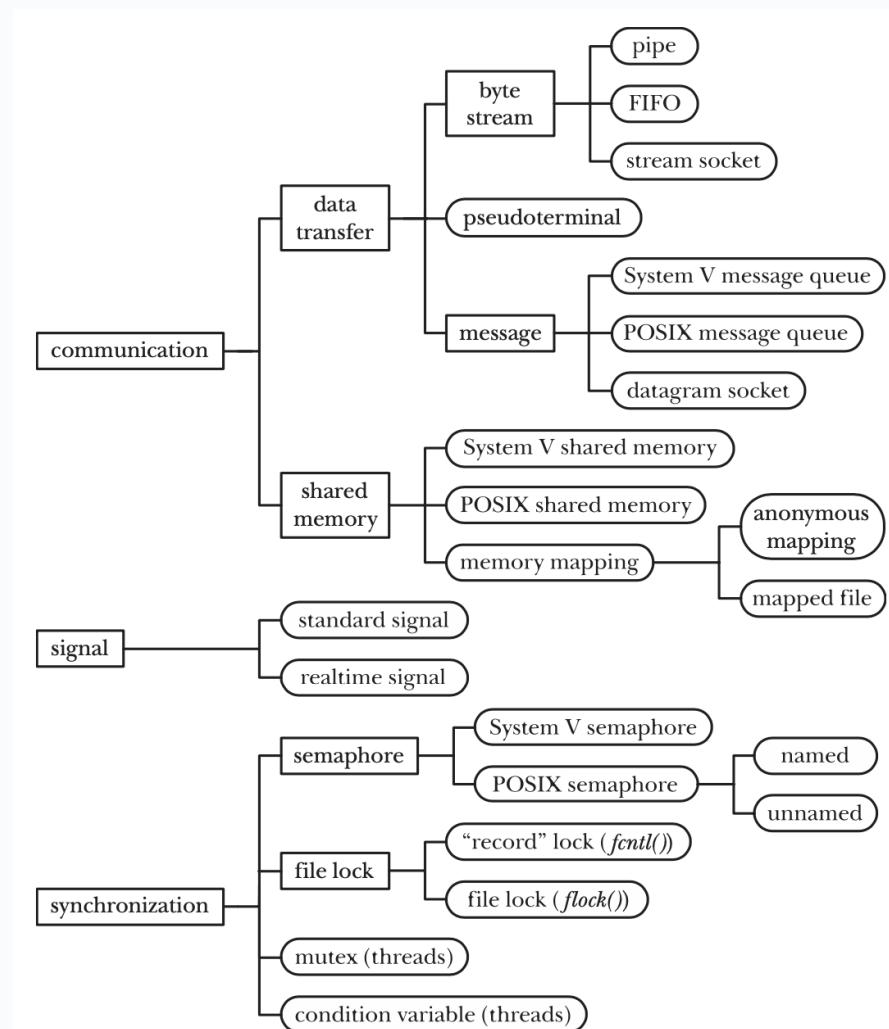
- Altri costrutti di sincronizzazione  
Ogni SO utilizza meccanismi diversi.

## LINUX.

In Linux, ci sono *tanti meccanismi*.

- Storicamente stratificati
  - Ereditati da System V
  - Parte di standard POSIX
- Ci concentreremo su:
- Pipe, FIFO (*Byte stream*)
  - Memory Mapping (*Shared memory*)
  - Sincronizzazione (semafori, eccetera...)

Ogni meccanismo ha una sua caratteristica diversa: alcuni presuppongono *legami di parentela*, altri senza, eccetera...



## Le pipe

# Concetto di Pipe

## Idea. (Pipe)

Le **pipe** sono la più vecchia e la più usata forma di IPC introdotta in Unix

- Permettono di scambiare **dati** (sotto forma di bytestream) tra processi
- Modello **produttore-consumatore**; **produttore** → **consumatore**
- Si usano con le stesse System Call dei file: **read**, **write**
- Risiedono in memoria; molto veloce!
- **Non sono persistenti**: quando i processi terminano, tutto ciò che rimane viene distrutto

## Limitazioni:

- Sono half-duplex (comunicazione *in un solo senso*)
- Utilizzabili solo tra processi *con un "antenato" in comune*
  - Si dice che le pipe sono **anonime**.

