# 7 - Descrizione, Controllo e Schedulazione dei processi

## Modello a due stati

Il compito principale di un sistema operativo è il controllo dell'esecuzione dei processi.
Nel modello a **2 stati** il processo può avere stato di:

- Not running
- Running

Nel primo stato (not running), il processo è caricato in memoria, dunque ha ricevuto una sua locazione e può essere in ready to execute oppure blocked poiché in attesa di I/O.
Tutti i processi caricati in memoria e non in esecuzione vengono inseriti all'interno di una coda con altri processi che si trovano nel loro stesso stato.
I processi passeranno nello stato di Running solamente quando la CPU sarà libera, dunque quando il processo in esecuzione ha terminato il suo task ed è stato deallocato, oppure per varie ragioni rimandato in stato di Not running, sarà poi compito del dispatcher trasferire in base a degli algoritmi specifici che non tengono conto del tempo di attesa il processo da trasferire alla CPU, mandandolo in esecuzione.

## Process Control Block

Il PCB è il descrittore di processo con il quale il sistema operativo può gestirlo, visibile unicamente a lui stesso.
L'**Identificatore del processo** (o Process IDentification, PID) è il valore numerico che descrive quel processo, parte da 0 fino ad estendersi al concetto di "genesi" (tutti i processi a partire dallo 0 generano altri processi). Ogni processo avrà un genitore a sua volta, tenendo traccia del PID del processo genitore.
Il processo contiene:

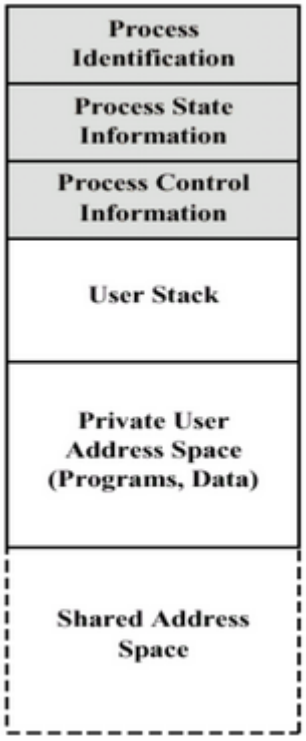
- **Registri di dati visibili all'utente** (dipendenti dall'architettura del calcolatore)
- **Registri di controllo e di stato**
  - PC: indirizzo della prossima istruzione da eseguire
  - Registri di stato: includono i flag per l'abilitazione dell'interrupt
  - Registri che contengono codici di condizione
- **Puntatori allo stack**
  Usato per procedure e funzioni

Per controllare un processo ci sono delle informazioni presenti all'interno di esso:

- **Schedulazione e info di stato**:

    - Stato del processo (running, ready, blocked, arrested etc.)
    - Priorità nelle code di scheduling
    - Info correlate alla schedulazione (tempo di attesa, tempo di esecuzione etc.)
    - Evento (del quale è in attesa)
- **Strutturazione dati**: puntatore ad altri processi (figli/padre o per l'implementazione delle code)
- **Comunicazione tra processi**: flag, segnali, messaggi per la comunicazione tra di loro
- **Privilegi** nell'utilizzo di determinati dispositivi
- **Gestione della memoria**: indirizzo di partenza (base) e indirizzo di fine (limite)
- **Contabilizzazione delle risorse**: la cronologia di utilizzo delle risorse, come file aperti, dispositivi I/O o altre risorse assegnate al processo

# Immagine dei processi in memoria




Ogni processo ha un suo stack utente, che contiene i punti di ritorno e le varie chiamate.
Lo stack di sistema serve per tenere traccia di quale è il punto di ritorno tra i diversi processi, ciò avviene nelle istruzioni macchina.
C'è anche il Shared Address Space, uno spazio di memoria condiviso dove i diversi processi possono leggere o scrivere, questo spazio può crescere dinamicamente, di questa area condivisa quando il processo muore possono accadere due cose:

- Muore definitivamente
- Chiede all'utente cosa vorrebbe farsene

# Creazione e terminazione dei processi

Ci sono possibili eventi che determinano la generazione di un processo:

- **Richiesta da terminale** (un utente accede al sistema e usa un app)
- **Il sistema operativo genera un processo sulla base della richiesta di un processo utente** (un processo prende i dati da un altro programma ed esegue delle azioni in modo indipendente, un esempio è la stampa)
- **Un processo utente genera un nuovo processo** (come il processo padre figlio)

Ci sono eventi anche che fanno terminare un processo:

- **Terminazione normale** (end)
- **Uscita dall'utente dall'applicazione**
- **Superamento del tempo massimo**, se un processo deve restituire dei risultati entro $n$ secondi e non ci riesce, il risultato diventa inutile e viene ucciso il processo responsabile
- **Memoria non disponibile**, se non si riesce ad ospitare un nuovo processo in memoria quello precedente termina
- **Violazione dei termini di memoria**, il processo inizierà a chiedere una locazione non data dal sistema operativo
- **Fallimento di un operazione** (di tipo aritmetico o I/O)
- **Terminazione del genitore**
- **Richiesta del genitore**

# Modello a 5 stati

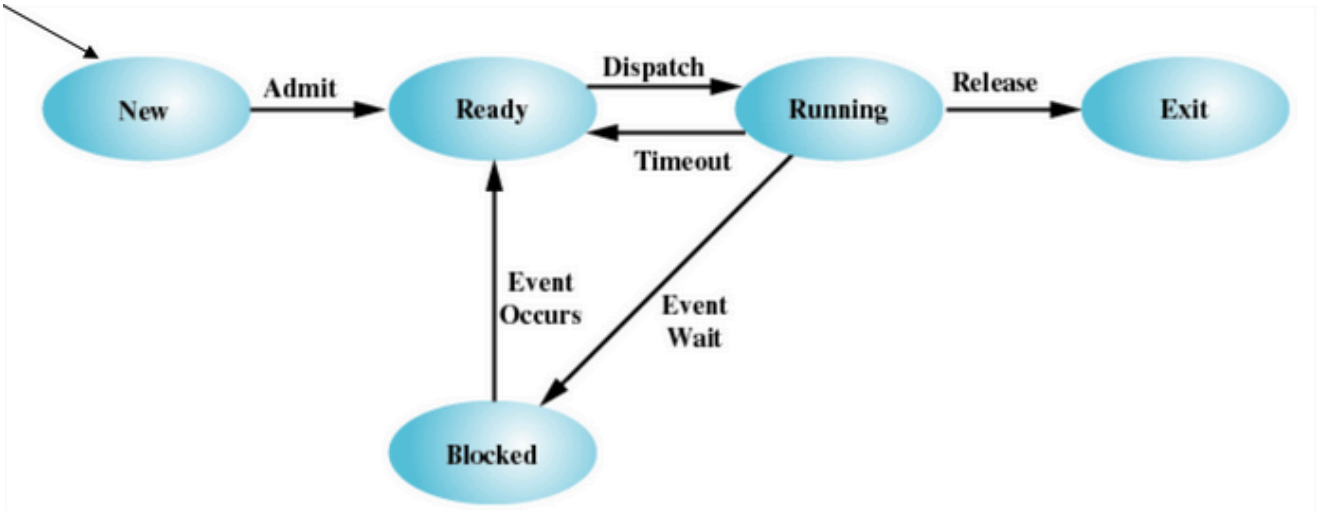



**Figure 3.6 Five-State Process Model**

La freccia che entra in **New** è la richiesta di creazione di un nuovo processo

1. Nel primo stato (**New**) è lo stato "nuovo", il SO crea un PCB quindi associa al processo il PID e tutti i componenti necessari, alloca e aggiunge una new entry nella tabella dei processi.
    - Il processo prima di entrare nello stato di ready, e quindi in stato di admit/activate, sostituisce i riferimenti simbolici delle variabili contenute nel programma con i

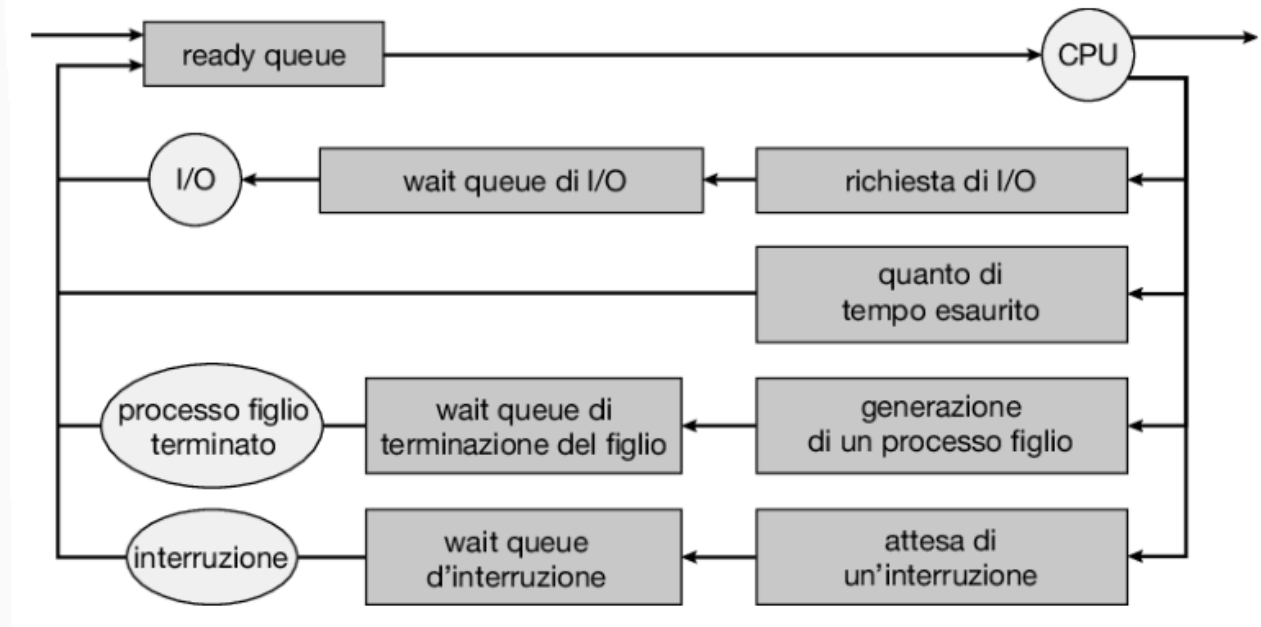
riferimenti agli indirizzi delle celle di memoria.

2. Nello stato di **Ready** il processo attende di essere assegnato a un'unità di elaborazione (CPU), dove attende finché non viene selezionato per essere eseguito dopo essere stato ammesso (admit/activate)
    - L'area di memoria che viene associata al processo si stima a seconda del linguaggio di programmazione utilizzato oppure se il programma è stato già mandato in esecuzione altre volte, ci si può basare sui dati precedenti, altrimenti si concede un'area di memoria di dimensione casuale.
3. Tramite il dispatcher il processo viene assegnato alla CPU ed entra in **esecuzione**(**Running**)
    - Nello stato di running un processo eseguirà le proprie istruzioni che possono essere di I/O oppure semplici operazioni su registri etc..
    - Il processo può entrare nello stato di **Blocked** e rimanere in attesa in RAM:
        - Se quest'ultimo necessita di uno spazio di memoria aggiuntiva (superiore alle stime del SO) allora entrerà nello stato di **Blocked** per poi essere rimandato in ready non appena ci sarà spazio sufficiente in memoria tale da permettere la sua esecuzione.
        - Se sta eseguendo operazioni di I/O e dunque in attesa di un evento (Event wait) che può verificarsi in qualsiasi momento
    - Il processo esce dallo stato di **Blocked** e ritorna nello stato di **Ready** solamente quando si verifica l'evento (event occurs) per cui era in attesa
    - Il processo può ritornare in stato di **Ready** per ragioni legate a politiche di scheduling oppure viene eseguita un'istruzione di sleep(), questo passaggio è chiamato **timeout**.
4. Il processo entra nello stato di **Exit** quando termina le istruzioni da eseguire oppure il genitore termina il figlio, quest'ultimo dunque rilascia le risorse (il SO può mantenere alcune info legate alla contabilità) e viene deallocato dalla memoria RAM per far spazio ad altri processi.

