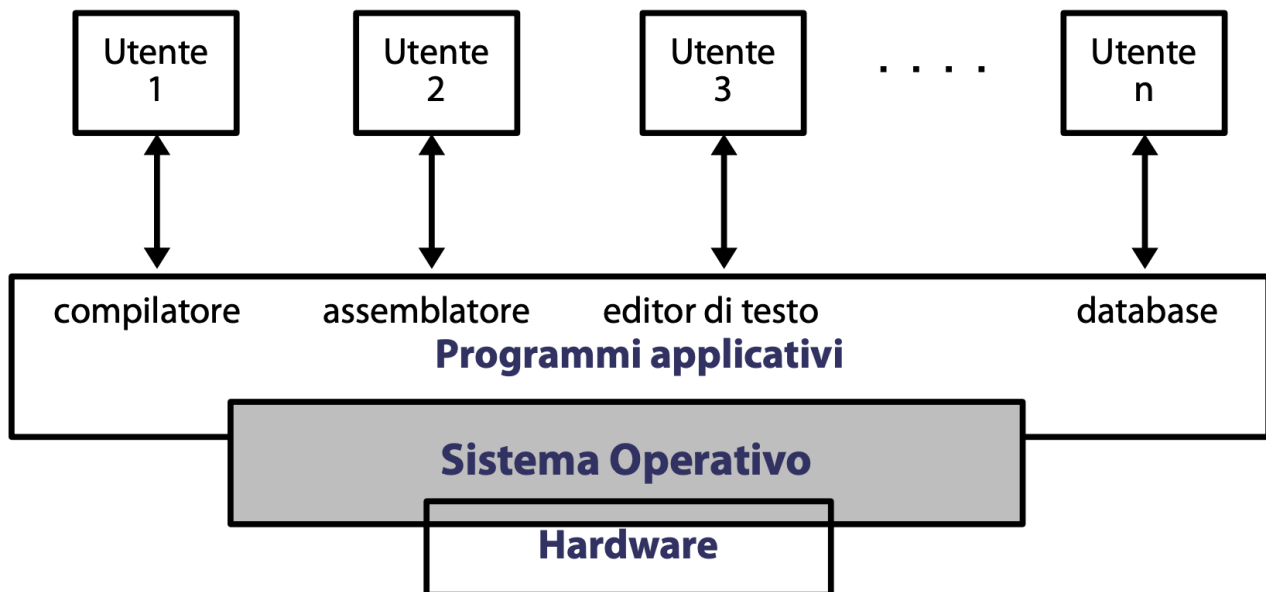


## T3 - Chiamate di sistema

### SO in un sistema di calcolo

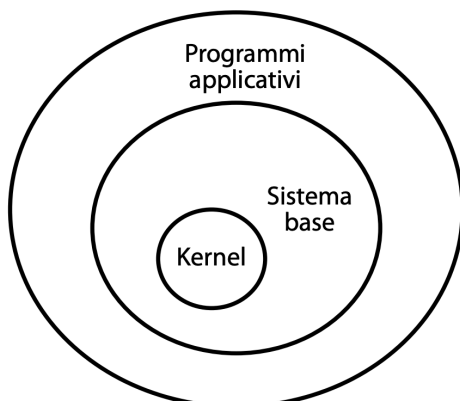


### Sistema Operativo

- *Kernel*: software mediatore fra applicazioni e hw.
- *Sistema di base*: permette ad un SO di avviarsi e di presentare una interfaccia testuale all'utente.

### Programmi applicativi

Tutto ciò che non è kernel o sistema di base.



## Kernel

- Driver dispositivi
- Gestore I/O
- Gestore dei processi

- Gestore del file system
- Gestore della memoria
- IPC

## Sistema base

- Librerie di sistema
- Caricatore dinamico
- Sistema di init
- Comandi di sistema
- Shell
- Terminale

## Programmi applicativi

- Compilatori
- Interpreti
- Ambiente grafico
- Suite di ufficio
- Browser
- Client e-mail

## Modello Client-Server

Una applicazione (di base e non) richiede un servizio al kernel. Il kernel elabora la risposta e la fornisce alla applicazione.

## User Mode e Kernel mode

### User Mode:

- Una applicazione esegue i suoi calcoli con privilegi ridotti.
- Non può alterare la memoria di altre applicazioni o del kernel.
- Non può eseguire istruzioni assembly legate all'I/O.
- Protezione contro usi maliziosi o sbadati.

### Kernel Mode:

- Quando una applicazione richiede un servizio del kernel, i suoi privilegi sono elevati al massimo rango.
- Esegue la porzione di codice del kernel corrispettiva al servizio richiesto, al termine del servizio, ritorna in user mode.

## User Space e Kernel Space

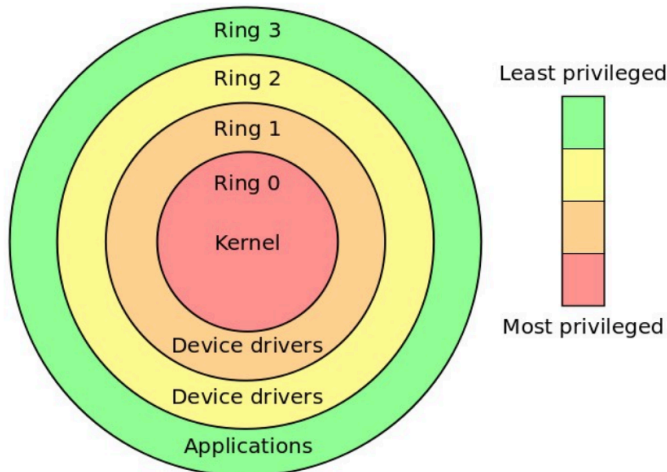
### User Space:

- Insieme di tutti gli indirizzi di memoria accessibili ad una applicazione.

#### Kernel Space:

- Insieme di tutti gli indirizzi di memoria accessibili al kernel.

## Livelli di privilegio nelle CPU Intel



**Ring 0:** kernel mode.

**Ring 3:** user mode.

## Chiamata di sistema

È l'unico meccanismo con cui una applicazione può richiedere un servizio al kernel.

- Salvataggio registri.
- Commuta user -> kernel.
- Esegue una funzione di servizio.
- Copia opzionalmente dati in memoria utente.
- Commuta kernel -> user.
- Ripristino registri ed esecuzione programma.

## Passaggio di parametri

Una chiamata di sistema accetta al più sei parametri.

Se ne servono di più, occorre usare un parametro come puntatore ad una struttura dati.

Il passaggio dei parametri avviene tramite registri.

Il registro **eax** contiene sempre un identificatore numerico della chiamata di sistema.

## Ingresso in Kernel Mode

- **Fino a Pentium:** Si è usata una interruzione software (*trap*), precisamente la 128 (*int \$0x80*).
- **Da Pentium 2:** Si usa l'istruzione assembly `sysenter`, ben più performante della eccezione. Ora si esegue la una funzione di servizio kernel `system_call()`. Per invocare la vera e propria funzione di servizio, che ha solitamente il prefisso `sys_`. Se ad esempio si invoca la chiamata di sistema `getpid()`, da qualche parte nel kernel esiste una funzione `sys_getpid()`.

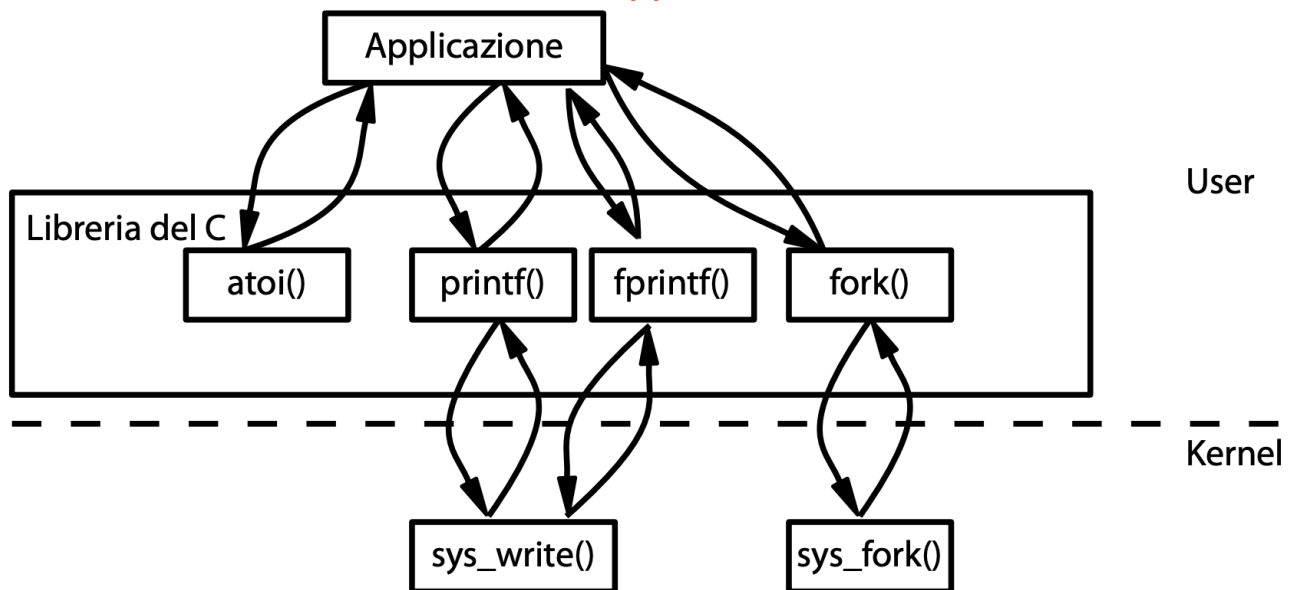
## Ritorno in User Space

Il valore di ritorno della funzione di lavoro è memorizzato nel registro **eax**, **rax** a 64bit.

Se si è usata l'istruzione `sysenter` per entrare in kernel mode, si usa ora l'istruzione `sysexit`.

## La libreria del C

Linux usa una libreria wrapper per facilitare l'accesso ai servizi del kernel: la libreria del C (**GNU C Library**).



