Fondamenti dei Sistemi Operativi

Manager della CPU e dei processi

Sommario

- Il concetto di processo
- Esecuzione sequenziale e concorrente di un programma
- Processi cooperanti: il paradigma produttore/consumatore
- Informazione di stato di un processo: il Process Control Block (PCB)
- La commutazione di contesto computazionale (Context Switch)
- Code di schedulazione
- Creazione, terminazione ed esecuzione di un processo
- Job Scheduler: algoritmi FCFS e SJF
- CPU Scheduler: il dispatcher
- CPU Scheduler: algoritmi RR, MRR, Priorità statica, Priorità dinamica
- Thread: il concetto, la struttura, i vantaggi, le applicazioni, il thread package
- Modelli di un multi-thread: *many-to-one*, *one-to-one*, *many-to-many*
- Organizzazioni di un multi-thread: *dispatcher-worker*, *team-line*, *pipe-line*
- Java threads

Fasi dell'esecuzione di un programma

submit fase in cui viene richiesta l'esecuzione del programma

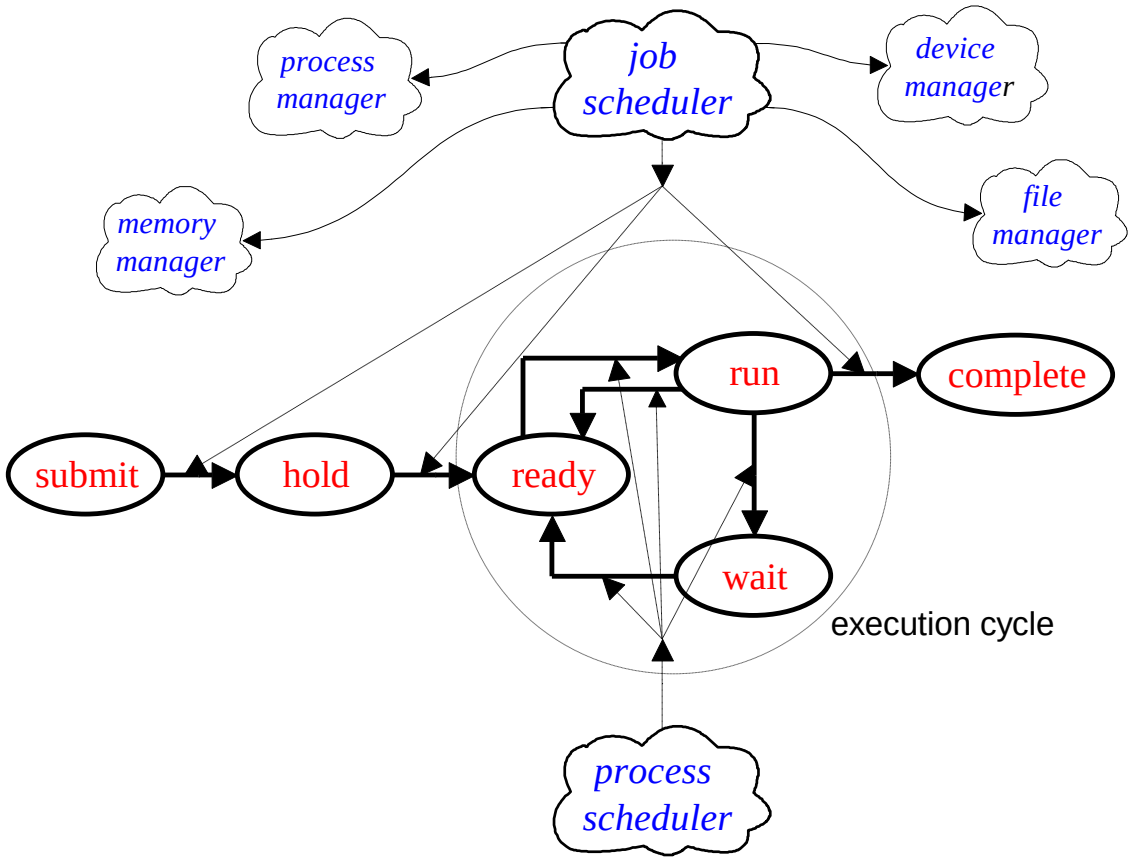
hold fase di incodamento delle richieste di esecuzione in attesa delle risorse necessarie (livello di multitasking, memoria, dispositivi, file)

ready fase di attesa della risorsa CPU da parte dei processi

run fase di utilizzo della CPU (un solo processo alla volta)

wait fase di attesa di eventi (I/O, disponibilità di risorse condivise, completamento processi cooperanti, ecc.)

complete fase di rilascio delle risorse



The process concept

Abbiamo visto che: un ***job*** è costituito da uno o più job-step (programmi) da eseguire in sequenza; un ***job-step*** (o programma) è costituito da uno o più processi (o task) che operano «contemporaneamente»; un ***task*** (o processo) è una attività elementare ed indipendente, con associate le risorse (memoria, file, dispositivi) richieste per la sua esecuzione.

Spesso, invece, si fa uso in maniera quasi intercambiabile dei termini *job*, *job-step*, *task*.

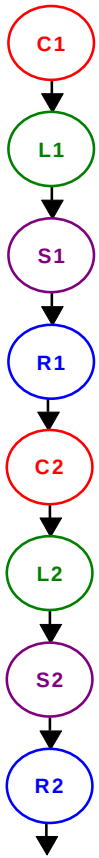
Ma qual è il vantaggio di suddividere un programma in diversi processi concorrenti?

Sequential execution

Realizzazione sequenziale di un sistema di acquisizione dati

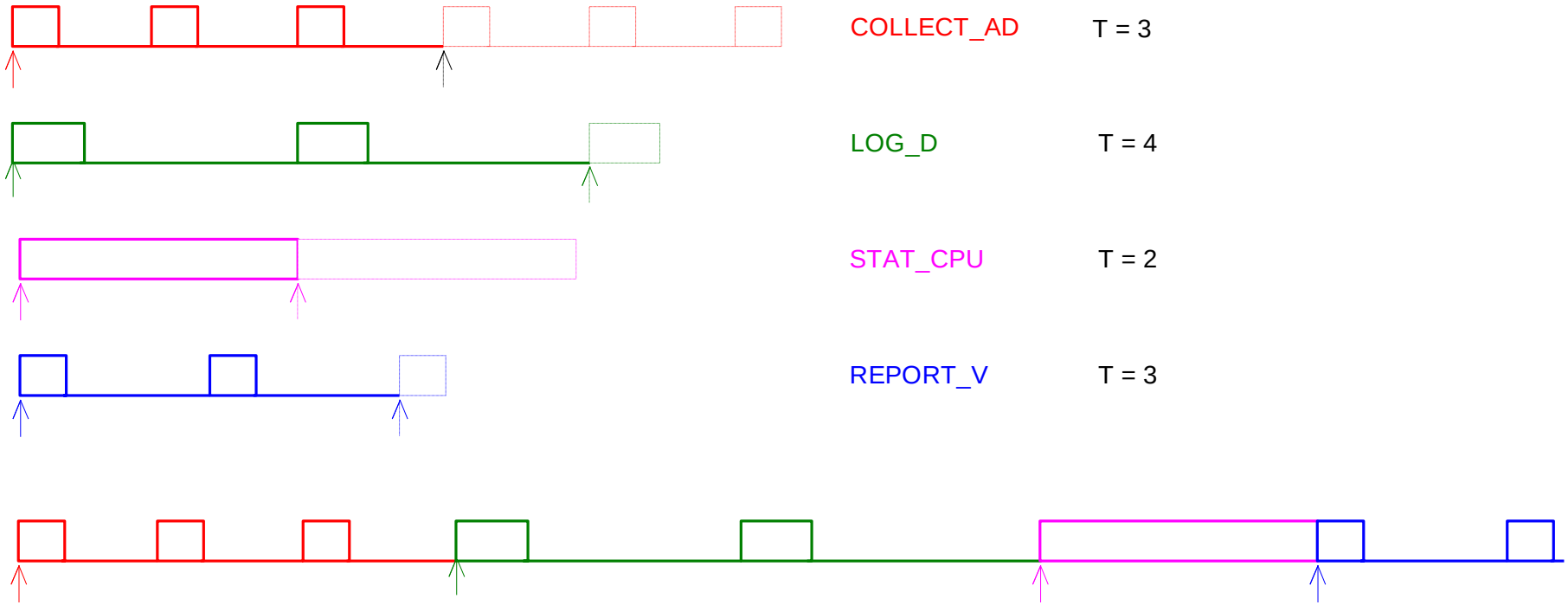
```
program          Acquisizione_sequenziale
   :
   :
     while true     [loop infinito]
        collect;    [lettura da convertitore analog-digital]
        log;        [memorizzazione su disco]
        stat;       [elaborazione statistica]
        report;     [visualizzazione risultati]
     end            [fine del loop]
end                 [fine Acquisizione_sequenziale]
```

