PROGRAM EXECUTION PHASES

submit fase in cui viene richiesta l'esecuzione del programma

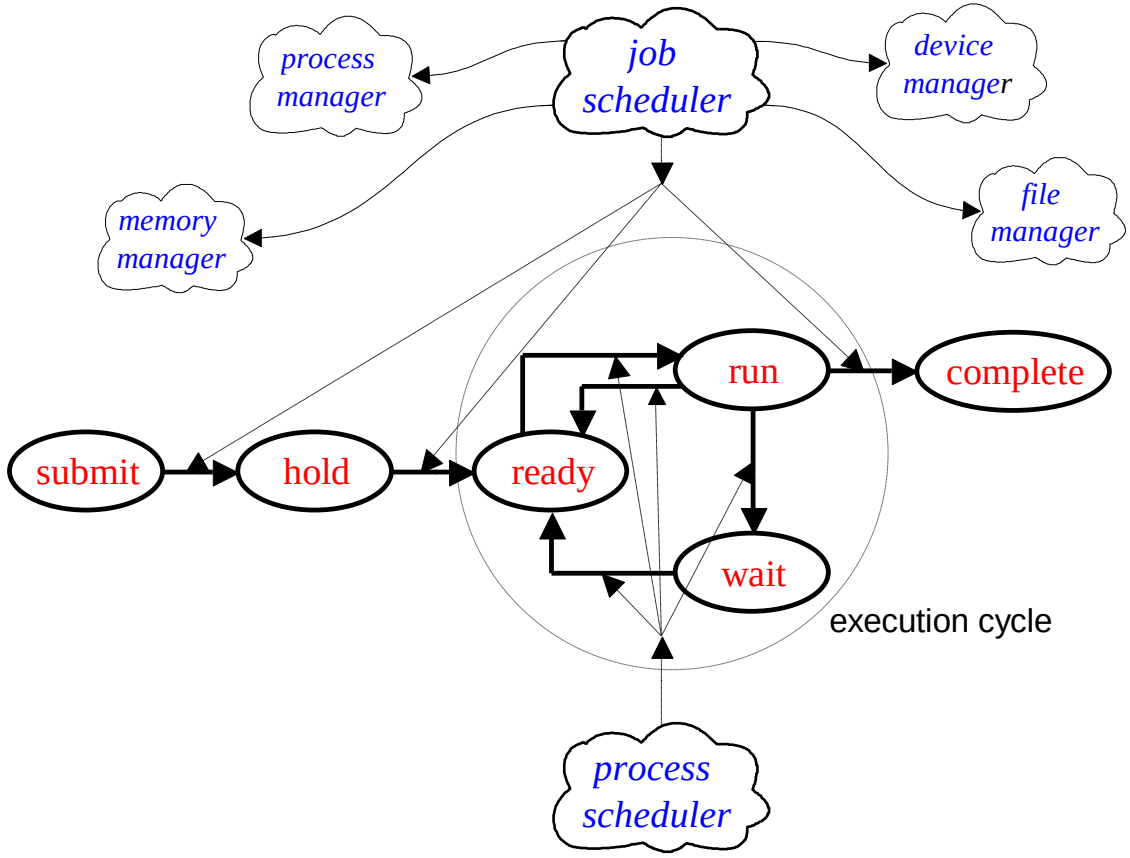
hold fase di incodamento delle richieste di esecuzione in attesa delle risorse necessarie (livello di multitasking, memoria, dispositivi, file)

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

Textbook uses the terms *job*, *job-step* and *process* almost interchangeably.

A job step may be made by a single process or by several concurrent processes.

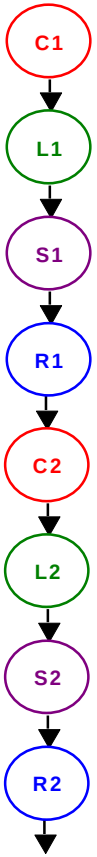
The advantages of segmenting a program in several concurrent processes.