## T4 - Struttura di un SO

### Stratificazione

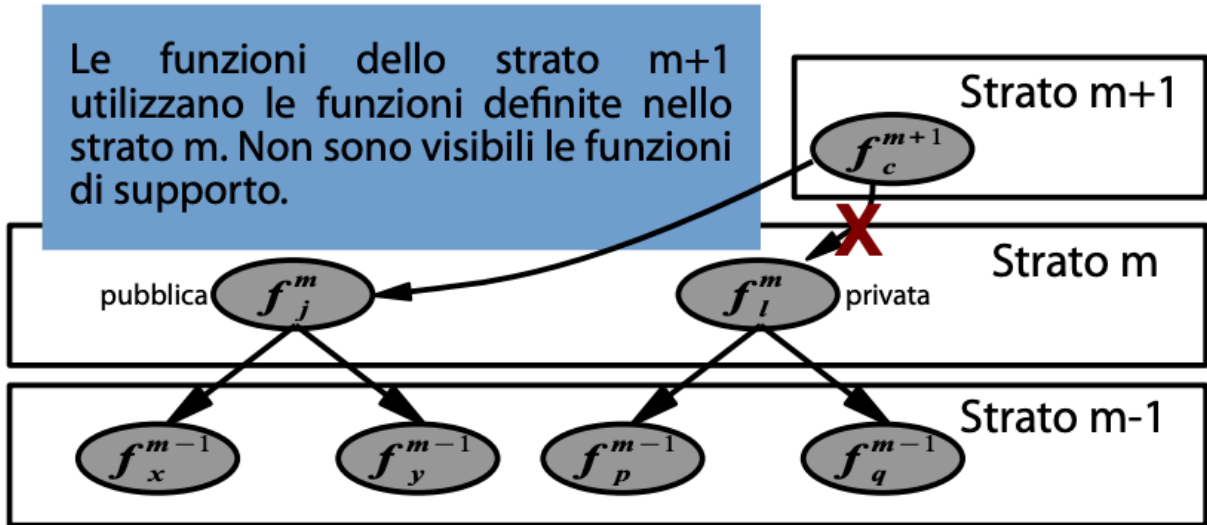
Un SO moderno è concepito per *strati* software successivi impilati uno sopra l'altro. L'insieme degli strati forma uno *stack software*.

Uno strato contiene l'implementazione delle funzione e delle strutture dati atte a fornire la funzionalità necessaria allo strato superiore.

Implementazione in due possibili modi:

- *Software*: libreria (statica o dinamica)

- **Hardware:** chip (ROM, EEPROM)



## Vantaggi

### Modularità

- **Nello sviluppo:** Il primo strato può essere progettato senza alcuna considerazione per il resto del sistema. Il secondo strato si appoggia sulle funzioni del primo.
- **Nel debugging:** In un sistema ad  $m$  strati, se i primi  $m - 1$  sono corretti ciò implica che l'eventuale errore stia nell' $m - m_n$  strato.

## Svantaggi

- Divisione degli strati, dove inizia e finisce uno strato?
- Riduzione dell'efficienza: ritardo nella fruizione del servizio. La funzione a strato  $m$  ci mette di meno ad essere "servita" rispetto ad una funzione nello strato  $m - m_n$ .

## Macro, Micro, Hybrid Kernel

### Macro Kernel (o monolitico)

I servizi vengono eseguiti in kernel mode. Le funzionalità essenziali del kernel sono contenute in una singola immagine eseguita al boot della macchina.

L'insieme delle applicazioni viene eseguito in user mode.

- Kernel Unix-Like (Linux, BSD, AIX, System V)

#### Vantaggi:

- Esecuzione dei servizi rapida (user  $\rightarrow$  kernel / kernel  $\rightarrow$  user)

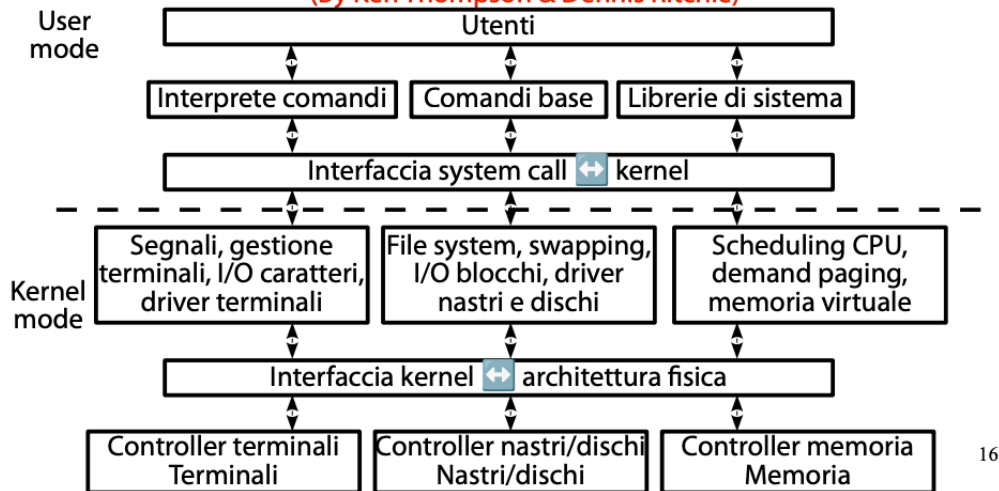
#### Svantaggi:

- Forte fragilità: un crash del kernel pianta la CPU.

- Dimensione kernel enorme, possibile riduzione di prestazioni.

## Esempio di Macro kernel: UNIX

(By Ken Thompson & Dennis Ritchie)



16

## Dimensione della code base di UNIX

Con l'aumentare dei servizi, il kernel dei sistemi UNIX è cresciuto notevolmente.

- Problemi funzionali
- Problemi di sicurezza
- Problemi prestazionali

Hanno portato all'uso di **moduli caricabili**, adozione dell'architettura basata su **Micro kernel**.

**Moduli caricabili:** file oggetto contenente funzionalità del kernel (file system, driver di dispositivi, algoritmi di schedulazione, ...). (Dis)attivabile a tempo di esecuzione (a mano oppure automaticamente).

- Linux, FreeBSD, Mac OS X

## Vantaggi

- Modularità: I vari componenti del sistema operativo, come i driver di dispositivi o le funzionalità di rete, sono separati in moduli che possono essere aggiunti o rimossi in fase di esecuzione.
- Flessibilità: I moduli possono essere caricati solo quando necessari.
- Migliore gestione delle risorse: kernel più leggero.
- Se un modulo crasha non compromette l'intero sistema.

## Svantaggi

- Sicurezza: può introdurre rischi di sicurezza, poiché caricare moduli non sicuri potrebbe compromettere il sistema operativo.
- Lieve perdita di prestazione dovuta alla attivazione e disattivazione dei moduli durante l'esecuzione.

## Micro Kernel

Il kernel esegue solo i servizi essenziali. Il resto è eseguito sotto forma di server applicativo.  
Il kernel diventa un *sistema di messaggistica* per i server, scheduler CPU e allocatore di memoria.

- Tru64 UNIX, Mach, Minix

## Vantaggi

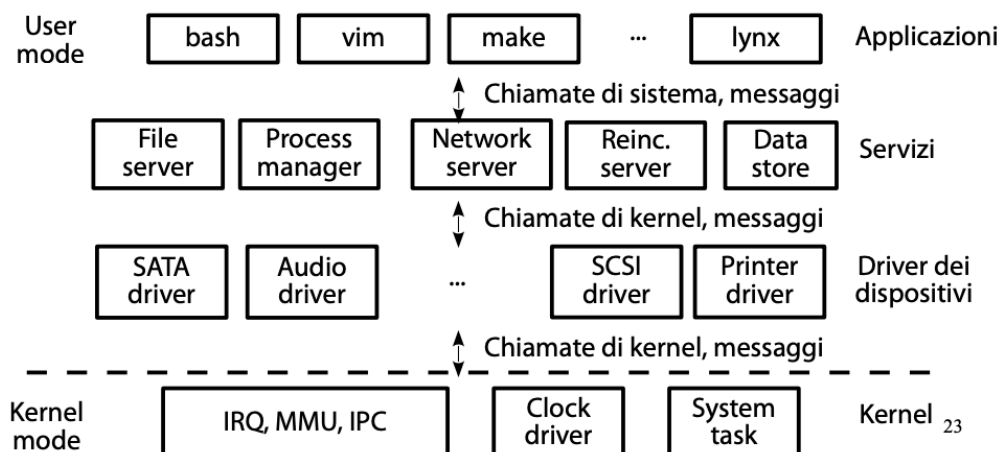
- Estendibili: nuovo servizio -> nuovo server, il kernel non è toccato.
- Kernel minimale, CPU cache friendly.
- SO robusto rispetto ai crash (muore il server, si ripara, se ne lancia una nuova istanza; nel frattempo, il SO continua ad eseguire).

## Svantaggi

- Meno performante.
- Presenza di commutazioni user -> kernel e kernel -> user per ogni chiamata di sistema e messaggio scambiato fra server.

### Esempio di Micro kernel: Minix

(By Andrew Tanenbaum)



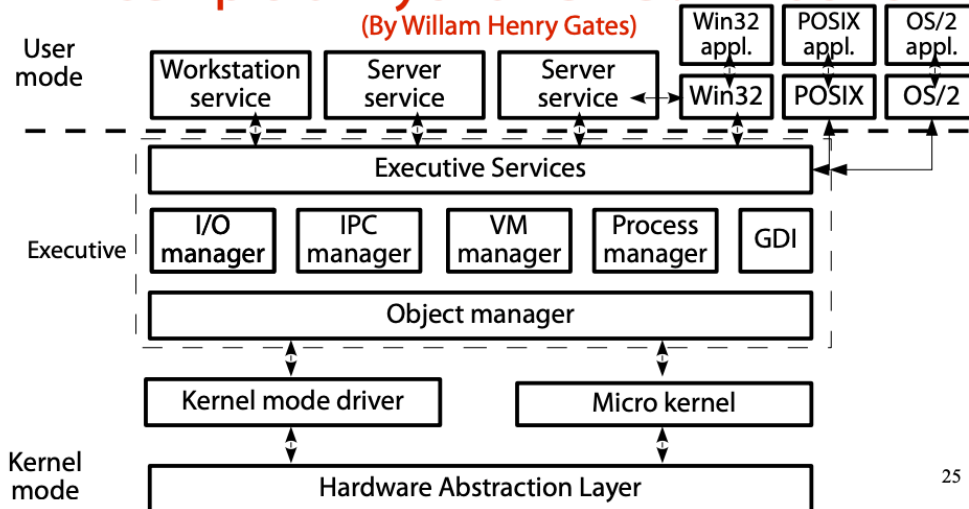
## Hybrid Kernel

Micro Kernel, ma i driver dei dispositivi eseguono in kernel mode.

Tentativo di combinare il meglio dei Macro e dei Micro kernel. Funzionalità, prestazioni e sicurezza si collocano come intermedie fra Macro e Micro kernel.