Grafo delle precedenze



Sequential execution

Timing hypothesis



Execution time

COLLECT_AD	T = 3
LOG_D	T = 4
STAT_CPU	T = 2
REPORT_V	T = 3
TOTAL CYCLE	T = 12

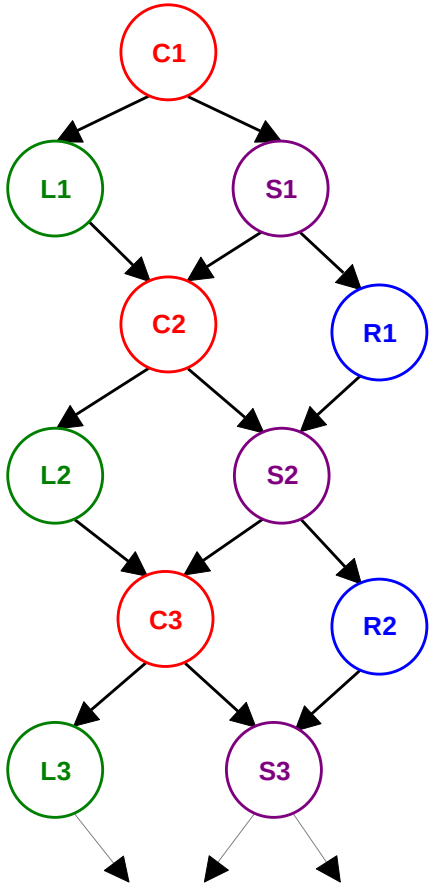
Concurrent execution

Realizzazione concorrente di un sistema di acquisizione dati

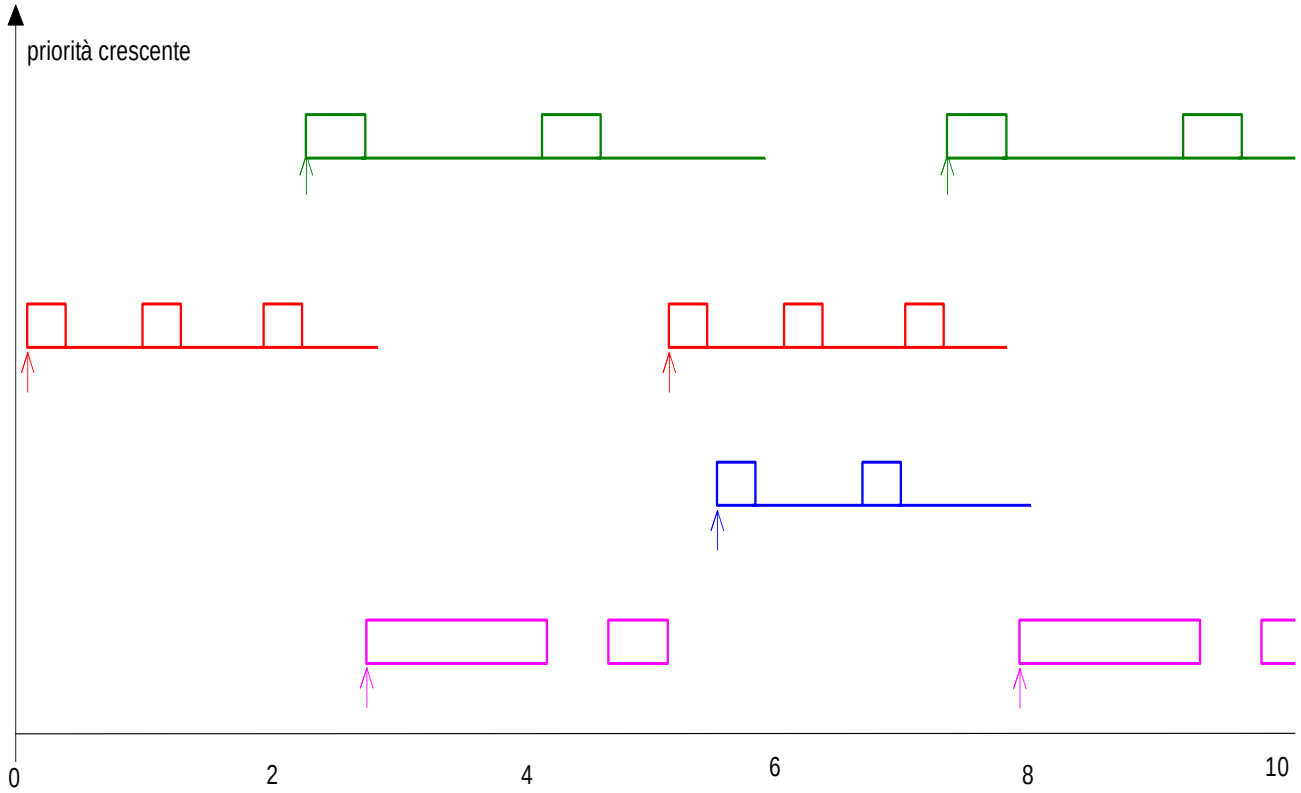
```
module   Acquisizione_concorrente
   :
   process collect;
       while true
        wait_signal_from (log,stat);
        collect_ad;
        send_signal_to (log.stat);
       end  [fine del loop]
   end;     [fine collect]
   process log;
       while true
        wait_signal_from (collect);
        log_d;
        send_signal_to (collect);
       end  [fine del loop]
   end;     [fine log]
   process stat;
       while true
        wait_signal_from (collect,report);
        stat_cpu;
        send_signal_to (collect,report);
       end  [fine del loop]
   end;     [fine stat]
   process report;
       while true
        wait_signal_from (stat);
        report_v;
        send_signal_to (stat);
       end  [fine del loop]
   end;     [fine report]
```

Grafo delle precedenze

Un solo buffer di accoppiamento tra processi



Concurrent execution

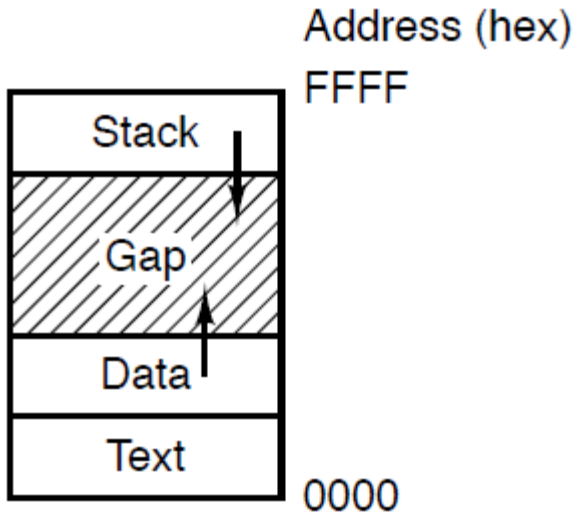


Cooperating Processes

- Un processo ***indipendente*** non può influenzare o essere influenzato dall'esecuzione di un altro processo.
- Un processo ***cooperante*** può influenzare o essere influenzato dall'esecuzione di un altro processo.
- Vantaggi della cooperazione tra processi
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience
- Paradigma dei processi cooperanti: un processo ***produttore*** crea l'informazione che viene depositata in un *buffer* e utilizzata da un processo ***consumatore***.
  - *unbounded-buffer* assume che non esista un limite pratico all'ampiezza del buffer.
  - *bounded-buffer* assume che esista un limite fisso del *buffer size*.
- Come si vedrà nel capitolo relativo alla "Sincronizzazione", l'accesso concorrente ai dati condivisi da parte dei due processi può produrre inconsistenza dei dati.
- Preservare la *data consistency* richiede "meccanismi" per garantire l'ordinata esecuzione di processi concorrenti.

Process and state information

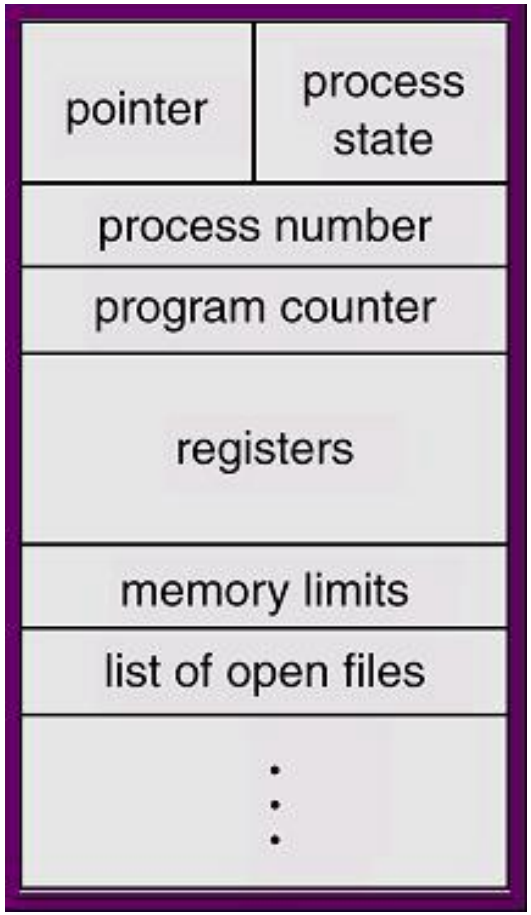
- In maniera informale ***un processo è definibile come un programma in esecuzione***; l'esecuzione di un processo deve progredire in maniera sequenziale.
- Un processo include:
  - **program code** (sezione del codice del programma)
  - **registri del processore (CPU)**, e, in particolare, il **program counter**
  - **process stack** (dati temporanei come le variabili locali, gli indirizzi di ritorno, etc.)
  - **data section** (variabili locali)
  - **heap** (memoria allocata dinamicamente in fase di esecuzione).



Process Control Block (PCB)

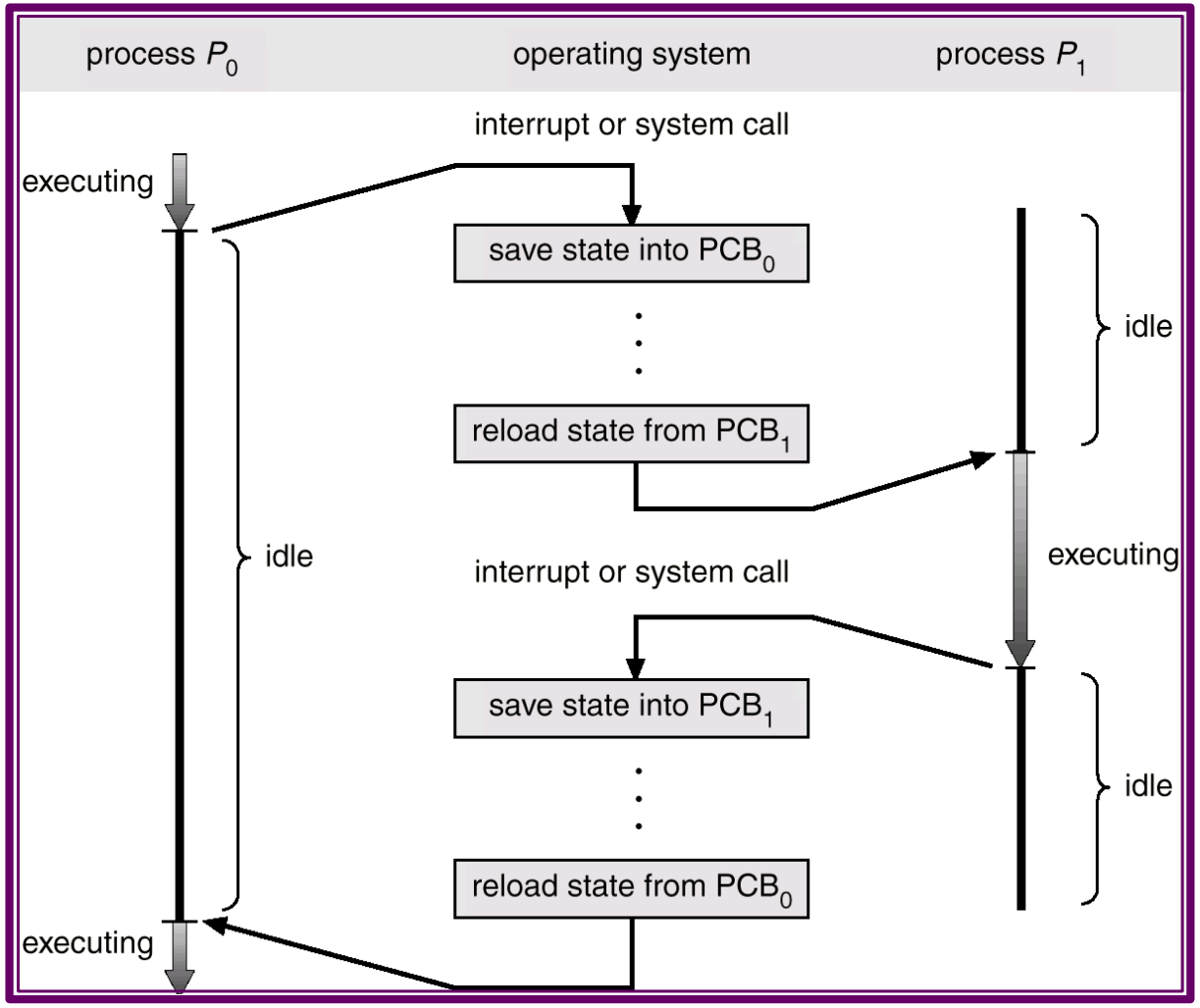
Informazione associata con ogni processo.