SEQUENTIAL EXECUTION

Realizzazione sequenziale di un sistema di acquisizione dati

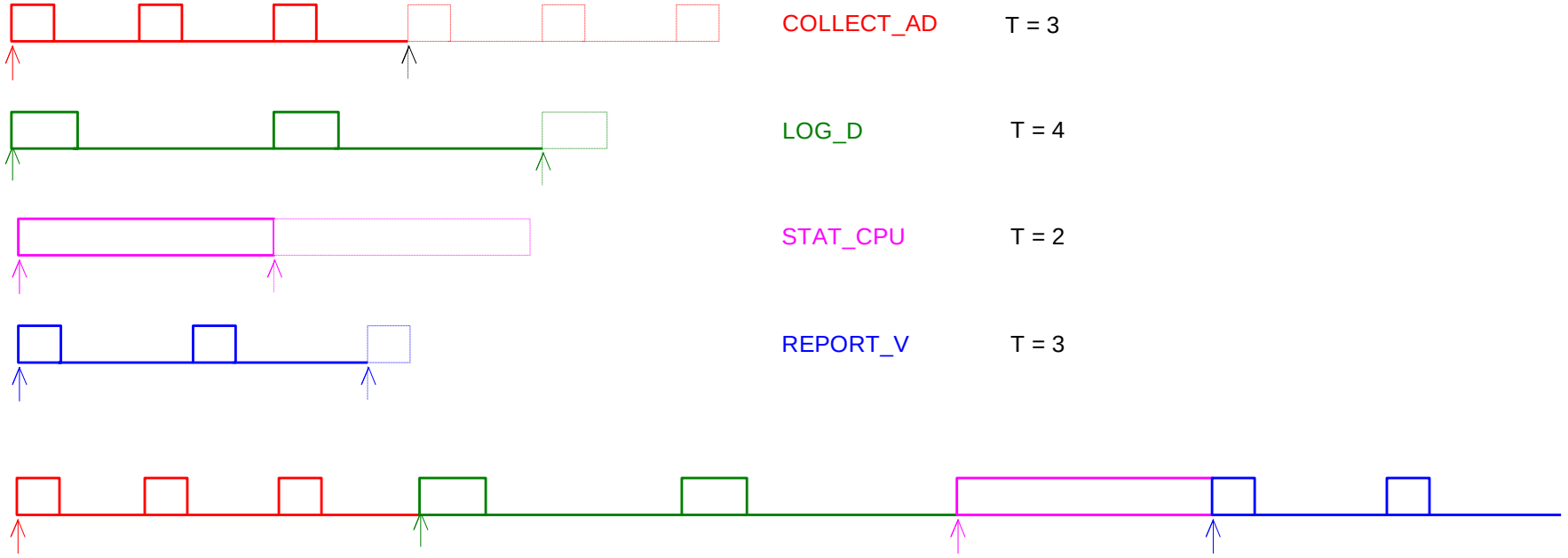
```
program          Acquisizione_sequenziale
   :
   :
     while true     [loop infinito]
        collect;    [lettura da convertitore analog-
digital]
        log;        [memorizzazione su disco]
        stat;       [elaborazione statistica]
        report;     [visualizzazione risultati]
     end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

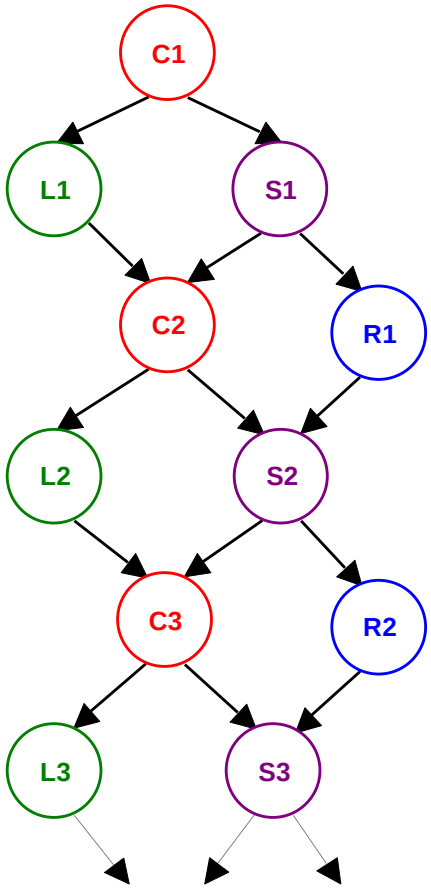
CONCURRENT EXECUTION

Realizzazione concorrente di un sistema di acquisizione dati

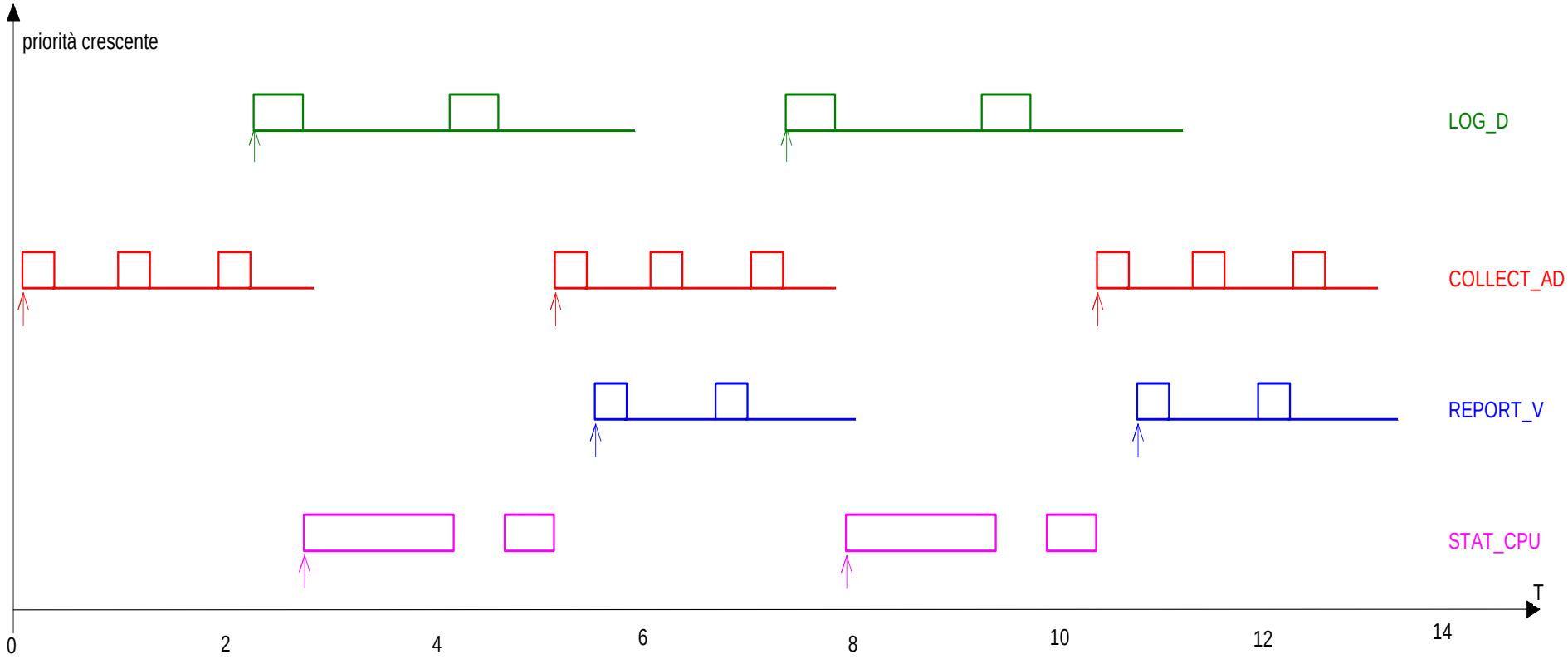
```
module              Acquisizione_concorrente
   :
   process collect;
       while true
        wait_signal_from (log,stat);
        collect_ad;
        send_signal_to (log.stat);
       end  [fine del loop]
   end;     [fine collect]
   process log;
       while true
        wait_signal_from (collect);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