# Strategie di accodamento

Tutti i processi che entrano in esecuzione possono terminare e rilasciare le risorse, andare in timeout e ritornare in ready oppure entrare in blocked, quando essi entrano nello stato di blocked vengono inseriti in una coda dove sono presenti tutti i processi in attesa di un evento. L'utilizzo di una sola coda renderebbe complicato l'identificazione del tipo di evento per cui un processo è in attesa, si sfruttano dunque più code per suddividere le tipologie di eventi che possono essere:

- Code di I/O
- Coda di attesa di terminazione di un processo figlio appena creato
- Coda di attesa di essere reinserito nella coda dei processi dopo essere stato rimosso forzatamente dalla CPU a causa di un'interruzione




Nei primi due casi il processo passa dallo stato d'attesa allo stato pronto ed è nuovamente inserito nella ready queue.
Un processo continua questo ciclo fino al termine della sua esecuzione, dove da questo punto viene rimosso da tutte le code e vengono deallocati il PCB e le varie risorse

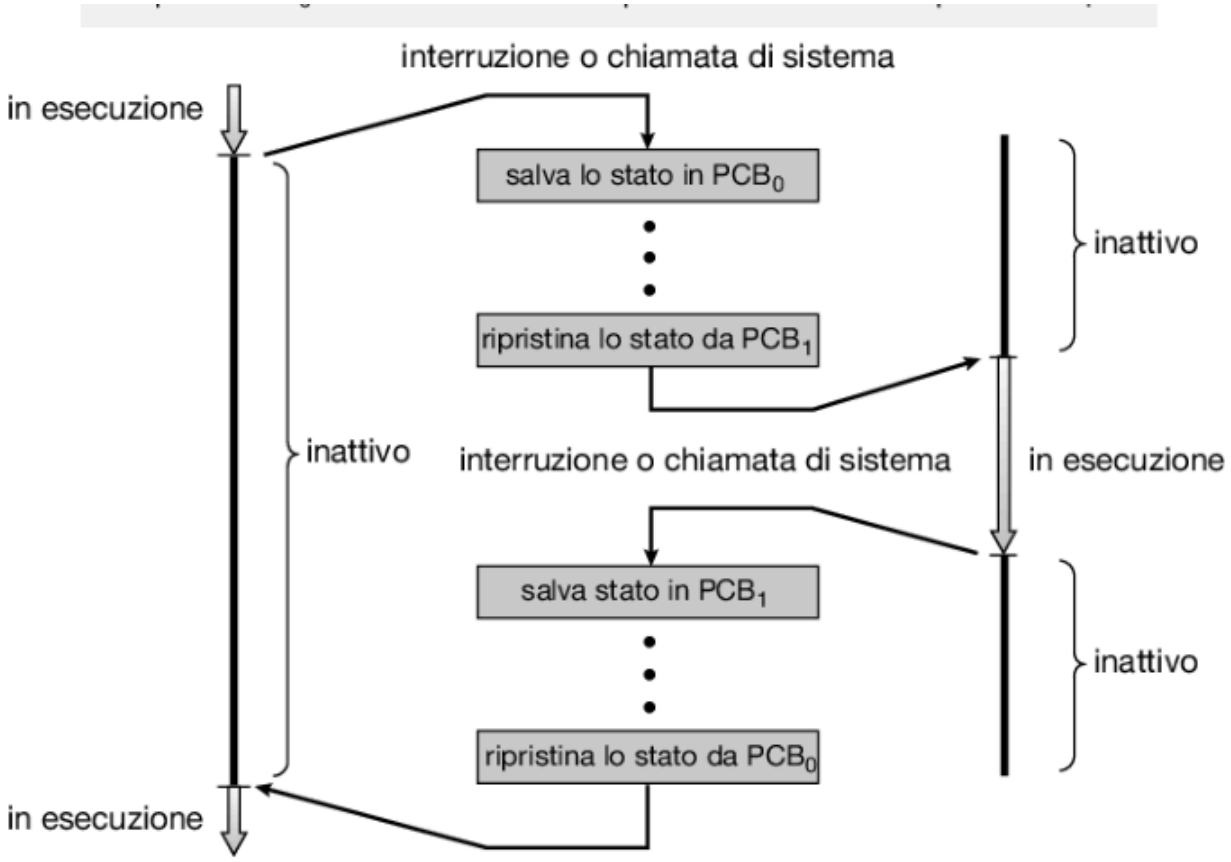
# Context switching

Le interruzioni forzano il sistema a sospendere il lavoro attuale della CPU per eseguire routine del kernel, sono eventi comuni nei sistemi general-purpose;
In presenza di una interruzione il sistema deve salvare il contesto del processo corrente per poterlo poi ripristinare quando il processo stesso potrà ritornare in esecuzione.
Il contesto è rappresentato all'interno del PCB del processo e comprende i **valori dei registri della CPU, lo stato del processo e informazioni relative alla gestione della memoria**. In termini generali, si esegue un salvataggio dello stato corrente della CPU, sia che essa esegua in modalità utente o in modalità di sistema;
In seguito, si attuerà un corrispondente ripristino dello stato per poter riprendere l'elaborazione dal punto in cui era stata interrotta.
Le possibili cause che possono causare l'interruzione di un processo sono:

- Clock interrupt: Il processo ha terminato il tempo a sua disposizione (torna in ready)
- I/O interrupt: Un operazione di I/O è terminata, il Sistema Operativo sposta il processo in attesa di tale evento da blocked a ready e decide se riprendere l'esecuzione del processo precedente
- Memory fault: L'indirizzo di memoria generato è sul disco e deve essere portato in RAM, il SO carica il blocco e nel frattempo il processo che ha generato la richiesta è in blocked, a termine del trasferimento andrà in ready
- Trap: errore di esecuzione (il processo potrebbe andare in exit)
- Supervisor call: un file viene aperto da un entità che ne prende unicamente il suo controllo, il processo utente va in blocked

In presenza di queste interruzioni il SO svolge delle operazioni in modalità supervisor al momento del cambio di processo in stato running

- Salvataggio del contesto del processo che abbandona la CPU (come il valore dei registri)
- Cambio del valore di stato nel PCB (da running a ready, blocked o exit)
- Spostamento del PCB in una nuova coda (ready or blocked) o deallocare le sue risorse (exit)
- Aggiornamento delle strutture dati (area dello stack)
- Selezione di un nuovo processo per lo stato running (dispatcher)
- Aggiornamento del suo stato nel PCB
- Ripristino del contesto




Il context-switch time è overhead (tempo non utile all'avanzamento dei processi), il sistema non svolge nessun compito utile all'utente;
Questo tempo dipende dalla complessità del SO e del suo hardware, dipendendo dalla velocità della memoria, dal numero di registri da copiare e dall’esistenza di istruzioni macchina appropriate.
Domanda da esame:
Chi fa i primi 4 punti? Il sistema operativo

# Gestione dei processi

I processi vengono quasi tutti eseguiti in modalità utente:

- Dal punto di vista della memoria può accedere solamente allo spazio di indirizzamento di un utente, non può indirizzare aree di memoria di altri processi o appartenenti al sistema operativo
- Dal punto di vista architetturale può accedere solo ed esclusivamente ai registri della CPU visibili all'utente

Alcuni processi avvengono in modalità kernel o controllo (solitamente i processi del sistema operativo, possono corrispondere ad un unico processo più processi del SO) ed è possibile che quest'ultimo possa:

- Gestire i processi: possono creare o distruggere i processi, schedulazione, cambio di contesto, sincronizzazione (gestire l'uso della stessa risorsa su due processi concorrenti)
- Gestire la memoria: allocare la memoria sufficiente a contenere il programma, trasferire i dati dal disco alla RAM e viceversa, gestione della paginazione e segmentazione
- Gestire l'I/O (i buffer e l'allocazione dei canali I/O)
- Gestire il supporto (interruzioni e compatibilità)

Ma perché deve essere un processo di un sistema operativo a crearlo e non un utente? Perchè accedere alle strutture come quelle del PCB può farlo unicamente il sistema operativo, dallo spazio utente non sono accessibili

## Creazione dei processi

In che stato ci troviamo? In quello di new

Per creare un processo bisogna:

1. Assegnare a un processo un PID unico, aggiornare quindi con una entry la tabella dei processi esistenti.
2. Allocare lo spazio per il processo e per tutti gli elementi della sua immagine (PCB, User Stack, Area di memoria dati e istruzioni, aree condivise). Alcuni elementi hanno una dimensione standard (come PCB), altre come lo user stack non possiedono una dimensione standard. Si può quindi effettuare una stima:
    - Si potrebbe ad esempio assegnare una dimensione random alla prima esecuzione, che se è minore di quello di cui ha realmente bisogno andrà in blocked. Si può da ciò dedurre di quanto spazio abbia bisogno dalla seconda esecuzione in poi.
    - Altre volte il processo padre, siccome già in esecuzione, potrebbe già conoscere già la dimensione del processo figlio, poiché era già nel suo spazio di indirizzamento. Possibilità di esecuzione:
        - Processo padre e figlio sono concorrenti, ovvero sono contemporaneamente nello stesso ciclo di esecuzione, e concorrono nell’uso di alcune risorse.
        - Il padre attende la terminazione del figlio, come quando il main invoca un metodo, attendendone la sua terminazione (stato di wait).
3. Inizializzare il PCB

- Lo stato del processore è inizializzato a 0
- Si indica la prossima istruzione al PC
- Puntatori allo stack
- Cambio di stato (in ready or ready-suspended)

4. Inserimento nella coda ready
5. Estende le strutture al fine della fatturazione o delle statistiche del software

# Terminazione dei processi

Un processo termina quando finisce l'esecuzione della sua ultima istruzione e inoltra la richiesta al sistema operativo di essere cancellato usando la chiamata di sistema exit();
A questo punto il processo figlio può riportare un'informazione di stato al processo genitore che la riceve attraverso la chiamata di sistema, wait().
Tutte le risorse del processo, incluse la memoria fisica e virtuale, i file aperti e le aree della memoria per l'I/O sono liberate dal sistema operativo.

La terminazione di un processo si può verificare anche in altri casi:
Un processo può causare la terminazione di un altro per mezzo di un'opportuna chiamata di sistema (per esempio TerminateProcess() in Windows), generalmente solo il genitore del processo che si vuole terminare può invocare una chiamata di sistema di questo tipo altrimenti gli utenti potrebbero causare arbitrariamente la terminazione forzata di processi di chiunque.
Occorre notare che un genitore deve conoscere le identità dei propri figli per terminarli, perciò quando un processo crea un nuovo figlio, l'identità di esso viene passata al processo genitore.

