



# Gestione dei processi

---

*Roberto De Virgilio*

*Sistemi operativi - 15 Novembre 2017*

# Introduzione

---

- Un sistema operativo consiste in un gran numero di *attività* che vengono eseguite più o meno *contemporaneamente* dal processore e dai dispositivi presenti in un elaboratore.
- Senza un modello adeguato, la coesistenza delle diverse attività sarebbe difficile da descrivere e realizzare.
- Il modello che è stato realizzato a questo scopo prende il nome di *modello concorrente* ed è basato sul concetto astratto di *processo*

# Processi e programmi

- **Definizione: processo**
  - E' un'attività controllata da un programma che si svolge su un processore
- **Un processo non è un programma!**
  - Un programma è un entità *statica*, un processo è *dinamico*
  - Un programma:
    - specifica un'insieme di istruzioni e la loro sequenza di esecuzione
    - non specifica la distribuzione nel tempo dell'esecuzione
  - Un processo:
    - rappresenta il modo in cui un programma viene eseguito nel tempo
- **Assioma di *finite progress***
  - Ogni processo viene eseguito ad una velocità finita ma sconosciuta

# Stato di un processo

- Ad ogni istante, un processo può essere totalmente descritto dalle seguenti componenti:
  - *La sua immagine di memoria*
    - la memoria assegnata al processo  
(ad es. testo, dati, stack)
    - le strutture dati del S.O. associate al processo  
(ad es. file aperti)
  - *La sua immagine nel processore*
    - contenuto dei registri generali e speciali
  - *Lo stato di avanzamento*
    - descrive lo stato corrente del processo: ad esempio, se è in esecuzione o in attesa di qualche evento

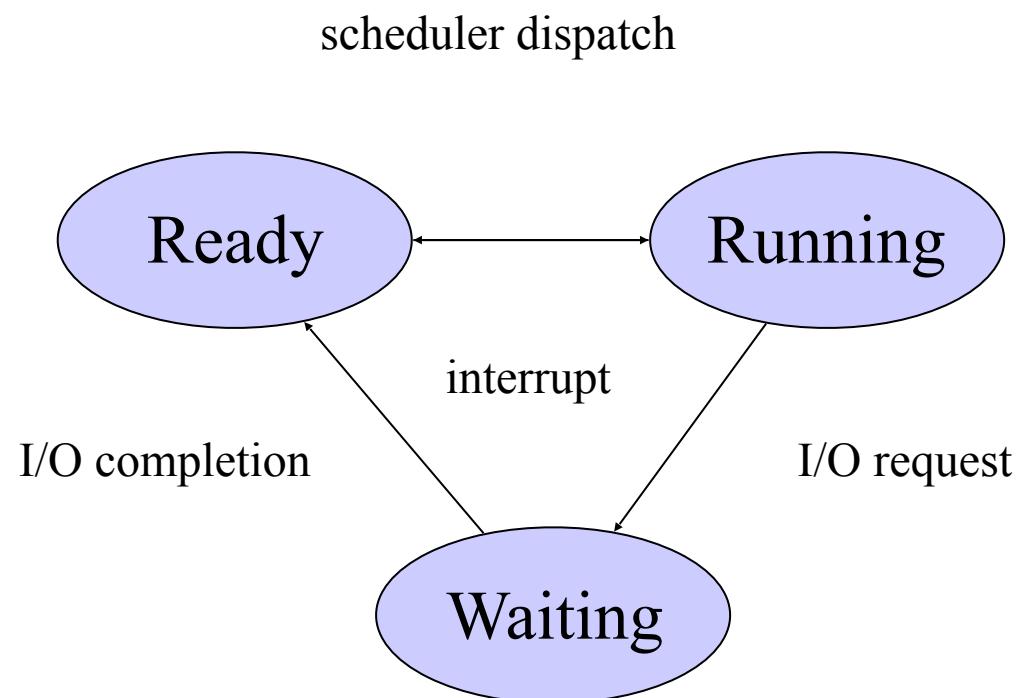
# Processi e programmi (ancora)

- **Più processi possono eseguire lo stesso programma**
  - In un sistema multiutente, più utenti possono leggere la posta contemporaneamente
  - Un singolo utente può eseguire più istanze dello stesso editor
- **In ogni caso, ogni istanza viene considerata un processo separato**
  - Possono condividere lo stesso codice ...
  - ... ma i dati su cui operano, l'immagine del processore e lo stato di avanzamento sono separati

# Stati dei processi (versione semplice)

- **Stati dei processi:**

- *Running*: il processo è in esecuzione
- *Waiting*: il processo è in attesa di qualche evento esterno (ad es. completamento operazione di I/O); non può essere eseguito
- *Ready*: il processo può essere eseguito, ma attualmente il processore è impegnato in altre attività



# Cos'è la concorrenza?

- Tema centrale nella progettazione dei S.O. riguarda la gestione di processi *multipli*
  - *Multiprogramming*
    - più processi su un solo processore
    - parallelismo apparente
  - *Multiprocessing*
    - più processi su una macchina con processori multipli
    - parallelismo reale
  - *Distributed processing*
    - più processi su un insieme di computer distribuiti e indipendenti
    - parallelismo reale

# Cos'è la concorrenza?

- **Esecuzione concorrente:**
  - Due programmi si dicono in esecuzione concorrente se vengono eseguiti in parallelo (con parallelismo reale o apparente)
- **Concorrenza:**
  - E' l'insieme di notazioni per descrivere l'esecuzione concorrente di due o più programmi
  - E' l'insieme di tecniche per risolvere i problemi associati all'esecuzione concorrente, quali *comunicazione* e *sincronizzazione*

# Dove possiamo trovare la concorrenza?

- **Applicazioni multiple**
  - la multiprogrammazione è stata inventata affinchè più processi indipendenti condividano il processore
- **Applicazioni strutturate su processi**
  - estensione del principio di progettazione modulare; alcune applicazioni possono essere progettate come un insieme di processi o thread concorrenti
- **Struttura del sistema operativo**
  - molte funzioni del sistema operativo possono essere implementate come un insieme di processi o thread

# Multiprocessing e multiprogramming: differenze?

- **In un singolo processore:**
  - processi multipli sono "*alternati nel tempo*" per dare l'impressione di avere un multiprocessore
  - ad ogni istante, al massimo un processo è in esecuzione
  - si parla di *interleaving*
- **In un sistema multiprocessore:**
  - più processi vengono eseguiti *simultaneamente* su processori diversi
  - i processi sono "*alternati nello spazio*"
  - si parla di *overlapping*

# Multiprocessing e multiprogramming: differenze?

- **A prima vista:**
  - si potrebbe pensare che queste differenze comportino problemi distinti
  - in un caso l'esecuzione è simultanea
  - nell'altro caso la simultaneità è solo simulata
- **In realtà:**
  - presentano gli stessi problemi
  - che si possono riassumere nel seguente:

*non è possibile predire la velocità relativa dei processi*

# Un esempio semplice

- Si consideri il codice seguente:

In C:

```
void modifica(int valore) {  
    totale = totale + valore  
}
```

- Supponiamo che:

- Esista un processo  $P_1$  che esegue `modifica (+10)`
- Esista un processo  $P_2$  che esegue `modifica (-10)`
- $P_1$  e  $P_2$  siano in esecuzione concorrente
- `totale` sia una variabile condivisa tra i due processi, con valore iniziale 100
- Alla fine, `totale` dovrebbe essere uguale a 100. Giusto?

In Assembly:

```
.text  
modifica:  
    lw $t0, totale  
    add $t0, $t0, $a0  
    sw $t0, totale  
    jr $ra
```

# Scenario 1: multiprogramming (corretto)

P1lw \$t0, totale	totale=100, \$t0=100, \$a0=10
P1add \$t0, \$t0, \$a0	totale=100, \$t0=110, \$a0=10
P1sw \$t0, totale	totale=110, \$t0=110, \$a0=10
S.O. interrupt	
S.O. salvataggio registri P1	
S.O. ripristino registri P2	totale=110, \$t0=? , \$a0=-10
P2lw \$t0, totale	totale=110, \$t0=110, \$a0=-10
P2add \$t0, \$t0, \$a0	totale=110, \$t0=100, \$a0=-10
P2sw \$t0, totale	totale=100, \$t0=100, \$a0=-10

## Scenario 2: multiprogramming (errato)

P1lw \$t0, totale

totale=100, \$t0=100, \$a0=10

S.O. interrupt

S.O. salvataggio registri P1

S.O. ripristino registri P2

totale=100, \$t0=? , \$a0=-10

P2lw \$t0, totale

totale=100, \$t0=100, \$a0=-10

P2add \$t0, \$t0, \$a0

totale=100, \$t0= 90, \$a0=-10

P2sw \$t0, totale

totale= 90, \$t0= 90, \$a0=-10



S.O. interrupt

S.O. salvataggio registri P2

S.O. ripristino registri P1

totale= 90, \$t0=100, \$a0=10

P1add \$t0, \$t0, \$a0

totale= 90, \$t0=110, \$a0=10

P1sw \$t0, totale

totale=110, \$t0=110, \$a0=10

## Scenario 3: multiprocessing (errato)

- I due processi vengono eseguiti simultaneamente da due processori distinti

Processo P1:

```
lw $t0, totale
```

```
add $t0, $t0, $a0
```

```
sw $t0, totale
```

Processo P2:

```
lw $t0, totale
```

```
add $t0, $t0, $a0
```

```
sw $t0, totale
```

- Nota:

- i due processi hanno insiemi di registri distinti
- l'accesso alla memoria su **totale** non può essere simultaneo

# Alcune considerazioni

- **Non vi è sostanziale differenza tra i problemi relativi a multiprogramming e multiprocessing**
  - ai fini del ragionamento sui programmi concorrenti si ipotizza che sia presente un "processore ideale" per ogni processo
- **I problemi derivano dal fatto che:**
  - non è possibile predire gli istanti temporali in cui vengono eseguite le istruzioni
  - i due processi accedono ad una o più risorse condivise

# Race condition

- **Definizione**
  - Si dice che un sistema di processi multipli presenta una *race condition* qualora il risultato finale dell'esecuzione dipenda dalla temporizzazione con cui vengono eseguiti i processi
- **Per scrivere un programma concorrente:**
  - è necessario eliminare le race condition

# Considerazioni finali

- **In pratica:**
  - scrivere programmi concorrenti è più difficile che scrivere programmi sequenziali
  - la correttezza non è solamente determinata dall'esattezza dei passi svolti da ogni singola componente del programma, ma anche dalle interazioni (volute o no) tra essi
- **Nota:**
  - Fare debug di applicazioni che presentano race condition non è per niente piacevole...
  - Il programma può funzionare nel 99.999% dei casi, e bloccarsi inesorabilmente quando lo discutete con il docente all'esame...
  - (... un corollario alla legge di Murphy...)

