Modulo di Sistemi Operativi per il corso di Master
RISS: Ricerca e Innovazione nelle Scienze della Salute
Unisa, 17-26 Luglio 2012



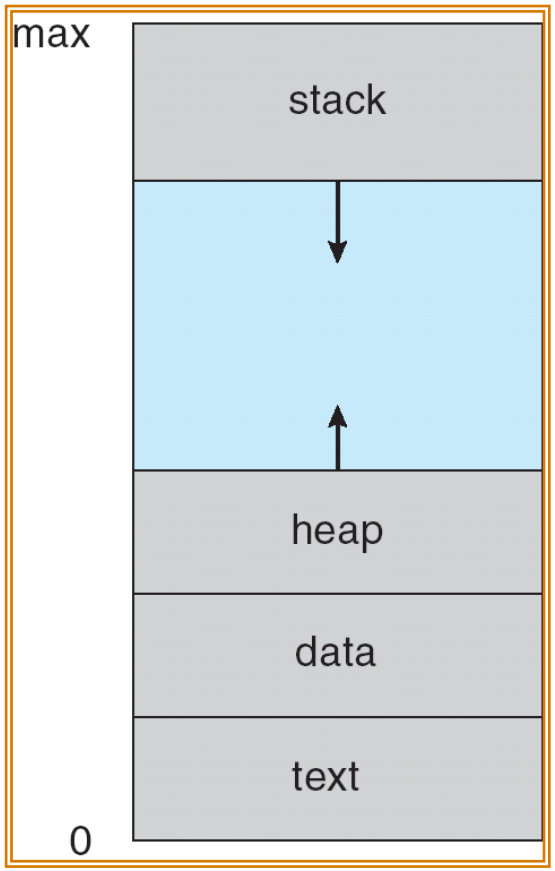
Sistemi Operativi

Rappresentazione e gestione delle attività e della computazione: processi e thread

Processi

- Un SO esegue una varietà di attività:
  - Sistemi batch – job
  - Sistemi time-sharing – programmi utenti o task
- I termini **job** e **processo**: intercambiabili
- Processo – un **programma in esecuzione**; l' esecuzione di un processo avviene in **modo sequenziale**
- Un processo include:
  - program counter
  - contenuto registri CPU
  - sezione testo
  - sezione dati
  - heap
  - stack

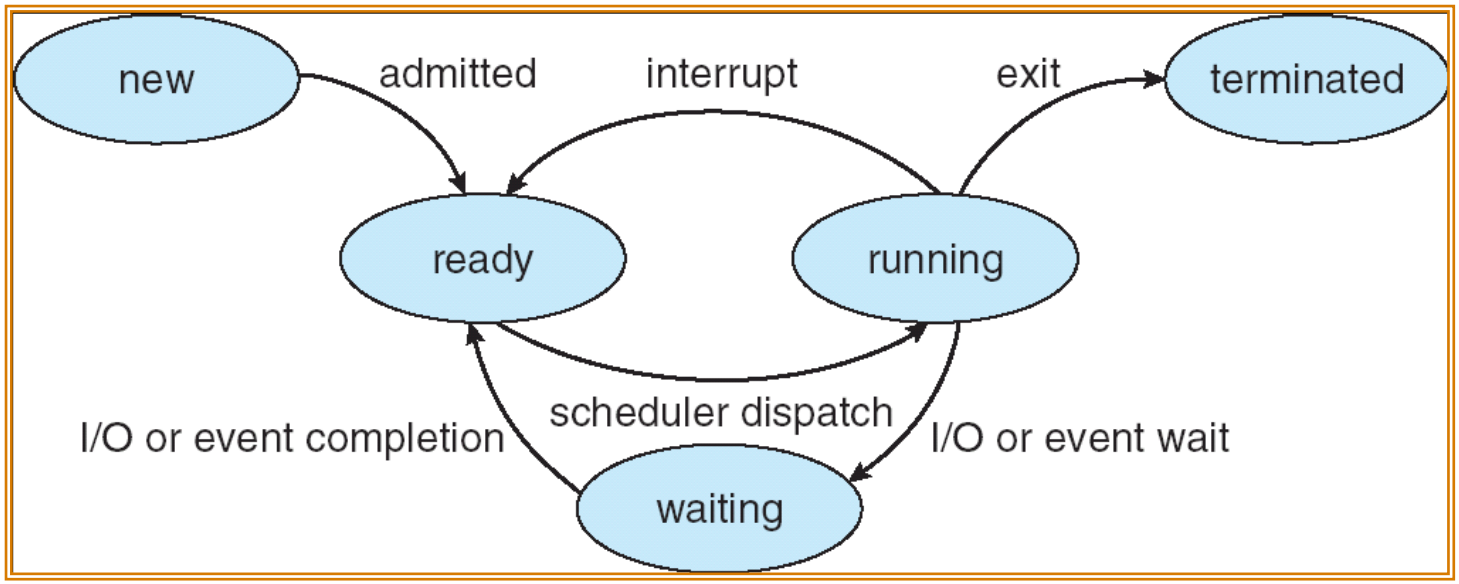
Immagine in memoria di un processo



Stati di un processo

- Durante l'esecuzione un processo cambia stato
  - **new**: Il processo è stato creato
  - **running**: Le sue istruzioni vengono eseguite
  - **waiting**: Il processo è in attesa di qualche evento
  - **ready**: Il processo è in attesa di essere assegnato ad un processore
  - **terminated**: Il processo ha terminato l'esecuzione

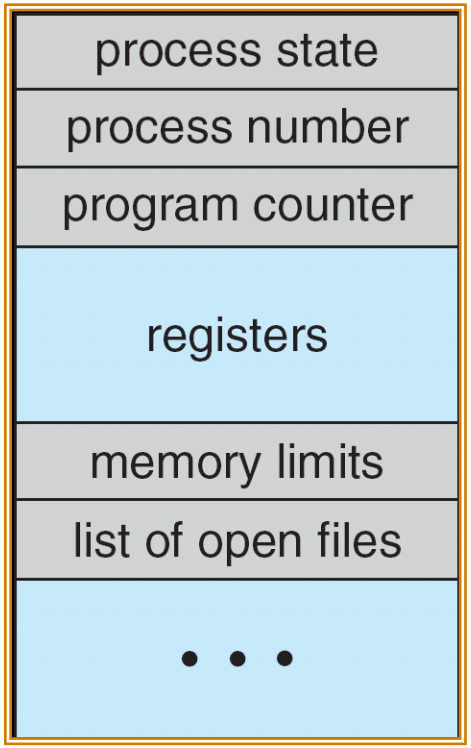
Diagramma di Stato dei Processi



Process Control Block (PCB)

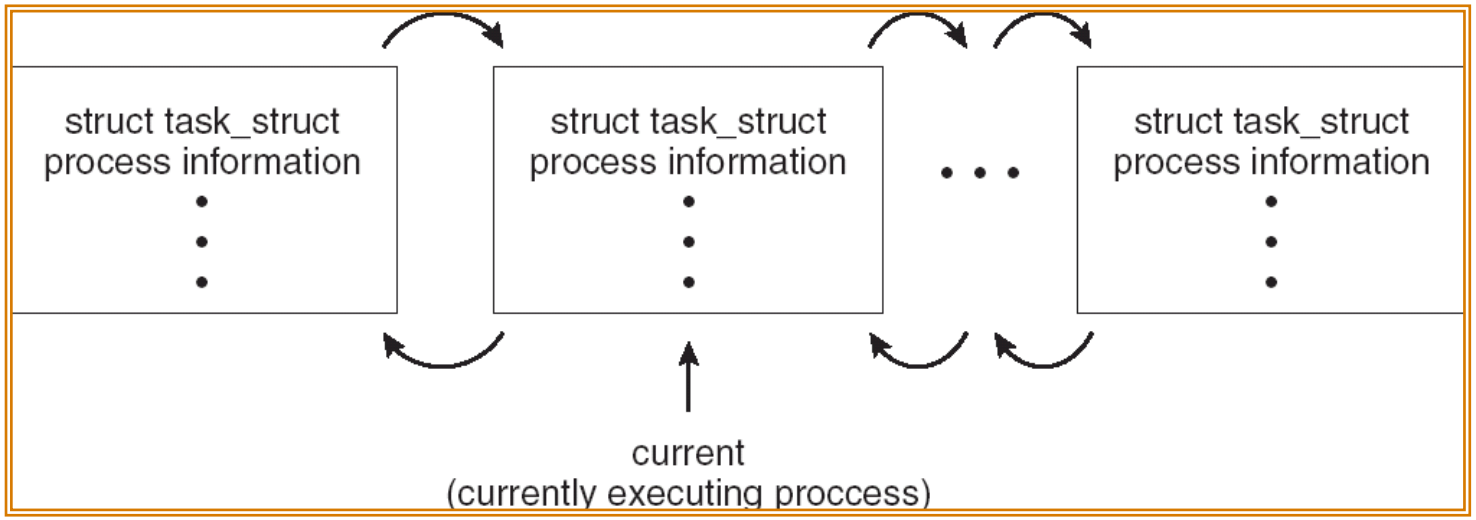
Contiene informazioni per la gestione del processo

- Stato del processo
- Program counter
- Registri della CPU
- Informazioni per lo scheduling CPU
- Informazioni per la gestione della Memoria
- Informazioni di contabilizzazione
- Informazioni sullo stato dell' I/O