## Come superare queste limitazioni?

- Le **FIFO** (o **named pipe**) possono essere utilizzati tra più programmi
  - Si identificano tramite un nome

## Primo Esempio di Pipe

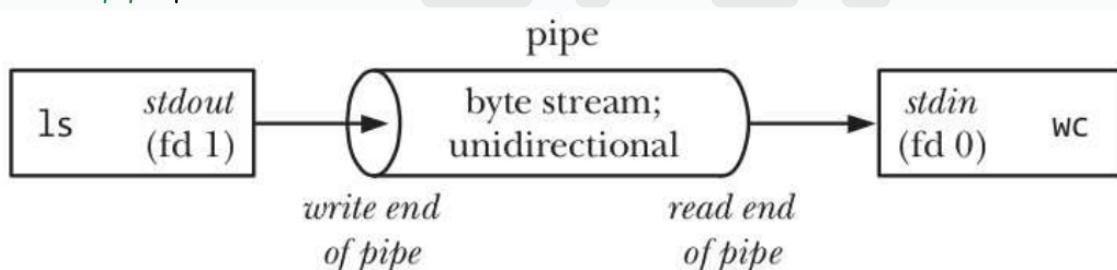
Le **pipe** sono comunemente usate nella shell, per redirezionare gli **stdout** e **stdin**, come visto con **Bash**.

## Esempio:

```
ls | wc -l
```

Per fare questa operazione, la shell:

- Usa due **fork** e **exec** per creare i processi **ls** e **wc**
- Crea una **pipe** per connettere lo **stdout** di **ls** con lo **stdin** di **wc**



- Studieremo questo **a basso livello**

# Definizione di Pipe

## Definizione:

Le pipe sono un *byte stream*

- Vi si scrivono/leggono byte
- Non solo caratteri stampabili  
Sono *unidirezionali*:
- Hanno un ingresso e una uscita  
Hanno *capacità limitata*:
- I dati accodati (scritti ma non ancora letti) *non possono eccedere una soglia*
- Soglia configurabile: 65 KB di default
  - Si può variare con **fcntl(fd, F\_SETPIPE\_SZ, size)**

# Sintassi in C per Pipe

## Creazione:

```
#include <unistd.h>
int pipe (int filedes [2]);
```

Ritorna due descrittori di file attraverso l'argomento fd (*passato per riferimento*; ricordiamoci che un vettore è un puntatore!)

- **fd[0]** è aperto in lettura (*out*)
- **fd[1]** è aperto in scrittura (*in*)
- L'output di **fd[0]** è l'input di **fd[1]**

## I/O su pipe.

Si usano le funzioni **read** e **write**

- Il valore di ritorno è il numero di byte scritti/letti

## Lettura:

- La **read** è bloccante finché non è letto almeno un byte. Se la pipe è morta, ritorna 0.

## Scrittura:

- Se la *pipe* è piena, la **write** è *bloccante*

## Chiusura di una pipe:

- Se leggo un pipe e ottengo 0, significa che *sono stati chiusi tutti i file descriptor*. Così posso effettuare l'*operazione di gestione* per la chiusura di pipe.

## Confronto Pipe-File.

- Sui *file* assumo che il contenuto sia *immutable*, letto in sequenza fino all'*EOF*.
- Sui *pipe* la *EOF* non esiste! Teoricamente potrei scriverci quanto voglio.

**Nota:** se scrivo su una *pipe* che non ha un *reader* (*fd[0]* è stato chiuso), il processo riceve *il segnale SIGPIPE* (*broken pipe*)

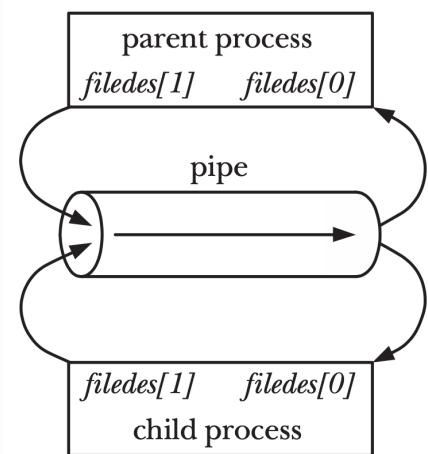
- Comporta la terminazione del processo, se non c'è un *Signal Handler* opportuno

## Prassi delle pipe

### Condivisione tra processi:

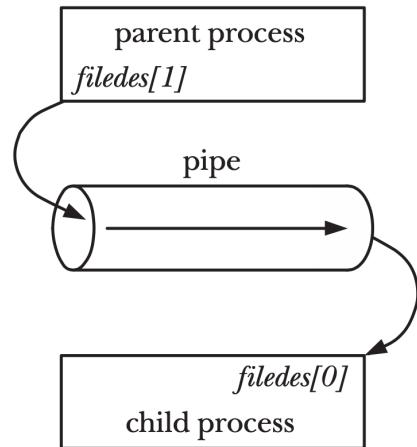
Per utilizzare una pipe tra più processi:

- Il processo padre *crea la pipe* e ottiene i due *fd*
- Esso fa una *fork*
- Entrambi i processi possono accedere alla *pipe* usando i due *fd*