# Interazioni tra processi

- E' possibile classificare le modalità di interazione tra processi in base a quanto sono "**consapevoli**" uno dell'altro.
- Processi totalmente "ignari" uno dell'altro:
  - processi indipendenti non progettati per lavorare insieme
  - sebbene siano indipendenti, vivono in un ambiente comune
- Come interagiscono?
  - competono per le stesse risorse
  - devono sincronizzarsi nella loro utilizzazione
- Il sistema operativo:
  - deve arbitrare questa **competizione**, fornendo meccanismi di **sincronizzazione**

# Interazioni tra processi

- **Processi "indirettamente" a conoscenza uno dell'altro**
  - processi che condividono risorse, come ad esempio un buffer, al fine di scambiarsi informazioni
  - non si conoscono in base ai loro id, ma interagiscono indirettamente tramite le risorse condivise
- **Come interagiscono?**
  - cooperano per qualche scopo
  - devono *sincronizzarsi* nella utilizzazione delle risorse
- **Il sistema operativo:**
  - deve facilitare la *cooperazione*, fornendo meccanismi di *sincronizzazione*

# Interazioni tra processi

- **Processi "direttamente" a conoscenza uno dell'altro**
  - processi che comunicano uno con l'altro sulla base dei loro id
  - la comunicazione è diretta, spesso basata sullo scambio di messaggi
- **Come interagiscono**
  - cooperano per qualche scopo
  - comunicano informazioni agli altri processi
- **Il sistema operativo:**
  - deve facilitare la *cooperazione*, fornendo meccanismi di *comunicazione*

# Proprietà

- **Definizione**
  - Una *proprietà* di un programma concorrente è un attributo che rimane vero per ogni possibile storia di esecuzione del programma stesso
- **Due tipi di proprietà:**
  - *Safety* ("nothing bad happens")
    - mostrano che il programma (se avanza) va "nella direzione voluta", cioè non esegue azioni scorrette
  - *Liveness* ("something good eventually happens")
    - il programma avanza, non si ferma... insomma è "*vitale*"

# Proprietà - Esempio

- **Consensus, dalla teoria dei sistemi distribuiti**
  - Si consideri un sistema con **N** processi:
    - All'inizio, ogni processo *propone* un valore
    - Alla fine, tutti i processi si devono accordare su uno dei valori proposti (*decidono* quel valore)
- **Proprietà di safety**
  - Se un processo decide, deve decidere uno dei valori proposti
  - Se due processi decidono, devono decidere lo stesso valore
- **Proprietà di liveness**
  - Prima o poi ogni processo corretto (i.e. non in crash) prenderà una decisione

# Proprietà - programmi sequenziali

- **Nei programmi sequenziali:**
  - le proprietà di *safety* esprimono la correttezza dello stato finale (il risultato è quello voluto)
  - la principale proprietà di *liveness* è la terminazione
- **Quali dovrebbero essere le proprietà comuni a tutti i programmi concorrenti?**

# Proprietà - programmi concorrenti

- Proprietà di **safety**
  - i processi non devono "*interferire*" fra di loro nell'accesso alle risorse condivise
  - questo vale ovviamente per i processi che condividono risorse (non per processi che cooperano tramite comunicazione)
- I meccanismi di sincronizzazione servono a garantire la proprietà di safety
  - devono essere usati propriamente dal programmatore, altrimenti il programma potrà contenere delle race condition

# Proprietà - programmi concorrenti

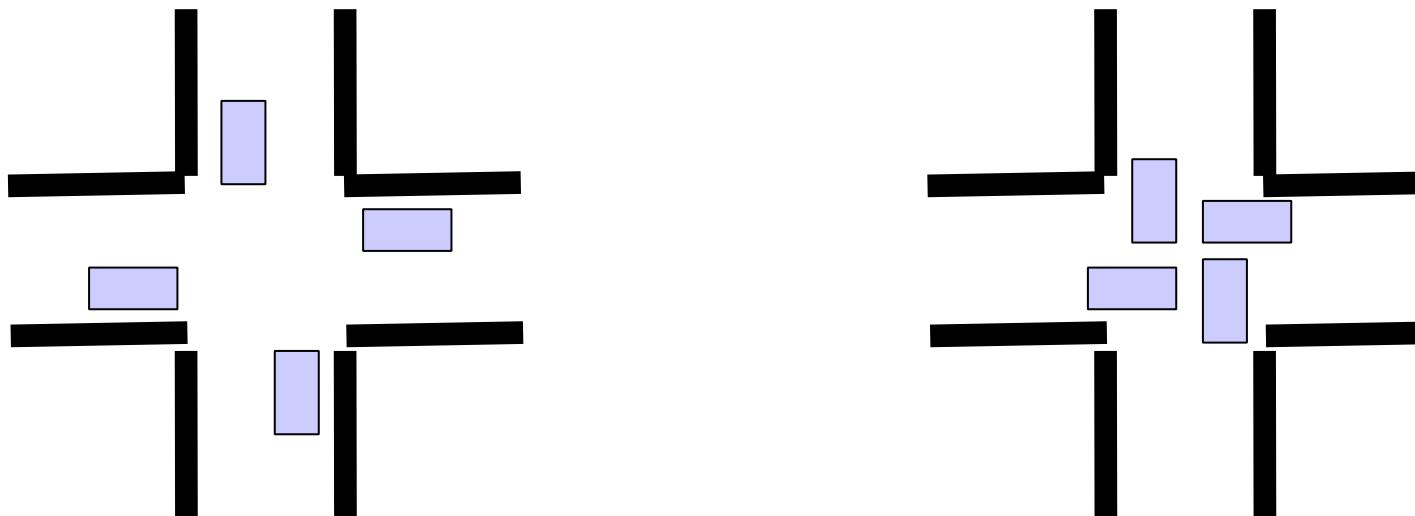
- Proprietà di *liveness*
  - i meccanismi di sincronizzazione utilizzati non devono prevenire l'avanzamento del programma
    - non è possibile che *tutti* i processi si "*blocchino*", in attesa di eventi che non possono verificarsi perché generabili solo da altri processi bloccati
    - non è possibile che *un* processo debba "*attendere indefinitamente*" prima di poter accedere ad una risorsa condivisa
- Nota:
  - queste sono solo descrizioni informali; nei prossimi lucidi saremo più precisi

# Mutua esclusione (safety)

- **Definizione**
  - l'accesso ad una risorsa si dice *mutualmente esclusivo* se ad ogni istante, al massimo un processo può accedere a quella risorsa
- **Esempi da considerare:**
  - due processi che vogliono accedere contemporaneamente a una stampante
  - due processi che cooperano scambiandosi informazioni tramite un buffer condiviso

# Deadlock (stallo) (liveness)

- **Considerazioni:**
  - la mutua esclusione permette di risolvere il problema della non interferenza
  - ma può causare il blocco permanente dei processi
- **Esempio: incrocio stradale**



# Deadlock (stallo)

- **Esempio:**
  - siano  $R_1$  e  $R_2$  due risorse
  - siano  $P_1$  e  $P_2$  due processi che devono accedere a  $R_1$  e  $R_2$  contemporaneamente, prima di poter terminare il programma
  - supponiamo che il S.O. assegni  $R_1$  a  $P_1$ , e  $R_2$  a  $P_2$
  - i due processi sono bloccati in attesa circolare
- **Si dice che  $P_1$  e  $P_2$  sono in deadlock**
  - è una condizione da evitare
  - è definitiva
  - nei sistemi reali, se ne può uscire solo con metodi "distruttivi", ovvero uccidendo i processi, riavviando la macchina, etc.

# **Starvation (inedia) (liveness)**

- **Considerazioni:**
  - il deadlock è un problema che coinvolge tutti i processi che utilizzano un certo insieme di risorse
  - esiste anche la possibilità che un processo non possa accedere ad un risorsa perché "sempre occupata"
- **Esempio**
  - se siete in coda ad uno sportello e continuano ad arrivare "furbi" che passano davanti, non riuscirete mai a parlare con l'impiegato/a

# Starvation (inedia)

- **Esempio**
  - sia **R** una risorsa
  - siano **P<sub>1</sub>**, **P<sub>2</sub>**, **P<sub>3</sub>** tre processi che accedono periodicamente a **R**
  - supponiamo che **P<sub>1</sub>** e **P<sub>2</sub>** si alternino nell'uso della risorsa
  - **P<sub>3</sub>** non può accedere alla risorsa, perché utilizzata in modo esclusivo da **P<sub>1</sub>** e **P<sub>2</sub>**
- **Si dice che P<sub>3</sub> è in *starvation***
  - a differenza del deadlock, non è una condizione definitiva
  - è possibile uscirne, basta adottare un'opportuna politica di assegnamento
  - è comunque una situazione da evitare

# Riassunto

Tipo	Relazione	Meccanismo	Problemi di controllo
processi "ignari" uno dell'altro	competizione	sincronizzazione	mutua esclusione deadlock starvation
processi con conoscenza indiretta l'uno dell'altro	cooperazione (sharing)	sincronizzazione	mutua esclusione deadlock starvation
processi con conoscenza diretta l'uno dell'altro	cooperazione (comunicazione)	comunicazione	deadlock starvation

# **Sezione 1**

---

## **1. Scheduler, processi e thread**

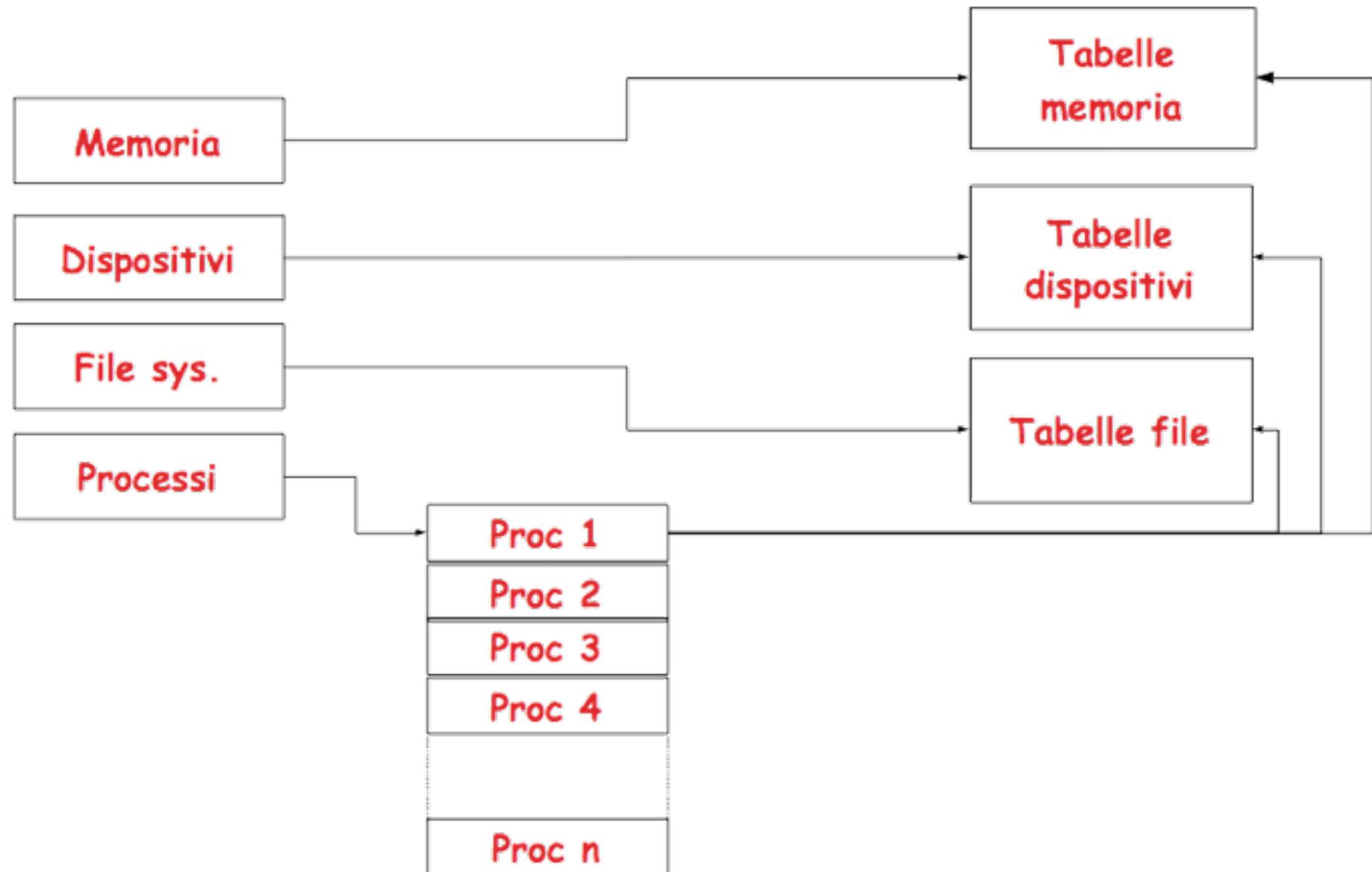
# Introduzione

- **Un sistema operativo è un gestore di risorse**
  - processore, memoria principale e secondaria, dispositivi
- **Per svolgere i suoi compiti, un sistema operativo ha bisogno di strutture dati per mantenere informazioni sulle risorse gestite**
- **Queste strutture dati comprendono:**
  - tabelle di memoria
  - tabelle di I/O
  - tabelle del file system
  - tabelle dei processi