- ✓ Il PCB è una struttura dati che si occupa di mantenere le informazioni relative ad ogni processo.
- ✓ Stato del processo (ready, running, waiting …).
- ✓ Program counter.
- ✓ Registri della CPU (accumulatori, registri indice, stack pointer, registri general purpose).
- ✓ Informazioni per la schedulazione della CPU (priorità del processo, puntatori alle code di schedulazione).
- ✓ Informazioni per la gestione della memoria centrale (valore dei registri base e limite, tabelle delle pagine o dei segmenti).
- ✓ Informazioni per l'accounting (quantità di CPU utilizzata, limiti di tempo, numero di processo).
- ✓ Informazioni sullo stato dell'I/O (lista dei dispositivi di I/O allocati al processo).



Context Switch

CPU Switches From Process to Process



Context Switch

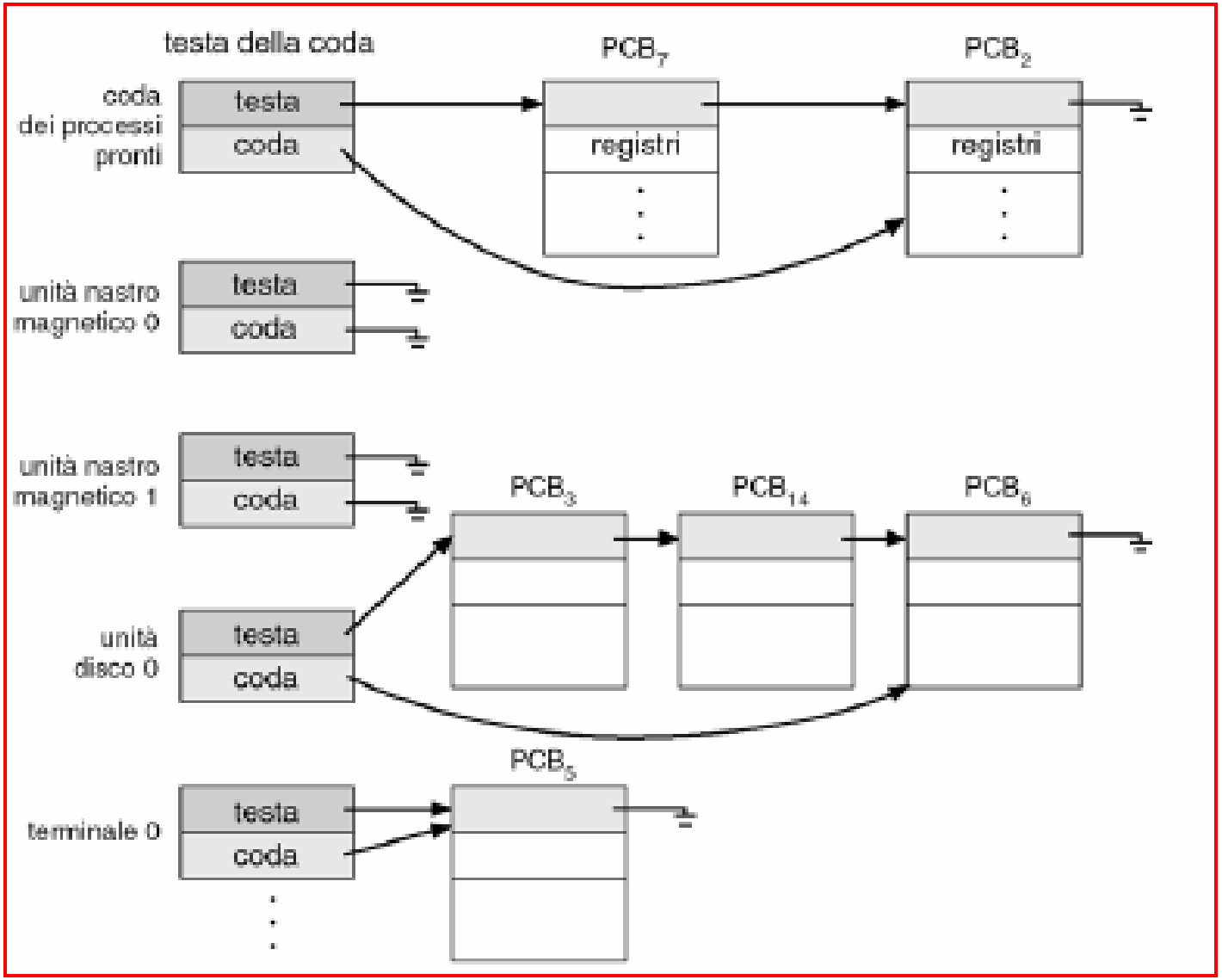
- Quando la CPU commuta da un processo ad un altro, il sistema deve salvare il **contesto** (o stato) del vecchio processo in esecuzione e caricare il contesto (precedentemente salvato) del processo che va in esecuzione.
- Il tempo per effettuare il *context-switch* è **overhead**: il sistema, cioè, non svolge lavoro "utile" durante la commutazione.
- Il tempo impiegato per il cambio di contesto dipende sensibilmente dall'hardware a disposizione.

Code di schedulazione

- **Coda di submit (*submit queue*)** – contiene tutti i processi immessi nel sistema.
- **Coda di hold (*hold queue*)** – contiene tutti i processi di cui sia stata effettuata l'analisi delle risorse richieste.
- **Coda dei processi pronti (*ready queue*)** – contiene tutti i processi che risiedono nella memoria centrale, pronti e in attesa di esecuzione.
- **Code dei processi in attesa (*wait queue*)** – contiene i processi in attesa di una particolare periferica di I/O o di un evento. Viene spesso suddivisa (*splitted*) in una sottocoda per ciascun dispositivo di I/O.

Il processo si muove fra le varie code.

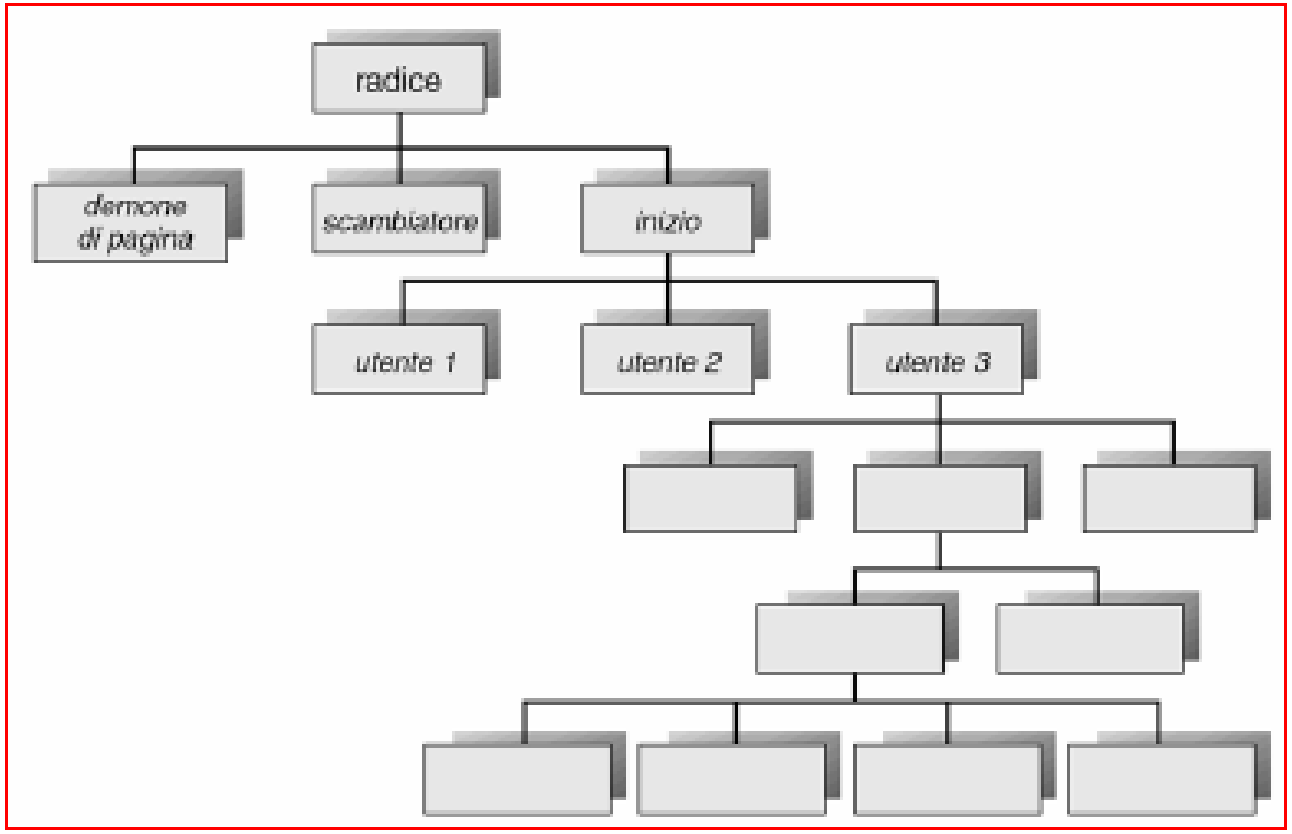
Code di schedulazione



Code dei processi pronti e delle periferiche

Creazione dei processi

- ✓ Tutti i SO offrono la possibilità di creare e terminare dinamicamente i processi.
- ✓ Durante la sua esecuzione, un *processo padre* crea *processi figli*, i quali a loro volta creano altri processi formando un *albero di discendenze*.