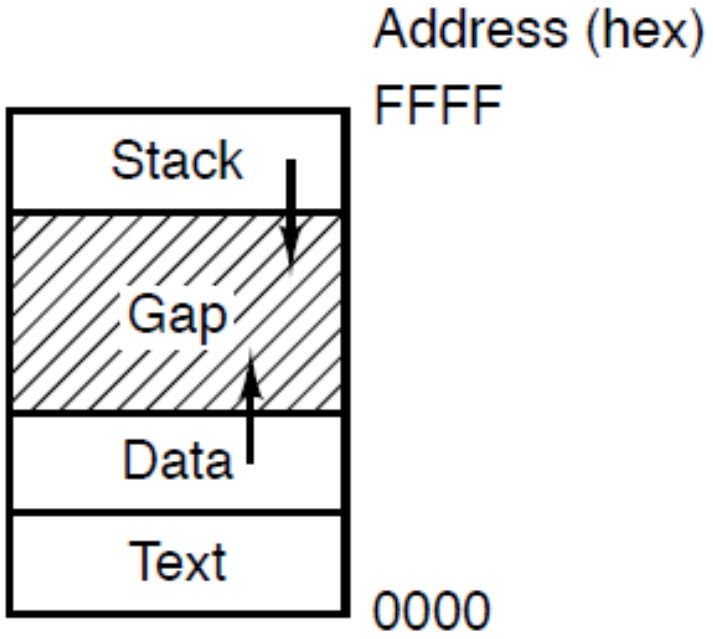


COOPERATING PROCESSES

- ***Independent*** process cannot affect or be affected by the execution of another process.
- ***Cooperating*** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience
- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process.
  - *unbounded-buffer* places no practical limit on the size of the buffer.
  - *bounded-buffer* assumes that there is a fixed buffer size.
- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