# Introduzione

- **Tabelle per la gestione della memoria**
  - allocazione memoria per il sistema operativo
  - allocazione memoria principale e secondaria per i processi
  - informazioni per i meccanismi di protezione
- **Tabelle per la gestione dell'I/O**
  - informazioni sullo stato di assegnazione dei dispositivi utilizzati dalla macchina
  - gestione di code di richieste
- **Tabelle per la gestione del file system**
  - elenco dei dispositivi utilizzati per mantenere il file system
  - elenco dei file aperti e loro stato

# Introduzione



# Descrittori dei processi

- Qual é la manifestazione fisica di un processo?
  1. il codice da eseguire (**segmento codice**)
  2. i dati su cui operare (**segmenti dati**)
  3. uno stack di lavoro per la gestione di chiamate di funzione, passaggio di parametri e variabili locali
  4. un *insieme di attributi* contenenti tutte le informazioni necessarie per la gestione del processo stesso
    - incluse le informazioni necessarie per descrivere i punti 1-3
- Questo insieme di attributi prende il nome di **descrittore del processo (Process Control Block)**

# Descrittori dei processi

- **Tabella per la gestione dei processi**
  - contiene i descrittori dei processi
  - ogni processo ha un descrittore associato
- **E' possibile suddividere le informazioni contenute nel descrittore in tre aree:**
  1. informazioni di **identificazione** di processo
  2. informazioni di **stato** del processo
  3. informazioni di **controllo** del processo

# Descrittori dei processi

- **Informazioni di identificazione di un processo**
  - *identificatore di processo* (*process id*, o *pid*)
    - può essere semplicemente un *indice* all'interno di una tabella di processi
    - può essere un *numero progressivo*; in caso, è necessario un mapping tra pid e posizione del relativo descrittore
    - molte altre tabelle del s.o. utilizzano il process id per identificare un elemento della tabella dei processi
  - *identificatori di altri processi logicamente collegati* al processo
    - ad esempio, pid del processo padre
  - *id dell'utente* che ha richiesto l'esecuzione del processo

# Descrittori dei processi

- **Informazioni di stato del processo**
  - *registri generali del processore*
  - *registri speciali, come il program counter e i registri di stato*
- **Informazioni di controllo del processo**
  - *Informazioni di scheduling*
    - stato del processo
      - in esecuzione, pronto, in attesa
    - informazioni particolari necessarie dal particolare algoritmo di scheduling utilizzato
      - priorità, puntatori per la gestione delle code
    - identificatore dell'evento per cui il processo è in attesa

# Descrittori dei processi

- **Informazioni di controllo del processo (continua)**
  - informazioni di gestione della memoria
    - valori dei registri base e limite dei segmenti utilizzati, puntatori alle tabelle delle pagine, etc.
  - informazioni di accounting
    - tempo di esecuzione del processo
    - tempo trascorso dall'attivazione di un processo
  - informazioni relative alle risorse
    - risorse controllate dal processo, come file aperti, device allocati al processo
  - informazioni per interprocess communication (IPC)
    - stato di segnali, semafori, etc.

# Scheduler

- E' la componente più importante del kernel
- Gestisce l'avvicendamento dei processi
  - decide quale processo deve essere in esecuzione ad ogni istante
  - interviene quando viene richiesta un'operazione di I/O e quando un'operazione di I/O termina, ma anche periodicamente
- NB
  - Il termine "scheduler" viene utilizzato anche in altri ambiti con il significato di "gestore dell'avvicendamento del controllo"
  - possiamo quindi fare riferimento allo "scheduler del disco", e in generale allo "scheduler del dispositivo X"

# **Schedule, scheduling, scheduler**

---

- **Schedule**

- è la sequenza temporale di assegnazioni delle risorse da gestire ai richiedenti

- **Scheduling**

- è l'azione di calcolare uno schedule

- **Scheduler**

- è la componente software che calcola lo schedule

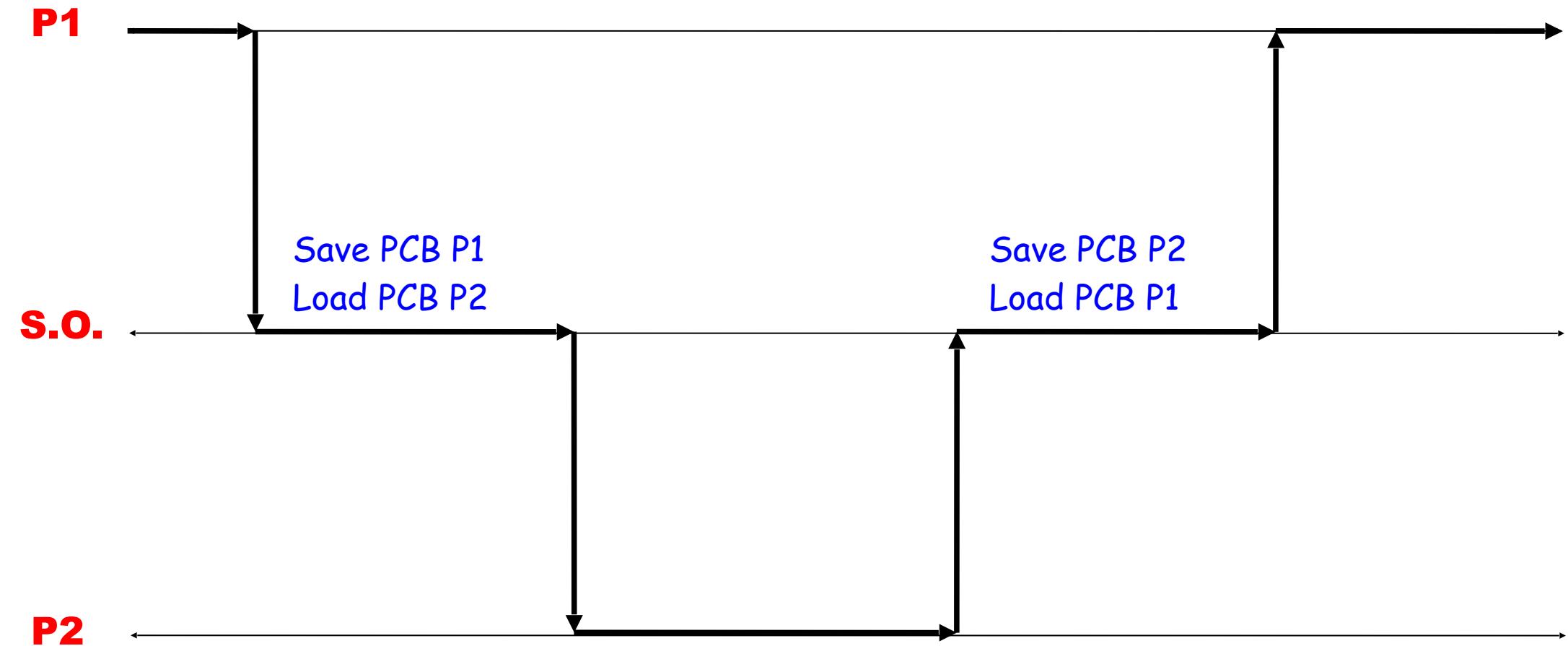
# Mode switching e context switching

- **Modalità utente e supervisore**
  - tutte le volte che avviene un interrupt (software o hardware) il processore è soggetto ad un *mode switching*
    - modalità utente → modalità supervisore
- **Durante la gestione dell'interrupt**
  - vengono intraprese le opportune azioni per gestire l'evento
  - viene chiamato lo scheduler
  - se lo scheduler decide di eseguire un altro processo, il sistema è soggetto ad un *context switching*

# Context switching

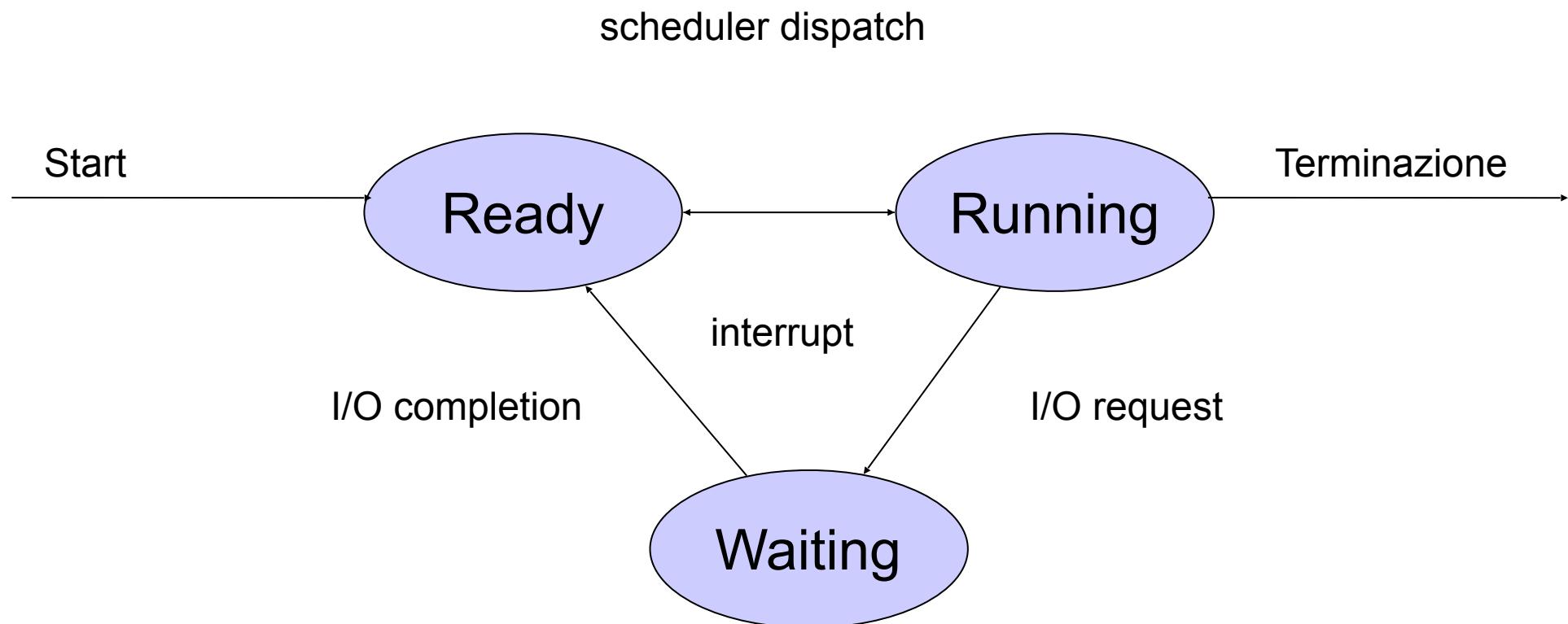
- Operazioni durante un context switching
  - lo stato del processo attuale viene salvato nel PCB corrispondente
  - lo stato del processo selezionato per l'esecuzione viene caricato dal PCB nel processore