### Nota.

- Solitamente un processo (e.g., padre) scrive, e un altro (e.g., figlio) legge
- Tecnicamente possibile che un processo legga e scriva
  - Crea però *problemi di sincronizzazione*
- Ogni processo chiude i *fd* che *non usa* (buona pratica)



## Esempio di Pipe

**Esempio:**

```

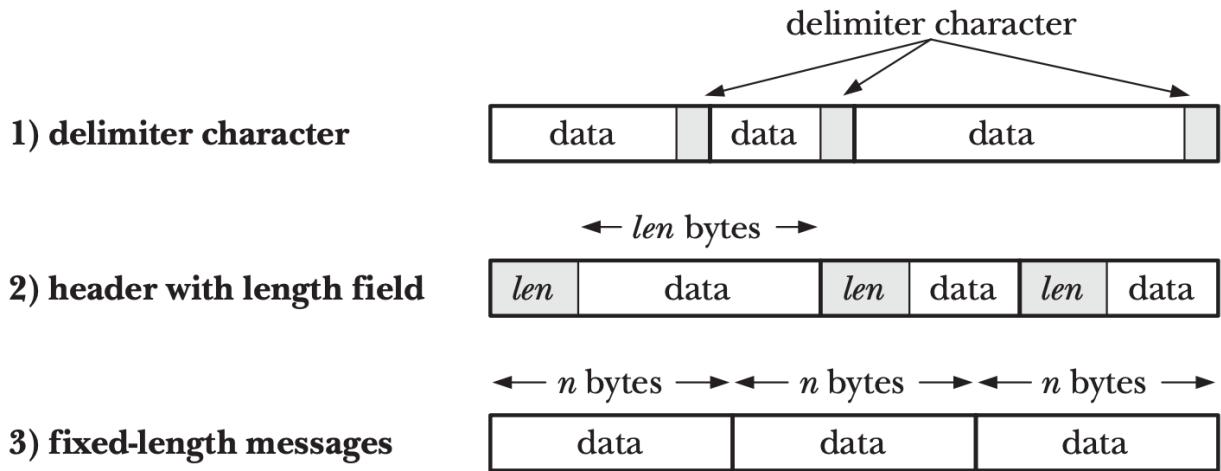
int pfd [2];
pipe ( pfd ); /* Crea la pipe */
switch ( fork () ) {
    case -1: exit(1);
    case 0:
        /* Child */
        close ( pfd [1]);
        /* Può ora leggere */
        break;
    default :
        /* Parent */
        close ( pfd [0]);
        /* Scrive nella pipe */
        break;
}

```

## Messaggi su pipe

### Messaggi su pipe

Ci sono diverse strategie per scambiare messaggi tramite *pipe*



1. Scelgo un byte per rappresentare la *fine* del messaggio (come **\0**)
2. Approccio complementare: prima scrivo il *la lunghezza del messaggio*, poi il messaggio effettivo
3. Ho  $n$  bytes fissati per messaggio. Il più semplice

## Esercizio sulle pipe

**Esercizio:** si crei un programma che genera un figlio. Il processo padre riceve una stringa da riga di comando e la passa al figlio tramite una *pipe*. Il figlio riceve la stringa e la stampa.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#define MAXLINE 1024

int main(int argc, char *argv[])
{
    int pfd[2], status;
    char line[MAXLINE];

    pipe(pfd);
    if (fork() > 0) { /* Padre */
        close(pfd[0]);
        write(pfd[1], argv[1], strlen(argv[1]));
        wait(&status);
    } else { /* Figlio */
        close(pfd[1]);
        read(pfd[0], line, MAXLINE);
        printf("Ricevuto: %s\n", line);
    }
    exit(0);
}
```

**Esercizio:** si crei un programma con due processi. Il processo padre riceve il nome di un file da riga di comando e ne passa il contenuto al figlio tramite una *pipe*. Il figlio riceve il contenuto e lo stampa.

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#define MAXSIZE 1000

int main(int argc, char * argv[]){
    int pfd[2], status;

    pipe(pfd);
    if(fork()>0){
        close(pfd[0]);
        FILE * fp;
        char line[MAXSIZE];

        fp=fopen(argv[1],"r");
        while(fgets(line,sizeof(line),fp)!=NULL) /* Itera riga per riga*/
            write(pfd[1],line,sizeof(line)); /* Scrive nella pipe */
        close(pfd[1]);
        wait(&status);
        exit(0);
    }
    else {
        char buffer [MAXSIZE];
        close(pfd[1]);
        while (read(pfd[0],buffer,sizeof(buffer)) > 0) /* Quando read ritorna 0, la
pipe è morta*/
            exit(0);
    }
}

```

## Le FIFO

### Motivazioni per le FIFO

#### **Richiamo.**

Facciamo un breve richiamo sulle *pipe*, per poter parlare delle *FIFO* e darne una motivazione.