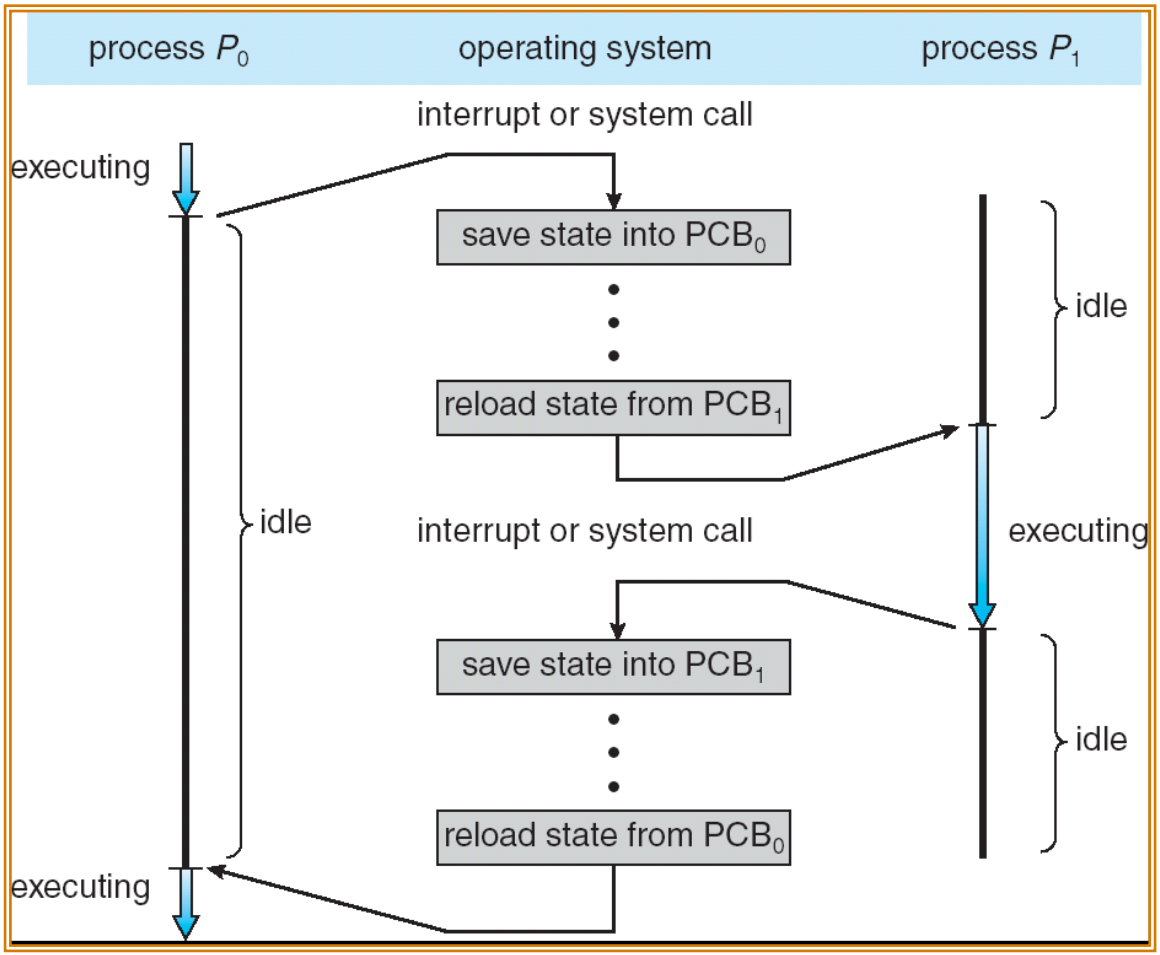


Rappresentazione di un processo in Linux

```
typedef task_struct {
        pid_t pid;                      /* process identifier
        long state;                     /* state
        unsigned int time_slice;        /* scheduling info
        struct files_struct *files;     /* open files
        struct mm_struct *mm;           /* address space
        …
        }
```



La CPU commuta da processo a processo

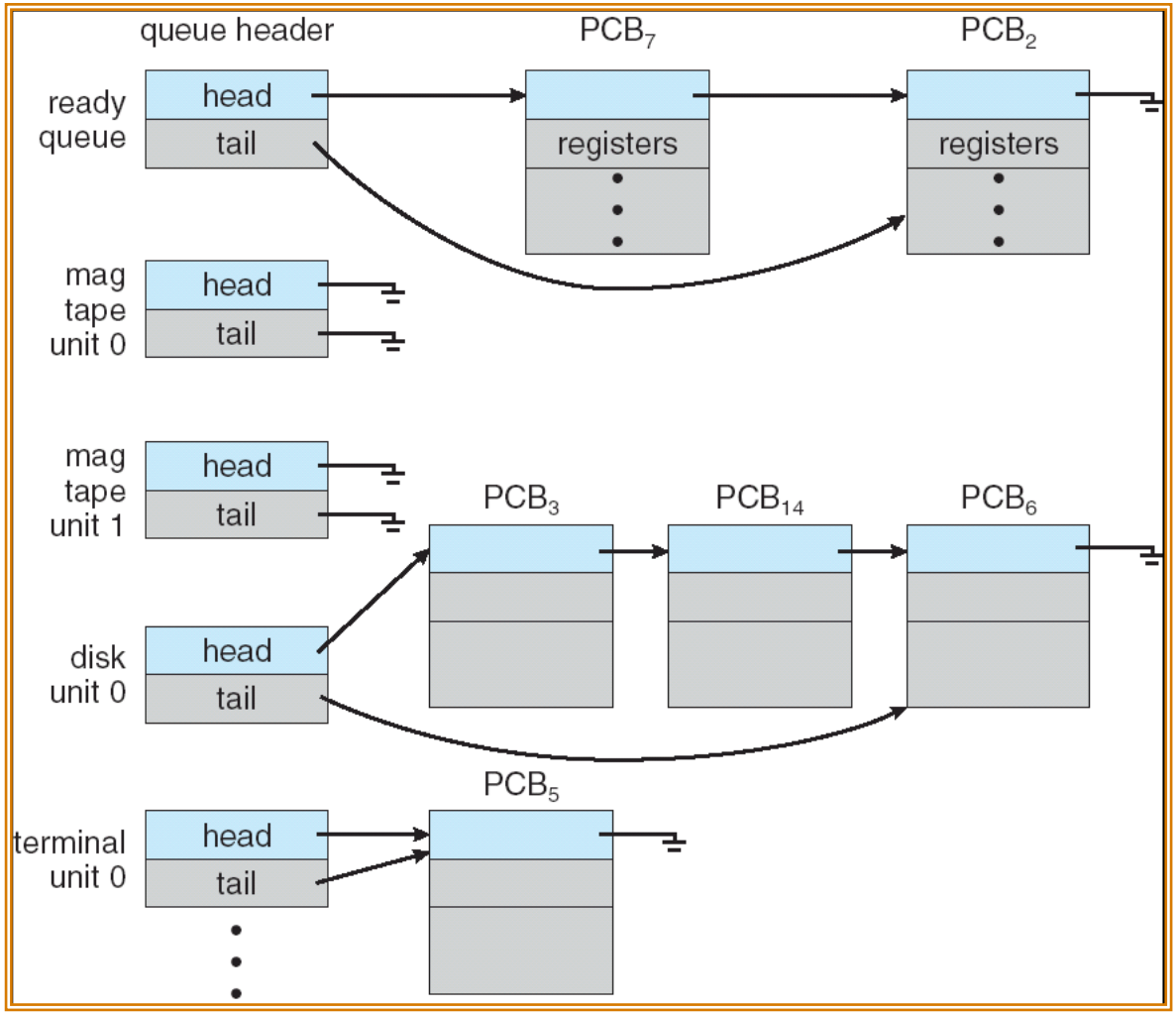


Code di scheduling per i processi

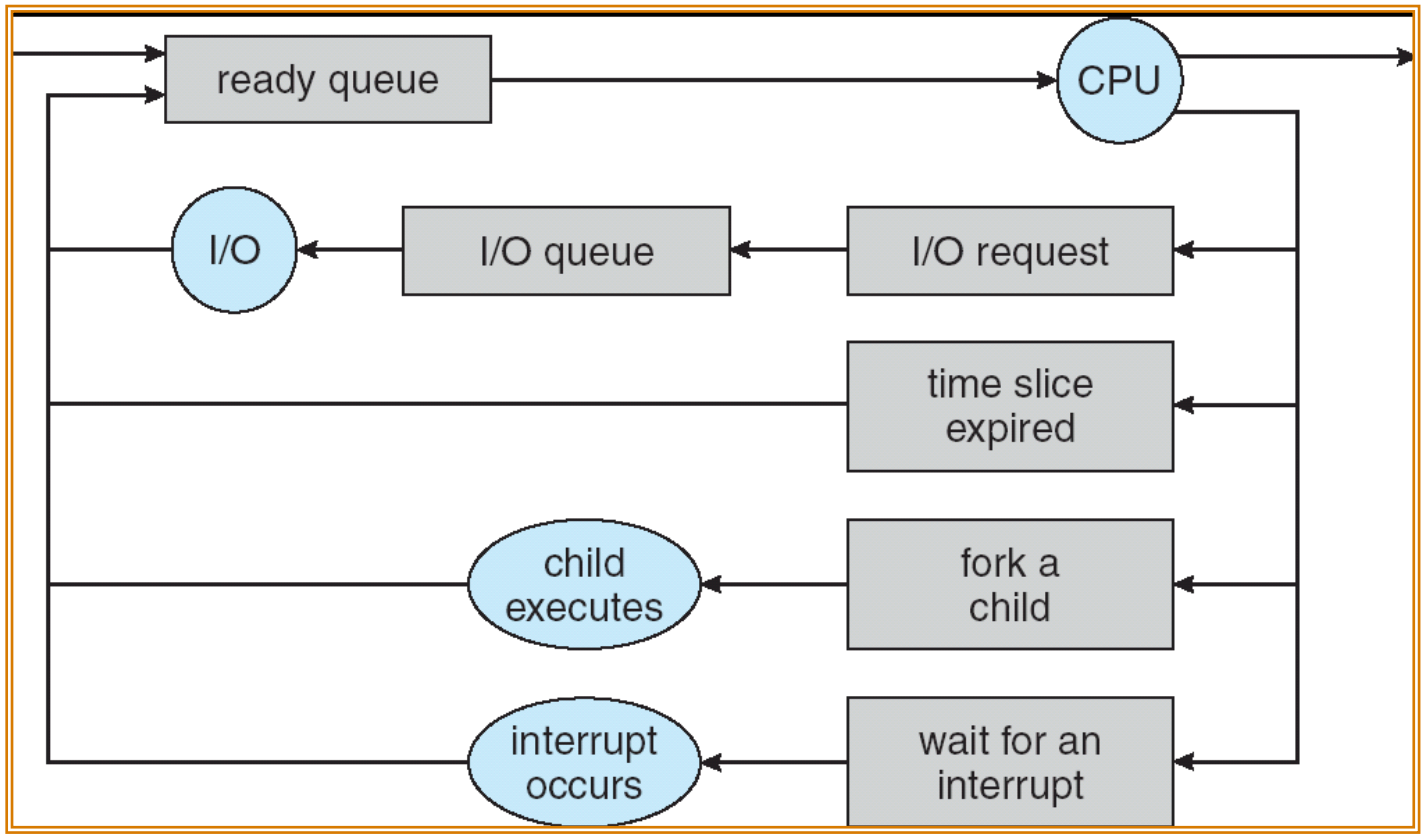
- **Job queue** – insieme di tutti i processi nel sistema
- **Ready queue** – insieme di tutti i processi in memoria centrale pronti per l'esecuzione
- **Code dei dispositivi** – insieme dei processi in attesa di qualche dispositivo di I/O