# Context switch



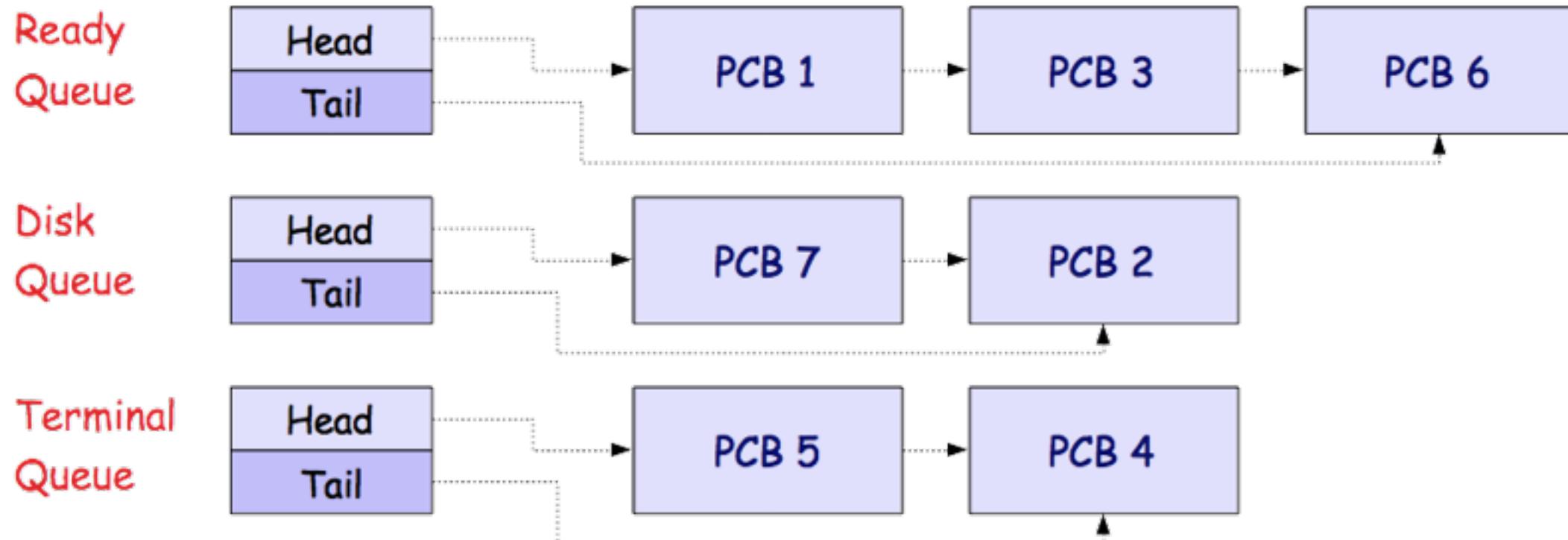
# Vita di un processo

- **Stati dei processi:**
  - *Running*: il processo è in esecuzione
  - *Waiting*: il processo è in attesa di qualche evento esterno (e.g., completamento operazione di I/O); non può essere eseguito
  - *Ready*: il processo può essere eseguito, ma attualmente il processore è impegnato in altre attività

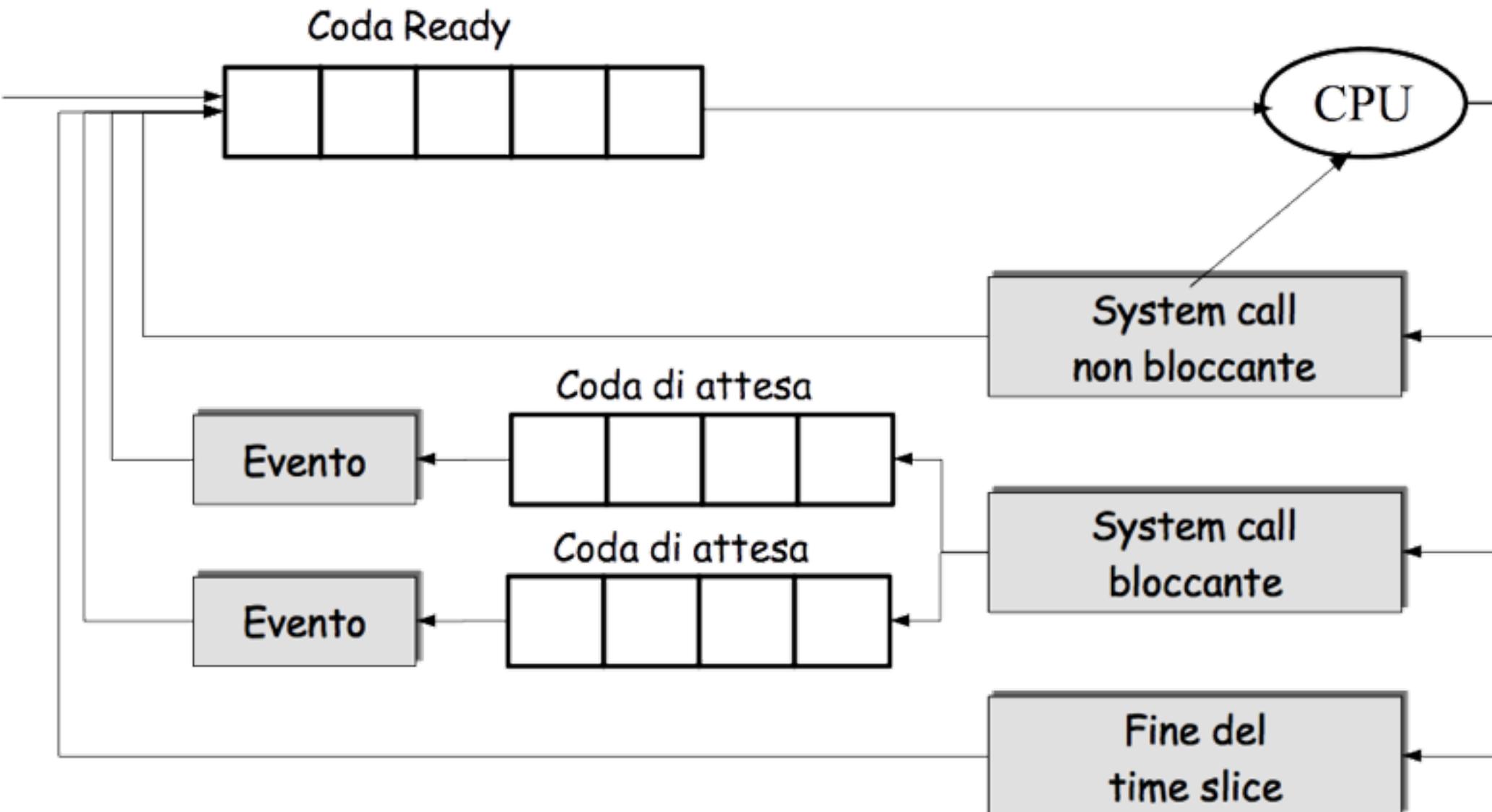


# Code di processi

- Tutte le volte che un processo entra nel sistema, viene posto in una delle code gestite dallo scheduler



# Vita di un processo nello scheduler



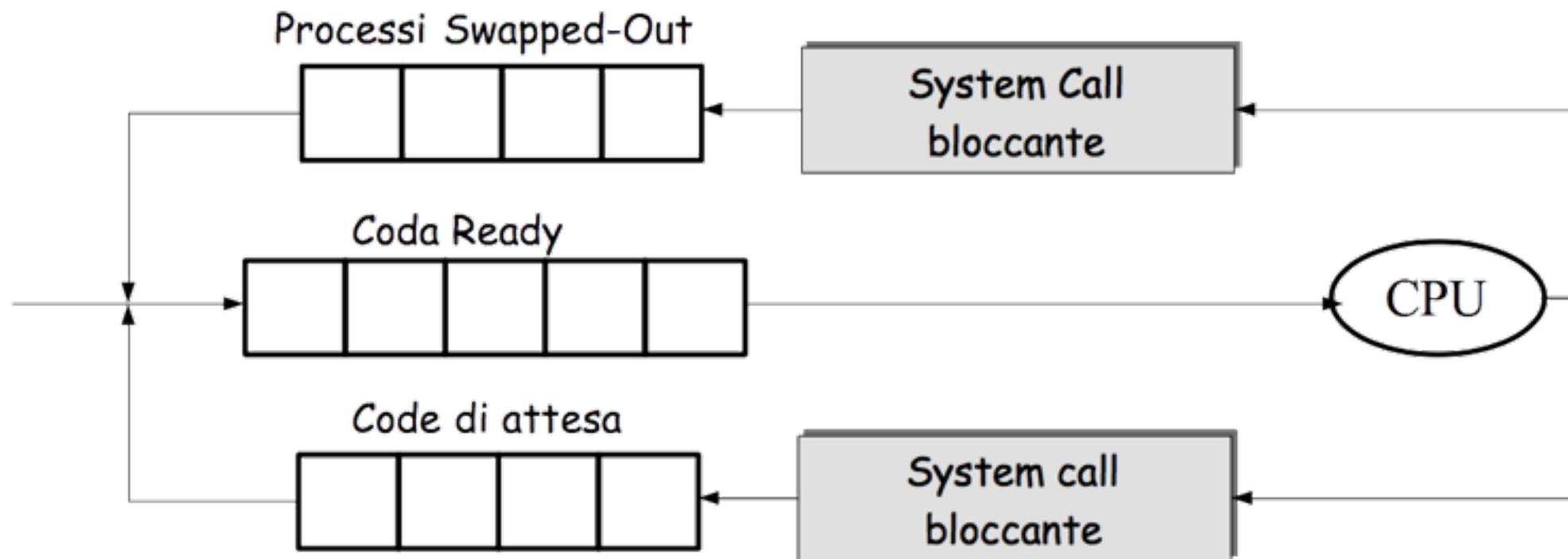
# Classi di scheduler

- **Long-term scheduler**
  - seleziona quali processi creare fra quelli che non hanno ancora iniziato la loro esecuzione
  - viene utilizzato per programmi batch
  - nei sistemi interattivi (UNIX), non appena un programma viene lanciato il processo relativo viene automaticamente creato
- **Short-term scheduler (o scheduler di CPU, o scheduler tout-court)**
  - seleziona quale dei processi pronti all'esecuzione deve essere eseguito, ovvero a quale assegnare il controllo della CPU