Un processo genitore può porre termine all'esecuzione di uno dei suoi processi figli per diversi motivi:

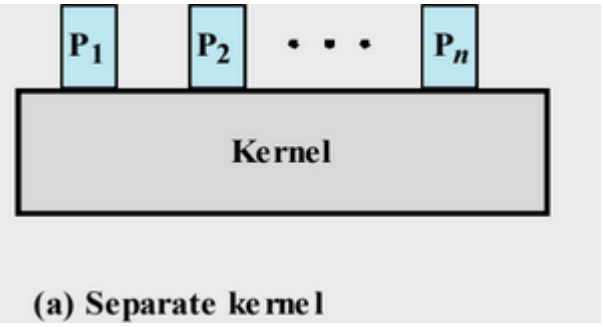
- Il processo figlio ha ecceduto nell'uso di alcune tra le risorse che gli sono state assegnate, questo richiede che il processo genitore disponga di un sistema che esamini lo stato dei propri processi figli
- Il compito assegnato al processo figlio non è più richiesto
- Il processo genitore termina e il sistema operativo non consente a un processo figlio di continuare l'esecuzione in tale circostanza. In alcuni sistemi, se un processo termina si devono terminare anche i suoi figli, indipendentemente dal fatto che la terminazione del genitore sia stata normale o anormale. Si parla di terminazione a cascata, una procedura avviata di solito dal sistema operativo.

Non possono esistere processi orfani (senza un padre) poiché dannosi (di solito associati a worm o virus), questi tipi di processi nel sistema operativo non sono più rintracciabili e si chiamano processi **zombie**.

# Modalità di esecuzione del SO

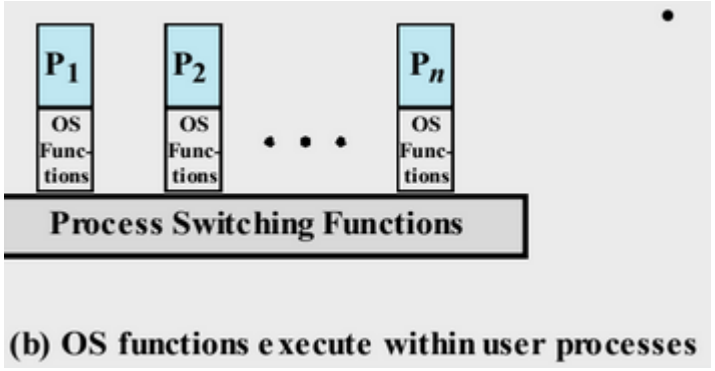
Nella prima versione il kernel era separato dai processi:

- Occupa una regione di memoria specifica
- Ha il proprio stack
- Viene eseguito come entità separata
- I processi generano una chiamata a sistema




Come si può risolvere la lentezza del kernel?
1. Nel primo modo i programmi, i dati e lo stack del kernel vengono iniettati (injected) nel processo che lavora in modalità **super utente**

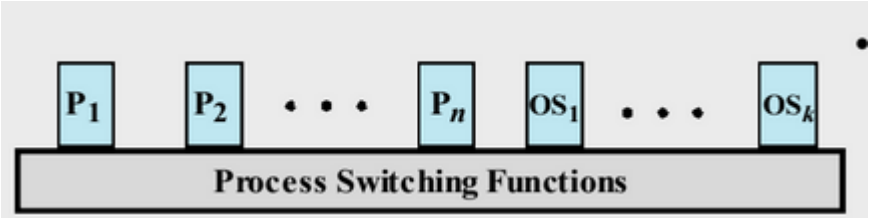
- Si prende la parte del kernel che si occupa del context switching facendo sì che ogni processo sesegua il cambio autonomamente senza nessuna chiamata al sistema.
  - Risolve l'overhead legato al context switching, ma la parte legata al context switching viene ripetuto per ogni processo, causando uno spreco di memoria
  - Quando vi è una chiamata a SO, il Kernel cambia il modo di esecuzione (salvataggio contesto utente, modo utente -> kernel ) **NB: non c'è un cambio di processo (non interviene lo schedulatore) poiché la funzione del kernel è nel processo utente**
  - Occupa molta memoria




2. Nel secondo modo il SO è basato su processi, dunque in RAM ci sono solamente alcune funzioni di esso

- Dato il sistema operativo, si individuano un insieme di istruzioni fondamentali per l'utilizzo ed un altro di funzioni usate meno spesso, ed invece che tenerle tutte assieme nel kernel si inseriscono in processi di sistema.
  - Consente di portare in memoria RAM solo le funzioni essenziali.

- L’indicazione dei dati facilita la comprensione del compito da svolgere.




La seconda metodologia è molto usata nei sistemi multiprocessori ed è per questo che nei sistemi SMP qualsiasi processore può eseguire un processo di sistema.

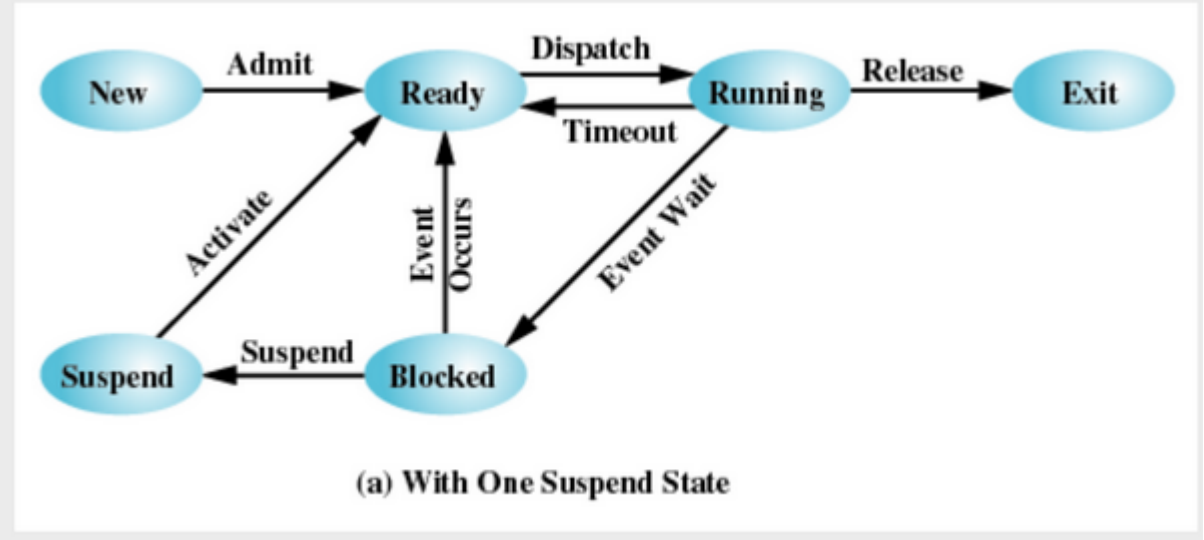
## Evoluzione del modello a 5 stati

Tutti i programmi per essere eseguiti devono risiedere in memoria tuttavia, con elevata probabilità, tutti i processi in memoria restano in attesa di operazioni di I/O o in generale di eventi.
**Soluzioni**:

- Espandere la memoria, ma più costoso e poco efficente
- Eseguire lo **swap**, un area di hard disk da utilizzare per le attività di scambio per i processi della RAM, aggiungendo al modello a 5 stati un nuovo stato chiamato **suspended**. Lo swapping è un ulteriore operazione di I/O.

Lo swap out è l'area di scaricamento del processo sul disco dallo stato blocked a suspended (dalla RAM al disco)
Lo swap in è l'area di scaricamento da RAM dallo stato suspended a blocked (dal disco alla RAM)
Chi è in blocked è sospeso in RAM, chi è in suspended sta nel disco

Ma tra processi in new e in suspended quale si sceglie da portare in ready? Se un processo è in suspended significa che è in attesa di un evento che se si verifica potrebbe andare in ready.




## Evoluzione a 7 stati

È possibile che lo stato suspended abbia anche due stati:
**Ready suspended**: il processo è pronto, ha tutte le risorse di cui ha bisogno di essere eseguito ma manca di memoria.
**Blocked suspended**:- Il processo è bloccato e sospeso, quindi attende un evento mentre è fuori dalla memoria principale.

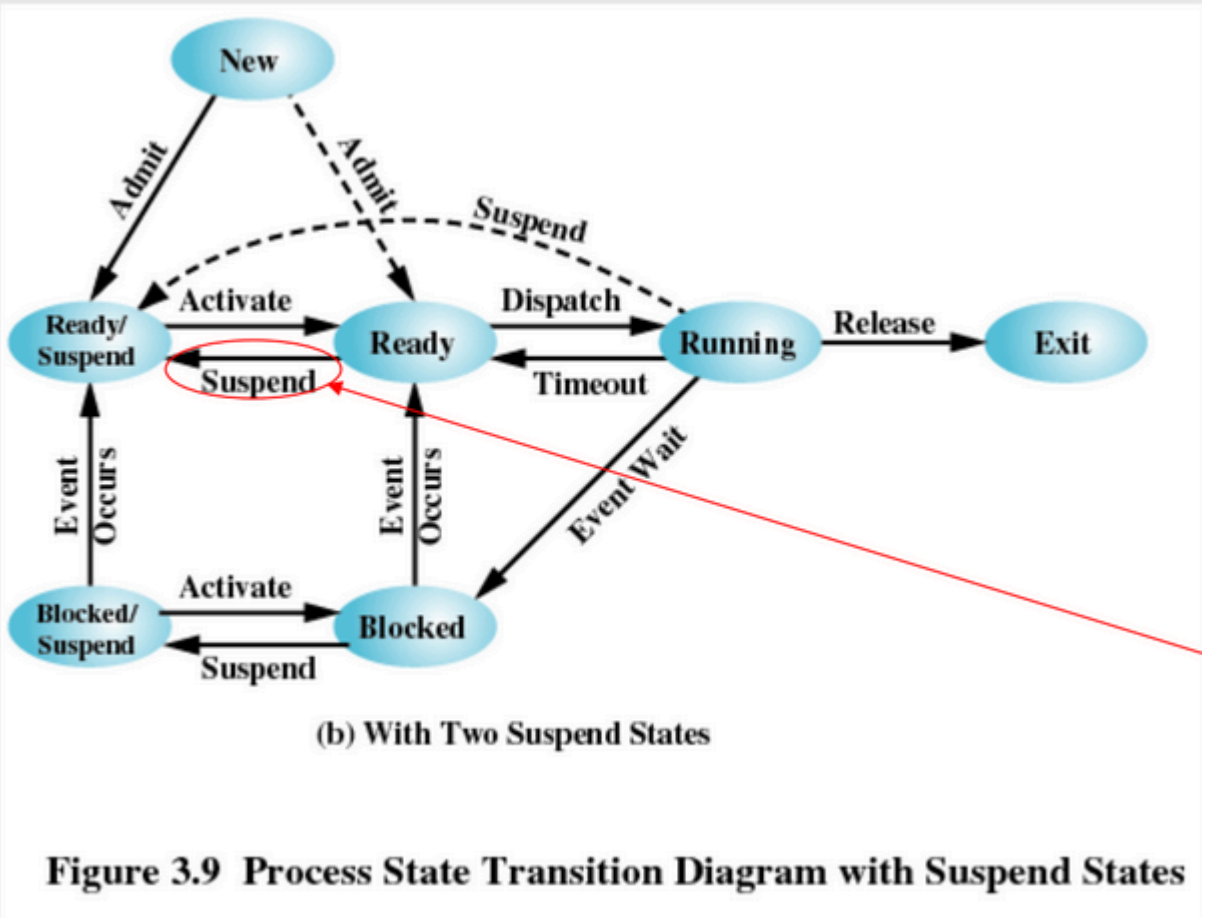



**Figure 3.9 Process State Transition Diagram with Suspend States**

Questo rende il modello più realistico, poiché i sistemi operativi spesso spostano processi fuori dalla memoria principale per ottimizzare l'uso delle risorse.
Ma perché un processo può andare da ready (quindi pronto) a ready/suspended?
Il processo avrà bisogno di più RAM o di una maggiore priorità