- I processi, durante la loro vita, **migrano tra varie code**

Ready Queue e varie code di I/O



Ciclo di vita di un processo





Schedulatori

- **Schedulatore a lungo termine** (o job scheduler) – seleziona i processi che devono essere caricati in memoria centrale (ready queue)
- **Schedulatore a breve termine** (o CPU scheduler) – seleziona il prossimo processo che la CPU dovrebbe eseguire

Schedulatori

- Lo schedulatore a breve termine viene invocato molto **frequentemente** (millisecondi) ⇒ deve essere veloce
- Lo schedulatore a lungo termine viene invocato **raramente** (secondi, minuti) ⇒ può essere più lento
- Lo schedulatore a lungo termine controlla il **grado di multiprogrammazione**
- I processi possono essere descritti come:
  - **Processi I/O-bound** – consumano più tempo facendo I/O che computazione, contengono molti e brevi CPU burst
  - **Processi CPU-bound** – consumano più tempo facendo computazione; contengono pochi e lunghi CPU burst

Sequenza alternata di CPU burst e di I/O burst

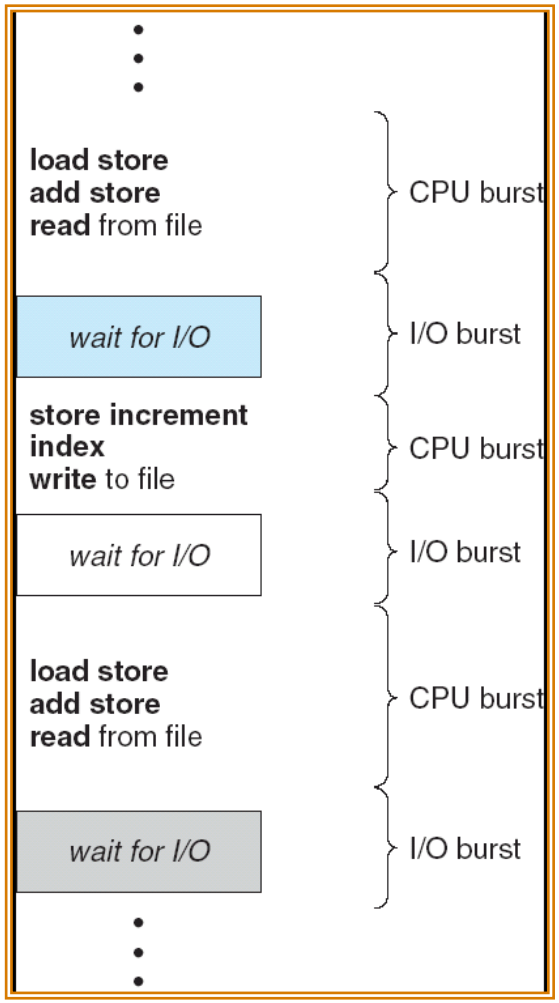
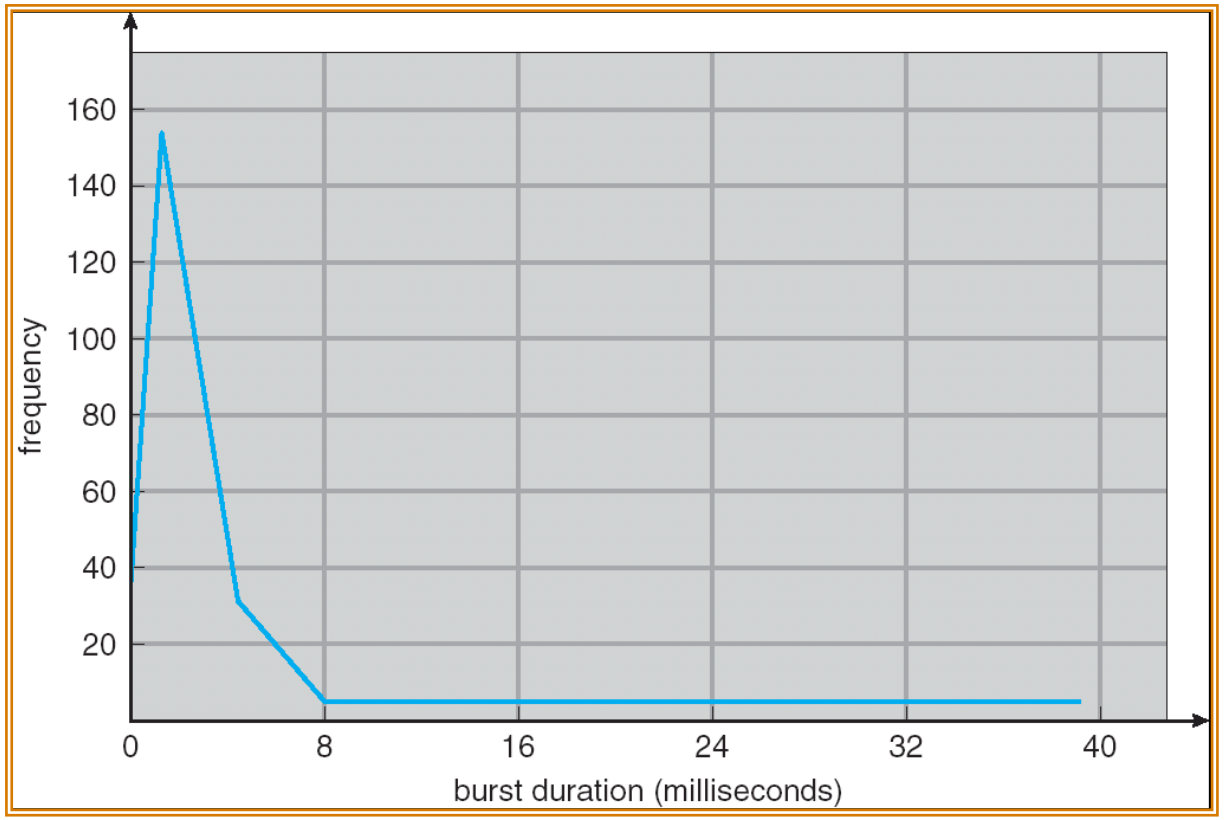


Diagramma delle durate dei CPU burst



Lo scheduler della CPU

- Il SO sceglie fra i processi in memoria che sono pronti per l'esecuzione ed assegna la CPU ad uno di essi.

- Lo scheduling della CPU può avvenire quando un processo:

  1. Passa dallo stato di esecuzione allo stato di attesa.
  2. Passa dallo stato di esecuzione allo stato di pronto.
  3. Passa dallo stato di attesa allo stato di pronto.
  4. Termina.

- La schedulazione nei punti 1 e 4 è detta **nonpreemptive** (senza diritto di **prelazione**).
- Tutte le altre schedulazioni sono dette **preemptive**.

Dispatcher

- Il **dispatcher** è il modulo del SO che dà il controllo della CPU ad un processo selezionato dallo scheduler. Questa funzione comprende:
  - cambio di contesto;
  - passaggio alla modalità utente;
  - salto alla corretta locazione nel programma utente per ricominciarne l'esecuzione.
- **Latenza del dispatcher** – tempo necessario al dispatcher per fermare un processo e cominciarne un altro. Dipende dal supporto hardware.

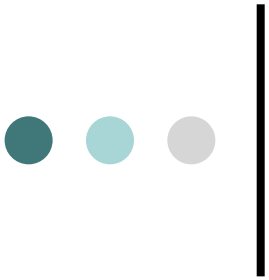
Criteri di scheduling

- **Utilizzo della CPU** – mantenere la CPU il più impegnata possibile.
- **Frequenza di completamento** (**throughput**) – numero di processi completati per unità di tempo.
- **Tempo di completamento** (**turnaround time**) – intervallo che va dal momento dell'immissione del processo nel sistema al momento del completamento.
- **Tempo di attesa** (**waiting time**)– somma dei tempi spesi in attesa nella coda dei processi pronti.
- **Tempo di risposta** – tempo che intercorre dalla formulazione della prima richiesta fino alla produzione della prima risposta, **non** l'output (per gli ambienti di time-sharing).