- Windows, Plan 9, OS X

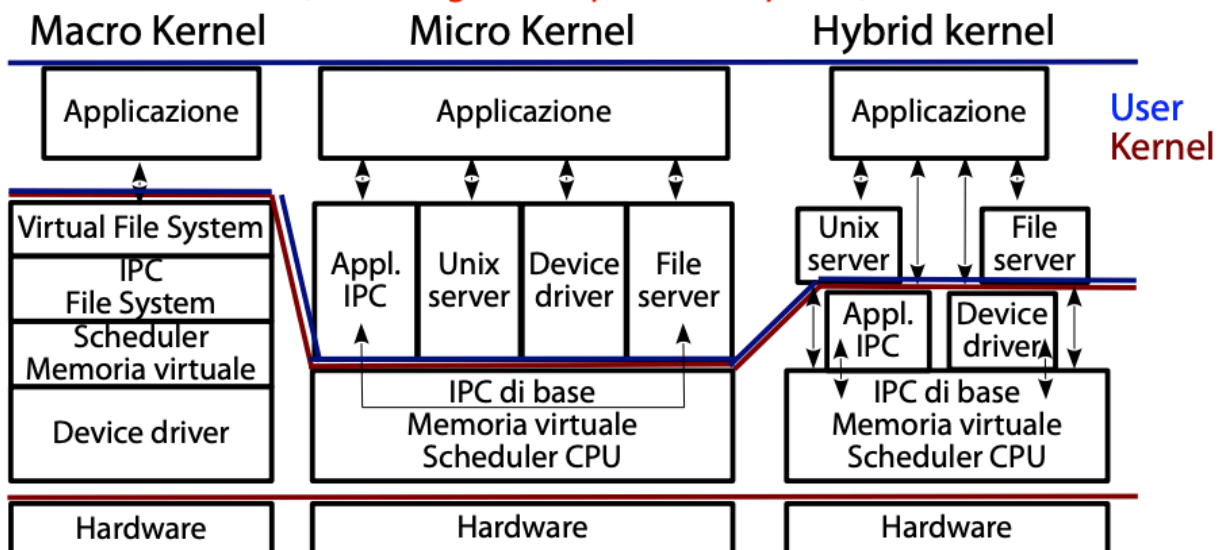
## Esempio di Hybrid kernel: Windows



25

## Macro vs. Micro vs. Hybrid

(Un'immagine vale più di 1000 parole)



26

## T5 - Processi

Il kernel di un SO deve saper gestire processi:

**Multiprogrammati:** Più applicazioni in esecuzione contemporaneamente.

**Multiutente:** Più utenti possono usare la macchina contemporaneamente.

**Time sharing:** "Piccole" porzioni di applicazione eseguite sequenzialmente.

**Astrazione** = *processo*

## Processo vs programma eseguibile

**Programma eseguibile:** file memorizzato su supporto secondario, contiene codice macchina da eseguire, alcune aree dati, una tabella di simboli utile per il debugging. Non va in esecuzione da solo.

**Processo:** la rappresentazione del kernel (in termini di strutture dati e funzioni di gestione) di un programma eseguibile *in esecuzione*.



# Rappresentazione

**Strutture dati:** Rappresentazione di uno “*stato interno*” (Bloccato? In esecuzione? Terminato?). Puntatori alle risorse in uso.

**Funzioni di gestione:** Creazione e terminazione. Esecuzione di un eseguibile. Comunicazione, sincronizzazione.

## PID

L'**identificatore di processo** è un numero *univoco* assegnato dal kernel a ciascun processo creato.

In quali stati si può trovare un processo?

**In esecuzione:** codice in esecuzione dal processore.

**Bloccato:** in attesa di un evento.

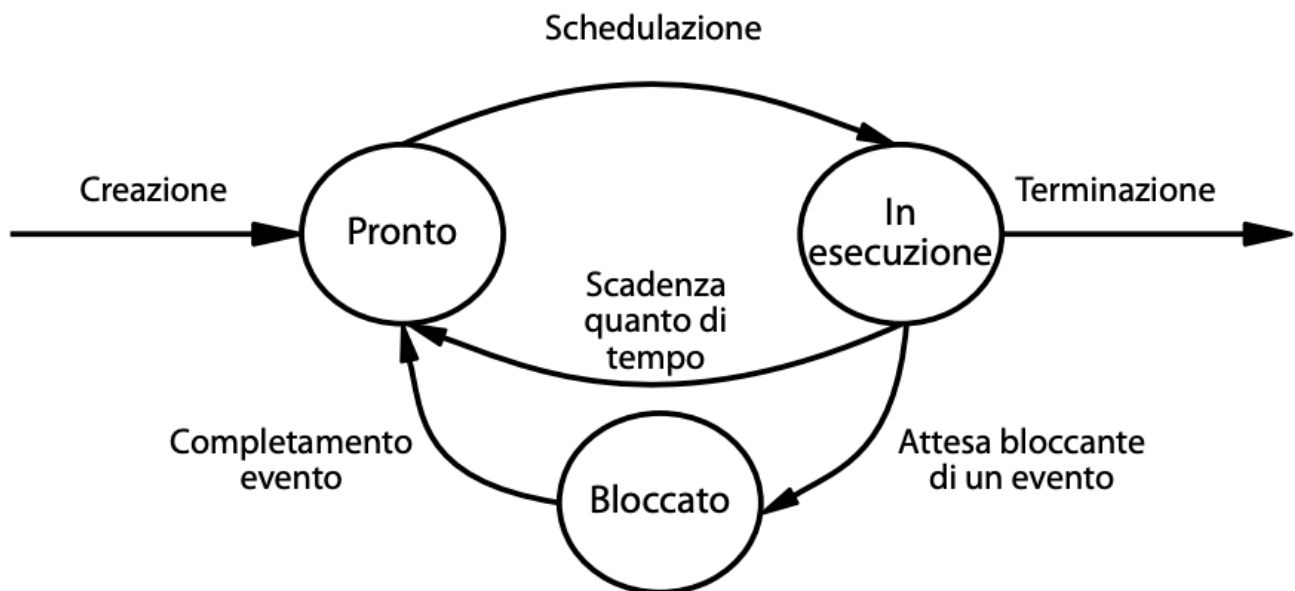
**Pronto:** si è verificato un evento ed il processo è pronto a proseguire/iniziare la sua esecuzione.

**Terminato:** terminato il codice da eseguire, `free()` della memoria e distruzione del processo.

## Automa

### Automa a stati finiti (DFA)

L'evoluzione fra stati si rappresenta mediante un automa a stati finiti deterministico (Deterministic Finite Automaton).

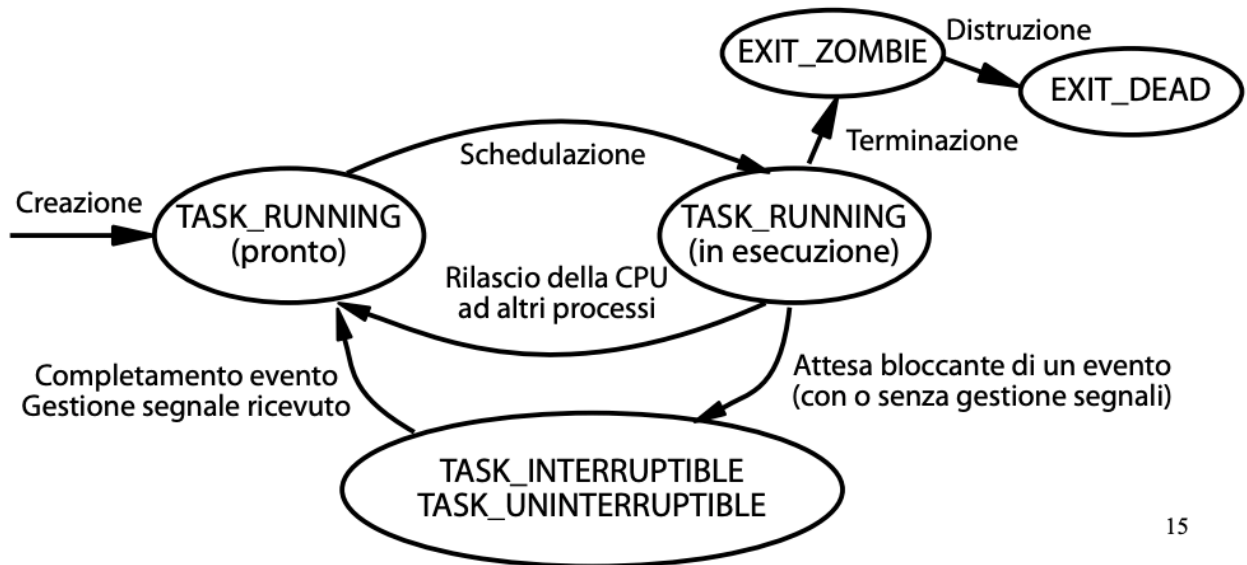


## Differenze in Linux

- **Esecuzione:** non vi è differenziazione fra processo “pronto” ed “in esecuzione”. Entrambi sono catturati da un unico stato, `TASK_RUNNING`.
- **Comunicazione:** un processo può comunicare con altri tramite la ricezione e l'invio di segnali.
  - `ctrl-C` segnale di interruzione. Se il processo sta ricevendo dati da una periferica ed esce così, può lasciare il kernel in uno stato inconsistente. L'attesa è gestita da due

stati: interrompibile mediante un segnale (`TASK_INTERRUPTIBLE`) e non interrompibile (`TASK_UNINTERRUPTIBLE`).

- **Sincronizzazione:** Un processo può crearne un altro ed aspettare (wait) che esso termini. Quando il processo creato termina, salva il suo stato di uscita in una struttura dati, in modo tale che il processo creatore possa leggerlo.
  - `EXIT_ZOMBIE`: il processo creato è morto ma le sue risorse non sono ancora deallocate.
  - `EXIT_DEAD`: il processo creato è morto e il creatore ha letto il suo stato di uscita. Le sue risorse possono essere deallocate.
- **Debugging:**
  - `TASK_STOPPED`: il processo è stato stoppato il processo è stato stoppato.
  - `TASK_TRACED`: il processo è tracciato da un debugger.



15

## Descrittore di processo

Al momento della creazione di un processo, il kernel del SO crea una struttura dati detta **Process Control Block (PCB)** che contiene:

- Informazioni di stato.
- Puntatori alle risorse prenotate.

In Linux, il descrittore dei processi è definito nella struttura dati `struct task_struct`.

## Funzionalità di base

### Creazione di processi

Un processo può creare altri processi. Il meccanismo di creazione è una vera e propria **clonazione (forking)**.

Processo **clonante** -> processo **padre**.

Processo **clonato** -> processo **figlio**.

Il processo figlio è una copia identica del processo padre. Entrambi i processi ripartono dall'istruzione successiva alla clonazione.

Nei sistemi UNIX si usa la chiamata di sistema `fork()` per creare una copia esatta di un processo. In Linux, è implementata dalla funzione di servizio `do_fork()`.

## ✎ Come si distingue padre e figlio?

`fork()` ritorna un valore mutevole.