Albero di processi su un sistema UNIX

Creazione dei processi

Per quanto attiene la **condivisione delle risorse**, si può fare in modo che:

- padre e figli condividano tutte le risorse;
- i figli condividano un sottoinsieme delle risorse del padre;
- padre e figli non condividano risorse (l'O.S. alloca nuove risorse per ciascun processo figlio).

Per quanto attiene l'**esecuzione**, si può fare in modo che:

- il padre continui l'esecuzione in modo concorrente ai figli;
- il padre attenda finchè alcuni o tutti i suoi figli terminino.

Per quanto attiene l'**address space**, si può fare in modo che:

- lo spazio degli indirizzi (il codice) del figlio sia un duplicato di quello del padre.
- lo spazio degli indirizzi del figlio sia diverso da quello del padre.

Creazione di un processo

- la chiamata di sistema ***fork*** crea un nuovo processo;
- la chiamata di sistema ***exec***, usata dopo la *fork*, carica un nuovo programma nello spazio di memoria del processo che la esegue.

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

int main(int argc, char *argv[])
{
int pid;

    /* genera un altro processo */
    pid = fork();

    if (pid < 0) { /* si è verificato un errore */
        fprintf(stderr, "Fork fallita");
        exit(-1);
    }
    else if (pid == 0) { /*  processo figlio */
        execlp("/bin/ls","ls",NULL);
    }
    else { /* processo padre */
        /* il processo padre attenderà il completamento del figlio */
        wait(NULL);
        printf("Figlio terminato");
        exit(0);
    }
}
```

Esempi di fork

Esercizio 1

Si consideri l'esecuzione di questo frammento di programma C nel sistema operativo Unix, supponendo che il processo che lo esegue abbia process-id uguale a 999 e che i processi creati successivamente abbiano process-id consecutivi e crescenti:

```
...
for (i = 0; i < 10; i++)
i = fork();
...
```

Dire come evolve il sistema durante l'esecuzione, cioè quanti processi vengono creati, da chi vengono creati, che process-id hanno, e come prosegue la loro esecuzione.

Esercizio 2

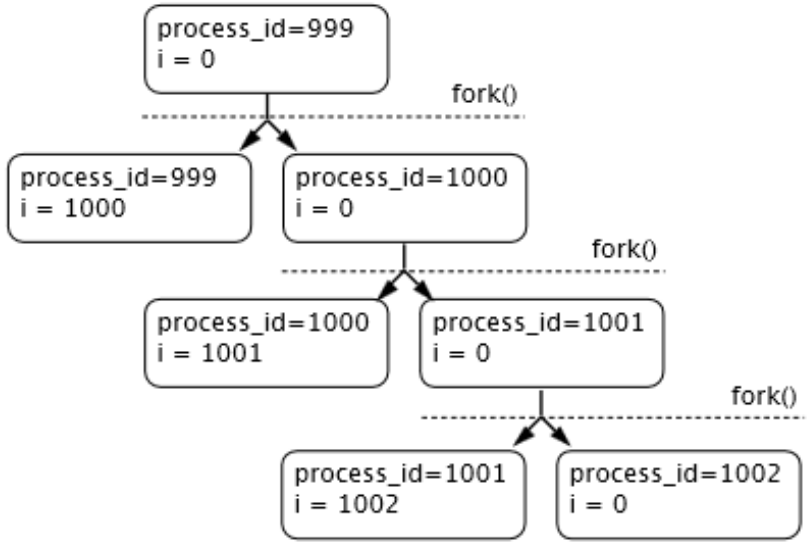
E' dato il seguente frammento di programma, eseguito nel sistema operativo Unix.

```
id = 0;          // programma "A"
if (fork() == 0)
   id = fork();
if (id == 0)
   execl("B",...);
else
   execl("C",...);
...
```

descrivere l'evoluzione del sistema durante l'esecuzione, cioè descrivere quali processi vengono creati, quali programmi eseguono e quale è il contenuto della memoria del sistema (aree codice e aree dati) durante e al termine della esecuzione del frammento, nell'ipotesi che non si verifichino errori.

Soluzione Esercizio 1

Alla prima esecuzione del ciclo, il processo crea una copia cui viene assegnato process-id = 1000. Nel processo creante (*processo padre*) la funzione fork restituisce il valore 1000, che viene assegnato alla variabile i. La variabile i viene incrementata e il ciclo for termina poiché i non è < 10; il processo prosegue con l'elaborazione successiva al ciclo.
Il processo creato (*processo figlio*) riceve dalla fork il valore 0 e lo assegna alla variabile i. Dopo l'incremento si ripete il ciclo, riproducendo il comportamento appena descritto. Il processo esegue la fork, crea un processo con process-id = 1001, valore che viene assegnato alla variabile i, causando la terminazione del ciclo for. Il processo creato riprende il ciclo con i = 1, e così all'infinito, o fino ad esaurimento delle risorse del sistema se i processi non terminano immediatamente l'esecuzione.



Soluzione Esercizio 2

Chiamiamo per semplicità *A* il processo che esegue il programma *A*, e *A1*, *A2*, etc., i processi creati a seguito dell'esecuzione della funzione fork(). L'esecuzione della istruzione

```
if (fork() == 0)
```

causa la creazione di un processo figlio *A1*, cui la fork restituisce il valore 0. Al processo *A* viene restituito un valore > 0 che coincide con l'identificatore di processo del processo *A1*. Il processo figlio *A1* quindi vede soddisfatta la condizione ed esegue l'istruzione

```
id = fork();
```

che causa la creazione del processo *A2*. Nel processo *A1* la variabile id riceve un valore > 0, mentre nel processo *A2* la variabile id riceve il valore 0.
Nel processo *A* la variabile id conserva il valore 0 assegnato dall'istruzione