## Pipe "normali"

- Possono essere utilizzate solo da processi che hanno un *"antenato" in comune* (sono anonime)
- Motivo: unico modo per ereditare descrittori di file

## Named pipe o FIFO

- Permettono a processi *non collegati* di comunicare
- Utilizzano il file system per *"dare un nome"* alla pipe
- Le *FIFO* sono un tipo di file
  - La macro **S\_ISFIFO** dopo una **stat** restituirà **true**
- La procedura per creare un fifo è simile alla procedura *per creare file*
  - In certi versi le FIFO possono essere trattati come file, similmente alle pipe

## Sintassi C per FIFO

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *path, mode_t mode);
```

Crea un *FIFO* dal **pathname** specificato

Argomento **mode** specifica i permessi (come nella **open**).

- E.g.: **S\_IRWXU**, **S\_IRGRP**, etc.

Valore di ritorno: 0 se successo, -1 se errore

### Utilizzo:

Come file e *pipe*: tramite **read** e **write**

Ogni processo che ha i permessi per **pathname** può usarla

1. **Apertura:** dopo essere state create con **mkfifo**, le FIFO vanno aperte con una **open** o una **fopen**

### Flag.

File aperto senza flag **O\_NONBLOCK**:

- Se il file è aperto in *lettura*, la **open si blocca** fino a quando un altro processo non apre la FIFO in scrittura
- Se il file è aperto in *scrittura*, la **open si blocca** fino a quando un altro processo non apre la FIFO in lettura

File aperto con flag **O\_NONBLOCK**

- Se il file è aperto in lettura, la **open** ritorna immediatamente
- Se il file è aperto in scrittura, e nessun altro processo è stato aperto in lettura, la **open** ritorna un messaggio di errore

### **Input/Output:**

- Con **read** e **write**
- I dati nella **FIFO** sono *bufferizzati dal kernel*

### **Importante:**

- Una **FIFO** ha un pathname ma è *solo un espediente* per permettere a diversi processi di accedervi
- Quando un **FIFO** viene chiusa (o i processi terminano) il nome del file persiste nel file system, ma esso *non contiene alcun dato*
- Non ho dei **file** presenti sul disco! Sono dei **file virtuali**

## Sintassi Bash per FIFO

Si possono creare e usare le **FIFO** in Bash in maniera semplice:

SHELL

```
mkfifo myfifo
tr 'aeiou' 'AEIOU' < myfifo &
man 2 pipe > myfifo
```

## Esercizi sulle FIFO

**Esercizio:** si crei un programma che:

1. se riceve **read** come argomento, stampa ciò che viene scritto su una pipe e
2. se riceve **write** come argomento, legge iterativamente una riga da tastiera e la scrive su una pipe.

**Soluzione.**

```
#include <stdio.h>
#include <sys/stat.h>
#include <string.h>
#include <stdlib.h>
#define FIFO "my-fifo"
#define BUF_SIZE 512

int main(int argc, char * argv[]){
    FILE * f;
    char buffer[BUF_SIZE];

    if (argc != 2 || ( strcmp(argv [1], "read")==0 && strcmp(argv [1], "write")==0 ) ){
        printf("Usage: fifo read|write\n");
        return 1;
    }

    if (mkfifo(FIFO, S_IRWXU)<0)
        perror("Warning. FIFO not created");

    if (strcmp(argv [1], "read")==0){
        f = fopen(FIFO, "r");
        if (f==NULL){
            perror("Impossible to open the FIFO");
            return 1;
        }

        printf("Read mode:\n");
        while(fgets(buffer,BUF_SIZE,f)≠NULL)
            printf("%s", buffer);

    }else{
        f = fopen(FIFO, "w");
        if (f==NULL){
            perror("Impossible to open the FIFO");
            return 1;
        }

        printf("Write mode. Write lines of text:\n");
        while(fgets(buffer, BUF_SIZE, stdin)≠NULL){
            fputs(buffer, f);
            fflush(f);
        }
    }
}
```

```
    }
    return 0;
}
```

**Esercizio:** si crei un programma che legge da una FIFO e stampa il contenuto in maiuscolo.

**Soluzione.**