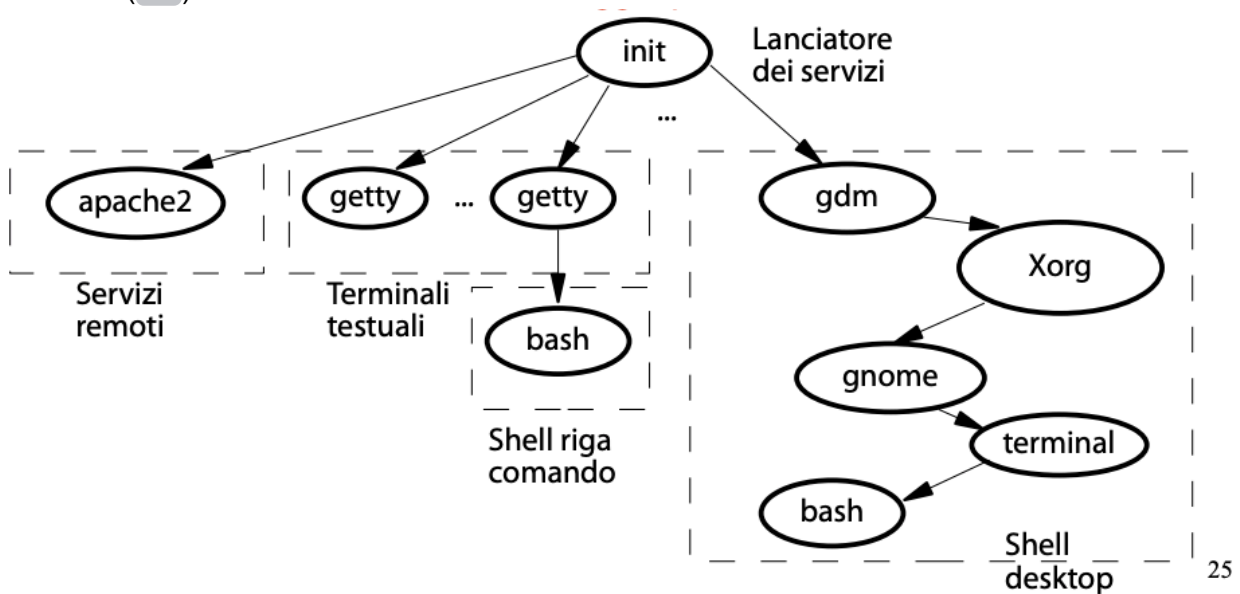
Nel processo padre: il PID del processo figlio.

Nel processo figlio: 0.

Errore: -1.

Nei SO UNIX l'organizzazione dei processi è ad *albero*. Un processo iniziale (*init*) è creato “a mano” dal kernel. Esso ha sempre PID = 1. Tale processo fa partire i servizi forniti dal computer tramite fork and exec. Fra i servizi vi è `getty` (gestore login su console).

Al termine del login, `getty` fa partire una shell (`bash`). La shell esegue un comando impartito dall'utente (`ls`).



## Esecuzione di programmi eseguibili

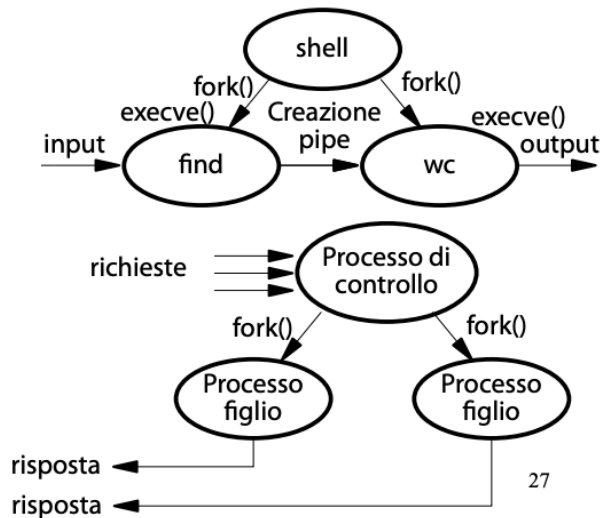
`fork()` non carica altri programmi. Il caricamento è gestito dalla chiamata di sistema `execve()`. Essa “sostituisce” in maniera efficiente le aree di codice e dati di un processo.

## Usi buoni del forking

Comandi composti in pipeline.

```
find . -name \*.c | wc -l
```

Server multiprocesso (Web, FTP, SSH, ...).



## Usi cattivi del forking

**Fork bomb:** tipo di processo che non fa altro che riprodurre processi figli, i quali a loro volta si riproducono. Meccanismo rozzo ma efficace di negazione del servizio (**Denial of Service, DoS**). Ogni processo consuma risorse e contribuisce al rallentamento della macchina.

Bash:

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

C:

```
while(1)
    fork();
```

La chiamata di sistema `exit()` termina un processo in maniera pulita.

Mette il processo in stato `EXIT_ZOMBIE`.

Attende la lettura del codice di uscita da parte del padre.

Mette il processo in stato `EXIT_DEAD`.

Rilascia le risorse.

Distrugge il PCB.

Schedula l'esecuzione di un altro processo.

Se un processo genitore termina, tutti i suoi figli diventano *orfani*.

**Reparenting:** processo orfano diventa figlio del processo `init`.

## Gruppi di processi e sessioni

Ogni comando dato all'interprete della shell crea un **gruppo** di processi, individuato da un *identificatore di gruppo* (*Process Group Identifier* **PGID**) pari al PID del primo processo nella pipeline (detto *process group leader*).

In questo modo è possibile inviare un segnale (e anche uccidere) a tutti i processi coinvolti in un comando.

```
ls -al | grep .bash | less -Mr
```

Una **sessione** è un insieme di gruppi di processi che condividono un terminale.

Una sessione è individuata da un *ID di sessione* (*session ID*, **SID**). Il SID è il PID del processo *session leader* che crea la sessione tramite `setsid()`.

## Gruppo in foreground e background

Un solo gruppo di processi può leggere dal terminale. Tale gruppo prende il nome di **foreground process group**. Gli altri gruppi di processi sono lanciati come **background process group**.

Lancio in *Background*:

```
ls -lR / > out.txt 2>err.txt &
```

Lancio in *Foreground*:

```
ls -lR / > out.txt 2>err.txt
```

**Esempio sessione:**

```
$ echo $$  
400  
$ find / 2> /dev/null | wc -l &  
[1] 659  
$ sort | uniq -c
```

Questo è il PID della **bash** (session leader).

Questa è una pipeline lanciata in background. Sono eseguiti due processi, legati in pipe. Il comando non può leggere input da terminale.

PID dell'ultimo processo della pipeline (**wc**). Il process group leader ha il PID precedente (658).

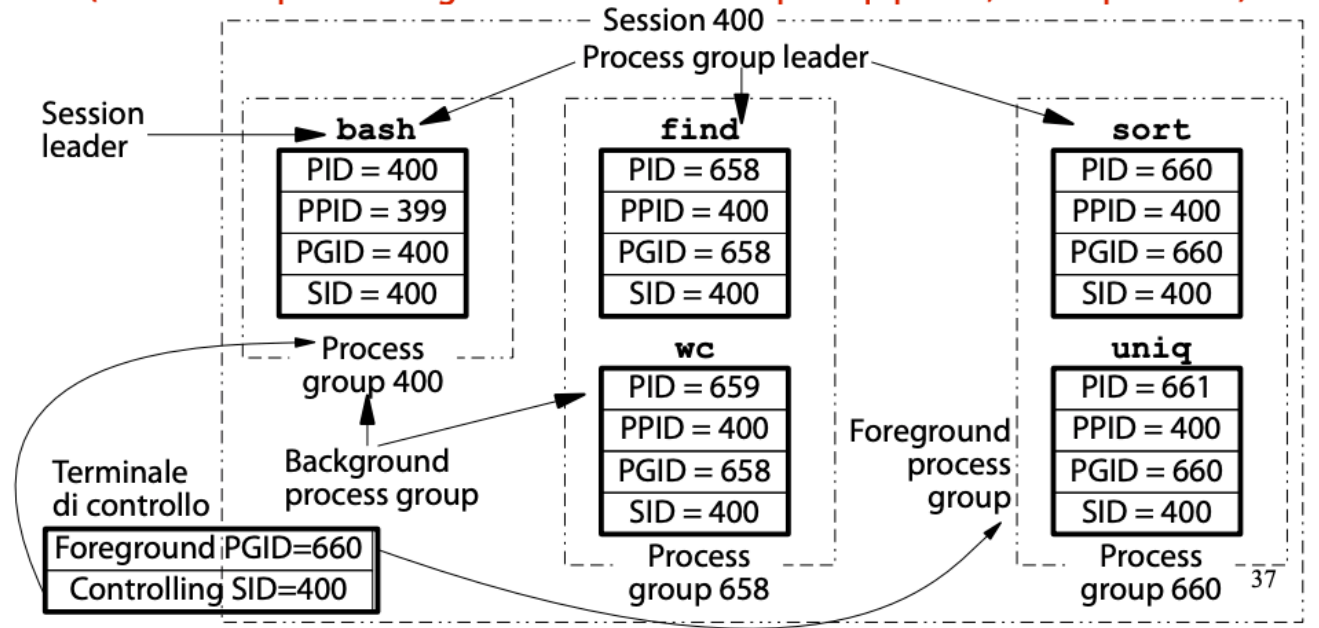
Questa è una pipeline lanciata in foreground. Sono eseguiti due processi, legati in pipe. Il comando può leggere input da terminale.

36

## Rappresentazione PID, PGID, SID

# Rappresentazione di PID, PGID, SID

(Sessione → processi legati al terminale, Gruppo → pipeline, PID → processo)



## T6 - Scheduling e dispatching

### Caratterizzazione dei processi

Durante la sua esecuzione, un processo si alterna in due fasi:

- **CPU Burst**: elaborazione user o kernel.
- **Wait**: attesa di I/O o evento.

Un processo si dice **vincolato (bound)** ad una **risorsa (resource)** o ad un **evento (event)** se la sua prestazione è correlata alla disponibilità della risorsa o al verificarsi dell'evento.

### Alcuni esempi di programmi

**CPU**: `factor` che fattorizza un numero intero è vincolato alla CPU.

**Disco**: `dd` che trasferisce dati a blocchi da/verso periferiche è vincolato al disco.

**Rete**: `wget` che scarica documenti dal web è vincolato dal collegamento alla rete.

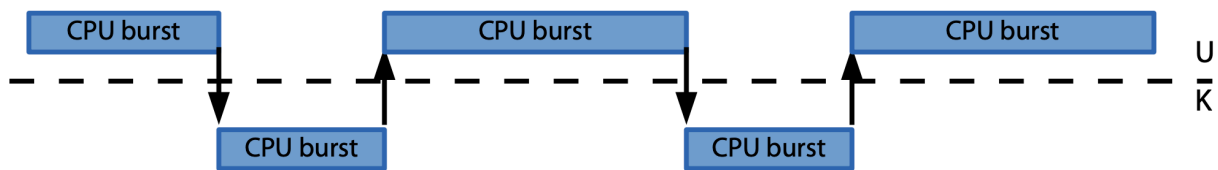
### Vincoli di interesse: CPU-bound e I/O-bound

Il comportamento di un processo si posiziona tra:

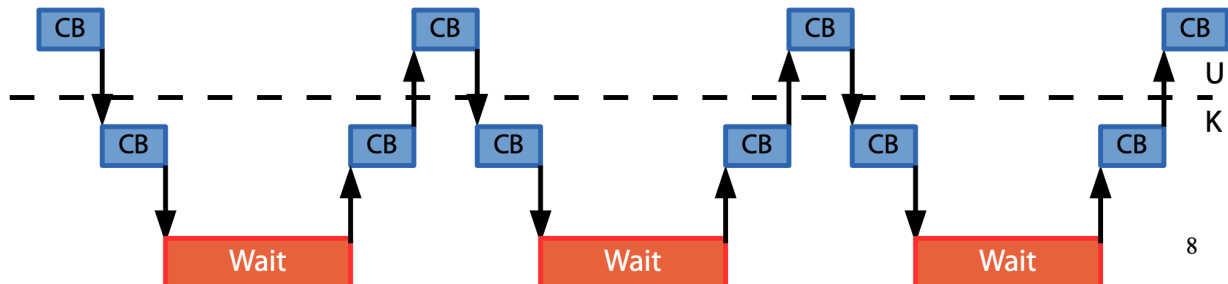
- **Processo CPU-bound**:
  - Tende a produrre poche sequenze di CPU burst lunghe.
  - Tende a fare poche richieste di I/O.
- **Processo I/O-bound**:
  - Tende a produrre tante sequenze di CPU burst brevi.