Criteri di ottimizzazione

- Massimizzare l'utilizzo della CPU.
- Massimizzare il throughput.
- Minimizzare il turnaround time.
- Minimizzare il tempo di attesa.
- Minimizzare il tempo di risposta.

Solitamente si ottimizzano i valori medi. A volte è opportuno ottimizzare i valori minimi o massimi. Per i sistemi interattivi andrebbe minimizzata la varianza



Algoritmi di scheduling

First-Come First-Served (FCFS)

Processo	CPU burst
P_1	24
P_2	3
P_3	3

- Se i processi arrivano all'istante 0 nell'ordine: P_1 , P_2 , P_3
  si ottiene il risultato mostrato nel seguente diagramma di Gantt:



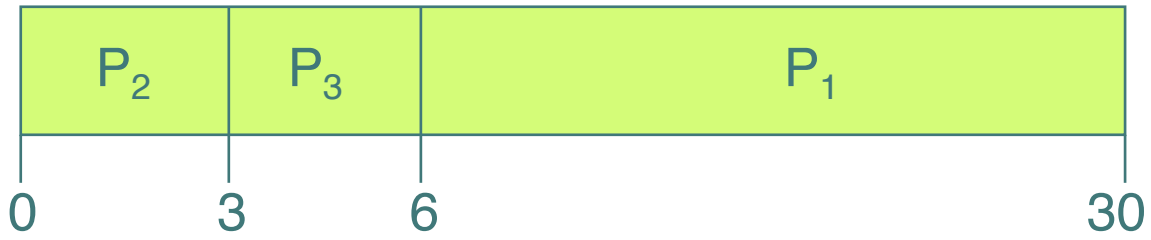
- Tempo di attesa per P_1 = 0; P_2 = 24; P_3 = 27
- Tempo di attesa medio: (0 + 24 + 27)/3 = 17

Caratteristiche di FCFS

Se i processi arrivano nell'ordine:

P_2 , P_3 , P_1

- Il diagramma di Gantt è:



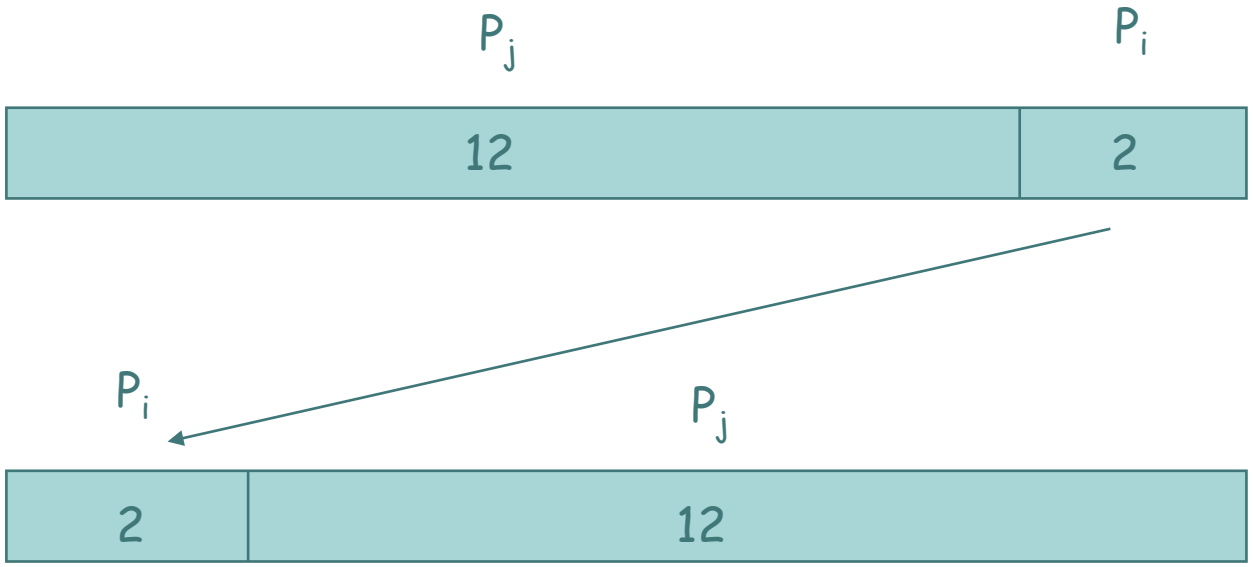
- Tempo di attesa per P_1 = 6; P_2 = 0; P_3 = 3
- Tempo di attesa medio: (6 + 0 + 3)/3 = 3
- Molto meglio del caso precedente.
- C'è un effetto di ritardo a catena (**convoy effect**) mentre tutti i processi attendono che quello grosso rilasci la CPU.

Shortest-Job-First (SJF)

- Schedula il processo con il prossimo CPU burst più breve.

- L'algoritmo SJF può essere:
  - **nonpreemptive** – quando un processo ha ottenuto la CPU, non può essere prelazionato fino al completamento del suo cpu-burst.
  - **preemptive** – quando un nuovo processo è pronto ed il suo CPU-burst è minore del tempo di cui necessita ancora il processo in esecuzione c'è prelazione. Questa schedulazione è anche detta **shortest-remaining-time-first**.
- SJF è ottimale – fornisce il minor tempo di attesa medio per un dato gruppo di processi.

SJF Ottimale

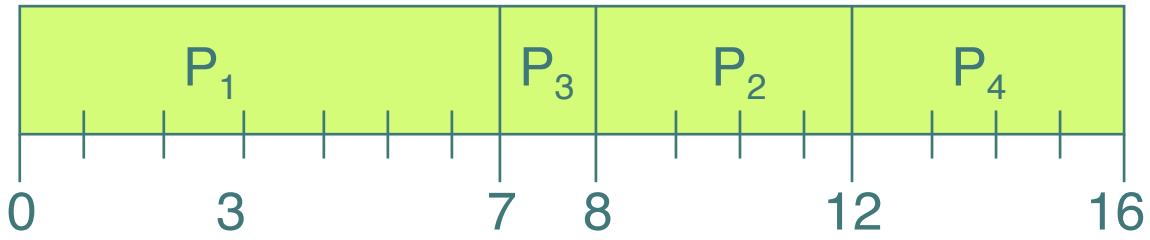


Spostando un processo breve prima di un processo lungo, il tempo di attesa del processo breve diminuisce più di quanto aumenti il tempo d'attesa per il processo lungo. Di conseguenza il tempo d'attesa medio diminuisce.

SJF Non-Preemptive

Processo	Tempo di arrivo	CPU burst
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)



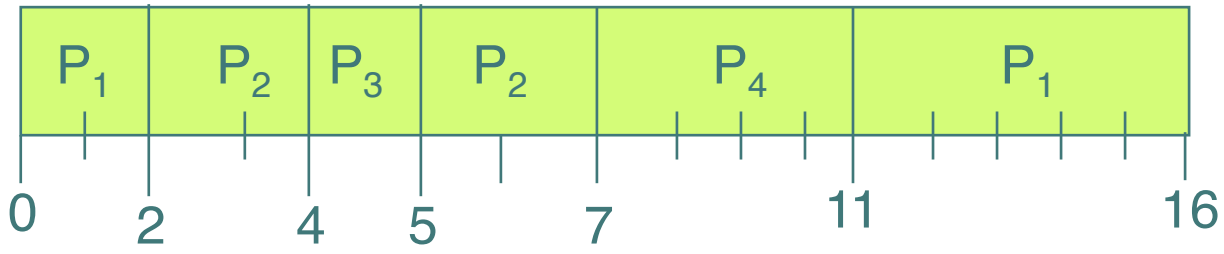
- Tempo di attesa medio = (0 + 6 + 3 + 7)/4 = 4

8-2 7-4 12-5

SJF Preemptive

Processo	Tempo di arrivo	CPU burst
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



- Tempo di attesa medio = (9 + 1 + 0 +2)/4 = 3

Stima del prossimo CPU burst

- É possibile fare solo una stima della lunghezza.
- Può essere fatta utilizzando la lunghezza dei precedenti CPU burst, usando una media esponenziale.

t_n = valore reale dell'ennesimo CPU burst
τ_{n+1} = valore previsto per il prossimo CPU burst
α = parametro con valore $0 \leq \alpha \leq 1$
τ_1 = previsione 1° CPU burst (valore di default)

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

Scheduling a priorità

- Si associa una priorità numerica a ciascun processo.
- La CPU viene allocata al processo con priorità più alta (più piccolo è il numero ≡ più alta è la priorità).
  - preemptive
  - nonpreemptive
- SJF è un algoritmo a priorità dove la priorità è il prossimo picco (previsto) di CPU.
- Problema ≡ Blocco indefinito (**starvation**) – processi a bassa priorità potrebbero non essere mai eseguiti.
- Soluzione ≡ Invecchiamento (**aging**) – accrescere gradualmente le priorità dei processi nel sistema.

Round Robin (RR)

- Ogni processo riceve la CPU per una piccola unità di tempo (**time quantum**), generalmente 10-100 millisecondi. Se entro questo arco di tempo il processo non lascia la CPU, viene interrotto e rimesso nella coda dei processi pronti.

- Se ci sono **n** processi nella coda dei processi pronti e il quanto di tempo è **q**, allora ciascun processo ottiene 1/**n** del tempo di CPU in parti lunghe al più **q**. Ciascun processo non deve attendere più di (**n** – 1) x **q** unità di tempo.

- Prestazioni:
  - **q** grande ⇒ FIFO
  - **q** piccolo ⇒ **q** deve essere grande rispetto al tempo di un context switch, altrimenti l'overhead diventa troppo elevato.