Dunque i passaggi di stato sono così definiti:

1. New

- new $->_{admit}$ ready: pronto + RAM disponibile
- new $->_{admit}$ ready-suspend: pronto + RAM non disponibile

2. Ready

- ready $->_{suspended}$ ready-suspend: necessità di maggiore memoria per allocare un processo più grande o con maggiore priorità
- ready $->_{dispatch}$ running: CPU disponibile e assegnata per l'esecuzione

3. Running

- running $->_{release}$ exit: processo termina la sua esecuzione
- running $->_{timeout}$ ready: processo messo in timeout
- running $->_{Eventwait}$ blocked: il processo è in attesa di un evento per proseguire
- running $->_{suspend}$ ready-suspend: il processo necessita di memoria aggiuntiva durante la sua esecuzione

4. Blocked

- blocked $->_{eventoccurs}$ ready: si verifica l'evento atteso da un processo
- blocked $->_{suspend}$ blocked-suspend: in attesa di un evento, caricato in HDD tramite **swap-out**

5. Blocked-suspend

- blocked-suspend $->_{eventoccurs}$ ready-suspend: il processo può riprendere la sua esecuzione poiché si è verificato l'evento, è in attesa di essere caricato in RAM
- blocked-suspend $->_{activate}$ blocked: il processo viene ricaricato in RAM poiché è stato fatto un riferimento a un indirizzo su disco

6. Ready-suspend

- ready-suspend $->_{activate}$ ready: il processo è pronto e può essere caricato in memoria poiché disponibile

7. Exit

- Terminazione del programma

# Scheduling

Con scheduling si intende un insieme di tecniche e di meccanismi interni del sistema operativo che amministrano l’ordine in cui il lavoro viene svolto, in pratica stabilisce l’ordine con il quale vengono ordinate tutte le code dei processi.

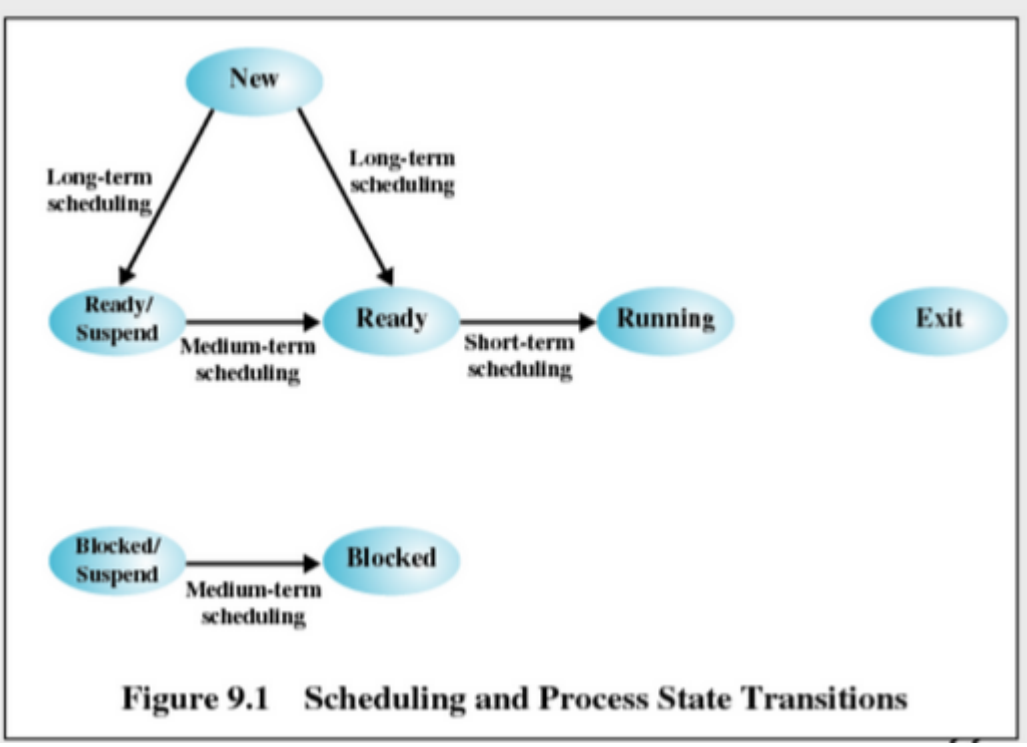



Figure 9.1 Scheduling and Process State Transitions

Tipicamente nei sistemi **multi-programmati**

- **New** è una coda perché ci possono essere più richieste di attivazione di più processi in contemporanea
- **Ready** è una coda, ci sono più processi in memoria pronti per essere mandati in esecuzione
- **Blocked** è una coda poiché ci sono più processi
- Ci sono altre code

**L'obiettivo primario dello scheduling è l'ottimizzazione delle prestazioni del sistema;**
Ad esempio per la coda di **Ready** lo scheduling è necessario per decidere **quale** processo mandare in esecuzione, in **New** invece è necessaria per determinare a quale processo assegnare la memoria

A seconda dei casi lo scheduling può assumere nomi differenti dunque criteri di scelta diversi a seconda del tipo di coda su cui si sta lavorando.

- **SLT**: scheduler di lungo termine
- **SMT**: scheduler di medio termine
- **SBT**: scheduler di breve termine
  Chi determina dove usare uno e dove usarne un altro? Si può risolvere il problema con i diversi algoritmi per decidere quale tra questi utilizzare, per farlo si raggruppano una serie di processi e si testano i vari algoritmi tenendo conto di quale performa meglio

## Scheduling a lungo termine

Stabilisce quale processo può essere ammesso all'interno del sistema e quindi passare dallo stato New a quello di Ready (o Ready Suspended).
Controlla il grado di multiprogrammazione (new ->ready)

- Più processi in memoria = minor percentuale di tempo di esecuzione per ognuno di essi

Stime effettuate dal programmatore (o dal sistema) forniscono informazioni sulle risorse necessarie alla esecuzione, come le dimensioni della memoria, il tempo di esecuzione totale, etc.;
Il lavoro dello scheduler a lungo termine si basa quindi sulla stima del comportamento globale dei job.

- In generale su 10 processi, ognuno tra questi avrà $\frac{1}{10}$ del tempo di esecuzione

Le motivazioni per cui viene impiegato sono principalmente:
1. Fornire alla coda dei processi pronti (e quindi allo scheduler di breve termine) gruppi di processi che siano bilanciati tra loro nello sfruttamento della CPU e dell'I/O in modo da usare tutte le risorse contemporaneamente;
2. Aumentare il numero di processi provenienti dalla coda batch, ovvero una sottocoda della coda dei ready in cui ci sono tutti i processi che vogliono usare la CPU, quando il carico della CPU diminuisce;
3. Diminuire (fino anche a bloccare) i lavori provenienti dalla coda batch, quando il carico aumenta e/o i tempi di risposta del sistema diminuiscono.
- La frequenza di chiamata dell'SLT è bassa e consente di implementare strategie anche complesse di selezione dei lavori e di dimensionamento del carico dei processi da inviare alla coda di ready

È ovvio quindi che gli scheduler devono parlarsi tra loro per sincronizzare l'entrata dei processi quando il carico diminuisce.
Inoltre è importante distinguere due tipi di processo:

- I/O bound, ovvero che fa un maggior utilizzo dei dispositivi di I/O
- CPU bound, che fa un maggior utilizzo di CPU

## Scheduling a medio termine

Stabilisce il numero di processi attivi presenti, in un certo istante, in memoria RAM; in pratica sceglie quale processo viene ammesso all'interno della memoria centrale.
Effettua i seguenti cambi di stato:

- Ready suspended ->Ready
- Blocked suspended ->Blocked

Proprio a causa del suo obbiettivo lo scheduler a medio termine è parte della funzione di swapping ed è basato sulla necessità di gestire il livello di multi-programmazione;
Utilizza le informazioni del PCB per stabilire la richiesta di memoria del processo, proprio perché uno dei campi è la stima della memoria necessaria.
La presenza di molti processi sospesi in memoria, riduce la disponibilità per nuovi processi pronti, in questo caso lo **scheduler di breve termine** è obbligato a scegliere tra i pochi

processi pronti.
L'SMT viene attivato quando:

- Si rende disponibile lo spazio in memoria, cioè se non ci sono processi in memoria oppure se dei processi in blocked-suspend/ready-suspend devono essere riportati in memoria;
- L’arrivo di processi pronti scende al di sotto di una soglia specificata.

Come fa il sistema di memoria virtuale a capire di quanta memoria ha bisogno il processo di swap-out? Tramite il PCB, poiché quest'ultimo contiene le informazioni sui limiti della memoria allocata, lo stato delle pagine e le richieste di memoria

## Scheduling a breve termine (dispatcher)

Stabilisce quale processo andrà ad impiegare la CPU, ovvero il prossimo processo da eseguire quindi il passaggio da:

- Ready -> Run

Questo algoritmo è eseguito molto frequentemente e viene sfruttato nel momento in cui un processo in esecuzione deve abbandonare la CPU e bisogna determinare il prossimo da mandare in esecuzione,
Viene invocato al verificarsi di:

- **Clock interrupt** (periodo di tempo di assegnazione della CPU terminato)
- **I/O interrupts**
- **Chiamata al sistema operativo** (spostamento in blocked del processo con una chiamata da un processo che non è di competenza della CPU)
  - Ad esempio la printf(), operazione che non esegue la CPU ma il monitor, oppure una scanf(), una fileopen(), inoltro di dati sulla scheda di rete ecc..
  - Questo genere di chiamate comportano il cambio di stato da **execute** a **blocked** del processo poiché in attesa di altri dispositivi
- **Signal (Trap)**

### Scheduling della CPU

Esistono due **decision mode** (Decisione di scheduling):

1. Non-Preemptive: Un processo nello stato di running abbandonerà tale stato solo al termine dell’esecuzione o se si blocca per una operazione di I/O. Il processo va in esecuzione ed utilizza la CPU fino a quando non viene eseguita l'ultima istruzione che indica la terminazione del programma.
2. Pre-emptive: Il processo attualmente nello stato di running può essere interrotto e spostato nello stato di ready dal SO, nessun processo può monopolizzare il processore ma dei processi che condividono dati hanno necessità di sviluppare meccanismi di

sincronizzazione dei processi
**Dispatcher**: modulo del SO che passa il controllo della CPU al processo selezionato dallo scheduler a breve termine attraverso:

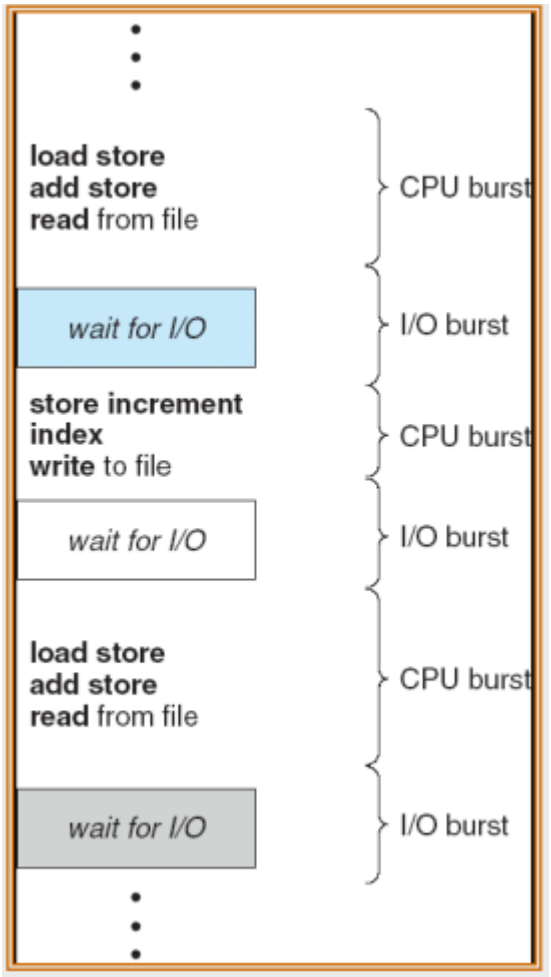
- switch del contesto
- switch del modo utente/supervisore
- Salto alla locazione opportuna del programma utente (contenuta nel PCB) per farlo ripartire, cioè recupero del valore del PC dal PCB

Il tempo che il dispatcher usa per fermare un processo e far partire un altro è chiamato **Dispatch latency**.