- Tende a fare molte richieste di I/O.

### Processo CPU-bound



### Processo I/O-bound



8

## Quali processi sono più importanti?

**Sistema desktop:** caratterizzato da *elevata interattività* con l'utente. Tale interattività si estrinseca in richieste di I/O a periferiche (terminale, disco, rete).

Si dovrebbero favorire i processi I/O-bound rispetto a quelli CPU-bound.

**Sistema di calcolo batch:** Duale del sistema desktop: interattività pressoché inesistente, necessità di completare quanti più processi di calcolo possibile. Si preferiscono i processi CPU-bound.

Il **grado di multiprogrammazione** è la somma di:

- $n$  processi in esecuzione.
- $n$  di processi pronti per l'esecuzione.  
 $n$  dipende da:
  - frequenza con cui un utente fa partire processi.
  - frequenza con cui i processi escono.

## Obiettivi del gestore dei processi

Reggere il grado di multiprogrammazione imposto dall'utente senza degradare le prestazioni proprie e dei processi.

Favorire i processi "giusti" per il tipo di SO considerato:

- Desktop → I/O-bound.
- Server → CPU-bound.

Come si raggiungono questi obiettivi?

**Schedulatore dei processi:** sceglie il prossimo processo da eseguire in maniera consona al tipo di SO in esecuzione.

**Dispatcher:** sostituisce in maniera efficiente il PCB del processo attuale con il PCB del processo scelto dallo schedulatore.

## Schedulatore

## Scenario

Un processo in kernel mode ha appena programmato il DMA controller per eseguire una operazione di I/O (lettura). La lettura è bloccante ed il processo non può più proseguire l'esecuzione fino all'ottenimento del dato.

Per non lasciare il processore inattivo, deve essere scelto e ripristinato un nuovo processo. Qui entra in gioco lo schedulatore, invocato dalla `sys_read()` (o chi per lei).

Lo schedulatore dei processi sceglie un processo ritenuto idoneo per l'esecuzione. Possibili criteri:

- È importante (priorità alta).
- Non esegue da tanto tempo.
- Scelto a caso.
- Scelto circolarmente.

Lo schedulatore invoca il dispatcher per sostituire il PCB del processo in esecuzione con il PCB del nuovo processo. A questo punto, il processo B è pronto per continuare l'esecuzione.

### Quando schedulare?

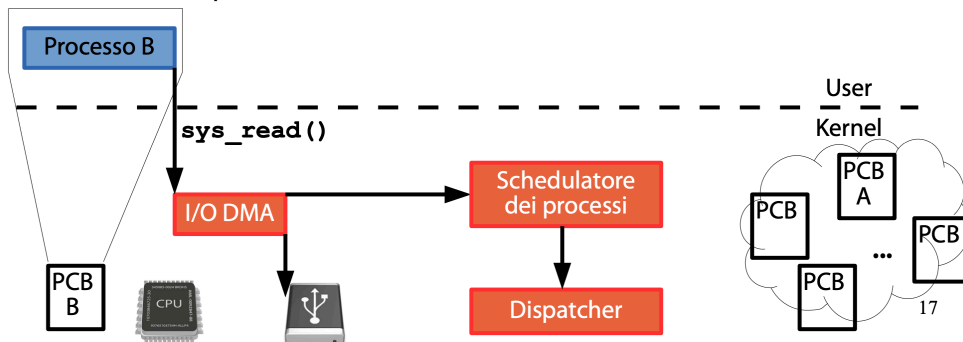
1. Un processo si blocca in attesa di un evento.
2. Un processo termina l'esecuzione.

Altrimenti il processo rimane *inattivo*.

## Prelazione

*Il kernel strappa con forza la CPU ad un processo*, l'atto di interrompere momentaneamente un processo troppo avido di CPU a favore di un altro che sta aspettando l'esecuzione.

Lo schema precedentemente visto non ha prelazione. Un processo CPU-bound può provocare starvation di altri processi.



## Scheduling con prelazione

Quando?

1. Un processo si blocca in attesa di un evento.
2. Un processo termina l'esecuzione.
3. È stato creato un nuovo processo.
4. Un processo è interrotto.
5. Un processo passa dallo stato bloccato allo stato pronto.

## Scheduling in Linux



È con prelazione. Implementato dalla funzione `schedule()`, definita in `$LINUX/kernel/sched/core.c`:

- Sceglie le `task_struct()` di un nuovo processo.
- Salva lo stato del processo attuale nella sua `task_struct`.
- Ripristina nei registri lo stato del nuovo processo.
- Salta alla prossima istruzione utile del nuovo processo.

La funzione `schedule()` è invocata nelle seguenti occasioni:

- Blocco dovuto da I/O.
- Blocco dovuto ad evento di sincronizzazione.
- Al termine di una chiamata di sistema.
- Al termine di un gestore delle interruzioni.

Lo scheduler non rischedula per forza, solamente se forzato dagli eventi (ad esempio, se un processo a priorità più elevata è pronto).

Il kernel usa un flag (`TIF_NEED_RESCHED`) della `task_struct` del processo in esecuzione per segnalare la necessità di rischedulare.

Il kernel controlla se il flag è impostato mediante la funzione `need_resched()` definita in `$LINUX/include/linux/sched.h`. Se la funzione ritorna 1 (TRUE) allora viene invocata la `schedule()`, altrimenti no.

## Funzioni rientranti

**Dati di fatto:**

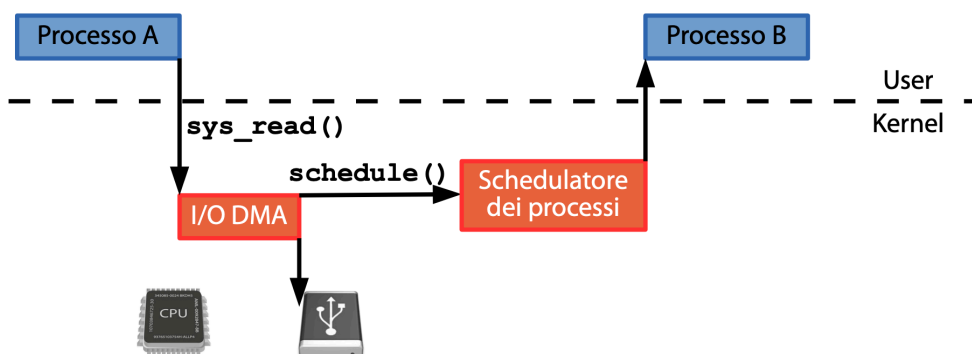
- Le macchine moderne sono multiprocessore.
- I SO moderni sono multiutente e time sharing.
- Le operazioni di I/O sono gestite in maniera asincrona tramite DMA e interruzioni.

**Conseguenza inevitabile:**

- **Funzione rientrante:** funzione del kernel può essere invocata di nuovo prima del termine di una sua precedente invocazione.

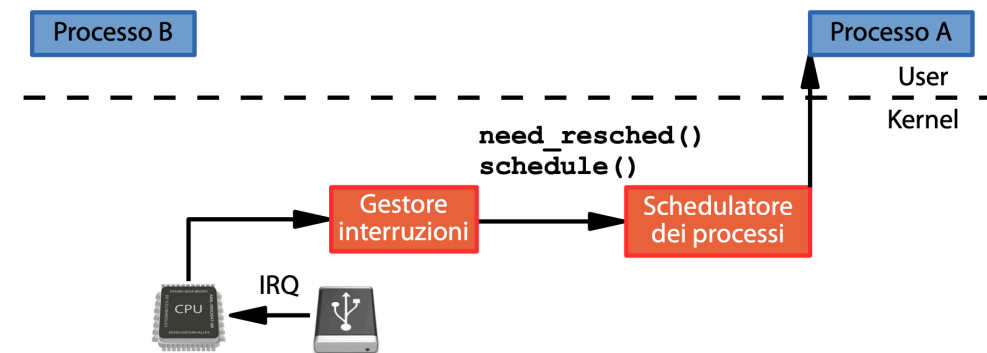
## Rientranza di `schedule()`

In questo istante, la funzione `schedule()` non è ancora terminata. La sostituzione dei processi provoca un salto diretto al processo B.



A  $\xrightarrow{\text{sys\_read()}}$  I/O DMA  $\xrightarrow{\text{schedule()}}$  Scheduler  $\rightarrow$  B

All'arrivo della richiesta di interruzioni da parte del disco, parte il gestore delle interruzioni. Al termine della sua esecuzione, lo scheduler dei processi rischedula, nel caso, il processo A.



Notare come `schedule()` sia stata nuovamente invocata prima che la precedente istanza di `schedule()` sia terminata.

Al termine della `schedule()`, il controllo è ritornato alle funzioni invocate dalla chiamata di sistema `sys_read()`.

## Vantaggi della rientranza

Il kernel esegue in maniera più efficiente su architetture [SMP](#).

## Svantaggi della rientranza

Le funzioni devono essere progettate in modo tale da essere eseguibili contemporaneamente:

- L'accesso a variabili globali va evitato come la peste, se possibile.
- Se proprio non si riesce ad evitare l'accesso a variabili globali, tale accesso deve essere serializzato.

La sua esecuzione su più CPU va impedita per evitare corruzione sui dati.

## Idle task

Se nessun processo è in grado di eseguire, il kernel esegue un processo speciale detto *swapper* o *idle task*.

- PID = 0
- Mette in idle il processore.

## Cambio di contesto

Lo scambio di due processi prende il nome di *cambio di contesto* (**context switch**).

**Contesto**: contenuto del PCB.

Operazioni svolte:

- Salvataggio del contesto del processo attuale.
- Ripristino del contesto del nuovo processo.

Il cambio di contesto impiega 1µs – 1ms (in base all'architettura hardware).

## Hop al nuovo processo

Uno dei campi contenuti nel contesto è il registro **Instruction Pointer**. Se tale registro fosse ripristinato per ultimo, la CPU salterebbe automaticamente alla prossima istruzione della nuova traccia. Nella sostanza, il salto avviene proprio così.