Esempio di RR con quanto di tempo pari a 20 millisecondi

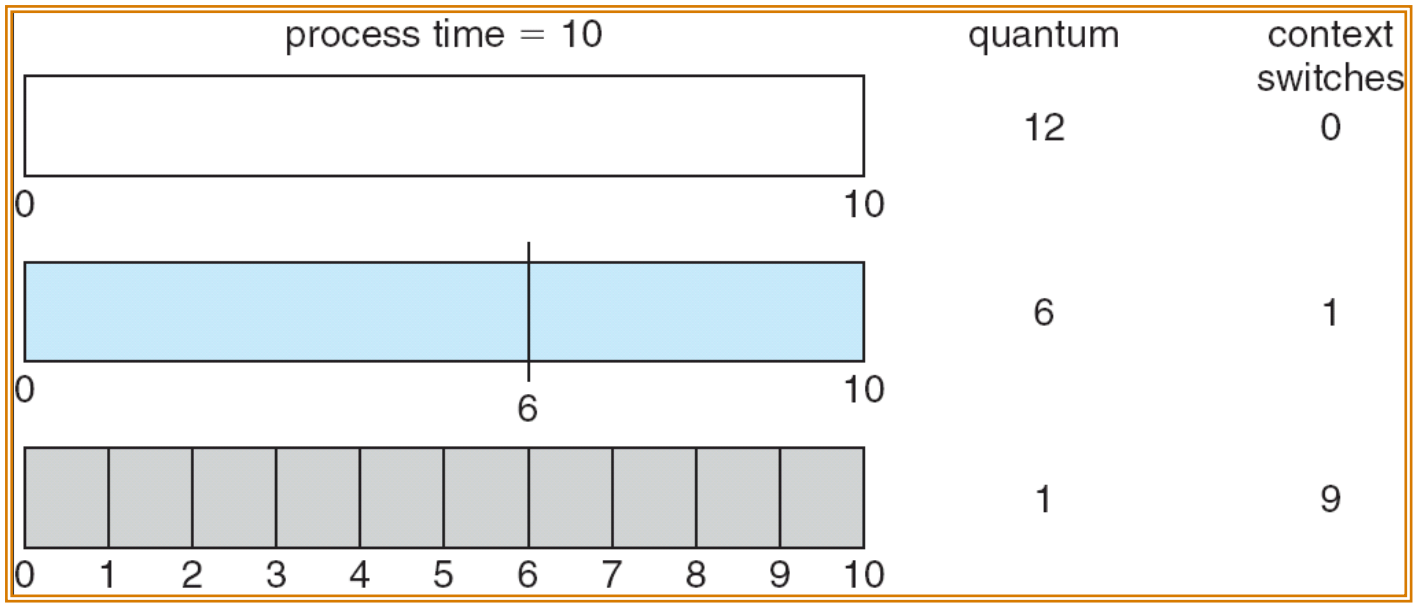
Processo	CPU burst
P_1	53
P_2	17
P_3	68
P_4	24

- Il diagramma di Gantt è:



- Tipicamente, il tempo medio di turnaround è più alto di SJF, ma il tempo di risposta è più breve.

Quanto di tempo e context switch



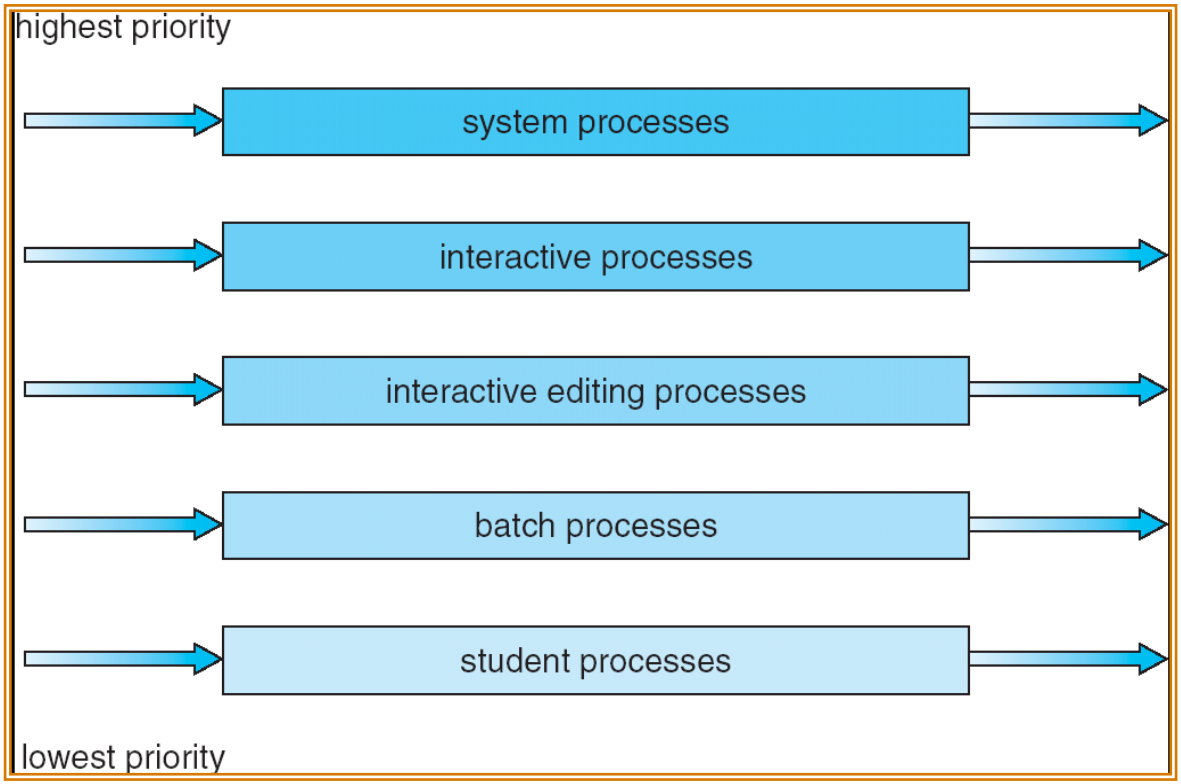
Scheduling a code multiple

- La coda dei processi pronti è partizionata in code separate:
  foreground (interattivi),
  background (batch – sullo sfondo).

- Ciascuna coda ha il proprio algoritmo di scheduling:
  - foreground – RR
  - background – FCFS

- Ci deve essere una schedulazione tra le code
  - A priorità fissa; (e.g., tutti i processi in foreground, poi quelli in background). Possibilità di starvation.
  - Time slice – ciascuna coda ha una certa quantità di tempo di CPU, che può schedulare fra i processi in essa contenuti;e.g., 80% del tempo di CPU per la coda foreground (RR), 20% background in FCFS.

Scheduling a code multiple



Code multiple con feedback

- Un processo può muoversi tra le varie code; l'aging potrebbe essere implementato in questo modo.

- Uno schedulatore a code multiple con feedback è definito dai seguenti parametri:

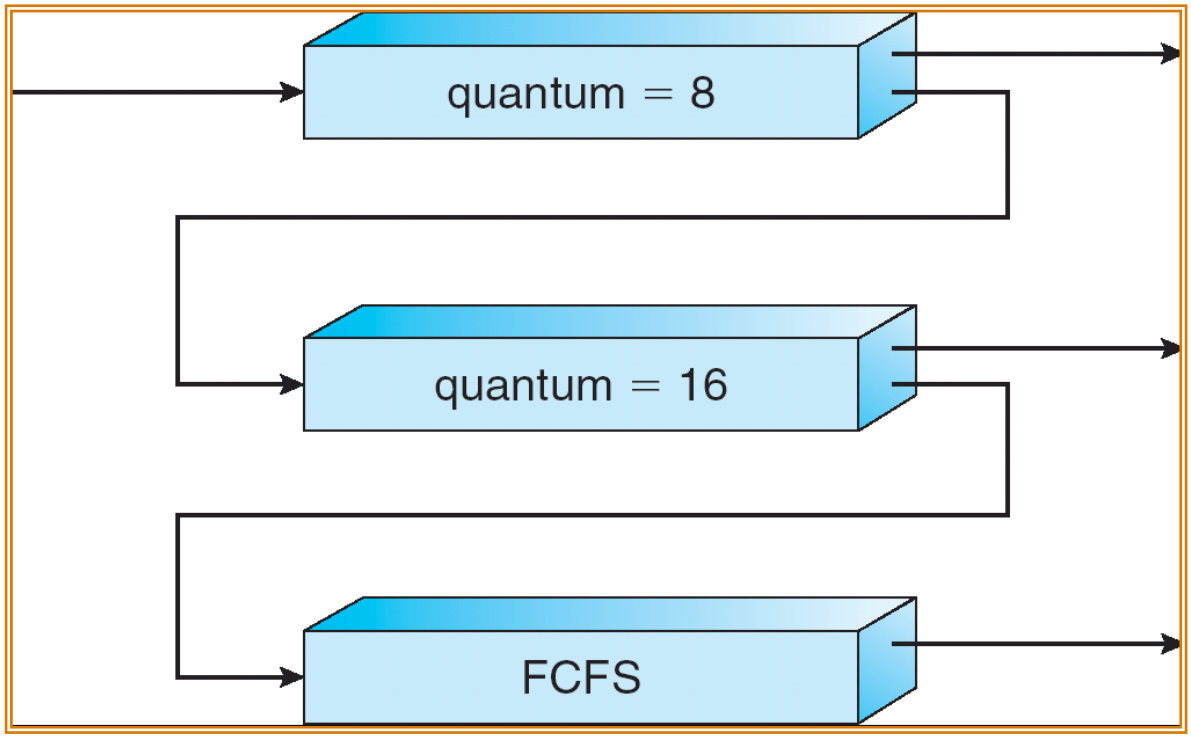
  - numero di code;
  - algoritmo di schedulazione per ciascuna coda;
  - metodo utilizzato per "far salire" un processo verso una coda a priorità più alta;
  - metodo utilizzato per spostare un processo in una coda a più bassa priorità;
  - metodo utilizzato per determinare in quale coda entrerà un processo quando avrà bisogno di un servizio.