## Scheduling della CPU (a breve termine)

Le attività di un processo durante il suo ciclo di esecuzione sono:

1. ciclo di elaborazione (CPU),  ovvero operazioni con la CPU
2. attesa di completamento di I/O, ovvero operazioni con dispositivi I/O




Lo scheduling della CPU riguarda la distribuzione delle sequenze di elaborazione della CPU, infatti deve esistere un equilibro tra una sequenza di istruzioni CPU burst e I/O burst. L'appellativo **burst** specifica una sequenza temporale continua in cui il processo esegue molte operazioni di tipo CPU o di tipo I/O.
Un processo si definisce **I/O bound** quando le operazioni che vuole svolgere sono dirette a dispositivi di I/O (Word è un processo di tipo I/O bound; usa schermo, tastiera

ecc... ), tipicamente sono molte sequenza di operazioni di CPU aventi una breve durata
Un processo si definisce **CPU bound** se l'insieme delle operazioni che deve svolgere sono prevalentemente sulla CPU (Un programma che analizza un'immagine è di tipo CPU bound), tipicamente sono poche sequenze di operazioni di CPU di lunga durata

## Valutazione delle prestazioni

Per la valutazione delle prestazioni si possono assumere vari tipo di approcci:

1. User-oriented (interesse in valutazioni del sistema dal punto di vista dell'utente cioè chi userà il SO)
   - Turnaround time: tempo intercorso tra la sottomissione di un processo e la sua terminazione. (*istante in cui parte una richiesta di attivazione di un processo rispetto all'istante in cui viene soddisfata la richiesta*). Questo parametro dipende anche dal progettista e dal tipo di requisiti (funzionali e non funzionali)
   - Tempo di risposta: per un processo interattivo è il tempo trascorso tra la sottomissione di richiesta e il ritorno del primo output. Il processo può continuare nel suo ciclo di esecuzione mentre produce output.
   - Deadlines: quando la deadline di un processo è specificata, lo scheduling deve fare in modo di completare (o avviare) il processo entro la deadline. Ad esempio i processi che hanno il compito di interfacciarsi con la rete e scaricare pacchetti in arrivo
2. System-oriented (nessun interesse verso l'utente ma si cerca di sfruttare al meglio la strumentazione fornita)
   - Throughput: numero di processi terminati per unità di tempo (**unità di tempo nei sistemi = 1 secondo**)
   - Utilizzo del processore: % del tempo in cui la CPU è impegnata
   - Per eseguire programmi degli utenti
   - Per eseguire moduli del sistema operativo
   - Evitare la starvation
   - Utilizzare tutte le risorse (dispositivi di I/O, CPU, ecc.)
   **Criteri di ottimizzazione:

- Massimizzare l'utilizzo della CPU
- Massimizzare il throughput (operazioni eseguite in un'unità di tempo)
- Minimizzare il turnaround time
- Minimizzare il tempo di risposta
  La principale motivazione per cui lo scheduling a breve termine viene impiegato è per **massimizzare le prestazioni** del sistema secondo un specifico insieme di obiettivi.
  Ogni quanto viene eseguito lo scheduling a breve termine? Ogni 5ms

## Valutazione delle prestazioni

**Tempo di ricircolo** (*Turnaround time*)

- Tempo trascorso da un processo all'interno del nostro sistema (dal suo avvio alla sua terminazione). Il progettista conosce i passaggi che fa un processo all'interno del nostro sistema quindi è possibile scomporre il tempo del runtime in voci intermedie:

$$T_{LOADING} + T_{READY} + T_{CPU} + T_{I/O}$$

- Con $T_{LOADING}$ indichiamo il tempo che ci vuole nell'attesa in stato new + l'attesa passata in stato di ready-suspended prima di essere caricato in memoria RAM
- Con $T_{READY}$ indichiamo il tempo trascorso in coda di ready
- Con $T_{CPU}$ indichiamo il tempo trascorso con la CPU
- Con $T_{I/O}$ indichiamo il tempo in cui aspetta i dispositivi di I/O

I primi due sono **overhead** poiché il tempo è dato da una serie di fattori, ad esempio il tipo di operazioni che svolge o deve svolgere quel processo, la presenza di altri processi più importanti, bassa velocità dei dispositivi hardware.

**Tempo di attesa**

Misura il tempo che un processo trascorre in attesa delle risorse a causa di conflitti con altri processi.

È la penalità che si paga per condividere risorse ed è espressa come:
$\text{tempo di ricircolo} - \text{tempo di esecuzione}$. Questa è una conseguenza della multi-programmazione poiché più processi necessitano di utilizzare la stessa risorsa.

Valuta in sostanza la sorgente di inefficienza, il tempo passato nelle code è legato all'inefficienza quindi quanto più si riesce a minimizzare i tempi d'attesa, più il sistema risulterà efficiente.

**Tempo di risposta**:

Un sistema può essere di due tipi, time-sharing o real-time (comunemente usato):

- Nei sistemi time-sharing si misura il tempo che trascorre dal momento in cui viene introdotto l’ultimo carattere al terminale all’istante in cui viene restituita la prima risposta, in quest'ultimo c'è una minore interazione con l'utente. (es: creazione di deep-fake). La presenza di più processi può impattare sul tempo di risposta poiché devono condividere le risorse per unità di tempo.
- Nei sistemi real-time si misura il tempo che trascorre dal momento in cui viene segnalato un evento esterno all’istante in cui viene eseguita la prima istruzione della relativa routine di gestione

## Priorità - Event Driven

A ogni processo viene assegnato un livello di **priorità**, lo scheduler sceglie sempre il processo pronto con priorità maggiore.

La priorità, presente all'interno del PCB, può essere assegnata dall’utente o dal sistema e può essere di tipo statico (immutabile) o dinamico (priorità modificabile in base all'esigenza).

La priorità dinamica varia in funzione:

- del valore iniziale

- delle caratteristiche del processo
- della richiesta di risorse
- del comportamento durante l’esecuzione

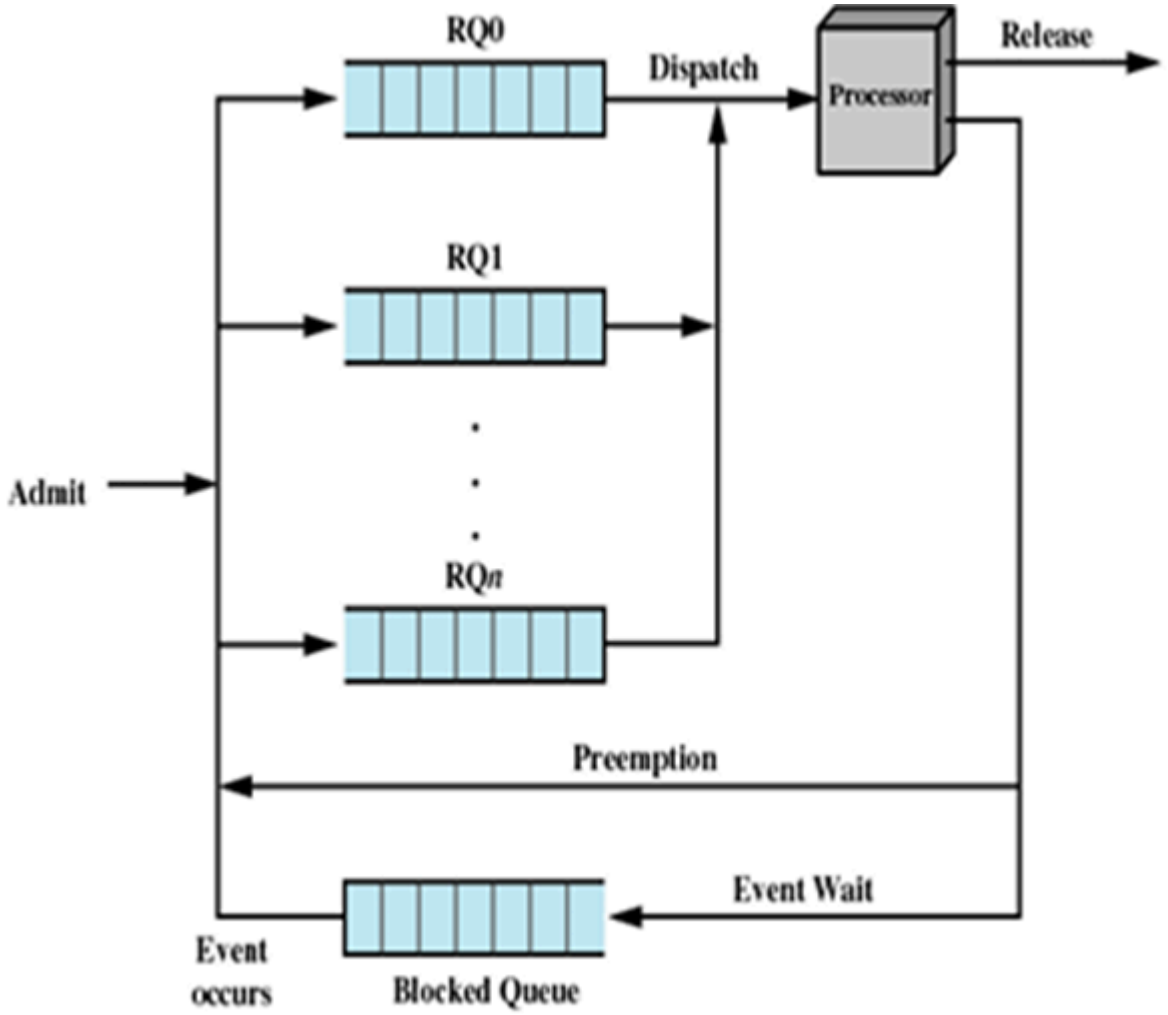
**Modello Event Driven**: modello usato nei sistemi che si basano sulle priorita’ e sulla risposta ad eventi esterni. Modello usato in cui il tempo di risposta del sistema è fondamentale.

Caratteristiche:

- Il sistemista può influire sull’ordine in cui uno scheduler serve gli eventi esterni modificando le priorità assegnate ai processi (come ad esempio, le operazioni legate ad un utente che nel nostro e-commerce è solito effettuare grandi ordini, hanno priorità maggiore).
- Le prestazioni sono dipendenti da una accurata pianificazione nell’assegnazione delle priorità, ovvero il sistemista ha un suo piano di priorità.
- Non è in grado di garantire il completamento di un processo in un intervallo di tempo finito dalla sua creazione.