```
id = 0;
```

poiché il processo *A* non è entrato all'interno della struttura condizionale if. Si ha quindi la seguente situazione:

- nel processo *A* `id = 0`,
- nel processo *A1* `id > 0`,
- nel processo *A2* `id = 0`.

I processi *A* e *A2* proseguiranno quindi eseguendo il codice del programma B, mentre il processo *A1* proseguirà eseguendo il codice del programma *C*.

Creazione di un processo

Quando viene creato un processo?

- **Al bootstrap** del sistema operativo
- All'esecuzione di una **system call apposita** (es., fork())
- **Su richiesta** da parte dell'utente
- **All'inizio di un job batch**

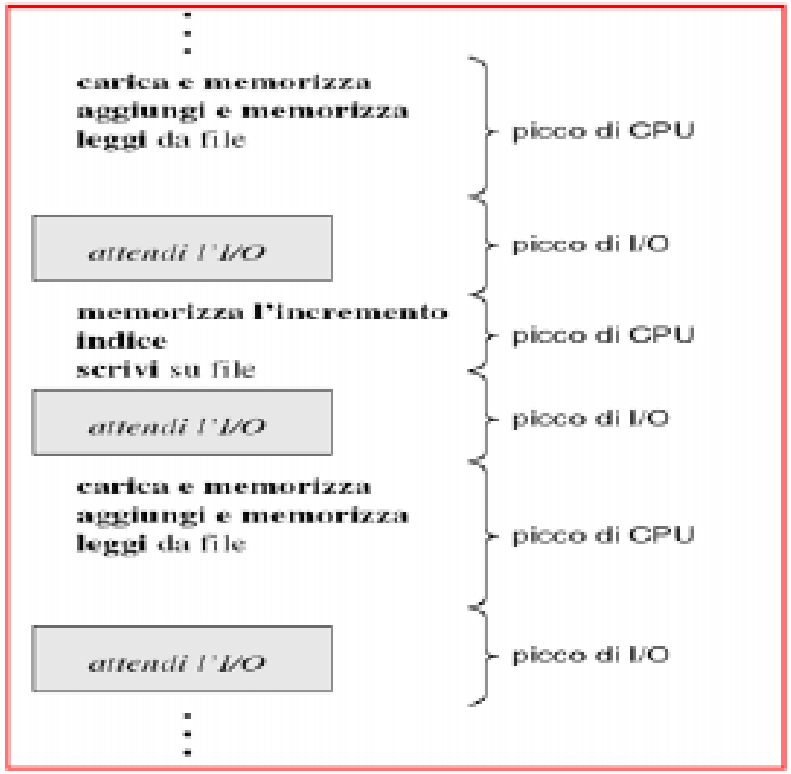
Terminazione dei processi

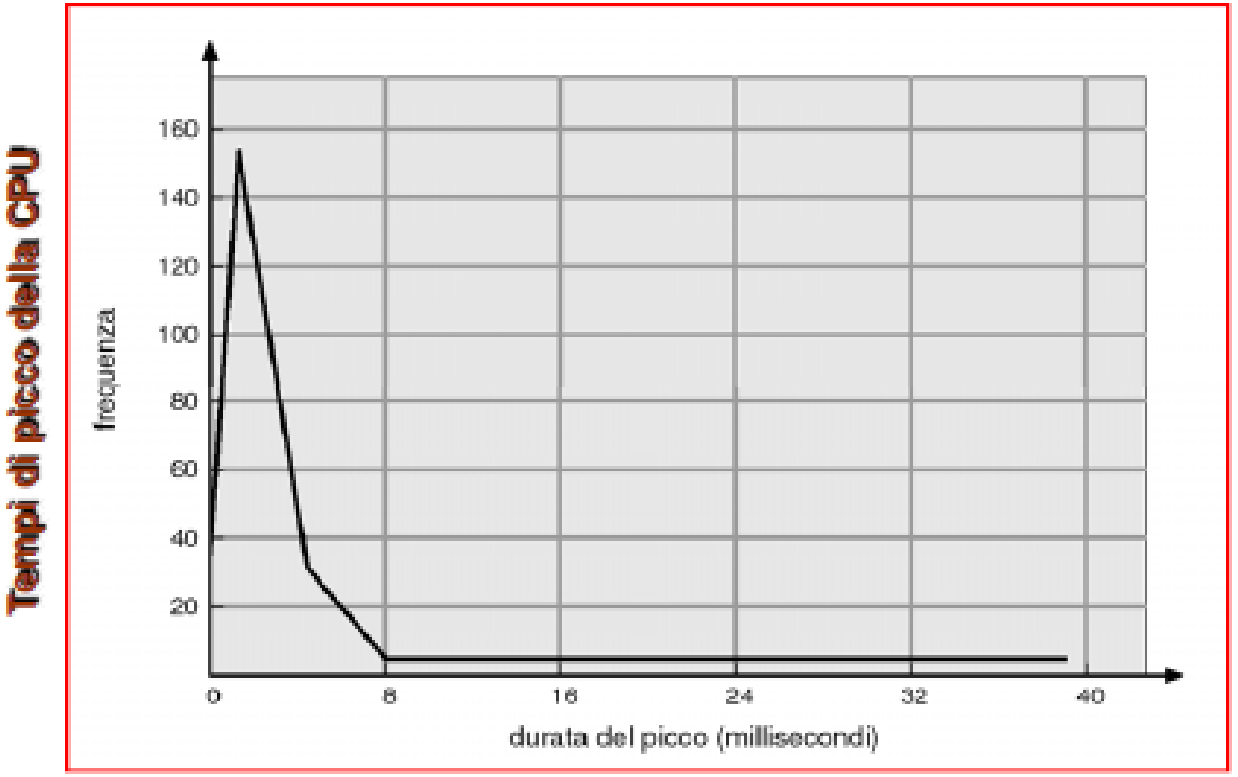
- Il processo esegue la sua ultima istruzione e chiede al sistema operativo di rimuoverlo dal sistema tramite la chiamata *exit*.
  - Il figlio restituisce un valore di stato al padre (che attende la terminazione tramite la chiamata *wait*).
  - Le risorse del processo sono deallocate dal sistema operativo.
- Il padre può terminare l'esecuzione di uno dei suoi figli per varie ragioni:
  - Il figlio ha ecceduto nell'uso di una risorsa a lui allocata.
  - Il compito assegnato al figlio non è più necessario.
  - Se il padre sta terminando:
    - alcuni sistemi operativi non permettono ad un figlio di proseguire. Tutti i figli terminano – terminazione a cascata (*cascading termination*).
    - alcuni altri sistemi operativi consentono l'"adozione" dei processi figli da parte del processo *init*.

Esecuzione di un processo

- Si può affermare che sostanzialmente l'esecuzione di un processo consiste in una **alternanza** di periodi di utilizzo della CPU (*CPU burst*) e di attese per il completamento di I/O (*I/O burst*).
- Sostanzialmente i processi si alternano tra cicli di picco di CPU e cicli di picco di I/O.
- Un processo termina con un ciclo di picco della CPU.
- Le durate dei picchi della CPU variano in base all'architettura e al processo, ma hanno un andamento comune.
- La curva dei CPU *burst* ha un andamento (iper)esponenziale con un elevato numero di picchi brevi e pochi picchi lunghi.
  - Un programma I/O-bound ha molti picchi brevi di CPU.
  - Un programma CPU-bound ha pochi picchi di lunga durata.

Esecuzione di un processo





Gli schedulatori

- **Long-term scheduler (o job scheduler)** – seleziona quale processo presente nella coda di *hold* deve essere inserito nella coda dei processi pronti.
  - Viene eseguito con una frequenza dell'ordine dei minuti
  - Controlla il livello di multiprogrammazione (numero di processi in memoria centrale)
  - Deve garantire una certa stabilità del grado di multiprogrammazione
  - Bilancia i processi **I/O bound** (molti e brevi utilizzi di CPU) e **CPU bound** (pochi e lunghi utilizzi di CPU).

- **Short-term scheduler (o CPU scheduler)** – seleziona quale processo deve essere eseguito e alloca la CPU a uno di essi.
  - Viene eseguito molto frequentemente (almeno una volta ogni 100 msec) e deve quindi essere molto veloce nello svolgere il proprio compito.

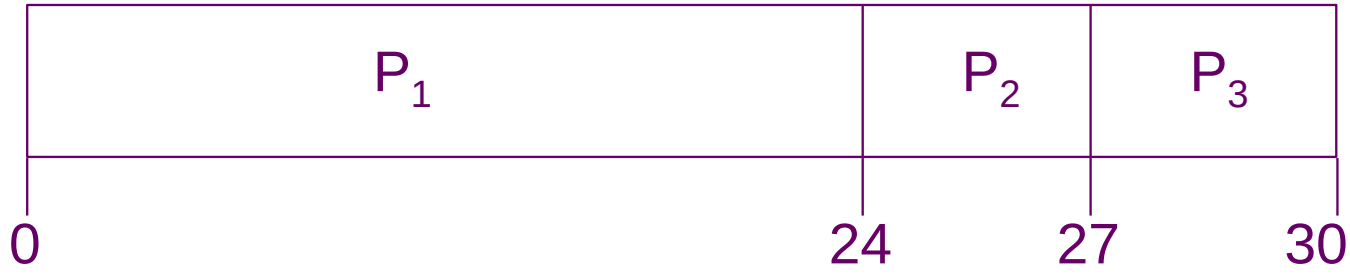
Job Scheduler algorithms

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

Supponiamo che i processi vengano eseguiti secondo l'ordine di arrivo: P_1 , P_2 , P_3.

Il diagramma di Gantt sarebbe:



Il ***tempo di attesa***: per P_1 = 0; per P_2 = 24; per P_3 = 27

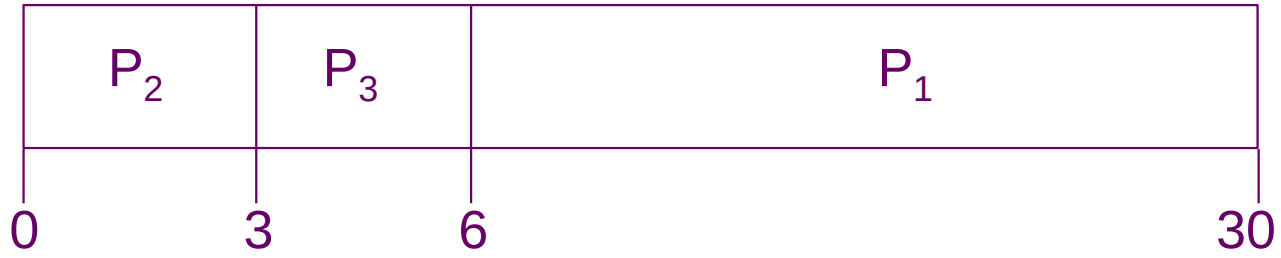
Il ***tempo medio di attesa***: (0 + 24 + 27)/3 = 17

Tale tempo medio di attesa dipende fortemente dalla durata dei processi e dall'uso che essi fanno della CPU durante il tempo di esecuzione.

Job Scheduler algorithms

Effetto "convoglio" dello Scheduling FCFS

Se si suppone che i processi arrivino nell'ordine P_2 , P_3 , P_1 . Il diagramma di Gantt sarà:



Il ***tempo di attesa***: per P_1 = 6; per P_2 = 0; per P_3 = 3
Il ***tempo medio di attesa***: (6 + 0 + 3)/3 = 3

Come si può osservare, il tempo medio di attesa è molto migliore di quello precedente. Ciò è dovuto al fatto che nel caso precedente l'uso della CPU da parte del processo P_1 frena l'esecuzione degli altre due processi che, dopo aver completato il loro I/O, rimangono nello stato di ready, attendendo il rilascio della CPU da parte di P_1.

Job Scheduler algorithms

Shortest-Job-First (SJF) Scheduling

- È un caso particolare del più generale **scheduling a priorità**. La priorità è, infatti, assegnata ai processi in ragione inversa del prossimo burst (previsto) di CPU. In altri casi il parametro in base al quale stabilire la priorità di un processo è la quantità di memoria richiesta, il numero di file aperti, il rapporto fra tempo dedicato all'I/O e tempo di utilizzo della CPU, o anche l'importanza del processo o altre forme di "precedenza". Se due o più processi hanno la stessa priorità, vengono schedulati in ordine FCFS. Pur non esistendo una regola univoca per definire quali sia il valore numerico che contraddistingue la priorità più alta, spesso quest'ultima è pari a Ø.
- L'estremizzazione del freno all'esecuzione dei processi a bassa priorità può dar luogo alla ***starvation*** (inedia indefinita). Per risolvere tale problema, viene adottata la tecnica dell'***aging*** (invecchiamento), che consiste nell'innalzare periodicamente la priorità di un processo a bassa priorità.

CPU Scheduler

- Il sistema operativo deve scegliere fra i processi in memoria che sono pronti per l'esecuzione ed assegnare la CPU ad uno di essi. Quest'ultimo compito viene svolto dal ***dispatcher***.
- Le decisioni sulla schedulazione della CPU possono avvenire nelle seguenti quattro circostanze:
  1. Quando un processo passa dallo stato di esecuzione alla coda di wait (I/O o attesa per la terminazione di un processo figlio).
  2. Quando un processo passa dallo stato di esecuzione alla coda di ready (interrupt).
  3. Quando un processo passa dalla coda di wait alla coda di ready (termine di una operazione di I/O).
  4. Quando un processo termina.