PROCESS AND STATE INFORMATION

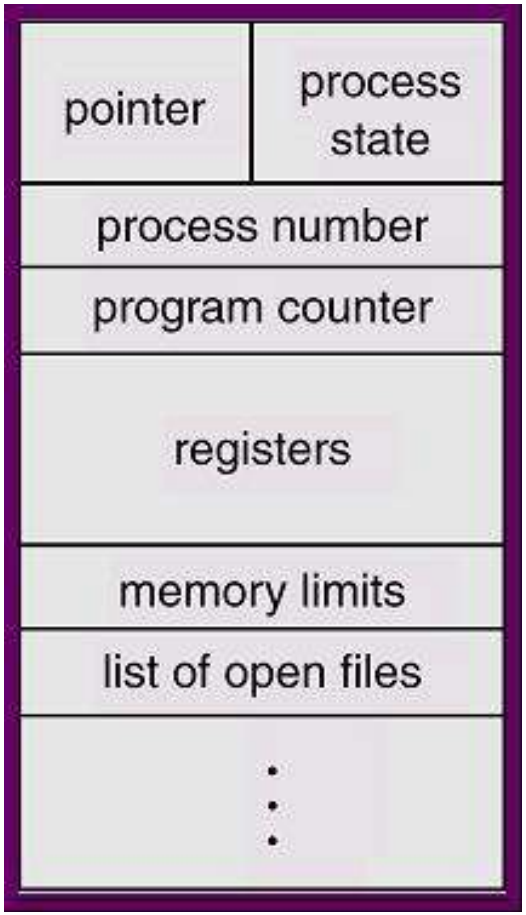
- Process – a program in execution; process execution must progress in sequential fashion.
- A process is an atomic and independent executing activity, with its own resources.
- A process includes:
  - program code (text section)
  - processor's registers including program counter
  - process stack (local variables, return address, etc.)
  - data section (global variables)
- As a process executes, it changes ***state***



Process Control Block (PCB)

Information associated with each process.

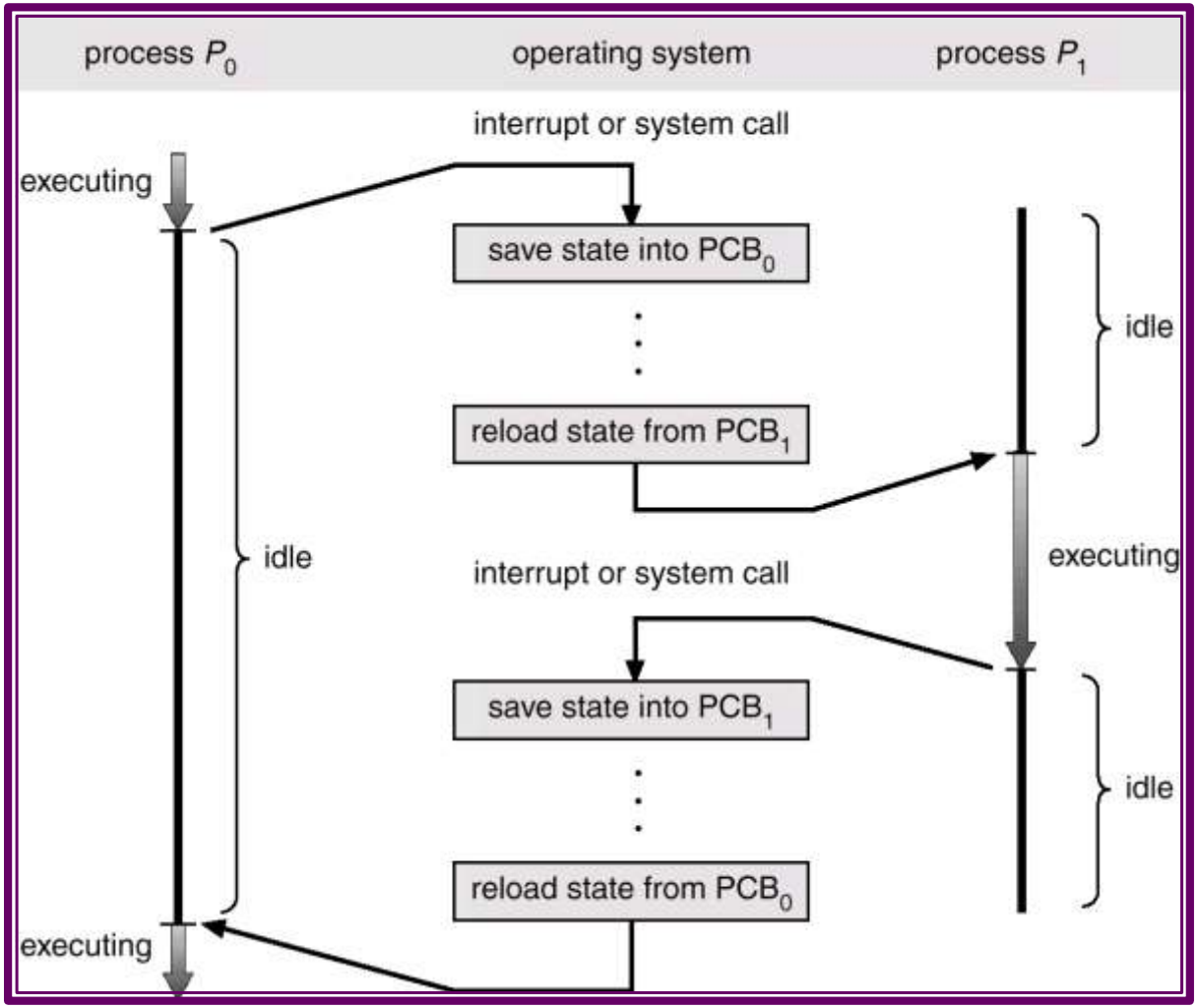
- ✓ Process state
- ✓ Program counter
- ✓ CPU registers
- ✓ CPU scheduling information
- ✓ Memory-management information
- ✓ Accounting information (CPU time, Memory, file, . . .
- ✓ I/O status information