La priorità è sempre associata ad un numero intero.
Se consideriamo uno schedulatore a breve termine, è possibile organizzare la coda di ready $RQ_n$ (in foto) per priorità, quindi coda con processi di priorità 1, coda con processi di priorità

2 etc.




- Preemptive (preazione): un processo può essere requisito della CPU se la priorità del nuovo processo in ready è maggiore.
  - Se il processo durante la sua esecuzione viene requisito della CPU, tutti i suoi date generati da operazioni su altri dati sono presenti all'interno dei registri dunque quest'ultimi, cioè il contesto computazionale, verrà salvato all'interno del PCB del processo.
  - In caso di una scanf() se l'interrupt avviene prima dell'inserimento del dato non accade nulla mentre se si verifica al dopo l'inserimento il problema non sussisterà poiché questo tipo di operazioni non sono gestite dalla CPU e il processo va in **blocked** quindi l'interrupt non riguarda quel processo ma quello in esecuzione.
  - Questo ramo in alcune politiche dei SO potrebbe non essere utilizzato
- Non-preemptive (senza-preazione)

Le priorità possono essere definite:

- Internamente al SO (utilizzando grandezze misurabili): quantità di memoria usata, file aperti, rapporto tra picchi medi di di I/O e di CPU
  - La priorità dunque viene data ai processi che occupano più spazio in memoria, che hanno un maggior numero di file aperti, ciò è necessario a tal fine di liberare più risorse possibili da poter poi assegnare ad altri processi

- Esternamente al SO rilevanza del processo, criticità.

Nel momento in cui gestiamo i processi con priorità basati su eventi si può generare un problema chiamato **starvation** cioè processi a bassa priorità che non vengono mai eseguiti. La soluzione applicabile è l'**aging**, al passare del tempo passato in stato di ready la priorità del processo aumenta

# Algoritmi di schedulazione

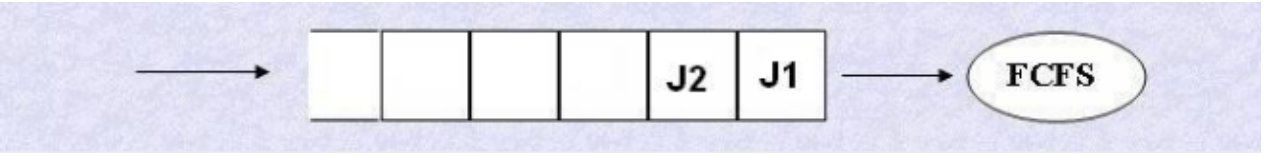
## FCFS(First Come First Served)

Con questo schema la CPU si assegna al processo che la richiede per primo, la realizzazione del criterio FCFS si basa su una coda FIFO (First In First Out):
Quando un processo entra nella Ready queue, si collega il suo PCB all'ultimo elemento della coda e quando la CPU è libera viene assegnata al processo che si trova alla testa della coda, rimuovendolo da essa.

L'algoritmo di scheduling FCFS è senza prelazione:
Una volta che la CPU è assegnata a un processo, questo la trattiene fino al momento del rilascio che può essere dovuto al termine dell'esecuzione o alla richiesta di un'operazione di I/O, questo comporta a:

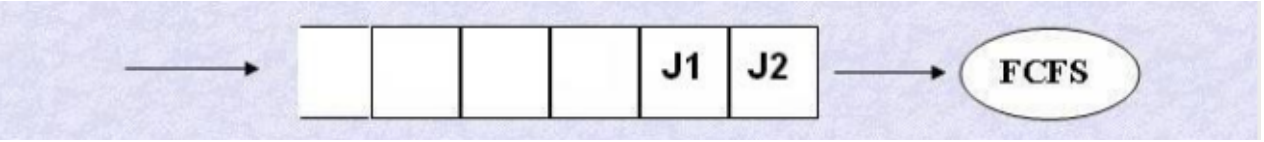
- basso sfruttamento dei componenti
- basso lavoro utile del sistema
  L'algoritmo FCFS risulta particolarmente problematico nei sistemi in time-sharing, dove è importante che ogni utente disponga della CPU a intervalli regolari, permettere a un solo processo di occupare la CPU per un lungo periodo condurrebbe a risultati disastrosi
  Altri lati negativi presenti sono:
- Sfavoreggiamento dei processi I/O Bound, che seppur richiedano poco tempo di assegnazione alla CPU saranno costretti ad aspettare.
- Tempo medio d'attesa spesso abbastanza lungo.
- Effetto convoglio: tutti i processi in coda attendono che un processo CPU-bound termini.
  **ESEMPIO**:
  Siano $J_1$ e $J_2$ due job con tempi di esecuzione totali rispettivamente pari a $t_1$ = 20 e $t_2$ = 2 unità di tempo.

1. **Primo caso**




Tempo di riclico di $J_1$ = 20; Tempo di riclico di $J_2$ = 22; Tempo medio di riclico= (20 + 22) / 2 = 21;
Tempo di attesa di $J_1$ = 0; Tempo di attesa $J_2$ = 20; Tempo medio di attesa = (0 + 20) / 2 = 10;

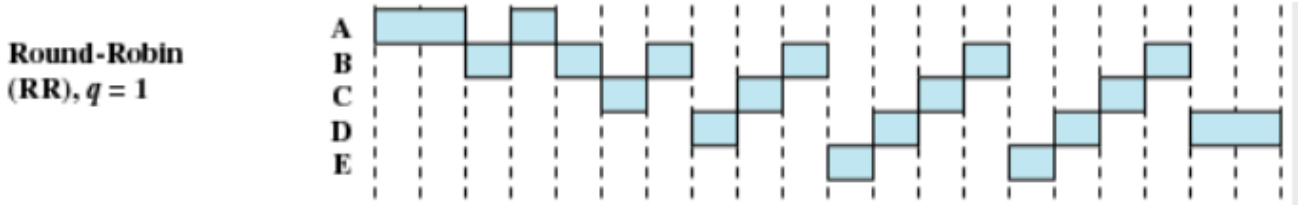
2. **Secondo caso**




Tempo di riclico di $J_2$ = 2; Tempo di riclico di $J_1$ = 22; Tempo medio di riclico= (2 + 22) / 2 = 12;
Tempo di attesa di $J_2$ = 0; Tempo di attesa $J_1$ = 2; Tempo medio di attesa = (0 + 2) / 2 = 1;

## Round Robin - Time Slice

Per ovviare ai problemi di efficienza generati dal FIFO, si fa uso della prelazione cioè la possibilità di requisire la CPU ad un processo, per farlo viene stabilito un quanto di tempo *q* con un valore determinato dal progettista e starà ad indicare il tempo di utilizzo della CPU per tutti i processi.

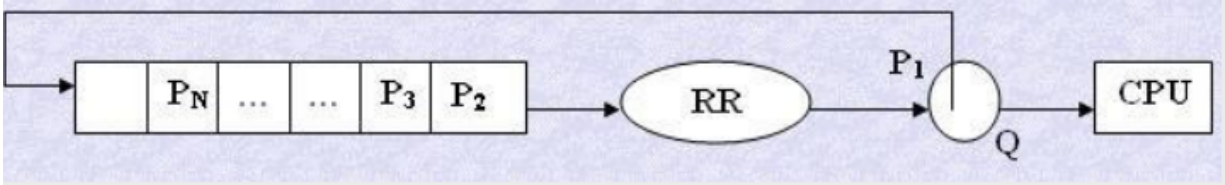



L'algoritmo Round Robin si basa sempre su FIFO ma fa uso di pre-emption basata sul clock (clock interrupt), ogni processo utilizza il processore per un dato intervallo di tempo (time slice), i valori tipici del time slice ricoprono un range dai 10ms ai 100ms;
Al verificarsi dell'interrupt, cioè ha raggiunto il tempo di utilizzo massimo, il processo in esecuzione viene portato nella coda di ready e il nuovo processo mandato in esecuzione sarà scelto in base a **FIFO**.

Con $n$ processi in ready e un time quantum $q$, ogni processo ottiene $\frac{1}{n}$ del tempo di CPU in frazioni di tempo al più pari a $q$.
Tramite questo meccanismo è possibile ottenere un tempo massimo di attesa in ready uguale a $(n-1) * q$ con $n$ = numero di processi e $q$ = quantità di tempo;
Per ottimizzare le prestazioni del sistema dunque è necessario agire sul quanto di tempo tenendo in considerazione sempre il numero di processi:

- con un valore di $q$ grande si ritorna ad una politica FCFS
- con un valore di $q$ piccolo si incrementa il numero di context switch ottenendo un tempo maggiore solo per effettuare un context switching rispetto al tempo di esecuzione di un processo
  La schedulazione Round-Robin fornisce una buona politica se adottata per lo schedulatore a breve termine:
- I processi più brevi, quelle di tipo I/O bound possono completare l'operazione molto più velocemente in un $q$ (buon tempo di risposta)
- I processi più lunghi sono forzati a passare più volte per la coda dei processi pronti (tempo proporzionale alle loro richieste di risorse)

- Per i processi interattivi lunghi, se l' esecuzione tra due fasi interattive riesce a completarsi in un $q$ , il tempo di risposta è buono
  La realizzazione di uno scheduler RR richiede il supporto di un Timer che invia un'interruzione alla scadenza di ogni q, forzando lo scheduler a sostituire il processo in esecuzione, lo stesso timer viene riazzerato se un processo cede il controllo al Sistema Operativo prima della scadenza del suo $q$.




## Highest Response Ratio Next

Questa è un altra politica di scheduling dove con $w$ indichiamo il tempo speso in coda di ready (attesa della disponibilità del processore) e con $s$ indichiamo il tempo di servizio previsto.
Si definisce il **Response Ratio** come:

$$RR = \frac{w + s}{s}$$

Quando un processo entra in coda di ready per la prima volta, $RR$ sarà uguale a 1 mentre più passa tempo nella coda di ready più $w$ aumenta quindi otterremo un rapporto maggiore a 1.
Sfruttando quest'algoritmo di schedulazione viene selezionato il processo che ha atteso, **in maniera proporzionata al lavoro che deve svolgere**, più tempo in ready, quindi con $RR$ maggiore rispetto a tutti gli altri, questo approccio favorisce chi ha un tempo di servizio molto basso ma il problema sorge all'inizio quando entra nella coda di ready, quindi con $w$ = 0 e $s$ = casuale otterremo

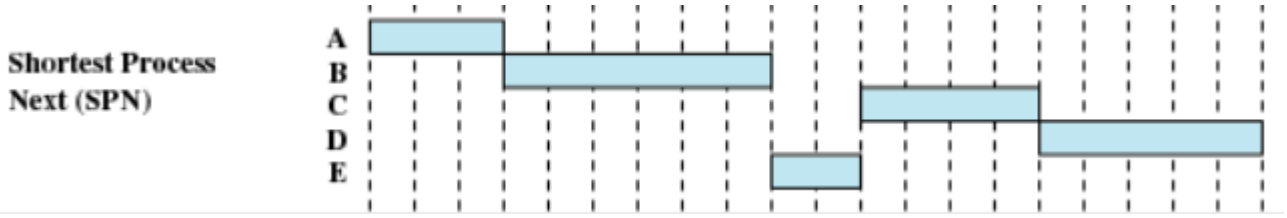
$$RR = \frac{0 + s}{s} \rightarrow \frac{s}{s} = 1$$

Quanto vale *s*?
Se è già stato in esecuzione allora abbiamo una statica, se invece è la prima verrà impostato un numero casuale che poi verrà modificato e utilizzato per una seconda esecuzione del processo

## Short Process Next (SPN)

Quest'algoritmo si basa su Highest Response Ratio Next ma presenta una miglioria, l'algoritmo HRRN richiede la valutazione di $s$, il problema fuoriesce quando durante l'esecuzione del processo, $s$ viene quasi interamente espletato, questo accade perché il calcolo del Response Ratio viene effettuato solamente una volta e il processo venendo eseguito più volte, si avvicinerà sempre di più al suo termine, dunque si tiene conto del "passato" del processo e non rappresenta il vero tempo di servizio previsto.