- La schedulazione ai punti 1 e 4 è detta ***nonpreemptive*** (senza sospensione dell'esecuzione): quando la CPU è assegnata ad un processo, esso non può essere interrotto finchè non termina il suo burst di CPU.
- Tutti gli altri scheduling sono ***preemptive***, cioè con sospensione dell'esecuzione.

Dispatcher

- Il ***dispatcher*** è il modulo del sistema operativo che dà il controllo della CPU ad un processo selezionato dal micro-schedulatore. Questa funzione comprende:
  - cambio di contesto;
  - passaggio dalla modalità supervisore alla modalità utente;
  - salto alla corretta locazione nel programma utente per ricominciarne l'esecuzione.
- Deve essere molto veloce, perchè viene molto frequentemente attivato.
- Richiede un tempo non trascurabile (***dispatch latency***) per sospendere un processo e passare all'esecuzione di un altro

Scheduling Criteria

- Gli algoritmi di schedulazione della CPU hanno diverse proprietà e possono favorire una categoria di processi piuttosto che un'altra.
- I criteri generali di comparazione possono essere riassunti in:
  - **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. Tloading+Tready+TCPU+TI/O
  - **Tempo di attesa** – somma dei tempi spesi in attesa nella coda dei processi pronti. Si tratta della variabile influenzata dall'algoritmo di schedulazione.
  - **Tempo di risposta** – tempo che intercorre dalla formulazione della richiesta fino alla produzione della prima risposta.

Optimization Criteria

In relazione ai precedenti criteri generali di schedulazione, gli algoritmi di scheduling possono mirare a:

- Max utilizzo della CPU
- Max throughput
- Min turnaround time
- Min tempo di attesa
- Min tempo di risposta

CPU scheduler algorithms

- ***round robin:*** la coda di READY è di tipo FIFO; il process scheduler assegna lo stesso *time slice* al processo correntemente primo in coda;

- ***round robin modificato:*** ad ogni processo viene attribuita una priorità nella coda di READY inversamente proporzionale al tempo di RUN utilizzato in precedenza;

- ***priorità statica:*** ad ogni processo viene attribuita una priorità, che il processo conserva per tutta la durata del ciclo di esecuzione;

- ***priorità dinamica:*** la priorità del processo viene stabilita in base al suo merito, calcolato, al termine di un intervallo statistico ΔT, in base al numero di volte che il processo ha completamente utilizzato il *time slice* assegnatogli.

CPU scheduler algorithms: Round Robin (RR)

Si tratta di un algoritmo di tipo FCFS con l'aggiunta della *preemption* per migliorare l'alternanza dei processi.

A ogni processo viene assegnata un quanto del tempo di CPU (*time slice*), generalmente 10-100 millisecondi. Se entro questo arco di tempo il processo non lascia la CPU, viene interrotto (mediante un interrupt) e rimesso in fondo alla 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* unità di tempo. Ciascun processo non deve attendere più di (*n*–1) x *q* unità di tempo.

Le prestazioni dipendono dalla dimensione del time slice *q*:

q grande ⇒ FIFO

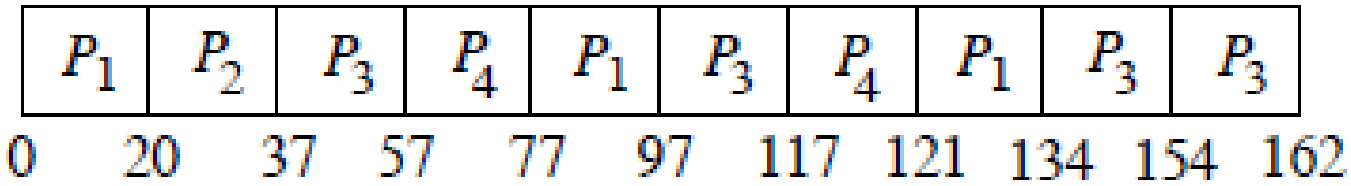
q piccolo ⇒ maggior effetto di "parallelismo virtuale" tra i processi, però aumenta il numero di context switch, e quindi l'overhead.

CPU scheduler algorithms: Round Robin (RR)

Esempio con time-slice = 20 msec

Processo	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

Diagramma di Gantt



Tipicamente, si ha un tempo di turnaround medio maggiore, ma minore tempo di risposta

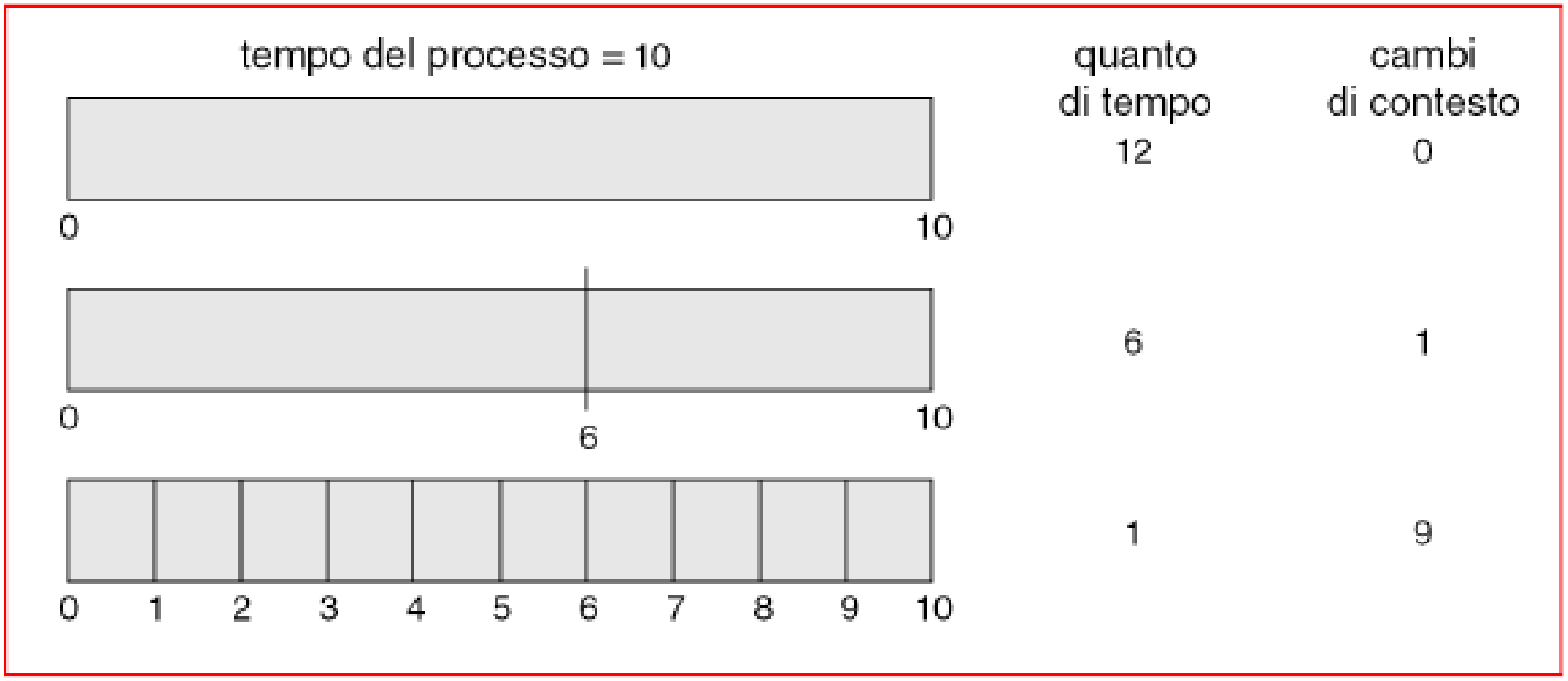
RR e context switching

Un time slice deve possedere una durata sufficientemente superiore rispetto a quella per il context switching:

$T_{context\_switching} \approx 10\% T_{time\_slice} \rightarrow 10\% T_{CPU}$ speso per cambi di contesto

Nei sistemi moderni:

$T_{time\_slice} \approx$ 10-100 ms, $T_{context\_switching} \approx$ 10 µs.



CPU scheduler algorithms: Modified Round Robin (MRR)

- ✓ Viene determinata la parte ($\Delta t'$) di time slice effettivamente impiegata dal processo.
- ✓ Tale valore viene impiegato per calcolare la priorità ($p \equiv k/\Delta t'$) da assegnare al processo la prossima volta che esso andrà in coda in attesa dell'attribuzione della CPU.
- ✓ Vengono favoriti i processi I/O-bound, che hanno manifestato, nella precedente occasione, attitudine a fare scarso uso della CPU.

CPU scheduler algorithms: Priorità statica

- Si associa una priorità numerica a ciascun processo.
- La CPU viene allocata al processo con la priorità più alta (più piccolo è il valore numerico più alta è la priorità).
- La CPU viene allocata al processo con la priorità più alta (più piccolo è il valore numerico più alta è la priorità) tra quello in esecuzione e quello che è in cima alla coda dei processi pronti.

La priorità può essere definita:

- ***internamente***, se vengono impiegati parametri misurabili del singolo processo, quali l'uso della CPU, i file da utilizzare, la quantità di memoria richiesta, l'impiego di dispositivi di I/O, ...
- ***esternamente***, quando vengono impiegati parametri quali la rilevanza del processo, la sua criticità, ...

CPU scheduler algorithms: Dynamic priority (process merit)

- Durante **l'intervallo statistico d'osservazione** ΔT, si calcola il valore dell'indicatore di merito per ciascuno dei processi in esecuzione.
- L'indicatore di merito:

$$R_i = n_i/N_i$$

  - N_i = numero di time slice attribuiti al processo i-esimo durante ΔT;
  - n_i = numero di time slice completati durante ΔT (negli altri casi, il processo è stato interrotto andando nello stato di WAIT)
  - con $n_i \leq N_i$

    $0 \leq R_i \leq 1$

CPU scheduler algorithms: Dynamic priority (process merit)

- la determinazione del merito viene usata per estrapolare al successivo intervallo statistico analogo comportamento del task;
- la coda di READY è organizzata in base alla priorità dei processi;
- viene attribuita priorità più alta ai processi con $R_i \rightarrow 0$.

RETROAZIONE: *Se tutti i valori di R_i sono addensati verso 0 (oppure verso 1), il SO regola il valore del time slice diminuendolo (oppure, nell'altro caso, aumentandolo).*