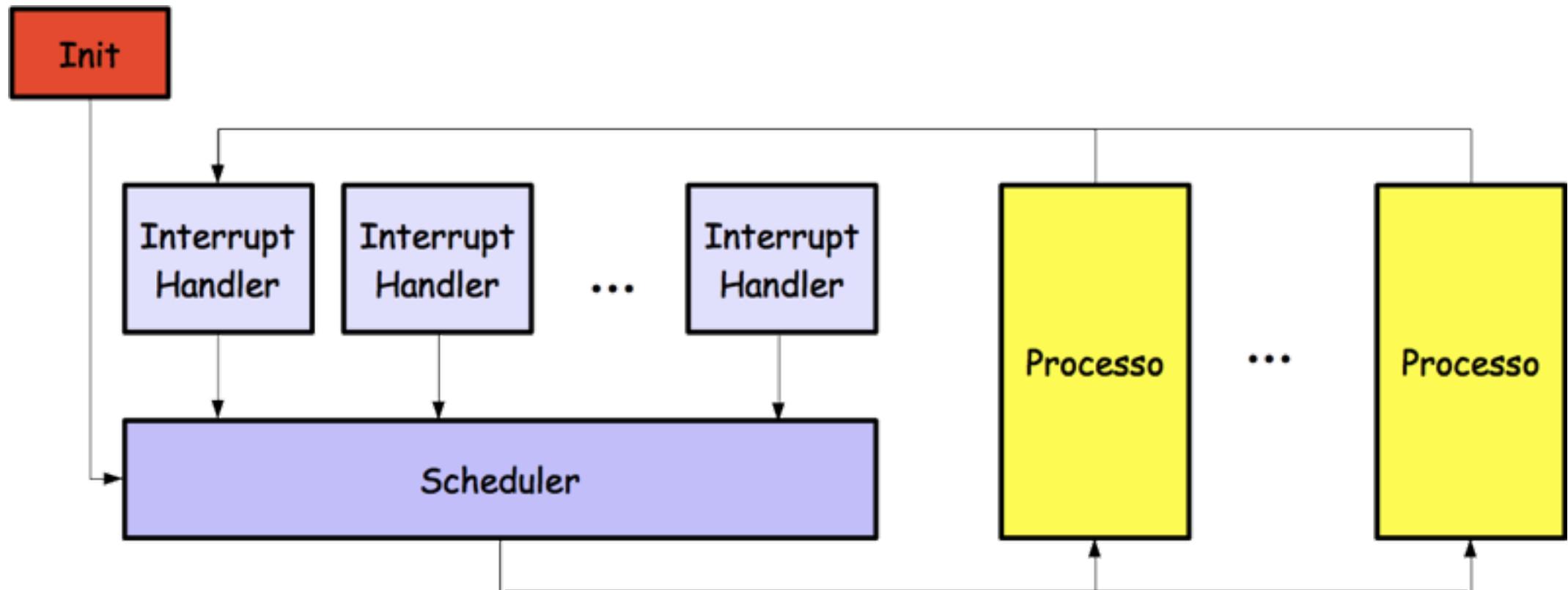
# Classi di scheduler

- **Mid-term scheduler**

- E' anche possibile definire uno scheduler a medio termine che gestisca i processi bloccati per lunghe attese
- per questi processi l'immagine della memoria può venire copiata su disco al fine di ottimizzare l'uso della memoria centrale



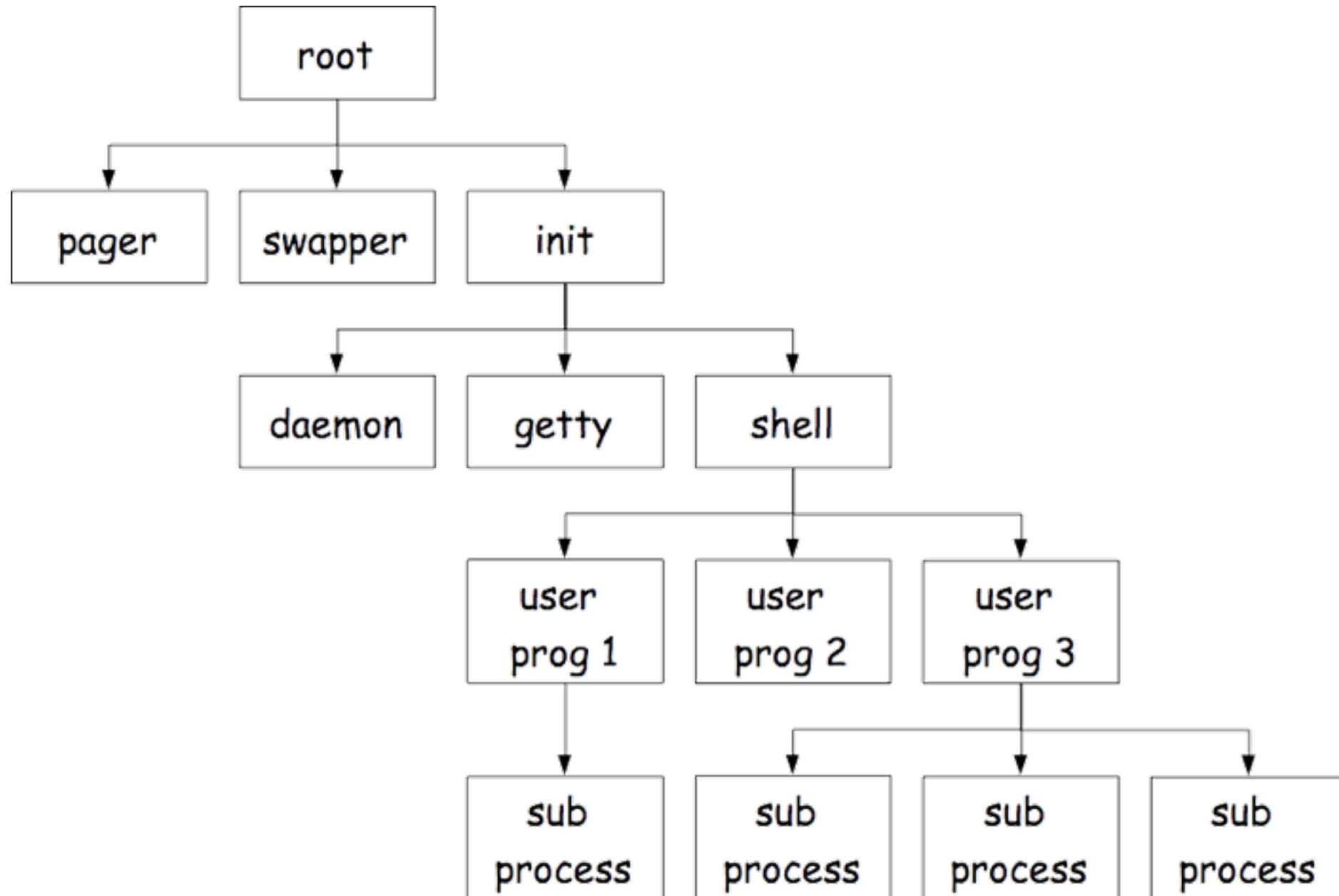
# Schema di funzionamento di un kernel



# Gerarchia di processi

- **Nella maggior parte dei sistemi operativi**
  - i processi sono organizzati in forma gerarchica
  - quando un processo crea un nuovo processo, il processo creante viene detto padre e il creato figlio
  - si viene così a creare un albero di processi
- **Motivazioni**
  - semplificazione del procedimento di creazione di processi
    - non occorre specificare esplicitamente tutti i parametri e le caratteristiche
    - ciò che non viene specificato, viene ereditato dal padre

# Gerarchia di processi: UNIX



# Processi e Thread

- La nozione di processo discussa in precedenza assume che ogni processo abbia una singola “linea di controllo”
  - per ogni processo, viene eseguite una singola sequenza di istruzioni
  - un singolo processo non può eseguire due differenti attività contemporaneamente
- Esempi:
  - scaricamento di due differenti pagine in un web browser
  - inserimento di nuovo testo in un wordprocessor mentre viene eseguito il correttore ortografico

# Processi e Thread

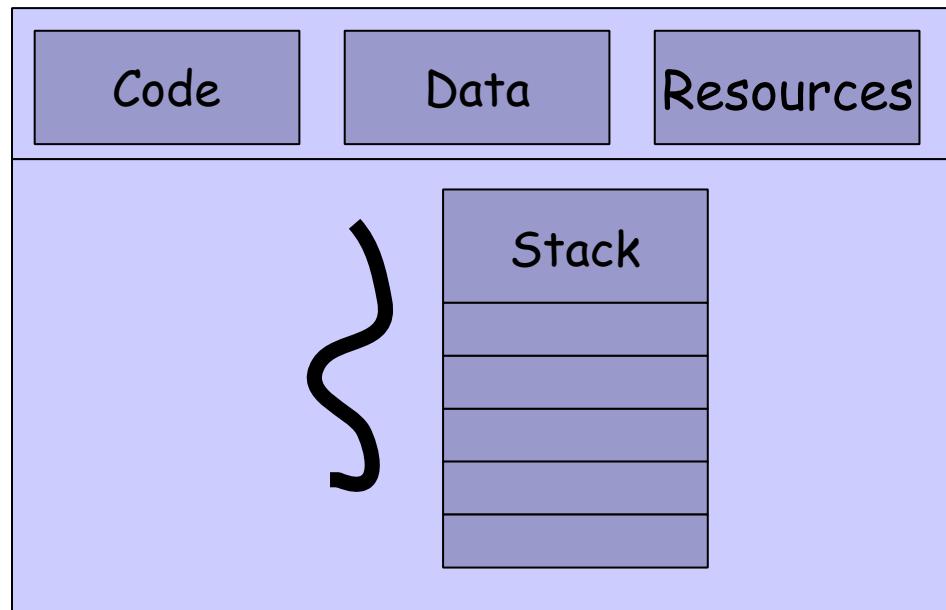
- **Tutti i sistemi operativi moderni**
  - supportano l'esistenza di processi multithreaded
  - in un processo multithreaded esistono molte "linee di controllo", ognuna delle quali può eseguire un diverso insieme di istruzioni
- **Esempi:**
  - Associando un thread ad ogni finestra aperta in un web browser, è possibile scaricare i dati in modo indipendente

# Processi e thread

- **Un thread è l'unità base di utilizzazione della CPU**
- **Ogni thread possiede**
  - la propria copia dello stato del processore
  - il proprio program counter
  - uno stack separato
- **I thread appartenenti allo stesso processo condividono:**
  - codice
  - dati
  - risorse di I/O

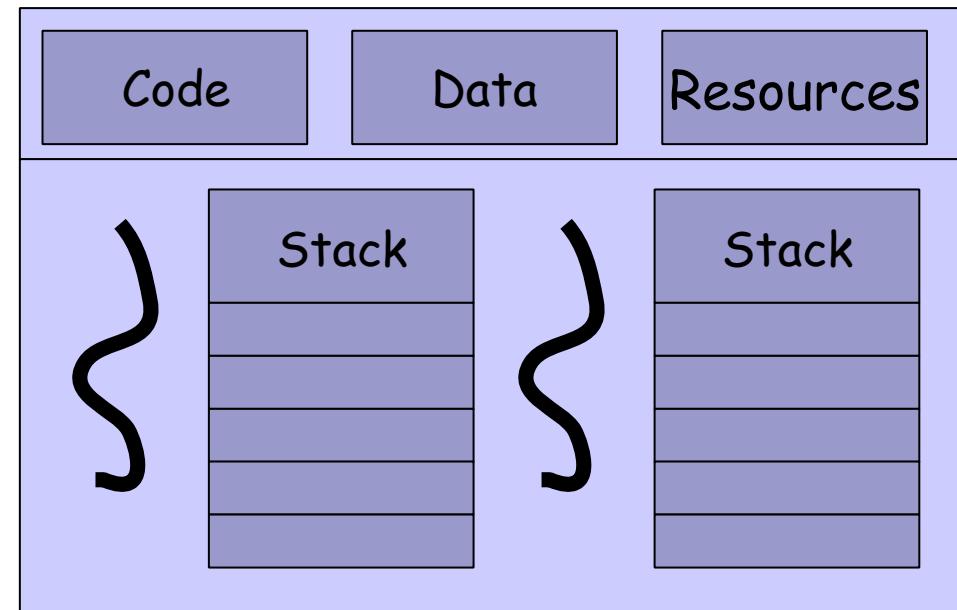
# Processi e thread

Processo



Single threaded

Processo



Multi threaded

# Benefici dei thread

- **Condivisione di risorse**
  - i thread condividono lo spazio di memoria e le risorse allocate degli altri thread dello stesso processo
  - condividere informazioni tra thread logicamente correlati rende più semplice l'implementazione di certe applicazioni
- **Esempio:**
  - web browser: condivisione dei parametri di configurazione fra i vari thread

# Benefici dei thread

- **Economia**
  - allocare memoria e risorse per creare nuovi processi è costoso
  - fare context switching fra diversi processi è costoso
- **Gestire i thread è in generale più economico, quindi preferibile**
  - creare thread all'interno di un processo è meno costoso
  - fare context switching fra thread è meno costoso
- **Esempio:**
  - creare un thread in Solaris richiede 1/30 del tempo richiesto per creare un nuovo processo