**Problema:** un processo appena creato (`fork()`) deve essere inizializzato prima di poter eseguire la prima volta.

Si usa un piccolo trucco:

- `push` sullo stack l'indirizzo della prossima istruzione da eseguire.
- `jump` (`JMP`) ad una funzione in cui vengono effettuati gli ultimi preparativi.
- La funzione invocata esegue un `return` (`RET`).
- L'**Instruction Pointer** è caricato con l'ultimo valore presente sullo stack (l'indirizzo desiderato).

La macro `switch_to()`, definita in `$LINUX/arch/x86/include/asm/switch_to.h` implementa questa variante.

La funzione inizializzatrice è: `ret_from_fork()`.

## T7 - Algoritmi di scheduling

Il kernel deve assegnare una risorsa ad  $n$  entità che la vogliono accedere.

Come vengono assegnate le richieste alle risorse?

### Lo scheduler

Gestisce l'accesso alle risorse:

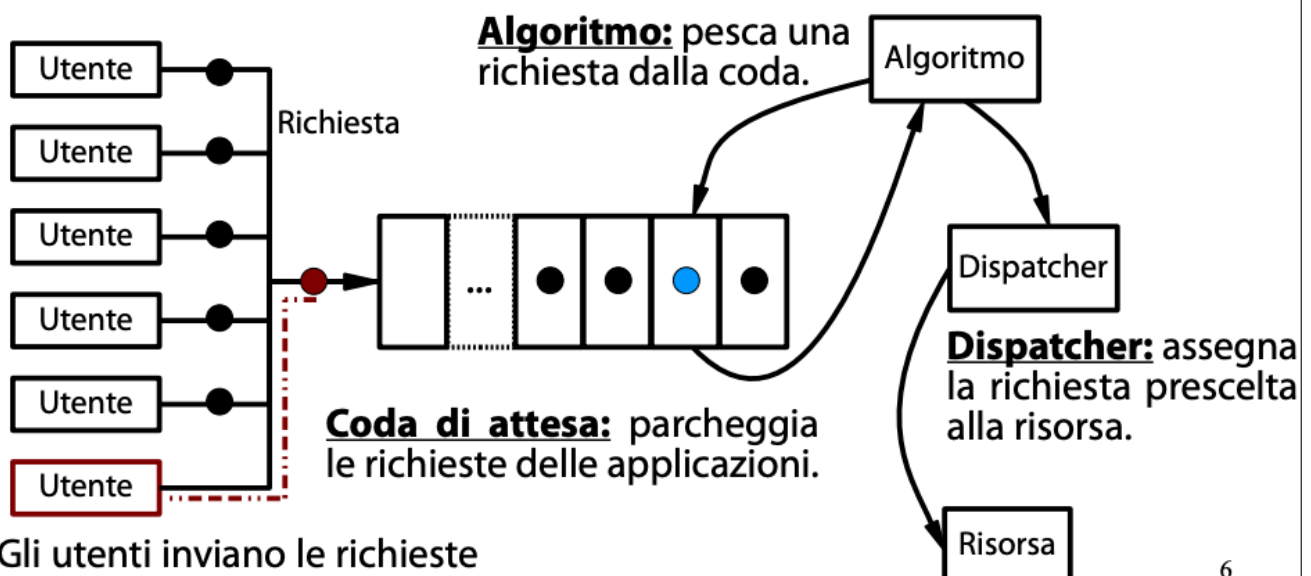
**Accodamento:** le richieste sono ricevute e "parcheeggiate".

**Algoritmo:** una richiesta è scelta dalla coda.

**Dispatching:** la richiesta è assegnata alla risorsa.

### Un semplice modello

(Scheduler = code di attesa + algoritmo + meccanismo di dispatch)



**CPU scheduling:** scelta del prossimo processo da eseguire.

- *Entità*: processi
- *Risorsa*: CPU  
**Job scheduling**: scelta del prossimo job da eseguire.
- *Entità*: job
- *Risorsa*: CPU  
**I/O scheduling**: scelta della prossima richiesta di disco (lettura/scrittura) da soddisfare.
- *Entità*: richieste di I/O
- *Risorsa*: disco  
**Memory scheduling**: scelta di una pagina di memoria da spostare in swap.
- *Entità*: pagine di memoria
- *Risorsa*: memoria

## Scheduling con e senza prelazione

**Con prelazione**: lo scheduler interrompe la fruizione della risorsa da parte di un utente per favorirne un altro.

**Senza prelazione**: lo scheduler non interrompe la fruizione della risorsa. L'utente continua ad usufruire della risorsa fino a quando:

- non termina la sua funzione
- non necessita di un'altra risorsa

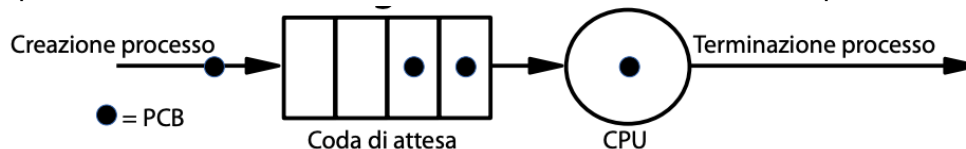
## Scheduling senza prelazione

**Scheduling cooperativo**: l'utente può decidere di assegnare la risorsa direttamente ad un altro utente, tramite lo **yielding**.

**Scheduling non cooperativo**: l'utente non influisce sulle scelte decisionali dello scheduler.

## Modello senza prelazione

I processi non effettuano richieste ad altre risorse. Non esiste prelazione.



Un **indice di prestazione** indica il livello di prestazione di un componente hw/sw. Nel caso di uno scheduler di CPU si considerano:

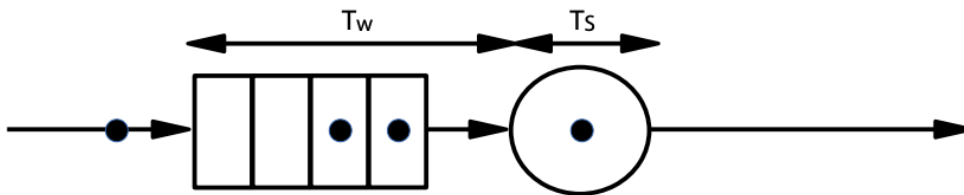
- *Attesa di un processo*
- *Produttività dello scheduler*
- *Utilizzo del processore*

## Processi in coda

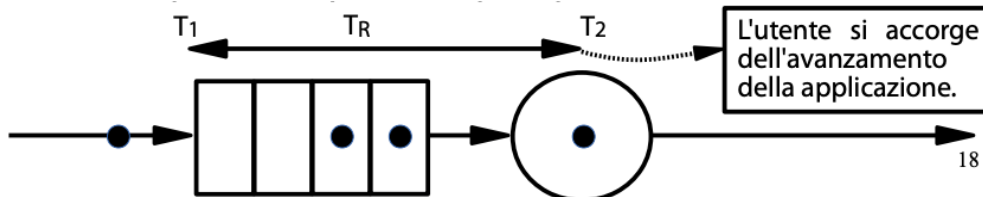
Il **numero di processi accodati** in un'istante  $t = t_1$ , misura il numero di processi precedenti al nostro processo accodato.

Il **tempo di attesa in coda** indica l'intervallo temporale tra l'ingresso in coda e l'associazione al processore (tempo in coda è tempo di "congestione" del processore).

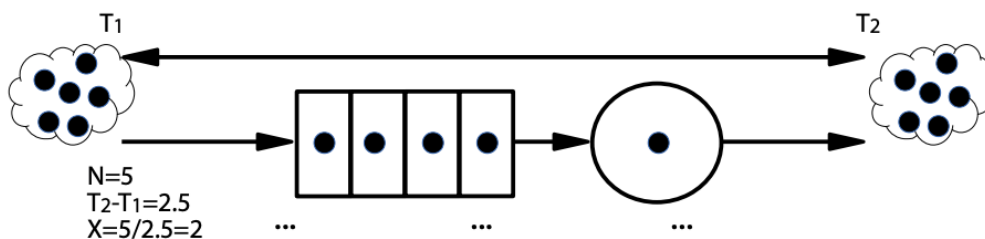
**Tempo di completamento:** somma dei tempi di attesa in coda e di servizio  $T_C = T_w + T_s$ , misura l'attesa complessiva di un processo.



**Latenza:** intervallo temporale fra l'ingresso di un processo in coda ed il calcolo del primo byte della risposta utile all'applicazione:  $T_R = T_2 - T_1$ , misura l'attesa di un processo ad iniziale la sua elaborazione e lo stato di congestione del processore. L'attesa percepita dall'utente.



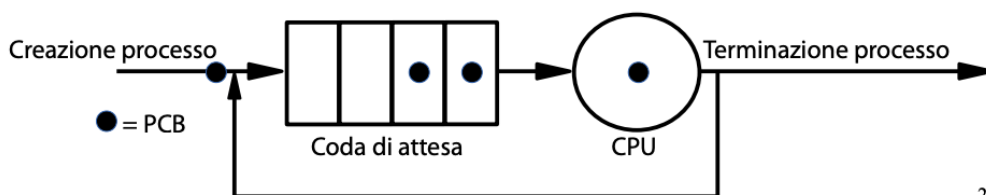
**Throughput:** numero di processi portati a termine da uno scheduler nell'unità di tempo  $X = \frac{N}{t_2 - t_1}$ , misura la produttività dello scheduler.



**Utilizzatore:** frazione di tempo in cui il processore è occupato ( $T_{occ}$ ) in un dato intervallo ( $T_{mis}$ ):  $p = \frac{T_{occ}}{T_{mis}}$ , misura l'utilizzo del processore.

## Modello con prelazione

Si deve prevedere il rientro di un processo nella coda di attesa, un processo può subire più attese in coda e ricevere più volte servizio dal processore.



## Throughput vs. Latenza

Sono *obiettivi contrastanti*:

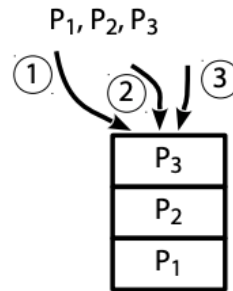
- Uno scheduler ad alto throughput serve tutto ciò che si presenta in coda → Non si fa distinzione fra processi → interattivi e non La latenza di tali processi aumenta.
- Uno scheduler a bassa latenza tende a scegliere i processi interattivi → Gli altri processi non avanzano → Il throughput dello scheduler cala.

## Algoritmi di base

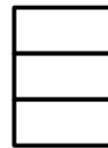
### First come, first served

Opera *senza prelazione*, un processo esegue fino all'I/O o alla fine.