Esempio di code multiple con feedback

- Tre code:
  - $\mathbf{Q_0}$ – RR con quanto di tempo: 8 millisecondi
  - $\mathbf{Q_1}$ – RR con quanto di tempo: 16 millisecondi
  - $\mathbf{Q_2}$ – FCFS

- Schedulazione
  - Un nuovo processo entra nella coda $\mathbf{Q_0}$. Quando schedulato ottiene la CPU per 8 millisecondi. Se non termina in 8 millisecondi, viene spostato nella coda $\mathbf{Q_1}$.
  - In $\mathbf{Q_1}$ il processo, quando schedulato, riceve la CPU per 16 millisecondi. Se non completa entro I 16 millisecondi, viene spostato nella coda $\mathbf{Q_2}$.

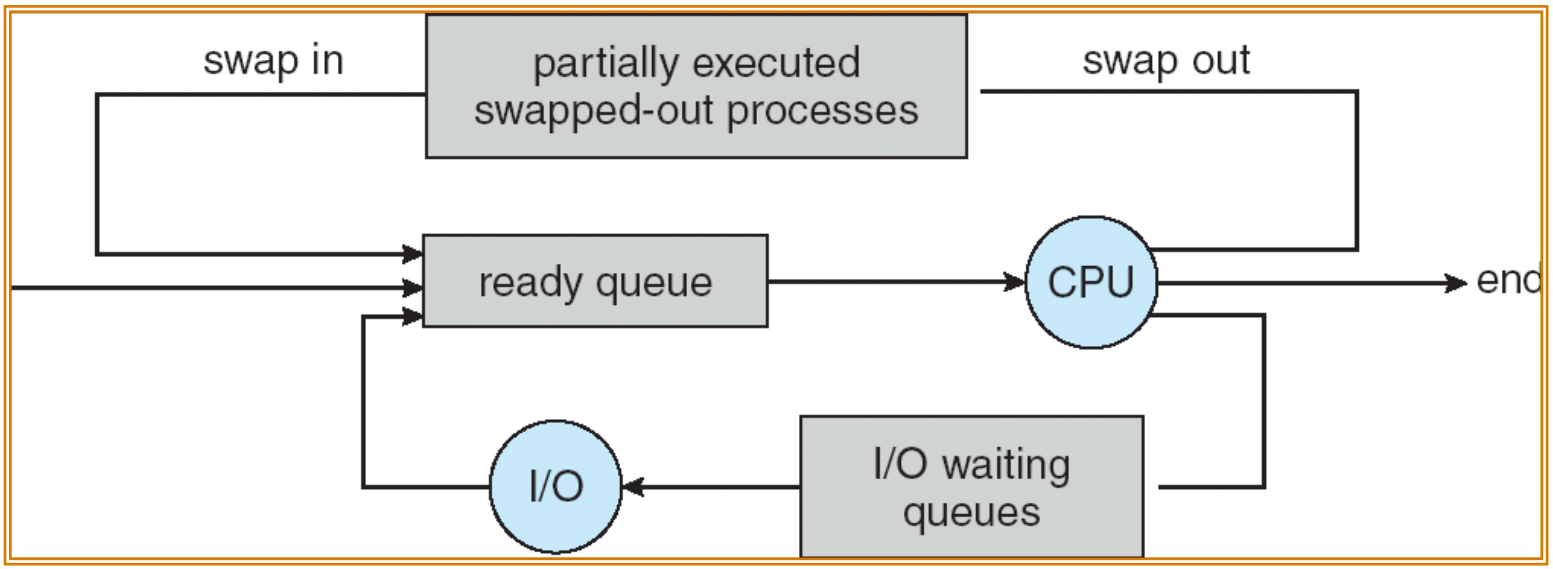
Code multiple con feedback



Scheduling in Solaris

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

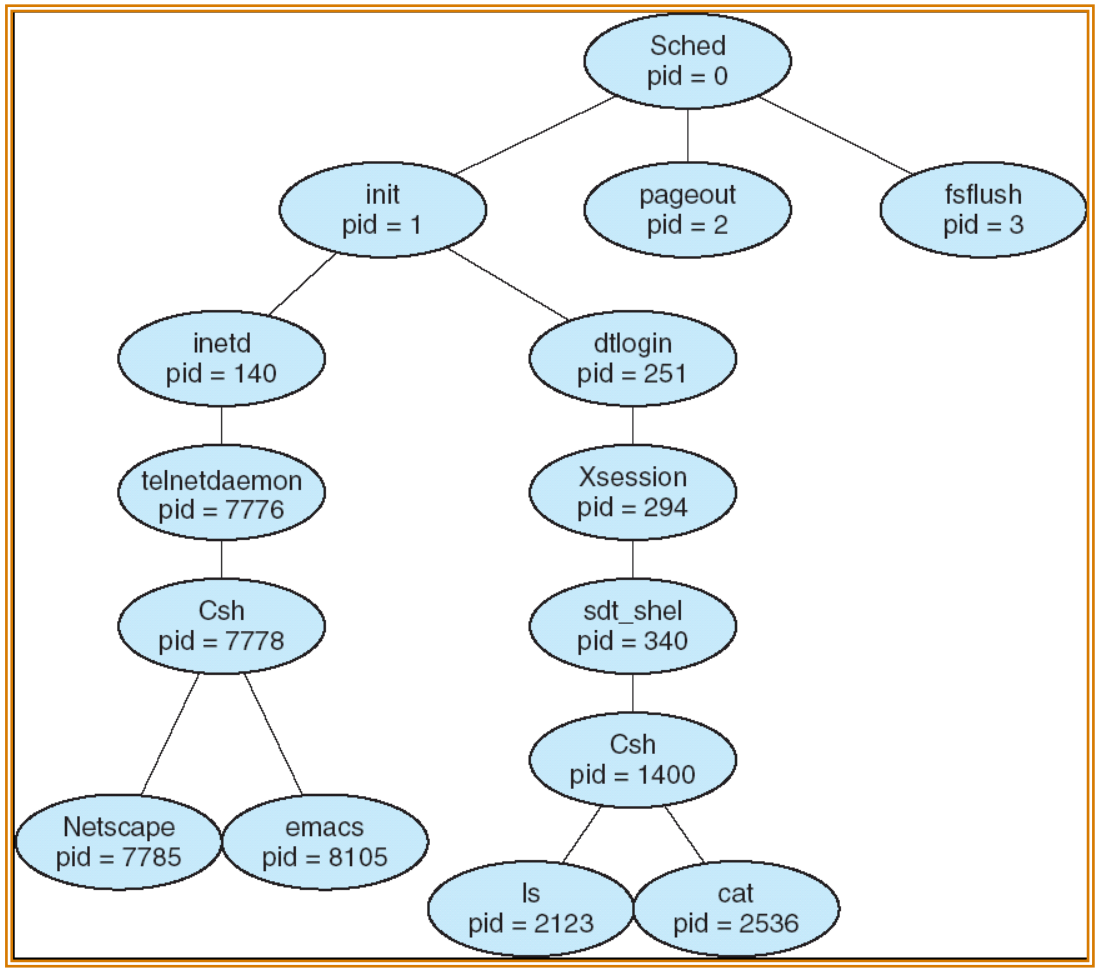
Schedulatore a Medio Termine



Creazione di processi

- Il sistema operativo fornisce meccanismi per creare e terminare processi
- Un processo padre crea processi figli che, a loro volta, creano altri processi, formando un albero di processi
- Ogni processo ha un identificatore (process identifier) PID

Un albero di processi di Solaris



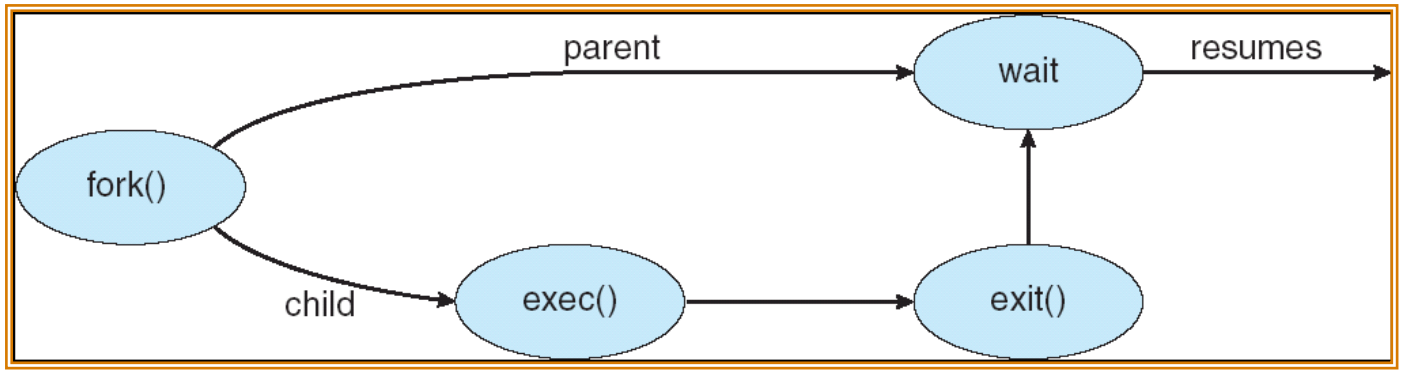
Creazione di processi: implementazione

- Condivisione di risorse
  - Padre e figli condividono tutte le risorse
  - I figli condividono un sottoinsieme delle risorse del padre
  - Padre e figlio non condividono niente
- Esecuzione
  - Padre e figli vengono eseguiti in concorrenza
  - Padre aspetta fino alla terminazione dei figli