# Processi vs Thread

- **Thread = processi "lightweight"**
  - utilizzare i thread al posto dei processi rende l'implementazione più efficiente
  - in ogni caso, abbiamo bisogno di processi distinti per applicazioni differenti

# Multithreading: implementazione

- Un sistema operativo può implementare i thread in due modi:
  - User thread  
(A livello utente)
  - Kernel thread  
(A livello kernel)

# User thread

- Gli user thread vengono supportati sopra il kernel e vengono implementati da una "thread library" a livello utente
  - la thread library fornisce supporto per la creazione, lo scheduling e la gestione dei thread senza alcun intervento del kernel
- Vantaggi:
  - l'implementazione risultante è molto efficiente
- Svantaggi:
  - se il kernel è single-threaded, qualsiasi user thread che effettua una chiamata di sistema bloccante (che si pone in attesa di I/O) causa il blocco dell'intero processo

# Kernel thread

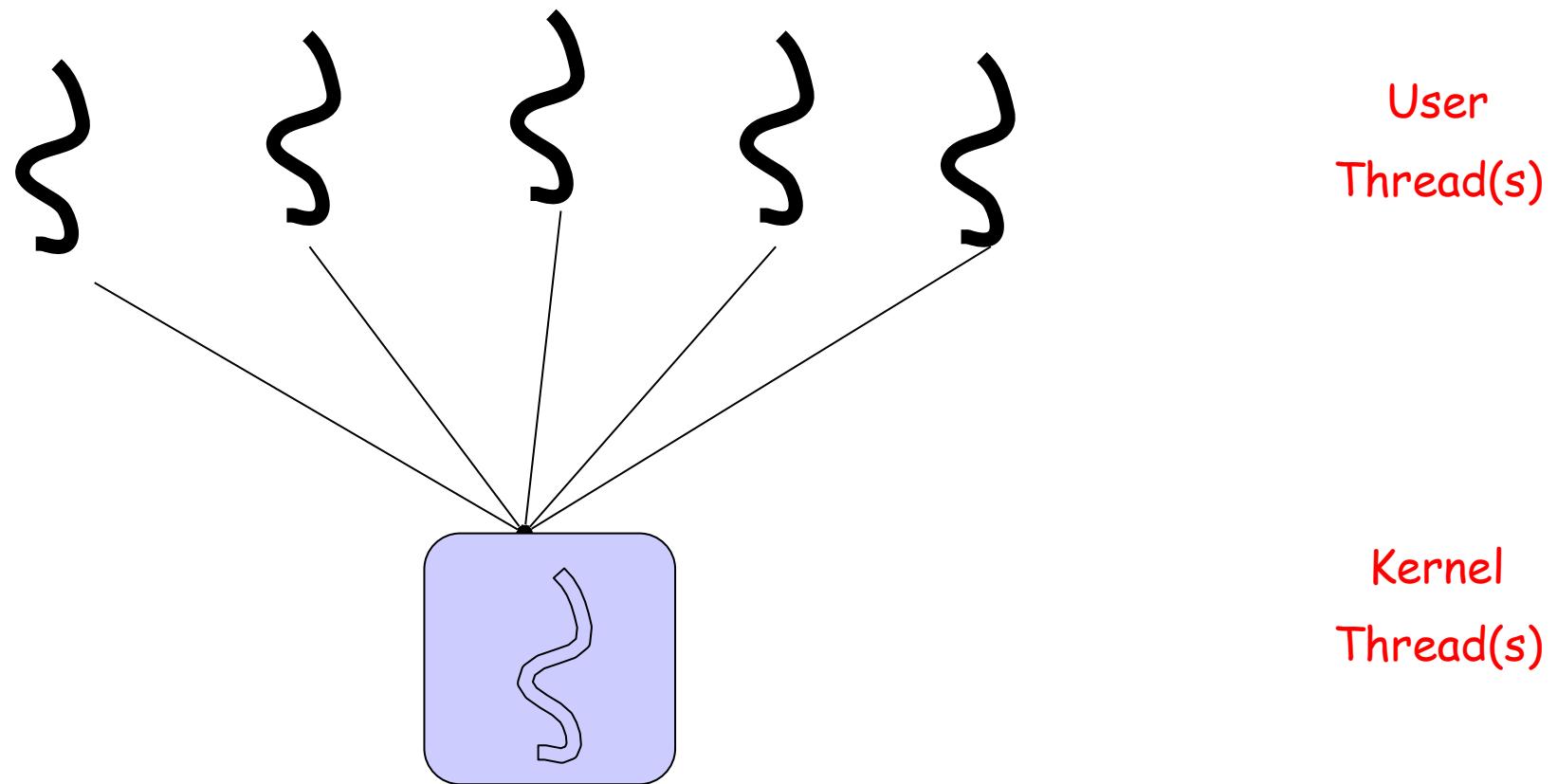
- I kernel thread vengono supportati direttamente dal sistema operativo
  - la creazione, lo scheduling e la gestione dei thread sono implementati a livello kernel
- Vantaggi:
  - poichè è il kernel a gestire lo scheduling dei thread, se un thread esegue una operazione di I/O, il kernel può selezionare un altro thread in attesa di essere eseguito
- Svantaggi:
  - l'implementazione risultante è più lenta, perché richiede un passaggio da livello utente a livello supervisore

# Modelli di multithreading

- Molti sistemi supportano sia kernel thread che user thread
- Si vengono così a creare tre differenti modelli di multithreading:
  - Many-to-One
  - One-to-One
  - Many-to-Many

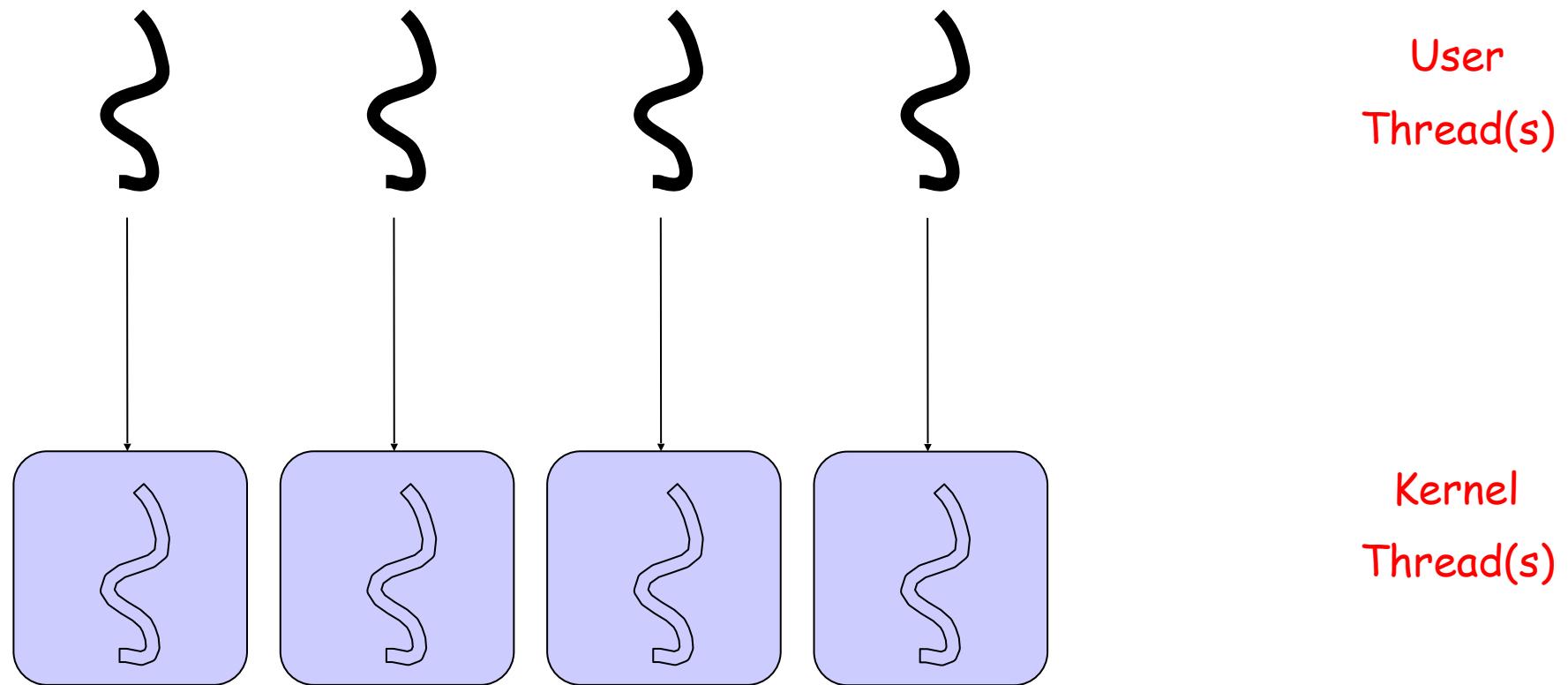
# Many-to-One Multithreading

- ◆ Un certo numero di user thread vengono mappati su un solo kernel thread
- ◆ Modello generalmente adottato da s.o. che non supportano kernel thread multipli



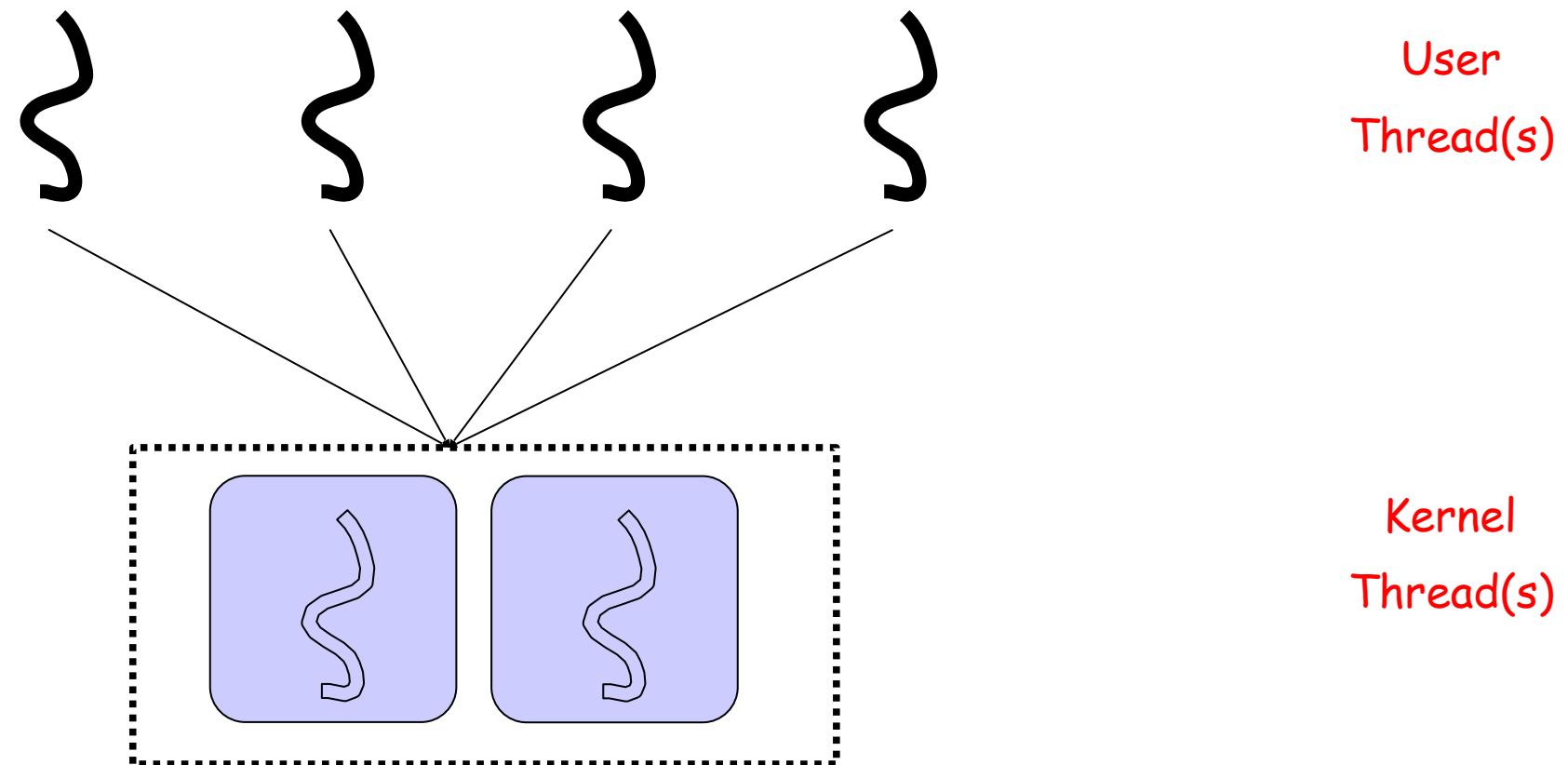
# One-to-One Multithreading

- ◆ Ogni user thread viene mappato su un kernel thread
- ◆ Può creare problemi di scalabilità per il kernel
- ◆ Supportato da Windows 95, OS/2