```
#include <stdio.h>
#include <ctype.h>
#include <sys/stat.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char * argv[]){
    int i, n, l;
    FILE * f;
    char buffer[512];

    i = mkfifo("myfifo", S_IRWXU);
    if (i<0){
        printf("Impossibile creare la FIFO\n"); /* Potrebbe già esistere */
    }

    f = fopen("myfifo", "r");
    if (f==NULL){
        printf("Impossibile aprire la FIFO\n");
        exit(1);
    }

    while(fgets(buffer,sizeof(buffer),f)!=NULL){
        l = strlen(buffer);
        for (i=0; i<l; i++)
            putc(toupper(buffer[i]), stdout);
    }
}
```

La si testi con: **echo "ciao mondo" > myfifo**

## Cenni di memoria virtuale

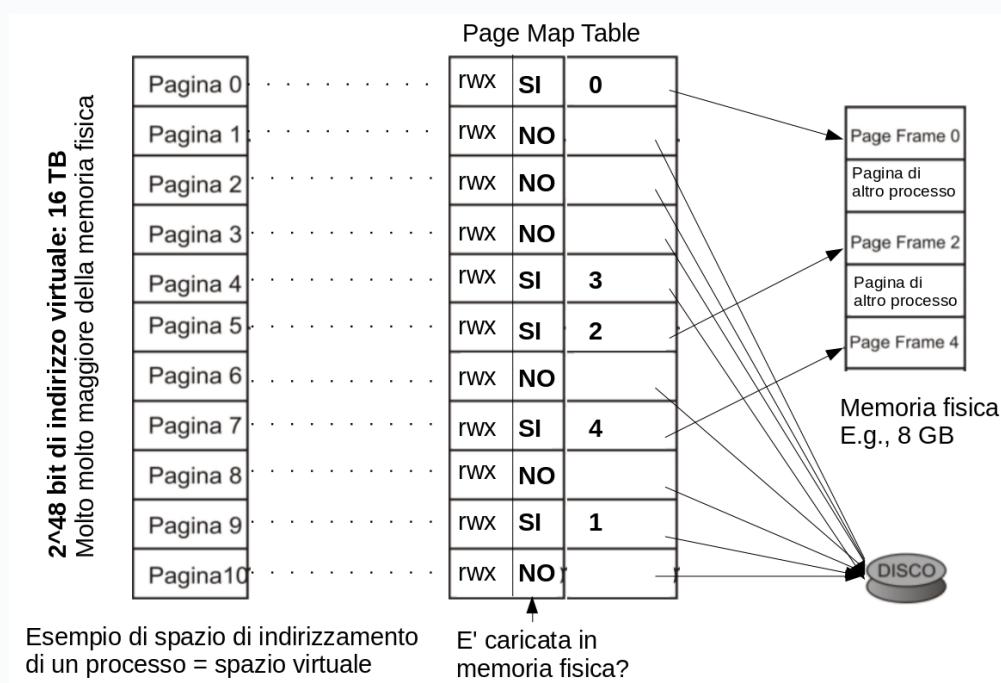
Diamo dei *cenni di memoria virtuale*, prima di poter parlare di *condivisione della memoria*.

I processi emettono *indirizzi virtuali*

- Permettono di indirizzare *più memoria di quella disponibile* (fisicamente)
  - Su architettura AMD64: 48bit; 256TB di memoria virtuale. La memoria fisica è di solito minore (e.g., 16 GB)
- Evitano che un processo acceda a memoria di altri

La *memoria virtuale* è divisa in *pagine* e una *tabella* mappa le pagine da *spazio di indirizzi virtuali a indirizzi fisici*

- Azione compiuta dalla *Memory Management Unit* in Hardware
  - Il sistema operativo interviene a *collocare pagine in memoria*

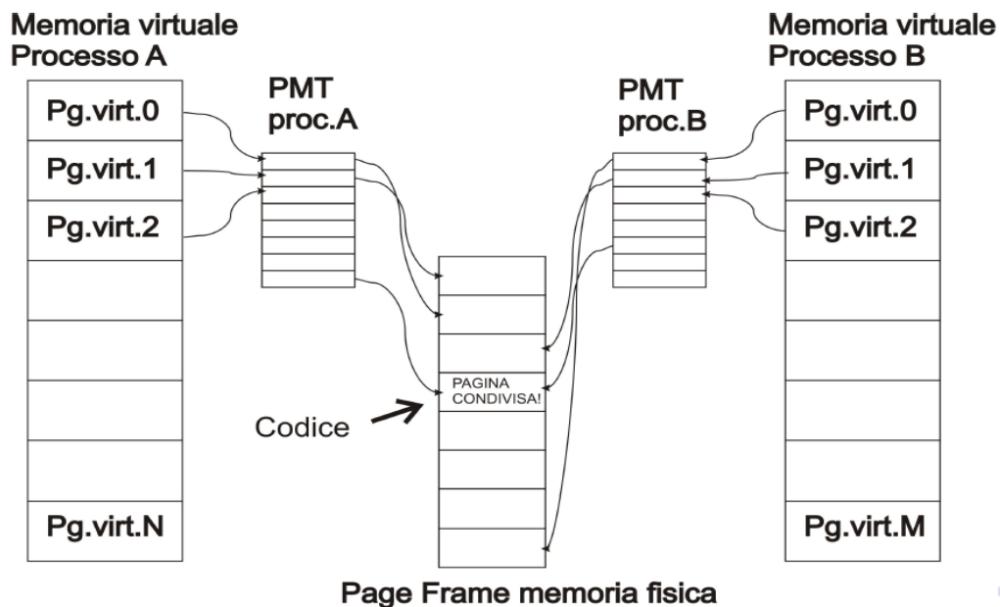


Ogni processo ha uno *spazio di indirizzi virtuali* dedicato

- C'è una tabella delle pagine per processo
- *Isolamento della memoria tra processi*
  - Essenziale per *sicurezza*
  - Non permette la condivisione di memoria
  - Dà al processo l'illusione di essere *l'unico processo*

Per condividere la memoria, è necessario *condividere una o più pagine*

- Il SO mette a disposizione delle System Call per questo scopo



Esistono due set di System Call per avere *memoria condivisa* tra processi in Linux:

- **shmget shmat shmdt ftok**
- **mmap munmap shm\_open shm\_unlink**: questa è l'opzione più moderna, di cui vedremo

L'approccio con **mmap** è più *moderno e flessibile*

In Windows si usa la System Call **CreateFileMapping**

## Memoria condivisa con **shmget**

*Nota. Opzionale*