Il processo scelto dalla coda di ready è quello con il più breve tempo di esecuzione (in run) stimato: più breve sequenza di operazioni svolte dal processore
I processi I/O bound sono quelli che richiedono un tempo di esecuzione inferiore poiché una volta eseguita l'istruzione (come scanf()) attenderanno in stato di blocked, dunque quest'ultimi verranno selezionati con una maggiore frequenza facendo aumentare la possibilità di **starvation** per processi fortemente CPU bound.
Lo SPN è ottimale nel senso che fornisce il tempo medio di attesa minimo, se compariamo tutti gli algoritmi sul tempo di attesa, questa sarà quello che ci darà il valore minimo, per le sue caratteristiche viene utilizzato nello scheduling a lungo termine.
Rispetto a scegliere inizialmente un tempo di utilizzo casuale che si tratta un'operazione molto semplice, stimare la durata della prossima sequenza di CPU è difficile (oltre che oneroso)

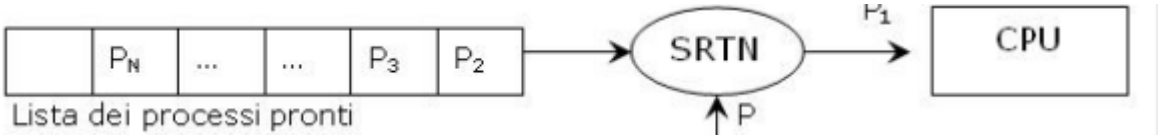



Ha due possibili schemi:

- Non preemptive;
- Preemptive: se arriva nuovo processo con una sequenza di CPU minore del tempo necessario per la conclusione della sequenza di CPU del processo attualmente in esecuzione, si ha il pre-rilascio della CPU a favore del processo appena arrivato. Questo schema è anche noto come Shortest-Remaining-Time-First (SRTF)

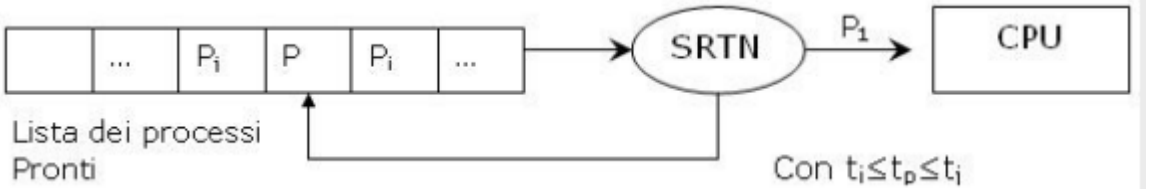
## Shortest Remaining Time Next

### SPN Con Prerilascio

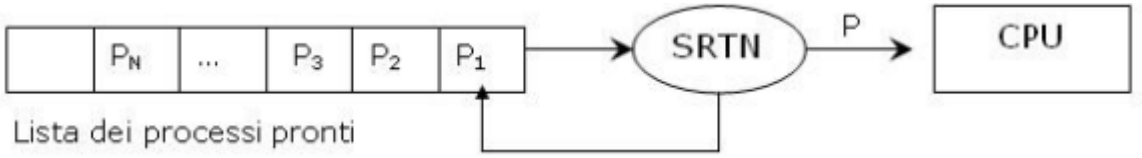



$P_i$: i-esimo processo con i = 1,…, N;
$t_i$: tempo di esecuzione dell'i-esimo processo;
Per ogni i, j = 1,...,N i ≤ j: $t_i \leq t_j$:
$t_p$: Tempo di esecuzione del processo P;
$t_1^r$ tempo di esecuzione residuo del processo $P_1$ ;

1. **Primo caso** : $t_1^r \leq t_p$

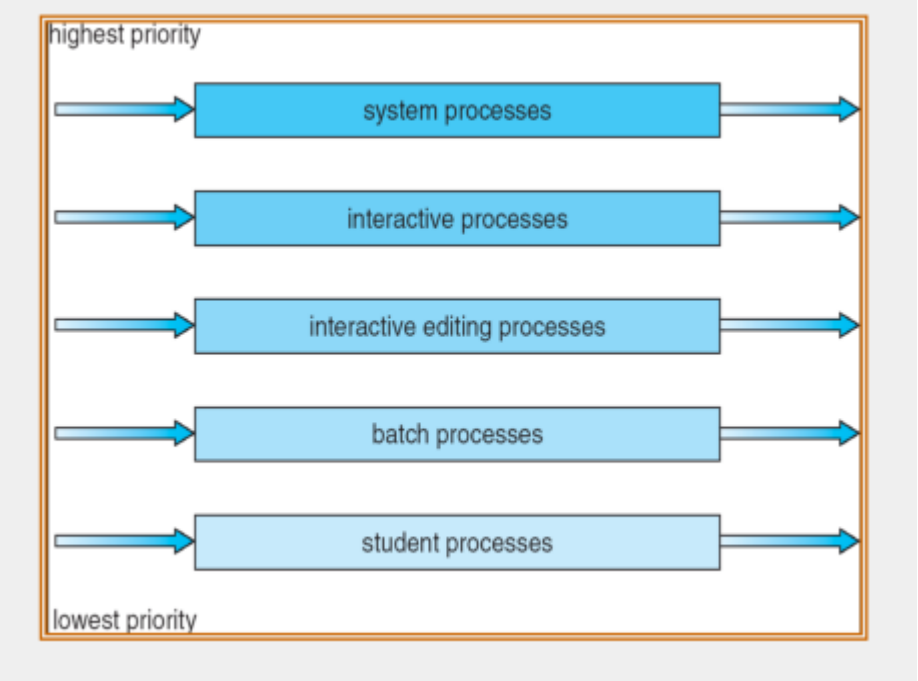


2. **Secondo caso** : $t_1^r > t_p$




## Schedulazione a code multiple

Esistono più tipi di code, ognuna suddivisa in base ad un criterio (la priorità ad esempio)
Alcuni dei processi possono anche cambiare la coda di appartenenza se stanno svolgendo funzioni differenti




La coda di ready è suddivisa in sotto-code suddividendo i processi in:

- Foreground (interactive)
  - processi in primo piano per l'utente che ci interagisce, l'interattività di un processo è dato dall'utilizzo del monitor, della tastiera, del touch-screen e da altri dispositivi possono interfacciarsi con l'utente esterno
- Background (batch)
  - processo che esegue operazioni non interattive, ad esempio il download delle mail, di un film ecc... attraverso la scheda di rete

Ogni coda ha un proprio algoritmo di schedulazione:

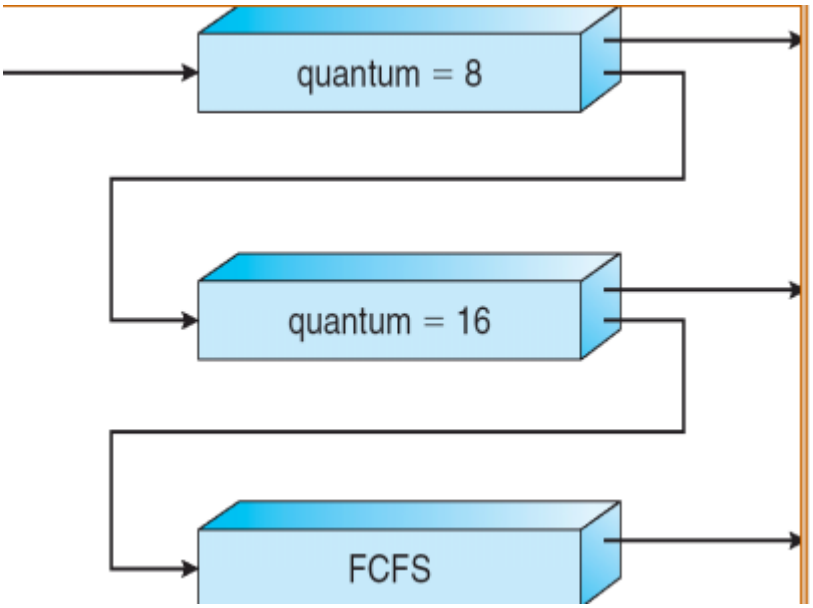
- Tipicamente i processi foreground applicano una politica RR poiché sono presenti più processi con cui l'utente sta interagendo, ad esempio più finestre aperte con un servizio di streaming e word in esecuzione;
- Quelli background applicano una politica FCFS

Lo scheduling deve essere effettuato tra le code:

- Scheduling per priorità fissa e con prelazione (i.e., serve all from foreground then from background). Possibilità di starvation.
- Time slice: ad ogni coda è associato un certo ammontare di tempo di CPU; ad esempio l'80% ai processi foreground in RR e il 20% ai processi in background in FCFS

## Schedulazione a code multiple con feedback

Per cambiare la priorità dei processi o la loro coda di appartenenza si utilizzano code **multiple con feedback**, il feedback è una retroazione




Dunque si ottengono delle code di diversi livelli (code multilevel-feedback) definite dai parametri:

- Numero di code
- Algoritmo di scheduling per ogni coda
- Metodi usati per l'up-grading e il down-grading di ogni processo
  **Esempio**:
- Tre code:
  - Q0 – RR con time quantum 8 milliseconds
  - Q1 – RR con ime quantum 16 milliseconds
  - Q2 – FCFS
- Scheduling
  - Un nuovo processo entra nella coda Q0 (FCFS).
  - Quando ottiene la CPU, la impegna per 8 ms. Se non termina entro gli 8 ms è spostato in Q1.
  - Il processo in Q1 viene nuovamente servito con politica FCFS e riceve la CPU per ulteriori 16 ms.
  - Se ancora non termina viene spostato in Q2.

Quello che può influenzare il cambiamento di coda di un processo può essere di vario tipo bisogna tenere in considerazione che i processi possono cambiare il tipo di operazioni in

base all'attività dell'utente, questo fa si che un l'operazione di scaricare dei dati di un processo inizialmente in foreground possa trasformarsi in uno background.
Durante l'esecuzione, attraverso la raccolta di dati statistici, il SO può anche determinare se un determinato processo richiede di un elevato utilizzo della CPU

# Linux Scheduling

Linux, come tutti i sistemi UNIX, supporta il multitasking con prelazione;
In tale sistema lo scheduler decide quale processo viene eseguito e quando.
Di norma si pensa allo scheduling come all'esecuzione e sospensione dei processi utente, ma un altro aspetto dello scheduling importante per il sistema Linux è l'esecuzione dei vari task del kernel, che include sia i task richiesti da un processo in esecuzione, sia i task interni eseguiti per conto del kernel stesso, come i task generati dal sottosistema di I/O di Linux.

Il sistema Linux adotta due distinti algoritmi di scheduling: uno è un algoritmo time sharing con prelazione per l'equa condivisione del tempo di CPU da parte di più processi, l'altro è concepito per elaborazioni in tempo reale dove le priorità assolute sono più importanti dell'equità.
L'algoritmo di scheduling adoperato per i task in time sharing ha beneficiato di forti innovazioni rispetto alle sue versioni che non offrivano un supporto adeguati ai sistemi SMP è chiamato **Completely Fair Scheduler (CFS)**.
L'algoritmo di scheduling real-time di Linux è molto più semplice rispetto all'algoritmo fair scheduling utilizzato per i processi time sharing standard.