Tale regolazione consente la migliore discriminazione tra i processi I/O Bound e CPU bound, permettendo un migliore utilizzo sia della CPU che dei dispositivi di I/O.

Multiple-processor scheduling

Se sono disponibili più CPU il problema della schedulazione diviene più complesso.

- **Processori omogenei**: uno qualunque dei processori disponibili può essere adoperato per eseguire uno dei processi pronti. In ogni caso un task può essere eseguito su uno specifico processore.
- **Suddivisione del carico** *(load sharing)*: Tutti i processori scelgono i processi dalla stessa coda di ready oppure viene creata una coda di ready per ciascun processore (rischio di asimmetria nel carico).
- ***Asymmetric multiprocessing***: solo un processore accede alle strutture dati del sistema diminuendo la necessità della condivisione dei dati.
- ***Symmetric multiprocessing***: ciascun processore schedula se stesso attingendo ad una coda di ready condivisa. In tal caso si pongono problemi di hardware speciale e di controllo degli accessi concorrenti alle risorse condivise da parte dei diversi processori.

Real-time scheduling

- Questo tipo di schedulazione viene richiesto dai **sistemi *hard real-time*** (in tempo reale stretto), che devono completare un'operazione critica entro una precisa quantità di tempo garantita (*deadline*).
  - Ogni processo viene avviato con un'asserzione sull'intervallo entro il quale deve essere completato e lo schedulatore effettua un *admission control* basato sul meccanismo della *resource reservation*.
  - Lo schedulatore deve garantire il rispetto delle tempistiche di ciascuna funzione dell'O.S. Una tale garanzia è impossibile in presenza di memoria di massa o memoria virtuale. I sistemi real-time utilizzano software specifico su hardware dedicato per ciascuna attività critica.
- **Sistemi *Firm real-time***: la violazione del requisito di completamento entro la deadline rende sempre più inutili i risultati ottenuti via via che cresce il ritardo (*time delay*) rispetto al tempo limite.
- ***Sistemi Soft real-time*** (in tempo reale lasco): richiedono che i processi critici ricevano priorità su quelli meno importanti.

More details will be given later.

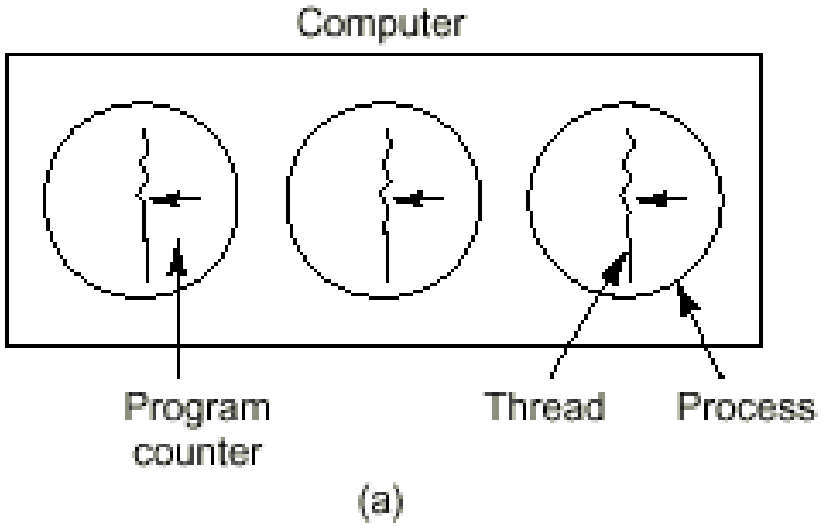
Comunicazione tra processi

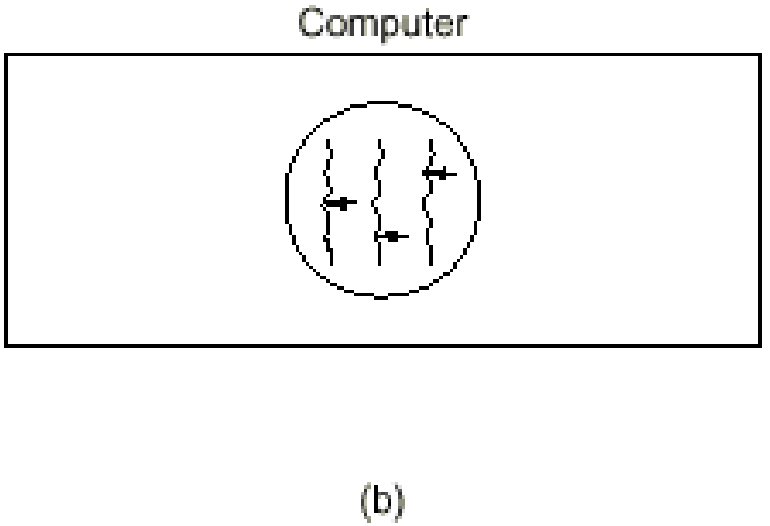
- La comunicazione tra processi può essere realizzata direttamente o attraverso un meccanismo (Inter Process Communication - ***IPC - facility***) per consentire ai processi di comunicare e sincronizzare le loro azioni mediante scambio di messaggi.
- Scambio di messaggi – un processo comunica con un altro senza ricorrere a dati condivisi (non si condivide lo spazio di indirizzi).
- La IPC fornisce due operazioni:
  - ✓ ***send*** *(messaggio)* – la dimensione del messaggio può essere fissa o variabile.
  - ✓ ***receive*** *(messaggio)*

More details will be given later
also referring to inter processes communication in a network.

Threads

- un processo è un programma la cui esecuzione si svolge seguendo un singolo **flusso di controllo (*thread*)** ⇨ *figura (a)*;
- se un processo ha più di un flusso di controllo, esso si dice che è costituito da più thread (***multi-thread***);
- un thread, a volte detto processo a peso leggero, è la singola sequenza di istruzioni che si svolge insieme ad altre sequenze dello stesso processo ⇨ *figura (b)*.



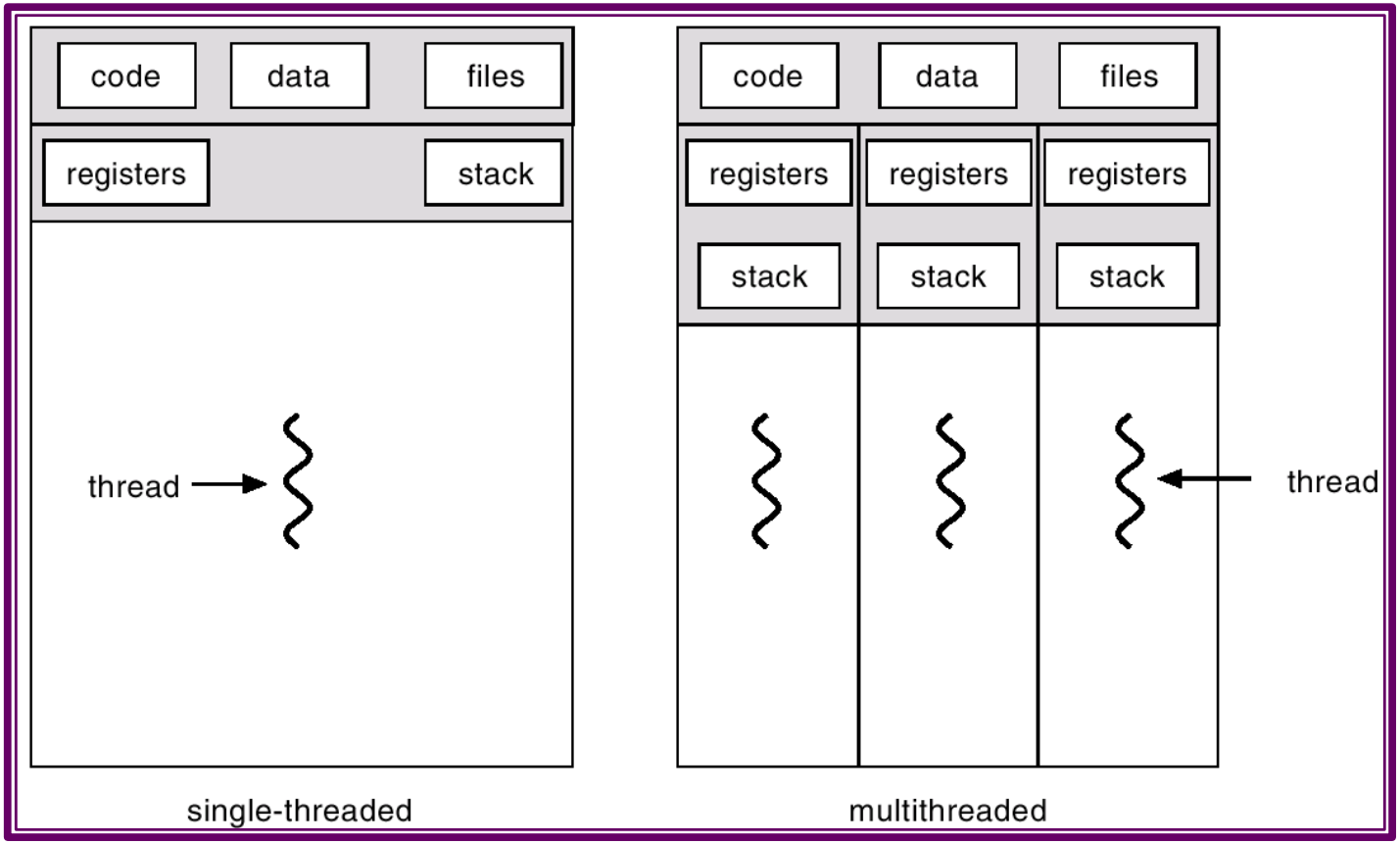


Threads

Thread = unità atomica di utilizzo della CPU.

- tutti i thread di uno stesso processo condividono:
  - la sezione del ***codice***,
  - la sezione dei ***dati***,
  - ***altre risorse*** (file utilizzati, segnali ricevuti, ecc.);
- ad ogni singolo thread sono associati:
  - il suo ***identificatore***,
  - il suo ***program counter***,
  - il suo ***insieme di registri***,
  - il suo ***stack***.

Threads



Un processo con un solo thread "pesante" e con molti thread a peso leggero

Nel passaggio da un thread all'altro, non interviene il SO, ma si simula il cambio di contesto computazionale. A tal fine, deve essere associato l'insieme dei valori dei registri della CPU ad ogni thread, in modo che il cambio di thread si riduca ad un semplice cambio del valore del registro Program Counter.

Threads

Generazione di thread vs generazione di processi figli

- La creazione di processi è un meccanismo dispendioso in termini di tempo e risorse necessarie.

- Quando le funzionalità da eseguire sono sostanzialmente analoghe ma semplicemente multiple, non ha senso duplicare l'intera struttura a corredo di un processo.
  - È il caso, ad esempio, di un demone HTTP che deve servire migliaia di client per restituire risorse complesse (immagini, audio, video).

- Generalmente è più opportuno replicare semplicemente i flussi di controllo di un medesimo processo.
  - È il caso dei server RPC ed RMI che sono tipicamente multi-thread.

Threads

Thread a livello user space o kernel space

La libreria di thread (**thread package**) a livello utente fornisce al programmatore le API per la creazione e gestione dei thread e può essere implementata come:

- **Libreria completamente residente all'interno dello user space** senza supporto da parte del kernel: invocare una funzione di libreria si traduce in una chiamata di funzione locale
  - il sistema operativo "vede" solo i processi e non i thread al loro interno;
  - ogni processo può avere un proprio algoritmo di scheduling;
  - il run time system si colloca al di sopra del kernel;
  - Un sistema multi-thread perde il controllo della CPU solo quando richiede un servizio del SO; l'intero sistema multi-thread va allora in wait (poiché, in genere, le chiamate a SO sono di tipo bloccante). Invece, le chiamate al run time system non bloccano il multi-thread, poiché il controllo passa ad un altro thread (che continua ad occupare la CPU)
  - Il processo è costituito dall'insieme dei thread e dal run time system, che fanno tutti parte dello stesso spazio indirizzi.
- **Libreria a livello kernel**: invocare una funzione delle API della libreria comporta una chiamata di sistema
- **Libreria realizzata tramite una combinazione run time system/kernel**