Processo	Durata
P <sub>1</sub>	24
P <sub>2</sub>	3
P <sub>3</sub>	3



Processo	Durata
P <sub>1</sub>	24
P <sub>2</sub>	3
P <sub>3</sub>	3



#### Vantaggi:

- semplice
- eseguibile su piattaforme non dotate di clock hardware
- non implementa prelazione
  - i processi non devono essere interrotti bruscamente
  - non serve un timer per avviare periodicamente il meccanismo di interruzione

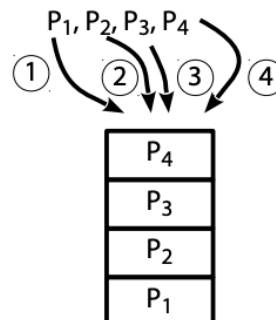
#### Svantaggi:

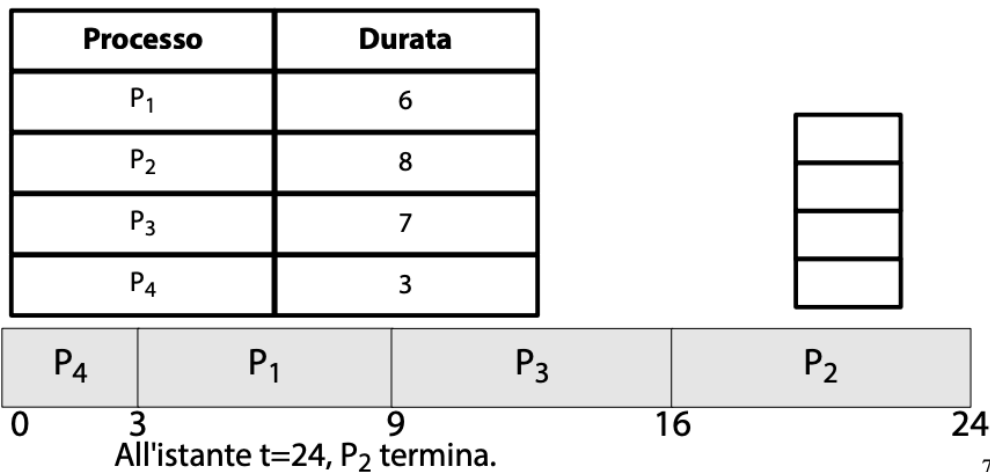
- Attesa non minima, varia al variare della composizione dei processi
- senza prelazione, monopolizza il processore e stalla l'esecuzione degli altri processi(starvation)

## Shortest job first

Opera *senza prelazione*, un processo esegue fino all'I/O o alla fine.

Processo	Durata
P <sub>1</sub>	6
P <sub>2</sub>	8
P <sub>3</sub>	7
P <sub>4</sub>	3





### Vantaggi:

- In assenza di prelazione, si può dimostrare che SJF è l'algoritmo di scheduling che fornisce tempo di attesa medio minimo.
  - Dati  $n$  processi  $P_1, P_2, \dots, P_n$ ;
  - $R_W(P_j)$  tempo di attesa del processo  $P_j$ ;
  - $T_W = \left(\frac{1}{n}\right) \times [T_W(P_1) + T_W(P_2) + \dots + T_W(P_n)]$  tempo medio di attesa dei processi al termine della schedulazione.
  - $T_W$  minimo fra tutti gli scheduler senza prelazione.

### Svantaggi:

- Richiede la conoscenza della durata del prossimo CPU burst di ciascun processo, va stimata.
- Lo scheduler eseguito non è SJF, bensì un "simil-SJF" con stima dei CPU burst.
- SJF considera l'intera durata del processo. Tuttavia, in uno scheduler con prelazione, i processi sono eseguiti "per piccoli pezzi" e le durate residue divergono rapidamente dalla durata iniziale.
  - L'informazione su cui fa conto SJF diventa molto rapidamente obsoleta.

## Stima durata prossimo CPU burst

**Media esponenziale**  $\tau_n$  dei CPU burst passati e presente.

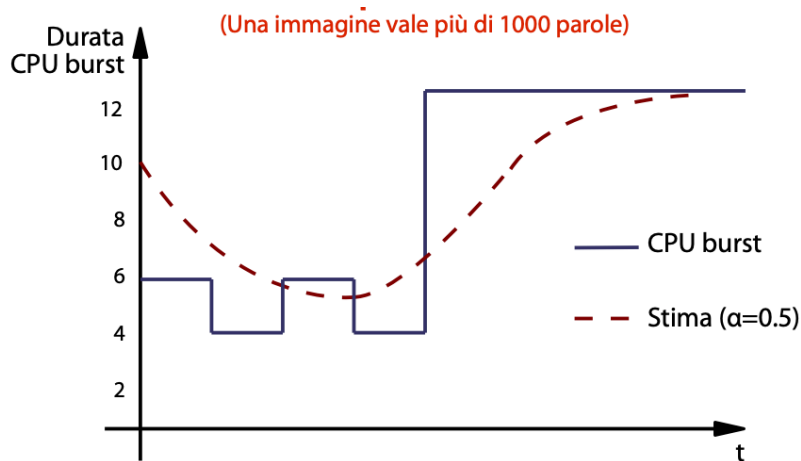
$$\tau_{n+1} = a \times \tau_n + (1 - a) \times \tau_n, a \in [0, 1]$$

$a$  "tara" il quantitativo di storia passata da prendere in considerazione.

$a = 0$  valore recente non ha effetto

$a = 1$  assenza di storia

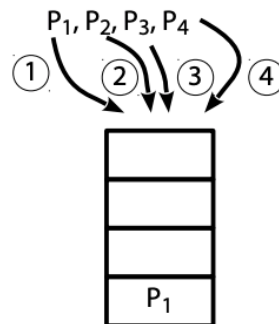
$a = \frac{1}{2}$  stesso peso per valore recente e storia passata



## Shortest remaining time first

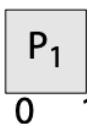
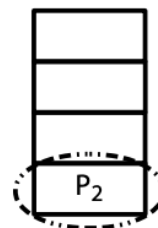
Opera *con prelazione*, un processo può essere interrotto dallo scheduler.

Processo	Durata	Residuo	Arrivo
$P_1$	8	8	0
$P_2$	4	4	1
$P_3$	9	9	2
$P_4$	5	5	3



$P_1$  viene eseguito, a  $t_2$  arriva  $P_2$  che, avendo un tempo residuo minore di  $P_1$ ,  $P_1$  torna in coda ( $P_1 = 7 < P_2 = 4$ ).

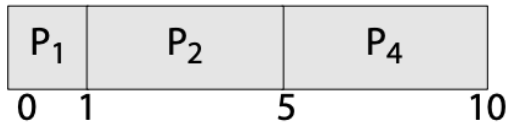
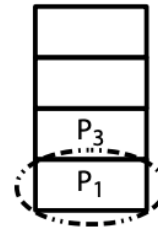
Processo	Durata	Residuo	Arrivo
$P_1$	8	7	0
$P_2$	4	4	1
$P_3$	9	9	2
$P_4$	5	5	3



$P_3$  e  $P_4$  si accodano, al termine di  $P_2$  viene eseguito  $P_4$ , quello con il tempo residuo minore ( $5_{P_4} < 7_{P_1} < 9_{P_3}$ ).



Processo	Durata	Residuo	Arrivo
P <sub>1</sub>	8	7	0
P <sub>2</sub>	4	0	1
P <sub>3</sub>	9	9	2
P <sub>4</sub>	5	0	3



Ora vengono eseguiti  $P_1$  e  $P_3$  che termina l'esecuzione.

**Vantaggi:**

- Diminuisce il tempo di attesa medio rispetto a SJF.

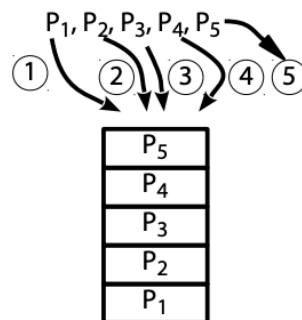
**Svantaggi:**

- Richiede la conoscenza della durata del CPU burst residuo di ciascun processo, va stimata.
- Lo scheduler eseguito non è SRTF, bensì un “simil-SRTF” con stima dei CPU burst.

## Priority

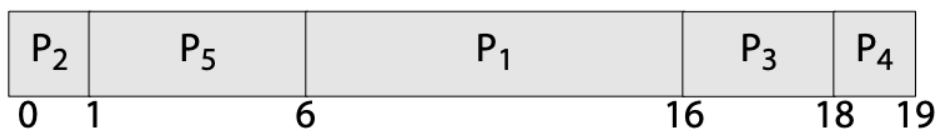
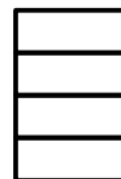
Opera *con o senza prelazione*.

Processo	Durata	Priorità
P <sub>1</sub>	10	3
P <sub>2</sub>	1	1
P <sub>3</sub>	2	4
P <sub>4</sub>	1	5
P <sub>5</sub>	5	2



Vengono eseguiti in base al valore della *priorità*: 1 → 5.

Processo	Durata	Priorità
P <sub>1</sub>	10	3
P <sub>2</sub>	1	1
P <sub>3</sub>	2	4
P <sub>4</sub>	1	5
P <sub>5</sub>	5	2



All'istante  $t=19$ ,  $P_4$  termina.

**Vantaggi:**

- Favorisce i processi ritenuti più importanti. Tale modo di operare è fondamentale nei sistemi interattivi.

- **Processo importante**: processo che interagisce “spesso” e direttamente con l'utente.

#### Svantaggi:

- È soggetto a starvation dei processi. Nella simulazione ora vista,  $P_4$  ha tempo di attesa pari a 18 → starvation. Se  $P_4$  fosse un terminale, stallerebbe per lungo tempo e l'utente non riuscirebbe ad interagire con il SO.

Gli scheduler con priorità necessitano di un ulteriore meccanismo per combattere l'attesa indefinita.

**Aging**: aumento graduale della priorità dei processi in attesa da lungo tempo.

Il valore della priorità può essere definito in due modi:

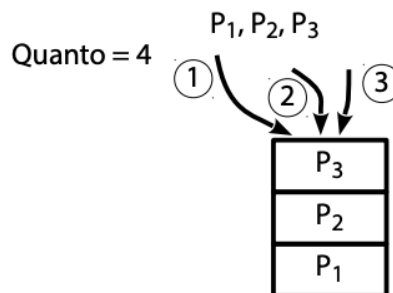
- **Internamente**: dal SO in base a grandezze misurabili.
  - Limiti di tempo, memoria, CPU burst.
  - Rapporto (avg I/O burst) / (avg CPU burst).
- **Esternamente**: in base a criteri esterni al SO.
  - Si associa ad ogni processo una priorità.
  - Sistemi UNIX: valore intero in  $[-20, 19]$  (più è basso, più è alta la priorità).

## Round Robin

Opera *con prelazione*, un processo può essere interrotto dallo scheduler.

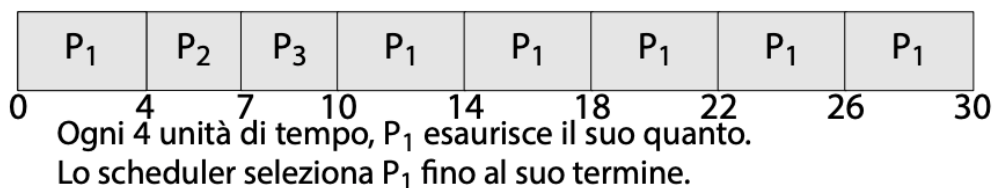
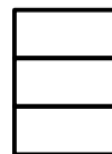
Ogni esecuzione è quantificata in un tempo detto *quanto*, al termine del quanto, si passa al processo successivo, rimettendo in coda il processo se non completato.