```
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
```

Crea un segmento di memoria condivisa.

Argomenti:

- **key**: identificativo definito dall'utente. Usare **IPC\_PRIVATE** se anonimo (usato solo con **fork**)
- **size**: dimensione della memoria condivisa
- **shmflg**: flag e permessi. **IPC\_CREAT** crea se non esistente.
  - Uso tipico **IPC\_CREAT | 0666**

Valore di ritorno:

- Un identificativo della zona create
- $-1$  se insuccesso

```
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Mappa il segmento di memoria virtuale nello spazio degli indirizzi del processo.

Argomenti:

- **shmid**: identificativo ritornato da **shmget**
- **shmaddr**: se non nullo, la memoria viene mappata a **shmaddr** (arrodonato per difetto al **page size**)
- **shmflg**: flag. **SHM\_RDONLY** mappa in Read Only

Valore di ritorno:

- L'indirizzo virtuale del segmento mappato
- $-1$  se insuccesso

```
#include <sys/shm.h>
int shmdt(const void *shmaddr);
```

Rimuove in mapping del segmento all'indirizzo virtuale **shmaddr**

Valore di ritorno:

- 0 in caso di successo
- $-1$  se insuccesso

```
#include <sys/ipc.h>
key_t ftok(const char *pathname, int proj_id);
```

Crea una **key** a partire da un path, garantendo che:

- Due path daranno sempre chiavi diverse
- Stesso path e stesso **proj\_id** darà sempre chiavi uguali
- Evita che programmi diversi che per sfortuna hanno scelto stessa chiave usino la stessa memoria

Argomenti:

- **pathname**: il path
- **proj\_id**: usato per creare la **key**. Non deve essere nullo.

Valore di ritorno:

- *key* in caso di successo
- *-1* se insuccesso

**Utilizzo:** Memoria condivisa con figlio creato tramite **fork**

```
C

int shmid = shmget(IPC_PRIVATE, 1*sizeof(int), IPC_CREAT | 0666);
void * shm = shmat(shmid, NULL, 0);
if (fork()){ /* Padre */

    ...
} else { /* Figlio */

    ...
}
shmdt(shm);
```

**Utilizzo:** Memoria condivisa tra due processi indipendenti

- Proceso creatore

```
C

key_t key = ftok(path, proj );
int shmid = shmget(key, size, IPC_CREAT | 0666);
void * data = shmat(shmid, NULL, 0);

...
shmdt(data);
```

- Proceso utilizzatore

```
C

key_t key = ftok(path, proj );
int shmid = shmget(key, size, 0666);
void * data = shmat(shmid, NULL, 0);

...
shmdt(data);
```

**Esercizio:** Si creino due programmi che hanno una memoria condivisa con **shmget**. Il primo programma permette di scrivere una stringa nella memoria, mentre il secondo permette di leggerla.