Creazione di processi: implementazione

- Spazio di indirizzamento
  - Lo spazio del figlio è un duplicato di quello del padre
  - Il figlio carica in esso un nuovo programma
- Esempi UNIX
  - La chiamata **fork** crea un nuovo processo
  - La chiamata **exec** viene usata dopo una **fork** per sostituire l'immagine in memoria del processo con un nuovo programma
  - La chiamate **wait** permette al padre di aspettare che il figlio termini

Creazione di un processo



Programma C che crea un processo figlio - Unix

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

int main( )
{
pid_t pid;

        /* fork a child process */
        pid = fork();

        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed\n");
                exit(-1);
        }
        else if (pid == 0) { /* child process */
                printf("I am the child %d\n",pid);
                execlp("/bin/ls","ls",NULL);
        }
        else { /* parent process */
                /* parent will wait for the child to complete */
                printf("I am the parent %d\n",pid);
                wait(NULL);

                printf("Child Complete\n");
                exit(0);
        }
}
```

Terminazione di Processi

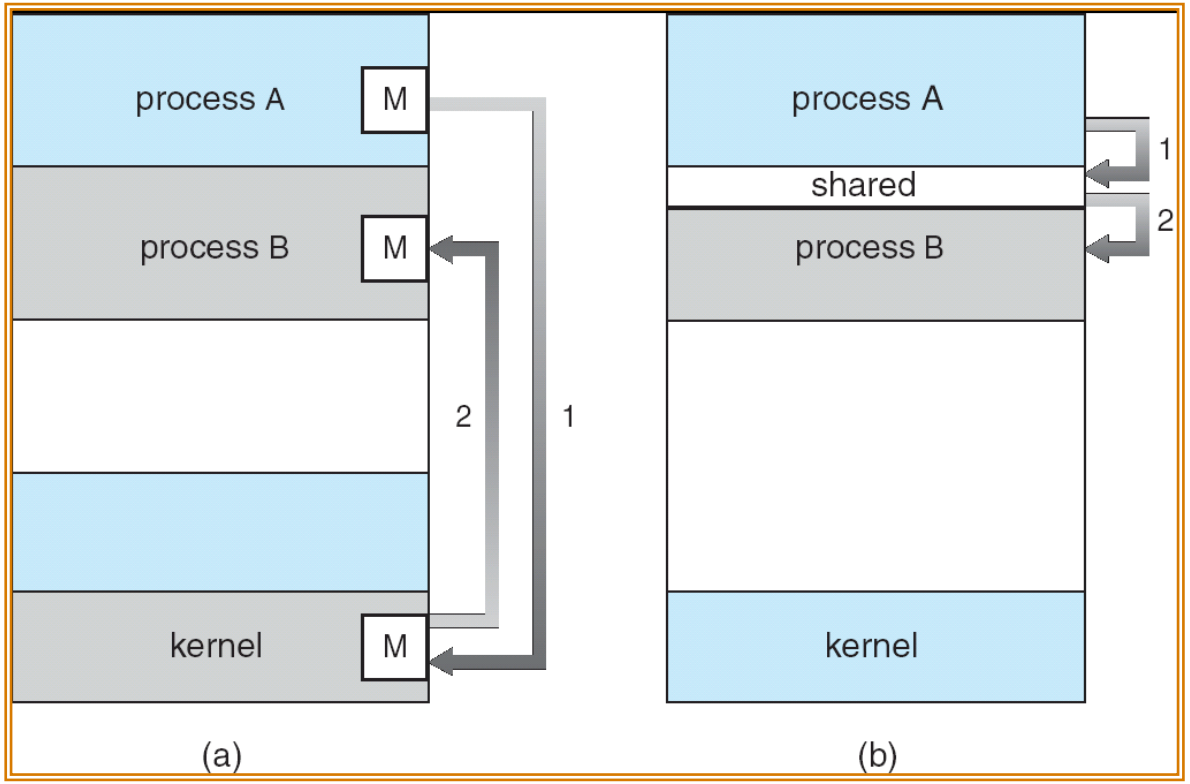
- Un processo esegue l'ultima istruzione e chiede al SO di eliminarlo (**exit**)
  - Trasmette un valore di output al padre (via **wait**)
  - Le risorse del processo sono recuperate dal SO
- Il processo padre può terminare l'esecuzione dei processi figli (**abort**)
  - Un figlio ha ecceduto nell'uso delle risorse
  - Il compito assegnato al figlio non è più richiesto
  - Se il padre ha terminato
    - Alcuni SO non permettono ai processi figli di continuare
    - Tutti i figli vengono terminati – terminazione a **cascata**

Processi cooperanti

- **Processi independenti** non possono influenzare o essere influenzati dall'esecuzione di altri processi
- **Processi cooperanti** possono essere influenzati dall'esecuzione di altri processi
- Vantaggi della cooperazione tra processi
  - Condivisione di informazione
  - Accelerazione delle computazioni
  - Modularità
  - Convenienza

Interprocess Communication (IPC)

Meccanismi per la comunicazione e la sincronizzazione tra processi

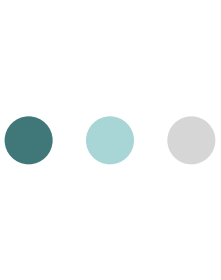


Produttore-Consumatore

- Nel paradigma dei processi cooperanti, un processo **produttore** produce informazioni che vengono consumate da un processo **consumatore**
  - **Buffer-illimitato** non pone limiti alla taglia del buffer
  - **Buffer-limitato** assume che la taglia del buffer sia fissata

Scambio di messaggi

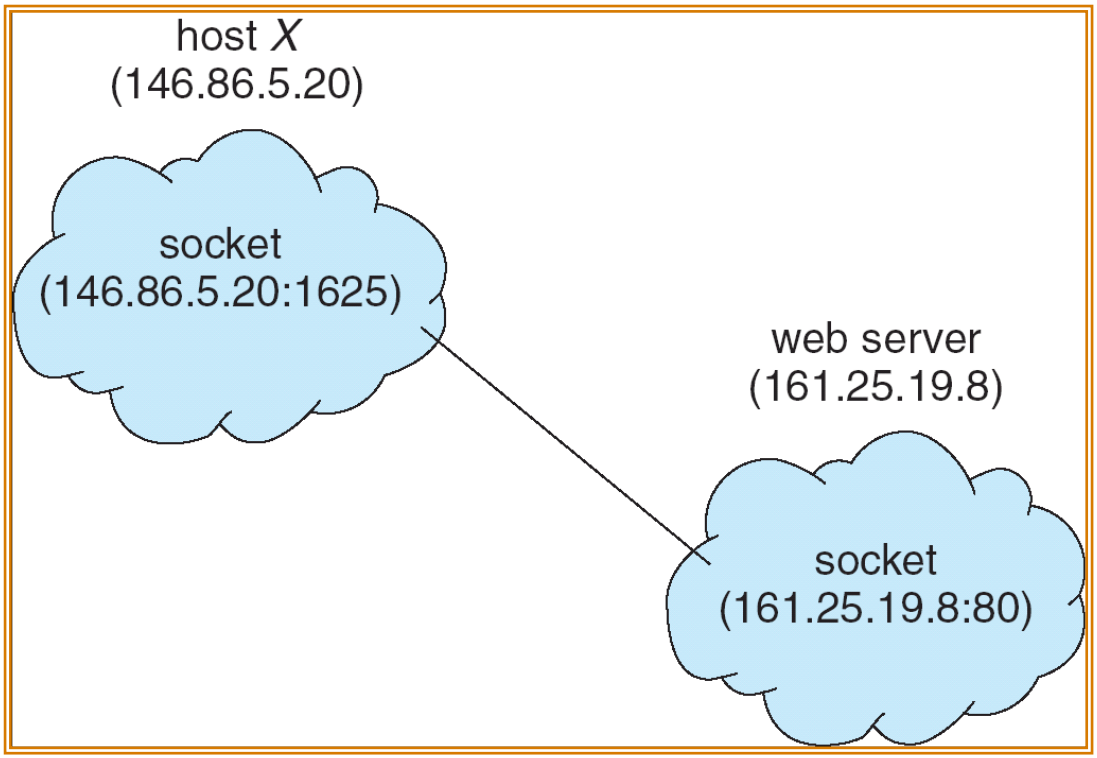
- Sistema di messaggi – I processi comunicano tra loro senza ricorrere a variabili condivise
- IPC fornisce due operazioni:
  - **send(message)** – message di taglia fissa o variabile
  - **receive(message)**
- Se **P** e **Q** desiderano comunicare, hanno bisogno di:
  - Stabilire un **canale di comunicazione** tra loro
  - Scambiare messaggi via send/receive



Comunicazioni Client-Server

- Socket
- Remote Procedure Call
- Remote Method Invocation (Java)

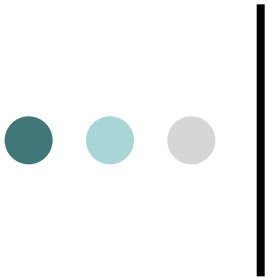
Comunicazione attraverso socket



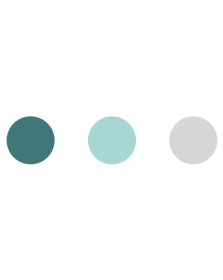
Comunicazione di basso livello: flusso non strutturato di bit

RPC vs RMI

- Le RPC seguono il paradigma della programmazione procedurale e permettono di invocare funzioni e procedure remote
- Le RMI sono orientate agli oggetti e consentono di invocare metodi su oggetti remoti
- I parametri per le procedure remote sono normali strutture di dati
- In RMI si possono passare oggetti come parametri ai metodi remoti



Thread

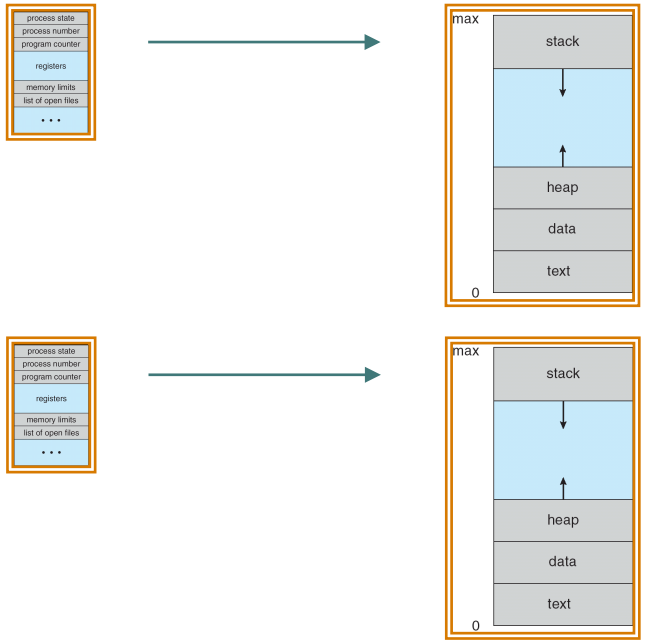


Applicazioni reali

- Browser web
  - Rappresentazione dati sullo schermo
  - Più finestre che caricano dati diversi
  - Reperimento di dati dalla rete
- Elaboratore testi
  - Acquisizione dati da tastiera
  - Visualizzazione sullo schermo
  - Correttore ortografico durante battitura
- Server web
  - Richieste multiple da gestire

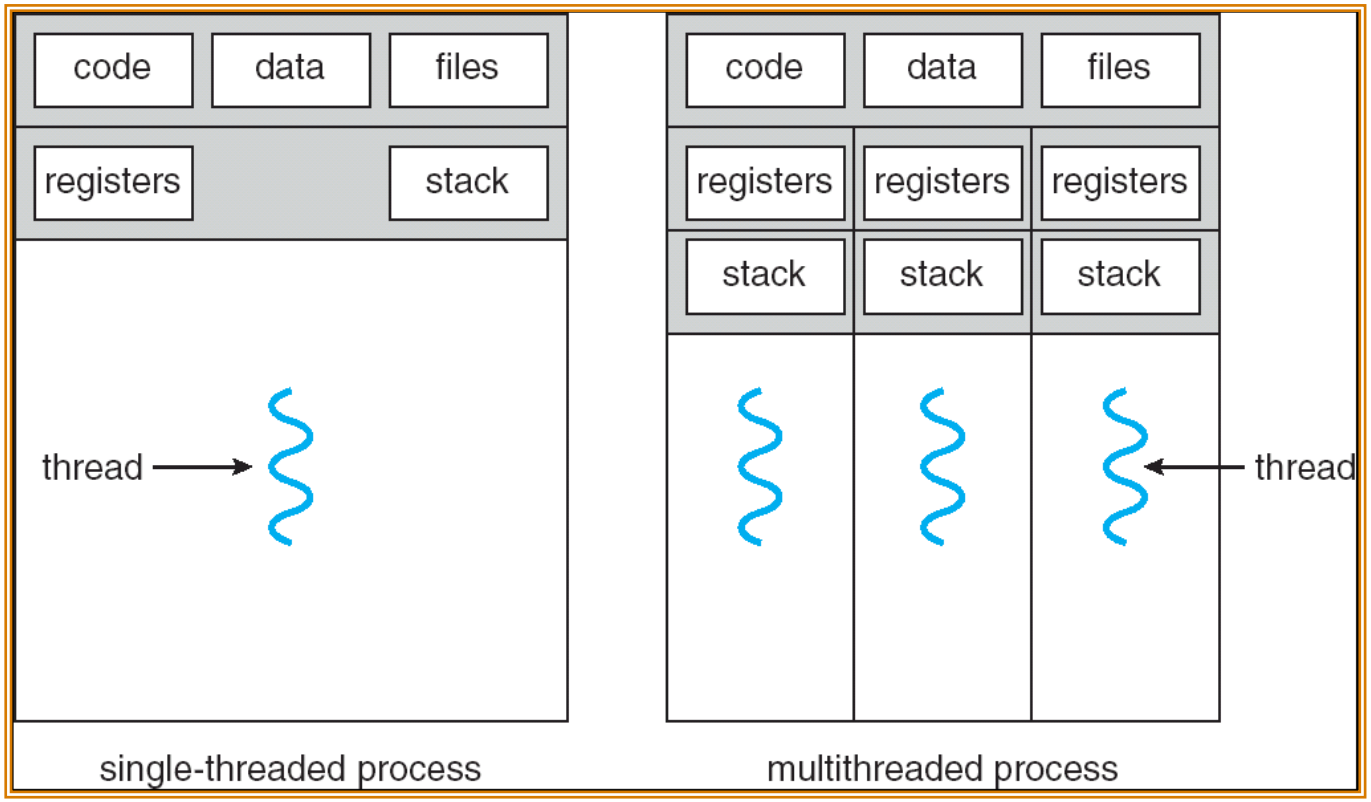
Creazione di nuovi processi

fork()

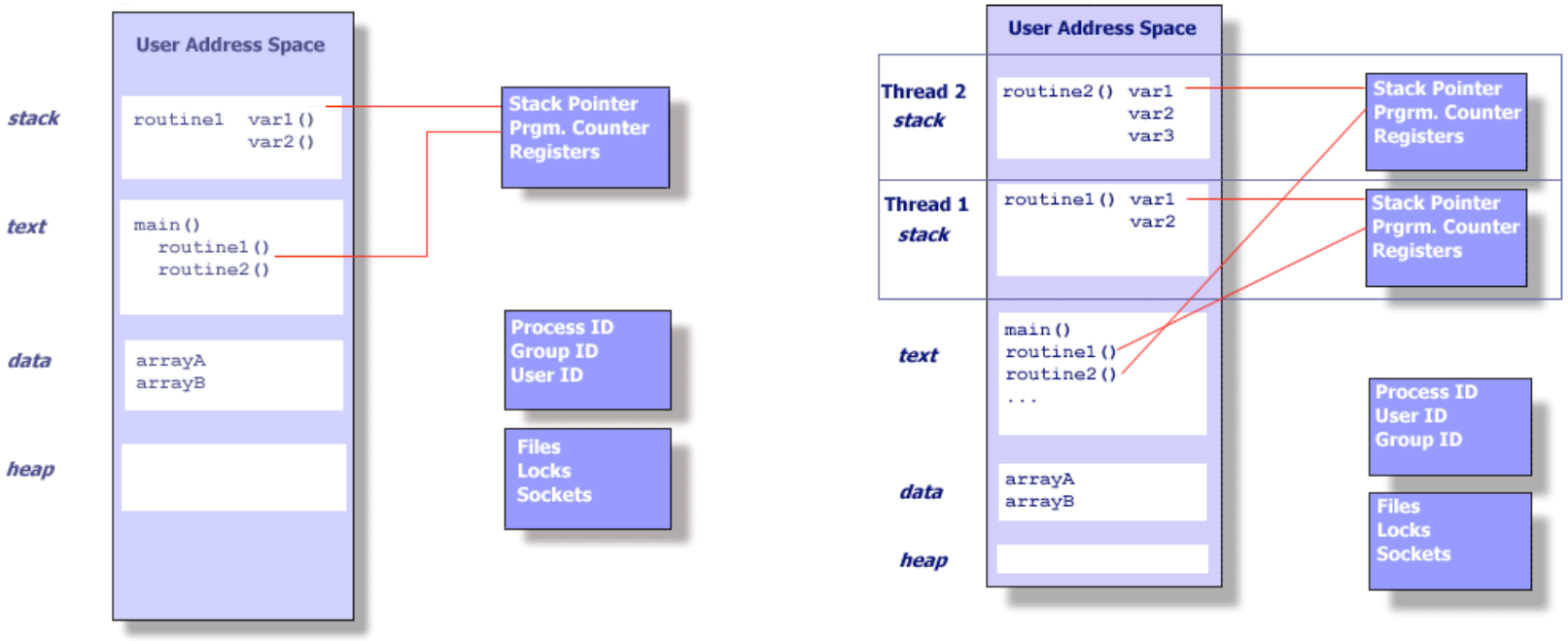


La creazione di un nuovo processo costa
I tempi di context switch sono elevati

Processi a singolo thread e processi multithread



Unix: processo a singolo thread e processo multithread





Benefici

- Prontezza di risposta
- Condivisione delle risorse
- Economia
- Uso di architetture multiprocessore

Thread vs Processi

La tabella che segue compara i tempi richiesti dall'esecuzione di **fork()** e **pthread_create()**, rispettivamente. I tempi riguardano la creazione di 50.000 processi/thread e sono espressi in secondi

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
IBM 375 MHz POWER3	61.94	3.49	53.74	7.46	2.76	6.79
IBM 1.5 GHz POWER4	44.08	2.21	40.27	1.49	0.97	0.97
IBM 1.9 GHz POWER5 p5-575	50.66	3.32	42.75	1.13	0.54	0.75
INTEL 2.4 GHz Xeon	23.81	3.12	8.97	1.70	0.53	0.30
INTEL 1.4 GHz Itanium 2	23.61	0.12	3.42	2.10	0.04	0.01

User Thread e Kernel Thread

- I thread possono essere distinti in thread a livello utente e thread a livello kernel
- I primi sono gestiti "senza l'aiuto" del kernel
- I secondi sono gestiti direttamente dal sistema operativo
- In ultima analisi deve esistere una relazione tra i thread utente e i thread del kernel

Problemi associati alla concorrenza

- L'accesso concorrente a dati condivisi può portare all'inconsistenza dei dati
- Sono pertanto necessari meccanismi che assicurino un'esecuzione ordinata dei processi cooperanti
- Oggetto di altri moduli del master … ☺