Caratteristiche dei thread

- Ciascuno dei thread è indipendente.
- Se un thread apre un file con privilegi di lettura, gli altri thread dello stesso processo possono leggere quel file.
- Quando un thread altera un dato in memoria, gli altri thread dello stesso processo vedono il risultato quando accedono a quel dato.
- Il vantaggio chiave dei thread è nelle prestazioni: la creazione di un nuovo thread in un processo esistente, la terminazione di un thread e il cambio di due thread richiedono molto meno tempo rispetto alle analoghe operazioni per i processi. Alcuni hanno valutato tale aumento di efficienza rispetto ad una implementazione paragonabile che non fa uso di thread, in un fattore 10.
- I thread migliorano anche l'efficienza della comunicazione tra processi, che normalmente richiede l'intervento del kernel, mentre, con un processo multi-thread, i vari thread possono comunicare senza intervento del kernel.

Tipiche applicazioni dei thread

Un esempio di applicazione dei thread è un file server di rete: per ogni nuova richiesta di file, viene creato un nuovo thread e molti thread vengono creati e distrutti in breve tempo. Se il server è multiprocessore, i thread appartenenti allo stesso processo possono essere eseguiti simultaneamente su diversi processori. Quando un programma è costituito di funzioni logicamente distinte, l'uso di thread è utile anche con un singolo processore.

In un foglio di calcolo, un thread gestisce i menu e legge l'input dell'utente e un altro esegue i comandi dell'utente e aggiorna il foglio di calcolo.

Nei browser del web, molte pagine contengono un buon numero di piccole immagini, ottenute con una connessione al sito relativo e la richiesta di un'immagine; con un processo multi-thread è possibile richiedere molte immagini allo stesso tempo, in quanto per piccole immagini il fattore dominante è il tempo necessario allo stabilirsi della connessione piuttosto che il tempo di trasmissione dell'immagine.

Politiche di scheduling dei thread

- Round robin;
- Priorità statica.

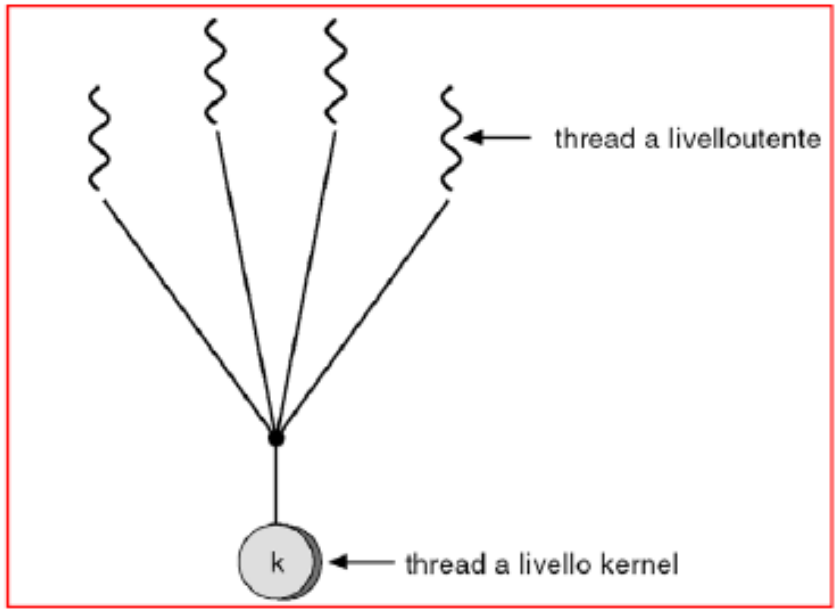
↳ Si può realizzare un sistema multi-thread con una certa politica di scheduling ed un altro sistema multi-thread che utilizza un altro tipo di politica di scheduling, a seconda della particolare applicazione (Questo è un ulteriore vantaggio dei thread).

↳ In Java lo scheduling utilizzato è quello a priorità. E' possibile assegnare ai singoli thread una priorità che va da 1 a 10. Un thread a priorità più alta ottiene il controllo della CPU e quello a priorità più bassa va nello stato di ready. Se il thread nello stato di run sospende o termina la sua esecuzione, il run time system passa il controllo della CPU al thread a priorità più alta nella coda di ready.

Multi-threading models 1/3

Many-to-One: Il modello molti-a-uno riunisce molti thread di livello utente in un unico kernel thread.

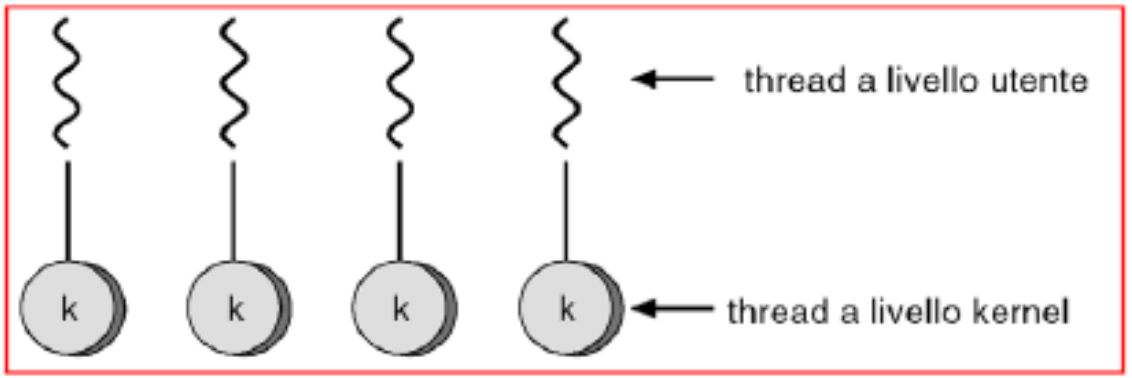
- Prima di sospendersi un thread avvia il successore. La commutazione è veloce perché avviene nello stesso spazio utente. Però l'intero processo si blocca se un thread esegue una chiamata di sistema bloccante.
- Non è possibile eseguire thread multipli in parallelo su più processori in quanto un solo thread per volta può accedere al kernel.



Multi-threading models 2/3

One-to-One: Il modello uno-a-uno mappa ciascun thread utente in un kernel thread

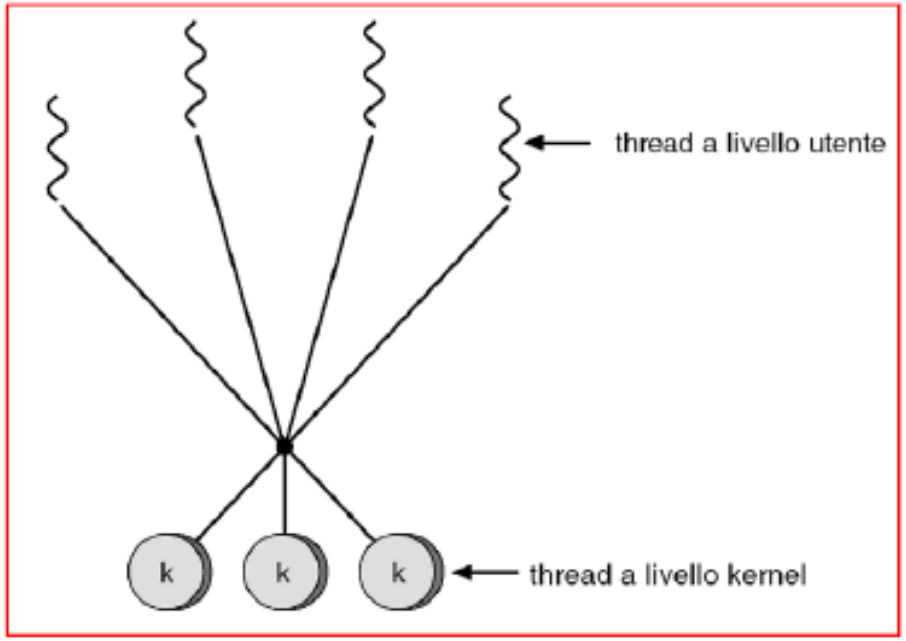
- Aumenta il livello di concorrenza possibile:
  - Se uno user thread si blocca, si blocca solo lui mentre un altro prosegue nella sua esecuzione.
  - Più thread possono essere eseguiti in parallelo su più processori.
- Obbliga alla mappatura di uno user thread su un kernel thread.
  - L'overhead della creazione di un kernel thread si ripercuote sull'applicazione.
  - In genere è posto un limite al multi-threading.



Multi-threading models 3/3

Many-to-Many: Il modello molti-a-molti permette di aggregare molti thread a livello utente verso un numero più piccolo o equivalente di kernel thread

- Permette al sistema operativo di creare un numero sufficiente di kernel thread in base alle caratteristiche dell'architettura.
- Non soffre di alcuna delle limitazioni dei precedenti modelli.



Multi-thread organizations 1/3

L'organizzazione Dispatcher/Worker

- multi-threading utilizzato in Java
- Prevede un thread principale (*dispatcher thread*) che gestisce l'esecuzione degli altri thread del processo (*worker thread*).
- Il dispatcher preleva le richieste di lavoro da una mailbox di sistema.
- Il dispatcher sceglie il worker thread inattivo che può eseguire tale lavoro, gli manda la richiesta e lo passa nello stato di ready.
- Il worker thread verifica la richiesta di lavoro e, se la può soddisfare, va in esecuzione.
- Nel caso in cui il worker selezionato non possa eseguire il lavoro richiesto (a causa, ad esempio, di lock), si mette nello stato d'attesa; A questo punto, viene invocato lo scheduler e viene mandato in esecuzione un altro thread, oppure il controllo passa nuovamente al dispatcher.

Multi-thread organizations 2/3

L'organizzazione a Team

- Ogni thread è specializzato in una operazione.
- Ciascun thread fa dispatcher di se stesso, analizzando la mailbox di sistema e prelevando richieste che può soddisfare o, se è già impegnato, mettendo in una coda la nuova richiesta.

Multi-thread organizations 3/3

L'organizzazione a Pipeline

- un lavoro viene eseguito in modo sequenziale da più thread, ognuno specializzato nell'esecuzione di una frazione di tale lavoro.
- In tal modo, è possibile soddisfare più richieste d'esecuzione di uno stesso lavoro in parallelo.

JAVA threads

I thread Java sono gestiti dalla Java Virtual Machine (JVM).

- Il ciclo d'esecuzione di un thread è composto da quattro stati:
  - Ready: Il thread è stato schedulato per essere eseguito;
  - Blocked: Il thread sta aspettando che un altro thread lo sblocchi;
  - Run: Il thread ha il controllo della CPU ed è attivo;
  - Dead: Il thread ha terminato la sua esecuzione oppure è stato ucciso.