### Programma 1:

```
C

#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
int main(){
    key_t key = ftok("test",123);
    if (key < 0){printf("Errore. Il file per la chiave esiste?"); exit(1);}
    printf("Key: %d\n", key);

    int shmid = shmget(key,1024,0666|IPC_CREAT);
    char *str = (char*) shmat(shmid,(void*)0,0);
    while(1){
        printf("Input Data : ");
        scanf(" %s", str);
    }
    shmdt(str);
    return 0;
}
```

### Programma 2:

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
int main(){
    key_t key = ftok("test",123);
    if (key < 0){printf("Errore. Il file per la chiave esiste?"); exit(1);}
    printf("Key: %d\n", key);

    int shmid = shmget(key,1024,0666|IPC_CREAT);
    char *str = (char*) shmat(shmid,(void*)0,0);
    while(1){
        printf("Premi enter per leggere");
        getchar();
        printf("Data: %s\n\n", str);
    }

    shmdt(str);
    return 0;
}
```

## Memoria condivisa con mmap

### Creazione di Memoria Condivisa

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

Crea una zona di memoria condivisa, mappandolo su un file. Comando **complesso** e **configurabile!**

Argomenti:

- **addr**: se non nullo, la memoria viene mappata a **addr** (arrotondato per difetto al **page size**)
  - Tipicamente lo si mette a **NULL**, l'indirizzo viene deciso dal **sistema operativo**.

- **length**: dimensione
- **prot**: può essere: **PROT\_READ**, **PROT\_WRITE**, **PROT\_EXEC**, **PROT\_NONE**
  - Cioè la pagina può essere letta, scritta, eseguita, non può essere acceduta
  - Normalmente **prot = PROT\_READ|PROT\_WRITE**
- **flags** determina come i cambiamenti sono visibili o meno ad altri processi
  - **MAP\_ANONYMOUS**, **MAP\_SHARED**, **MAP\_PRIVATE**
  - Normalmente si impone **flags=MAP\_ANONYMOUS | MAP\_SHARED** o **MAP\_SHARED**
- **fd** e **offset** sono usati per mappare la memoria su un **file** (diventerà l'identificativo globale)
  - Normalmente si impone **fd** e **0**, o **-1** e **0** in caso di anonimità.

Valore di ritorno:

- L'indirizzo virtuale del segmento mappato (andrà convertito col meccanismo di *casting*)
- **-1** se insuccesso

## Rimozione di Memoria Condivisa

```
C
int munmap(void *addr, size_t length);
```

Rimuove la mappatura e rende disponibile la memoria all'indirizzo **addr**

## Casi d'Uso della **mmap**

**Utilizzo:** ci sono tre modi per usare la **mmap**

1. Zona di memoria *anonima*: utilizzata con **fork**

```
C
mmap(NULL, size, PROT_READ|PROT_WRITE,
      MAP_SHARED|MAP_ANONYMOUS, -1, 0);
```

2. Zona di memoria *mappata su file*:

```
C
fd = open("/home/martino/file.txt", O_RDWR|O_CREATE);
mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
```

3. Zona di memoria mappata *su file temporaneo*:

```
fd = shm_open("temporaneo.txt", "rw");
mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
```

Adesso li vediamo in dettaglio

## Zona di Memoria Anonima

### 1. Zona di memoria anonima:

- La zona di memoria non ha un nome
- *Solo* un figlio nato con una **fork** può accedervi
- *Semplicissimo da usare*
- Praticamente è come una *pipe* bidirezionale

```
void* shmem = mmap(NULL, size, PROT_READ | PROT_WRITE,
MAP_SHARED | MAP_ANONYMOUS, -1, 0);

if (fork()) { /* Padre */
    ...
} else { /* Figlio */
    ...
}
munmap(shmem, size);
```

**Nota:** Dopo la **fork**, la memoria dei processi è indipendente. Solo con **mmap** è possibile creare una zona di memoria condivisa.

## Zona di Memoria Condivisa

### 2. Zona di memoria mappata su file:

*Idea.*

- Si usa *un file* come contenitore
- Il contenuto della zona condivisa verrà salvato su file
- *Più programmi* possono accedere alla memoria condivisa
  - Il path va identificatore (diventa l'espeditore)
- Efficace ma lento a causa del *disco*
- E' necessario che il file sia grande a sufficienza per contenere la regione mappata
  - Per assicurarci di questo useremo le funzioni **truncate**, **ftruncate**

```
#include <unistd.h>
int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);
```

Assicura che il file aperto **fd** o il file **path** si lungo almeno **length**.

- Se necessario il file è troncato
- Se necessario esteso e riempito con caratteri **'\0'** (o **0x00** in hex).

### Modello tipico.

```
fd = open(path, O_RDWR|O_CREAT); /* Non è una fopen()*/
ftruncate(fd, size);
void* shmem = mmap(NULL, size, PROT_READ|PROT_WRITE,
                   MAP_SHARED, fd, 0);
...
munmap(shmem, size);
```

## 3. Zona di memoria mappata su file temporaneo

- A volte è necessario avere un **file temporaneo identificabile** che risiede in memoria
  - Per utilizzo da parte di più programmi
- Una zona di memoria mappata **su file** è **inutilmente lenta**
  - **Se** non è necessario che i dati della zona di memoria sopravvivano

Si possono utilizzare le funzioni **shm\_open** e **shm\_unlink**, **fatte ad-hoc** per il **file system virtuale**.

- Creano e rimuovono un file temporaneo
- Che si trova nella cartella **/dev/shm/** che ha montato un FS temporaneo (**tmpfs**)
- I dati sono **in memoria**