# Many-to-Many Multithreading

- ◆ Riassume i benefici di entrambe le architetture
- ◆ Supportato da Solaris, IRIX, Digital Unix



## **Sezione 2**

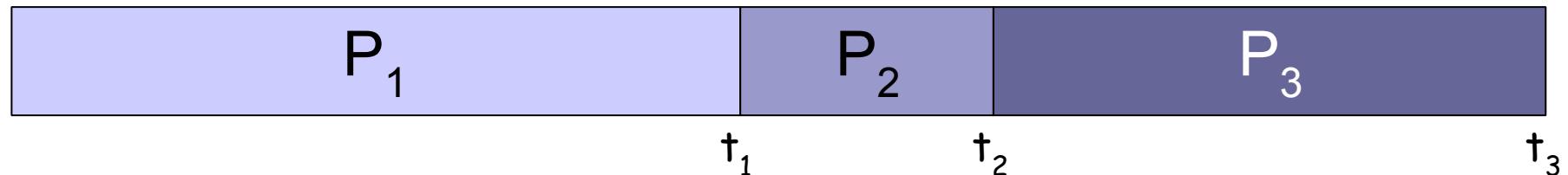
---

### **2. Scheduling**

# Rappresentazione degli schedule

- **Diagramma di Gantt**

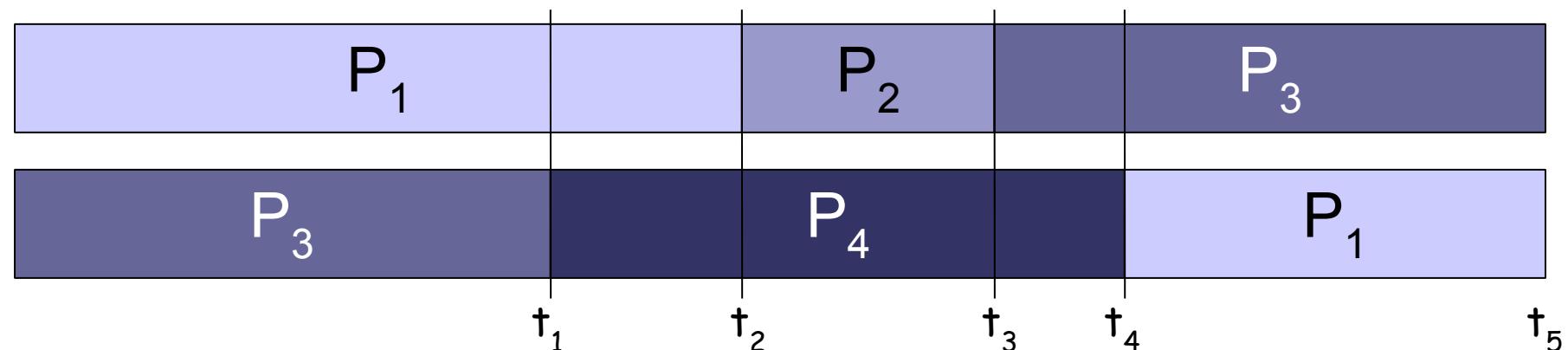
- per rappresentare uno schedule si usano i diagrammi di Gantt



- in questo esempio, la risorsa (es. CPU) viene utilizzata dal processo  $P_1$  dal tempo 0 a  $t_1$ , viene quindi assegnata a  $P_2$  fino al tempo  $t_2$  e quindi a  $P_3$  fino al tempo  $t_3$

# Rappresentazione degli schedule

- **Diagramma di Gantt multi-risorsa**
  - nel caso si debba rappresentare lo schedule di più risorse (e.g., un sistema multiprocessore) il diagramma di Gantt risulta composto da più righe parallele



# Tipi di scheduler

- **Eventi che possono causare un context switch**
  1. quando un processo passa da stato running a stato waiting  
(system call bloccante, operazione di I/O)
  2. quando un processo passa dallo stato running allo stato ready  
(a causa di un interrupt)
  3. quando un processo passa dallo stato waiting allo stato ready
  4. quando un processo termina
- **Nota:**
  - nelle condizioni 1 e 4, l'unica scelta possibile è quella di selezionare un altro processo per l'esecuzione
  - nelle condizioni 2 e 3, è possibile continuare ad eseguire il processo corrente

# Tipi di scheduler

- Uno scheduler si dice **non-preemptive o cooperativo**
  - se i context switch avvengono solo nelle condizioni 1 e 4
  - in altre parole: *il controllo della risorsa viene trasferito solo se l'assegnatario attuale lo cede volontariamente*
  - Windows 3.1, Mac OS y con  $y < 8$
- Uno scheduler si dice **preemptive** se
  - se i context switch possono avvenire in ogni condizione
  - in altre parole: *è possibile che il controllo della risorsa venga tolto all'assegnatario attuale a causa di un evento*
  - tutti gli scheduler moderni

# Tipi di scheduler

- **Vantaggi dello scheduling cooperativo**
  - non richiede alcuni meccanismi hardware come ad esempio timer programmabili
- **Vantaggi dello scheduling preemptive**
  - permette di utilizzare al meglio le risorse

# Criteri di scelta di uno scheduler

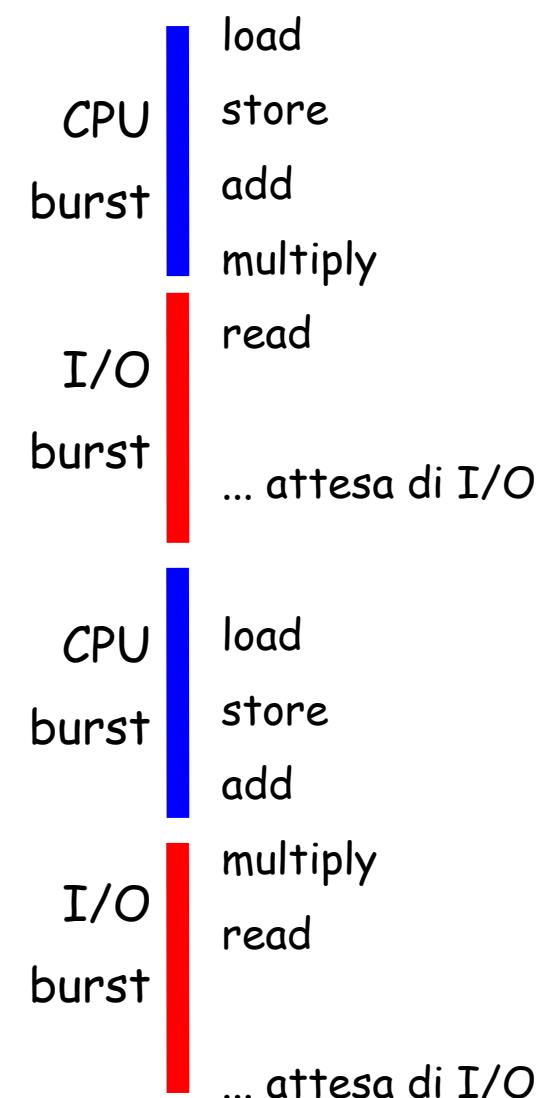
- **Utilizzo della risorsa (CPU)**
  - percentuale di tempo in cui la CPU è occupata ad eseguire processi
  - deve essere massimizzato
- **Throughput**
  - numero di processi completati per unità di tempo
  - dipende dalla lunghezza dei processi
  - deve essere massimizzato
- **Tempo di turnaround**
  - tempo che intercorre dalla sottomissione di un processo alla sua terminazione
  - deve essere minimizzato

# Criteri di scelta di uno scheduler

- **Tempo di attesa**
  - il tempo trascorso da un processo nella coda ready
  - deve essere minimizzato
- **Tempo di risposta**
  - tempo che intercorre fra la sottomissione di un processo e il tempo di prima risposta
  - particolarmente significativo nei programmi interattivi, deve essere minimizzato

# Caratteristiche dei processi

- **Durante l'esecuzione di un processo:**
  - si alternano periodi di attività svolte dalla CPU (*CPU burst*)...
  - ...e periodi di attività di I/O (*I/O burst*)
- **I processi:**
  - caratterizzati da CPU burst molto lunghi si dicono *CPU bound*
  - caratterizzati da I/O burst molto lunghi si dicono *I/O bound*



# Scheduling

- **Algoritmi:**

- First Come, First Served
- Shortest-Job First
  - Shortest-Next-CPU-Burst First
  - Shortest-Remaining-Time-First
- Round-Robin

# First Come, First Served (FCFS)

- **Algoritmo**
  - il processo che arriva per primo, viene servito per primo
  - politica senza preemption
- **Implementazione**
  - semplice, tramite una coda (politica FIFO)
- **Problemi**
  - elevati tempi medi di attesa e di turnaround
  - processi CPU bound ritardano i processi I/O bound

# First Come, First Served (FCFS)

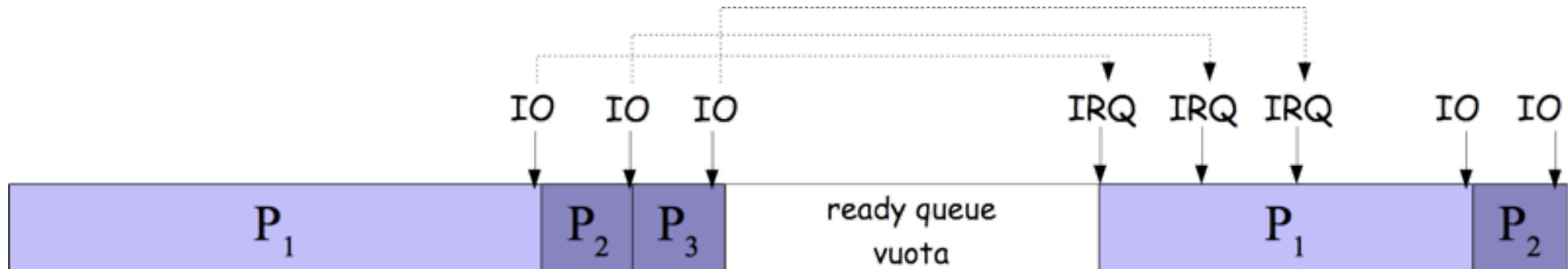
- **Esempio:**

- ordine di arrivo:  $P_1, P_2, P_3$
- lunghezza dei CPU-burst in ms: 32, 2, 2
- Tempo medio di turnaround:  $(32+34+36)/3 = 34 \text{ ms}$
- Tempo medio di attesa:  $(0+32+34)/3 = 22 \text{ ms}$