Processo	Durata
$P_1$	24
$P_2$	3
$P_3$	3



Processo	Durata
$P_1$	24
$P_2$	3
$P_3$	3

Quanto = 4



#### Vantaggi:

- Fornisce generalmente attese molto basse.
- Con  $n$  processi ed un quanto di tempo pari a  $q$ , l'attesa di un processo è limitata superiormente a:

$$(n - 1) \cdot q$$

### Svantaggi:

- Fornisce tempi di completamento di processi CPU-bound più lunghi rispetto a FCFS.
  - FCFS esegue i CPU burst "in toto", fino alla fine, mentre RR si alterna fra processi.

## Quanti di tempo fissi e variabili

**Quanto fisso:** scelto una volta per tutte e non cambia mai. È molto semplice da gestire, ma un quanto fisso non va mai bene per tutte le categorie di processi.

**Time slice variabile:** il kernel calcola dinamicamente l'intervallo massimo di esecuzione, in funzione della tipologia di processo e della sua priorità di esecuzione.

## Impatto del quanto sulle schedulazioni

Il tempo di completamento medio tende a diminuire se la maggior parte dei processi finisce il CPU burst entro lo scadere del quanto.

**Regoletta empirica:** l'80% dei processi dovrebbe avere CPU burst minori del quanto.

## Scheduling multilivello

### Categorizzazione dei processi nei SO moderni

**Processi interattivi** (*foreground*): devono andare in esecuzione il più presto possibile.

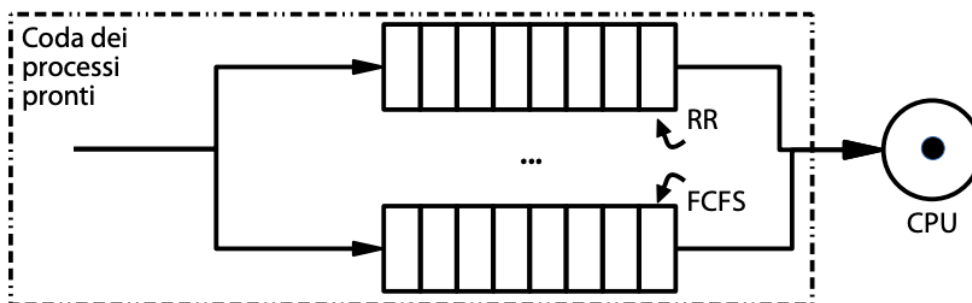
**Processi non interattivi brevi** (*background*): devono andare in esecuzione meno spesso degli interattivi.

**Processi non interattivi lunghi** (*batch*): vanno in esecuzione solo se non ne esistono di interattivi e non interattivi brevi.

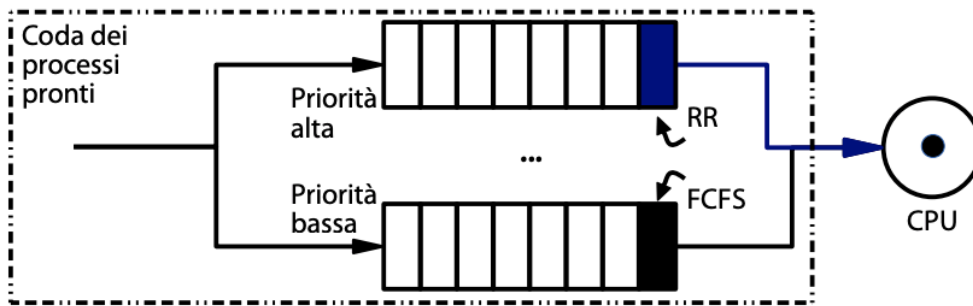
È necessario saper schedulare prima i processi appartenenti alle classi più importanti.

### Scheduler multilivello

Si usano più code, una per ogni categoria di processi.



1. sceglie la coda da cui pescare un processo.
2. sceglie un processo dalla coda, associando diversi livelli di priorità(fissi) alle code. Si sceglie la coda a priorità più alta in cui è presente un processo.

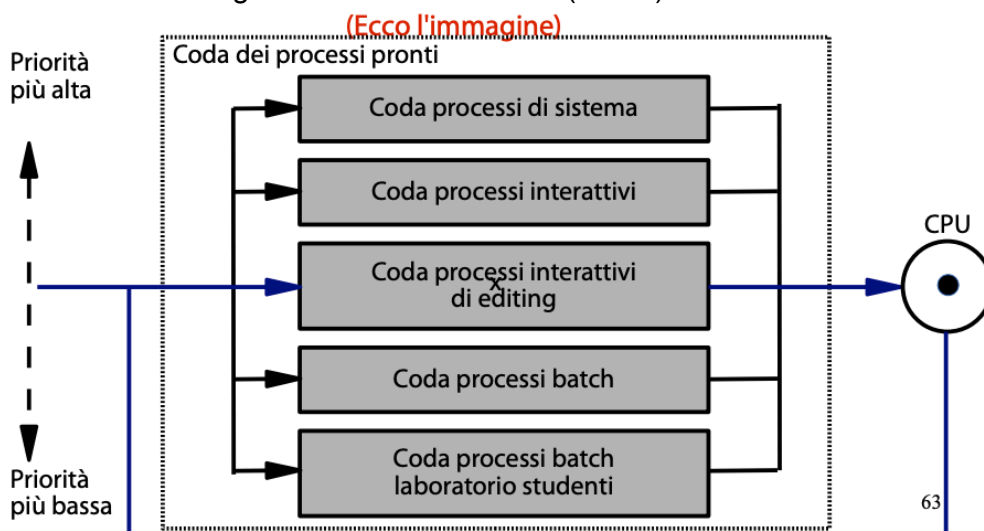


Dopo l'esecuzione, il processo è inserito di nuovo nella stessa coda da cui proviene.

#### Esempio:

Si suddivide la coda di pronto in cinque code distinte, con priorità decrescente in termini di importanza.

3. Processi di sistema (RR).
4. Processi interattivi (RR).
5. Processi interattivi di editing (RR).
6. Processi in background (FCFS).
7. Processi batch degli studenti di laboratorio (FCFS).



#### Vantaggi:

- I processi importanti sono serviti per prima, essendo incanalati nelle code a priorità più alta.
- Per tali processi, la latenza è molto bassa.

#### Svantaggi:

- I processi non importanti soffrono di starvation.
  - Per tali processi, non vi è garanzia sulla latenza.
  - È necessario integrare lo scheduler con un meccanismo di aging.
- Se la natura del processo (CPU-bound, I/O-bound) è variabile nel tempo, lo scheduler multilivello non è efficace.

## Scheduler multilivello con retroazione

È una variante dello scheduler multilivello in cui il processo può rientrare in una coda diversa da quella in cui si trovava precedentemente.

La retroazione tende a raggruppare i processi con caratteristiche di CPU burstiness simili simili nelle stesse code.

- Processi con CPU burst lunghi si muovono verso le code a priorità più bassa.
  - Processi con CPU burst brevi si muovono verso le code a priorità più alta.
- La retroazione è tanto più efficace quanto meno frequente è il cambio di natura dei processi.

#### Esempio:

Si usano tre code di scheduling di priorità 0, 1, 2.

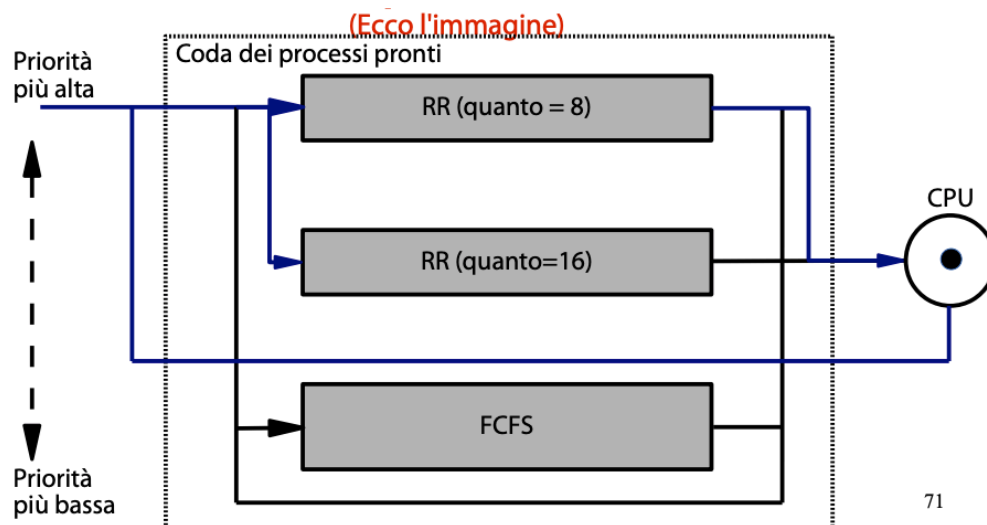
0. Round Robin con quanto 8
1. Round Robin con quanto 16
2. FCFS

Se un processo non finisce entro il suo quanto assegnato, viene spostato nella coda successiva. Se per 10 volte il processo non esaurisce il suo quanto di tempo, viene spostato nella coda precedente.

Quindi:

- I processi con CPU burst brevi (I/O bound, interattivi con il terminale) sono serviti molto rapidamente.
- I processi con CPU burst un po' più lunghi sono serviti rapidamente.
- I processi con CPU burst lunghi (batch) sono serviti solo se il sistema non è altrimenti impegnato.

Un sistema con CPU burst variabile si adatta lentamente alla sua coda ottimale.



## Scheduling in sistemi SMP

### Coda di pronto globali

**Coda globale:** unica coda di pronto condivisa da tutti i processori.

**Coda di pronto distribuita per CPU:** ne esiste una per ogni processore della macchina.

- Lo scheduler può operare su più code allo stesso istante.
- Non esistono ritardi dovuti a uso non corretto delle cache del processore.
- Soluzione preferita dai SO moderni su architetture SMP.

Come bilanciare i processi sulle diverse code?

### Predilezione

**Predilezione** (*CPU affinity*): lo scheduler dei processi cerca di eseguire il processo sempre sullo stesso processore.

**Predilezione debole** (*soft affinity*): lo scheduler rischedula il processo sempre sulla stessa CPU, a meno di un bilanciamento delle code di scheduling.

- scheduler di Linux

**Predilezione forte** (*hard affinity*): l'utente impone che il processo esegua sempre su una data CPU.

## Bilanciamento del carico

Per bilanciare il carico della predilezione i processi vengono spostati (*migrati*) da un processore all'altro, in modo che il carico sui processori sia simile.

Attenzione a bilanciare troppo spesso, *il bilanciamento annulla i benefici della predilezione*.

**Migrazione guidata**(*push migration*): un processo dedicato controlla periodicamente la lunghezza delle code. In caso di sbilanciamento, sposta i processi in modo da bilanciare il carico.

**Migrazione spontanea**(*pull migration*): o scheduler sottrae un PCB ad una coda sovraccarica.