Linux implementa le due classi di scheduling in tempo reale richieste da posix.1b: CFS e RR.
In entrambi i casi, ogni processo ha una priorità in aggiunta alla sua classe di scheduling.
Lo scheduler esegue sempre il processo con la priorità più alta, oppure, a parità di priorità, il processo che attende da più tempo, l'unica differenza fra lo scheduling FCFS e quello RR è che un processo FCFS continua l'esecuzione fino alla terminazione o finché non si blocca, mentre un processo RR può essere sospeso allo scadere del suo quanto di tempo, in questo caso è posto alla fine della coda di scheduling; la conseguenza è che fra i processi RR della stessa priorità si effettua il time sharing.

Lo scheduling real-time di Linux è in tempo reale:

- Debole (soft)
  - Lo scheduler offre rigide garanzie sulle priorità relative dei processi in tempo reale, ma il kernel non fornisce alcuna garanzia sulla rapidità con cui un processo in tempo reale pronto per l'esecuzione sarà effettivamente eseguito.
- Non in tempo reale stretto (hard)
  - Nei sistemi in tempo reale hard, invece, viene garantito un ritardo massimo entro il quale un processo pronto per l'esecuzione deve essere eseguito.

## Time-sharing

A differenza dei sistemi real-time, questi applicano un concetto di equità, pur utilizzando la priorità per ogni processo. Il concetto di equità è dato in merito alle risorse poiché i processi sono tutti "**uguali**". Per questa ragione, l'algoritmo **non viene utilizzato** dal mondo consumer, ma viene sfruttato per la gestione dei server, per fare un esempio.
Il concetto del time-sharing è condividere il tempo di utilizzo delle risorse disponibili, il suo funzionamento si basa sul **partizionamento** del tempo basato sui crediti e il processo con **più crediti** viene schedulato:
Ad ogni interruzione, dovuta allo scadere del tempo, i crediti vengono decrementati di una unità; quando i credit sono pari a 0, il processo viene sospeso e viene selezionato un altro processo. Periodicamente, i crediti vengono riassegnati a tutti i processi (utente e di sistema) con la regola: $Crediti = crediti/2 + priorità$ .In questo modo i processi blocked acquisiscono maggior priorità.

### Real-time

Il real-time impone al sistema il rispetto di scadenze ferree. Se un'operazione non viene completata entro la deadline, il risultato potrebbe risultare inutile o addirittura dannoso. Ad esempio durante la visione di uno streaming se arriva un pacchetto che rappresenta il frame 2 mentre noi ci troviamo al frame 5, analizzarlo e mostrarlo a schermo sarebbe dannoso poiché ormai non serve più e mostrerebbe un'immagine vecchia.
Gli algoritmi di tipo real-time sono utilizzati nei dispositivi del mondo consumer grazie alla definizione dello standard POSIX1.b.
Alcuni algoritmi possono anche essere di tipo soft real-time, essi non garantiscono il completamento dei processi entro le loro deadlines. In questi casi, il dato arrivato in ritardo può comunque esser utile e dunque non scartato.

La sigla POSIX (ossia interfaccia portabile del sistema operativo) rappresenta una serie di standard creati essenzialmente per sistemi operativi della famiglia UNIX. I sistemi compatibili con POSIX devono implementare lo standard di base (POSIX.1): è questo il caso di Linux e MacOS, esistono poi numerose estensioni degli standard di base, tra queste l’estensione real-time (POSIX1.b) e quella per i thread (POSIX1.c, meglio nota come Pthreads).

## Schedulazione multiprocessore

### Classificazione dei sistemi multi-processore

I sistemi multi-processore sono classificati in base al **tipo di connessione** tra i processori e alla loro gestione:

1. **Processori debolmente accoppiati o distribuiti, o cluster**: ogni processore ha la **propria memoria** e i propri **canali di input/output (I/O)**, un esempio sono i sistemi distribuiti o cluster, dove ogni nodo può operare autonomamente.
2. **Processori specializzati per funzioni**: questi processori svolgono **funzioni specifiche**, come l'input/output (I/O), sono **controllati da un processore master** che coordina il loro

funzionamento e riceve i servizi da questi processori specializzati, un esempio è un processore I/O che si occupa delle operazioni di lettura/scrittura dei dati.

3. **Processori fortemente accoppiati**: i processori **condividono la memoria centrale**,accedono alla stessa memoria, sono **sotto il controllo del sistema operativo**, che gestisce e sincronizza i processi tra i vari processori, un esempio sono i sistemi SMP (Symmetric Multi-Processing), dove tutti i processori lavorano su un unico sistema operativo e risorse condivise.

### Granularità

La **granularità** indica quanto sia stretta l'interazione tra i processori nei sistemi paralleli. Ci sono quattro livelli principali di granularità:

1. **Parallelismo indipendente**: non c'è **sincronizzazione** tra i processi, è tipico dei sistemi **time-sharing**, dove ogni processo può essere indipendente e assegnato a qualsiasi processore. Dal punto di vista dello **scheduling**, non fa differenza se i processi vengono eseguiti su uno o più processori.
2. **Parallelismo a grana grossa e molto grossa**: i processi hanno una **bassa sincronizzazione** tra loro, è paragonabile a un processore **multi programmato**, dove più programmi vengono eseguiti indipendentemente su uno o più processori, l'interazione tra i processi è limitata.
3. **Parallelismo a grana media**: una **applicazione** è suddivisa in **thread** che **interagiscono frequentemente** tra loro, la **frequenza di interazione** tra i thread è maggiore rispetto al parallelismo a grana grossa. Lo **scheduling** diventa critico, poiché una gestione inefficiente può ridurre le prestazioni complessive del sistema.
4. **Parallelismo a grana fine**: è tipico delle applicazioni ad **elevato parallelismo**, dove l'interazione tra processori è molto stretta, è spesso utilizzato in **applicazioni sperimentali o specializzate** per ottenere il massimo delle prestazioni, richiede **alta sincronizzazione** e meccanismi di coordinamento molto efficienti.

Esistono due tipi di politiche di assegnazione dei processi ai rispettivi processori:

- Assegnazione statica: nel momento in cui un processo viene assegnato ad un determinato processore, esso potrà essere eseguito solo e soltanto da quel determinato processore, per ogni processore vi è dunque una coda di ready. Nonostante potrebbe portare a situazioni in cui ci sono processi in coda, ma con processori inutilizzati, questa metodologia di assegnazione evita che i dati, magari presenti nella cache L2 di un processore X, debbano essere spostati nella cache L2 di un altro processore, a causa della coda unica, vi è quindi poco overhead nella schedulazione.
- Assegnazione dinamica à tutti i processori condividono una stessa coda globale. Non si presenta svantaggio se i PCB sono in RAM condivisa.

## Assegnazione dei processi al processore

La caratteristica principale degli SMP è la simmetria perfetta di tutti i core quindi un processo può andare in esecuzione su qualsiasi core.
Se i processori sono tutti **uguali** (SMP) possono essere trattati come un pool di risorse e i processi possono essere eseguiti su ogni processore, l'assegnazione di un processo ad un core può avvenire in due modo:

1. Assegnazione permanente (statica): Un processo viene eseguito sempre su un specifico processore, se assegno il processo i-esimo al core $n$(ad esempio $n = 3$) significa che quel processo può essere eseguito solo su quel core specifico, questo comporta alla creazione di code per ogni core presente, la presenza di più code diminuisce l'overhead nella gestione dello scheduling, se una coda ha terminato i processi da eseguire rimane inutilizzato mentre un altro ha un carico arretrato. Questo tipo di assegnazione è limitante ma ha un vantaggio, infatti è molto probabile che le istruzioni e i dati di un processo sono all'interno della cache L2, invece con una coda unica per ogni core c'è la possibilità di non avere i dati in cache facendo verificare un miss
2. Assegnazione dinamica: Questo tipo di assegnazione presenta una coda globale quindi unica per ogni core, non c'è uno svantaggio rispetto al caso precedente se i PCB sono in RAM condivisa (**area di memoria condivisa ma non cella condivisa**) dunque è immediato, se un processo chiede di essere eseguito e e il core è libero allora verrà eseguito ma se dei processi concorrenti su core diversi vogliono utilizzare un dato presente in memoria condivisa possono generare dei conflitti a seconda del tipo di operazione come read after read / read after write / write after write / write after read
    - Dati due processi, a seconda del tipo di operazioni e su chi arriva prima si può generare o no un conflitto

## Architettura Master/slave

Con questo tipo di architettura viene scelto un processo coordinatore che gestirà le attività degli altri, quindi tutte le istruzioni del kernel verranno eseguite su un particolare processore, inoltre il **master** avrà il compito di occuparsi anche dello scheduling.
Se il master coordina gli altri processi, gli slave inviano richieste di servizio al master, uno slave può anche richiedere dei compiti al master invece di aspettare l'arrivo di nuovi compiti da svolgere
Vantaggi:

- Solo il master ha il controllo della memoria e dei dispositivi di I/O (gestione conflitti semplificata), sono sincronizzati dal SO
  Svantaggi:
- Un fallimento del master determina il fallimento dell'intero sistema, un problema sul core master è bloccante per tutti gli altri
- Il master può diventare il collo di bottiglia delle prestazioni

## Architettura Peer (peer)

Lo scheduling viene realizzato facendo in modo che lo scheduler di ciascun processore esamini la ready queue e selezioni un thread da eseguire.
Si noti che questo approccio offre due possibili strategie per organizzare i thread da selezionare per l'esecuzione:

1. Tutti i thread possono trovarsi in una ready queue comune;
2. Ogni processore può avere una propria coda privata di thread.

Occorre assicurarsi che due processori distinti non scelgano di eseguire lo stesso thread e che i thread nella coda non vengano persi, per prevenire ciò, è possibile utilizzare una forma di lock per proteggere la ready queue comune da questa race condition, ma il lock sarebbe tuttavia molto conteso poiché tutti gli accessi alla coda richiederebbero il possesso del lock e l'accesso alla coda condivisa costituirebbe probabilmente un collo di bottiglia per le prestazioni.
La seconda opzione permette a ciascun processore di schedulare i thread da una coda di esecuzione privata e pertanto non risente dei possibili problemi di prestazioni associati a una coda condivisa, per questa ragione si tratta dell'approccio più comune sui sistemi che supportano SMP.

## Scheduling dei threads

### Applicazione dei threads

I thread sono processi a **peso leggero**, un thread difatti non dispone delle risorse ma è solo una sequenza di istruzioni.
Su un processore singolo i thread permettono di sovrapporre sequenze di I/O e di esecuzione, mentre nei sistemi multiprocessore i thread possono essere eseguiti in parallelo su processori diversi per migliorare le prestazioni complessive

### Approcci per l'assegnazione dei threads

- Local scheduling: ogni processore ha un **proprio carico di lavoro** e i thread sono assegnati localmente a ogni processore, potendo prevedere anche code specifiche per ogni processore
- Gang scheduling: un gruppo di thread correlati viene eseguiti simultaneamente su processori diversi
- Scheduling Threads assegnati ad uno specifico processore: i thread sono associati **in modo statico** a specifici processori, riducendo il costo di contesto (passaggio tra processori) ma rendendolo meno flessibile
- Scheduling dinamico: I thread **non sono assegnati staticamente**, quindi il numero di thread può essere modificato **durante l'esecuzione** per adattarsi ai carichi di lavoro.