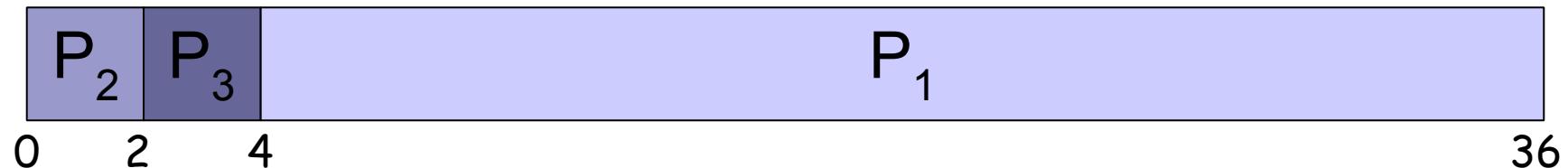
# First Come, First Served (FCFS)

- Supponiamo di avere
  - un processo CPU bound
  - un certo numero di processi I/O bound
  - i processi I/O bound si "mettono in coda" dietro al processo CPU bound, e in alcuni casi la ready queue si può svuotare
  - *Convoy effect*



# Shortest Job First (SJF)

- **Algoritmo (Shortest Next CPU Burst First)**
  - la CPU viene assegnata al processo ready che ha la minima durata del CPU burst successivo
  - politica senza preemption
- **Esempio**
  - Tempo medio di turnaround:  $(0+2+4+36)/3 = 7 \text{ ms}$
  - Tempo medio di attesa:  $(0+2+4)/3 = 2 \text{ ms}$



# Shortest Job First (SJF)

- L'algoritmo SJF
  - è *ottimale* rispetto al tempo di attesa, in quanto è possibile dimostrare che produce il minor tempo di attesa possibile
  - ma è *impossibile da implementare* in pratica!
  - è possibile solo fornire delle *approssimazioni*
- Negli scheduler long-term
  - possiamo chiedere a chi sottomette un job di predire la durata del job
- Negli scheduler short-term
  - non possiamo conoscere la lunghezza del *prossimo* CPU burst.... ma conosciamo la lunghezza di quelli *precedenti*

# Shortest Job First (SJF)

- **Calcolo approssimato della durata del CPU burst**
  - basata su *media esponenziale* dei CPU burst precedenti
  - sia  $t_n$  il tempo dell'n-esimo CPU burst e  $\tau_n$  la corrispondente previsione;  $\tau_{n+1}$  può essere calcolato come segue:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

- **Media esponenziale**
  - svolgendo la formula di ricorrenza, si ottiene

$$\tau_{n+1} = \sum_{j=0..n} (1-\alpha)^j t_{n-j} + (1-\alpha)^{n+1} \tau_0$$

da cui il nome media esponenziale

# **Shortest Job First (SJF)**

- **Spiegazione**
  - $t_n$  rappresenta la storia recente
  - $T_n$  rappresenta la storia passata
  - $\alpha$  rappresenta il peso relativo di storia passata e recente
  - cosa succede con  $\alpha = 0, 1$  oppure  $\frac{1}{2}$ ?
- **Nota importante:**
  - SJF può essere soggetto a starvation!

# Shortest Job First (SJF)

- Shortest Job First "approssimato" esiste in due versioni:
  - *non preemptive*
    - il processo corrente esegue fino al completamento del suo CPU burst
  - *preemptive*
    - il processo corrente può essere messo nella coda ready, se arriva un processo con un CPU burst più breve di quanto rimane da eseguire al processo corrente
    - "Shortest-Remaining-Time First"

# Scheduling Round-Robin

- E' basato sul concetto di quanto di tempo (o time slice)
  - un processo non può rimanere in esecuzione per un tempo superiore alla durata del quanto di tempo
- Implementazione (1)
  - l'insieme dei processi pronti è organizzato come una coda
  - due possibilità:
    - un processo può lasciare il processore *volontariamente*, in seguito ad un'operazione di I/O
    - un processo può *esaurire il suo quanto di tempo* senza completare il suo CPU burst, nel qual caso viene aggiunto in fondo alla coda dei processi pronti
  - in entrambi i casi, il prossimo processo da eseguire è il primo della coda dei processi pronti

# Scheduling Round-Robin

- **La durata del quanto di tempo è un parametro critico del sistema**
  - se il quanto di tempo è breve, il sistema è meno efficiente perché deve cambiare il processo attivo più spesso
  - se il quanto è lungo, in presenza di numerosi processi pronti ci sono lunghi periodi di inattività di ogni singolo processo,
    - in sistemi interattivi, questo può essere fastidioso per gli utenti

# Scheduling Round-Robin

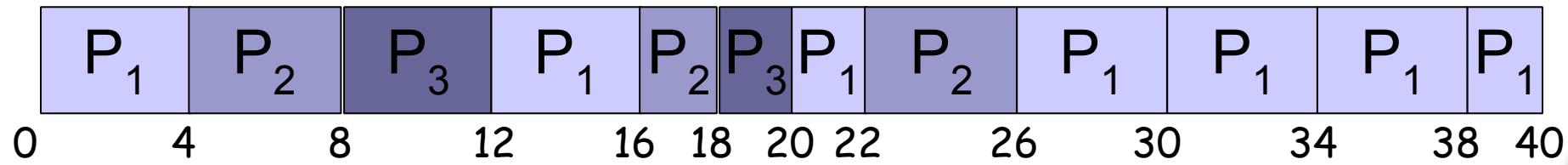
- **Implementazione (2)**

- è necessario che l'hardware fornisca un timer (interval timer) che agisca come "sveglia" del processore
- il timer è un dispositivo che, attivato con un preciso valore di tempo, è in grado di fornire un interrupt allo scadere del tempo prefissato
- il timer viene interfacciato come se fosse un'unità di I/O

# Scheduling Round-Robin

## ◆ Esempio

- tre processi  $P_1$ ,  $P_2$ ,  $P_3$
  - lunghezza dei CPU-burst in ms ( $P_1$ : 10+14;  $P_2$ : 6+4;  $P_3$ : 6)
  - lunghezza del quanto di tempo: 4
  - Tempo medio di turnaround  $(40+26+20)/3 = 28.66$  ms
  - Tempo medio di attesa:  $(16+16+14)/3 = 15.33$  ms
    - NB (supponiamo attese di I/O brevi, < 2ms)
  - Tempo medio di risposta: 4 ms



# Scheduling a priorità

- **Round-robin**
  - fornisce le stesse possibilità di esecuzione a tutti i processi
- **Ma i processi non sono tutti uguali:**
  - usando round-robin puro la visualizzazione dei un video MPEG potrebbe essere ritardata da un processo che sta smistando la posta
  - la lettera può aspettare  $\frac{1}{2}$  sec, il frame video NO

# Scheduling a priorità

- **Descrizione**
  - ogni processo è associato una specifica priorità
  - lo scheduler sceglie il processo pronto con priorità più alta
- **Le priorità possono essere:**
  - *definite dal sistema operativo*
    - vengono utilizzate una o più quantità misurabili per calcolare la priorità di un processo
    - esempio: SJF è un sistema basato su priorità
  - *definite esternamente*
    - le priorità non vengono definite dal sistema operativo, ma vengono imposte dal livello utente

# Scheduling a priorità

- **Priorità statica**
  - la priorità non cambia durante la vita di un processo
  - problema: processi a bassa priorità possono essere posti in *starvation* da processi ad alta priorità
  - Leggenda metropolitana: IBM 7094
- **Priorità dinamica**
  - la priorità può variare durante la vita di un processo
  - è possibile utilizzare metodologie di priorità dinamica per evitare *starvation*

# Scheduling a priorità

- **Priorità basata su aging**
  - tecnica che consiste nell'incrementare gradualmente la priorità dei processi in attesa
  - posto che il range di variazione delle priorità sia limitato, nessun processo rimarrà in attesa per un tempo indefinito perché prima o poi raggiungerà la priorità massima

# Scheduling a classi di priorità

- **Descrizione**

- e' possibile creare diverse classi di processi con caratteristiche simili e assegnare ad ogni classe specifiche priorità
- la coda ready viene quindi scomposta in molteplici "sottocode", una per ogni classe di processi

- **Algoritmo**

- uno scheduler a classi di priorità seleziona il processo da eseguire fra quelli pronti della classe a priorità massima che contiene processi

# Scheduling Multilivello

- **Descrizione**

- all'interno di ogni classe di processi, è possibile utilizzare una politica specifica adatta alle caratteristiche della classe
- uno scheduler multilivello cerca prima la classe di priorità massima che ha almeno un processo ready
- sceglie poi il processo da porre in stato running coerentemente con la politica specifica della classe

# Scheduling Multilivello - Esempio

- **Quattro classi di processi (priorità decrescente)**
  - processi server (priorità statica)
  - processi utente interattivi (round-robin)
  - altri processi utente (FIFO)
  - il processo vuoto (FIFO banale)

# Scheduling Real-Time

- **In un sistema real-time**
  - la correttezza dell'esecuzione non dipende solamente dal valore del risultato, ma anche dall'istante temporale nel quale il risultato viene emesso
- **Hard real-time**
  - le deadline di esecuzione dei programmi non devono essere superate in nessun caso
  - sistemi di controllo nei velivoli, centrali nucleari o per la cura intensiva dei malati
- **Soft real-time**
  - errori occasionali sono tollerabili
  - ricostruzione di segnali audio-video, transazioni interattive

# Scheduling Real-Time

- **Processi periodici**
  - sono periodici i processi che vengono riattivati con una cadenza regolare (periodo)
  - esempi: controllo assetto dei velivoli, basato su rilevazione periodica dei parametri di volo
- **Processi aperiodici**
  - i processi che vengono scatenati da un evento sporadico, ad esempio l'allarme di un rilevatore di pericolo

# Esempi di scheduler Real-Time

- **Rate Monotonic:**

- è una politica di scheduling, valida alle seguenti condizioni
  - ogni processo periodico deve completare entro il suo periodo
  - tutti i processi sono indipendenti
  - la preemption avviene istantaneamente e senza overhead
- viene assegnata staticamente una priorità a ogni processo
- processi con frequenza più alta (i.e. periodo più corto) hanno priorità più alta
- ad ogni istante, viene eseguito il processo con priorità più alta (facendo preemption se necessario)

- **Earliest Deadline First:**

- è una politica di scheduling per processi periodici real-time
- viene scelto di volta in volta il processo che ha la deadline più prossima
- viene detto "a priorità dinamica" perchè la priorità relativa di due processi varia in momenti diversi



```
Current conditions at Pescara, Italy (IBP) 42.26N 014.12E 11M (IBP)
Last updated Feb 10, 2012 - 02:50 PM EST / 2012-02-10 1950 UTC
Temperature: 1 C
Relative Humidity: 80%
Wind: from the W (270 degrees) at 15 MPH (13 KT) gusting to 45 KPH
Weather: light snow grains
Sky conditions: overcast
Su Mo Tu We Th Fr Sa      Su Mo Tu We Th Fr Sa
feb 29e 30 31 01 02 03 04   mar 04 05 06 07 08 09 10
  05 06 07 08 09 10 11    11e 12 13 14 15 16 17
  12 13 14 15 16 17 18    18 19 20 21 22 23 24
  19 20 21 22 23 24 25    25 26 27 28 29 30 31
mar 26 27 28 29 01 02 03   apr 01e 02 03 04 05 06e 07

silvio@Stain ~ $ cd Video
silvio@Stain ~ /Video$ movgrab http://vimeo.com/27998081

Formats available for this Movie: flv
Selected format: flv
Progress: 61.47% 15.4M of 25.1M 693.6K/s
```