```
#include <sys/mman.h>
int shm_open(const char *name, int oflag, mode_t mode);
int shm_unlink(const char *name);
```

Creano e rimuovono zone di memoria temporanee.

- Semantica analoga a **open** e **unlink**
- Operano su zone di memoria/file temporanee
- **name** è un nome di file, non un path completo
  - Tutte le zone di memoria sono file sotto **/dev/shm/**

### Flusso tipico

```
C  
int fd = shm_open(nome, O_RDWR, 0);  
ftruncate(fd, size);  
void * mem = mmap(NULL, size, PROT_READ | PROT_WRITE,  
                  MAP_SHARED, fd, 0);  
...  
munmap(shmem, size);  
  
/* Non obbligatorio */  
shm_unlink(nome)
```

## Esercizi con Memoria Condivisa

**Esercizio:** Si creino due programmi che hanno una memoria condivisa con **mmap** e **shm\_open**.

Il primo programma permette di scrivere una stringa nella memoria, mentre il secondo permette di leggerla.

### Programma 1:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>
#include <fcntl.h> /* Per O_RDWR */
int main(){
    int fd;
    char * mem;

    fd = shm_open("mymem", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    ftruncate(fd, 512);
    mem = (char *) mmap(NULL, 512, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);
    while(1){
        printf("Scrivere: ");
        scanf("%s", mem);
    }
    munmap(mem, 512); /* Inutile causa loop infinito*/
}
```

## Programma 2:

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>
#include <fcntl.h> /* Per O_RDWR */
int main(){
    int fd;
    char * mem;

    fd = shm_open("mymem", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    ftruncate(fd, 512);
    mem = (char *) mmap(NULL, 512, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);
    while(1){
        printf("Premi enter per leggere");
        getchar();
        printf("Data: %s\n\n", mem);
    }
    munmap(mem, 512);
}

```

**Nota:** Compilare con **gcc prog.c -lrt -o prog**

Include **librt, libposix4 - POSIX.1b Realtime Extensions library**

---

## Problematiche della Memoria Condivisa

L'utilizzo della memoria condivisa è complesso

- Preferire le *pipe* o *FIFO* quando possibile

La memoria condivisa ha problemi di *sincronizzazione* e *race conditions*

- Processi concorrenti possono leggere dati in stato inconsistente
- Mentre un altro processo li stava modificando
- Come abbiamo già visto coi segnali

**Esempio:**

Time	$\tau_x$	$\tau_y$
$t_1$	READ (A)	—
$t_2$	$A = A - 50$	—
$t_3$	—	READ (A)
$t_4$	—	$A = A + 100$
$t_5$	—	—
$t_6$	WRITE (A)	—
$t_7$	—	WRITE (A)

**LOST UPDATE PROBLEM**

La variabile  $A$  viene incrementata di 100.

- Il decremento di 50 viene perso

Per ovviare a questi problemi esistono le *tecniche di sincronizzazione*

- Permettono di evitare che un processo sia interrotto mentre effettua un'operazione critica
  - Permettono a un processo di attendere il verificarsi di una condizione
- 

## Domande

Le *pipe* sono

- Monodirezionali • Bidirezionali • Dipende dai parametri di creazione

**Risposta.** Monodirezionali

Le *pipe* sono identificate da un nome?

- Si • No

**Risposta.** No

Le *FIFO* sono identificate da?

- Un ID numerico • Una stringa • Un Path

**Risposta.** Un Path

Si consideri il seguente codice C che opera sulla FIFO **myfifo**:

```
int n;
FILE * f = fopen("myfifo", "r");
fscanf (f, "%d", &n);
```

Che operazione compie?

- Crea la FIFO myfifo
- Scrive un intero in myfifo
- Legge un intero da myfifo

**Risposta.** Legge un intero da myfifo

Una zona di memoria condivisa creata tramite **shmget** e **shmat** può essere condivisa anche tra processi senza legami di parentela?

- Si
- No

**Risposta.** Non posso rispondere (parte saltata)

Le funzioni **shmat** e **mmap** hanno valore di ritorno:

- char \*
- int
- int\*
- void\*
- void

**Risposta.** void\* (puntatore a void)

Si consideri il seguente spezzone di codice:

```
int fd = open("/tmp/mymem", O_RDWR|O_CREAT);
ftruncate(fd, 64);
void* shmem = mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_SHARED,
fd, 0);
sprintf( (char*)shmem, "Ciao Mondo!");
```

Dopo che il programma è terminato, il contenuto della zona di memoria presiste?

- Si, in memoria
- Si, nel file /tmp/mymem
- No

**Risposta.** No