CONTEXT SWITCH

CPU Switches From Process to Process

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process.
- Context-switch time is overhead; the system does no useful work while switching.
- Time dependent on hardware support.



CREAZIONE DEI PROCESSI

Quando viene creato un processo

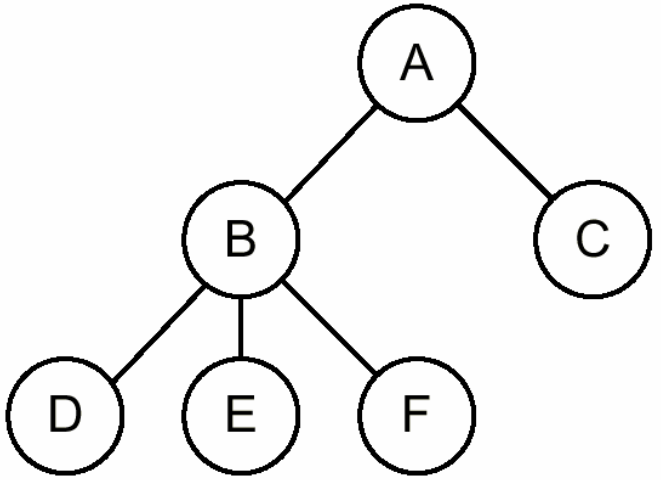
- **Al boot** del sistema (intrinseci, daemon)
- Su esecuzione di una **system call apposita** (es., fork())
- **Su richiesta** da parte dell'utente
- **All'inizio di un job batch**

CREAZIONE DI UN PROCESSO: LA CHIAMATA FORK

```
pid = fork();
if (pid < 0) {
              /* fork fallito */
}   else if (pid > 0) {
              /* codice eseguito solo dal padre */
}   else {
              /* codice eseguito solo dal figlio */
}
              /* codice eseguito da entrambi */
```

PROCESS CREATION

- Parent process create children processes, which, in turn create other processes, forming a tree of processes.
- Resource sharing
  - Parent and children share all resources.
  - Children share subset of parent's resources.
  - Parent and child share no resources.
- Execution
  - Parent and children execute concurrently.
  - Parent waits until children terminate.
- Address space
  - Child duplicate of parent.
  - Child has a program loaded into it.
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program.



A process tree.

Process A created two child processes, B and C.
Process B created three child processes, D,E, and F.

PROCESS TERMINATION

↳ Process executes last statement and asks the operating system to decide it (**exit**).

- Terminazione volontaria—normale o con errore. I dati di output vengono ricevuti dal processo padre (che li attendeva con un wait).
- Terminazione involontaria: errore fatale (superamento limiti, operazioni illegali, . . .)
- Terminazione da parte di un altro processo (uccisione)
- Terminazione da parte del kernel (es.: il padre termina, e quindi vengono terminati tutti i discendenti: terminazione a cascata)

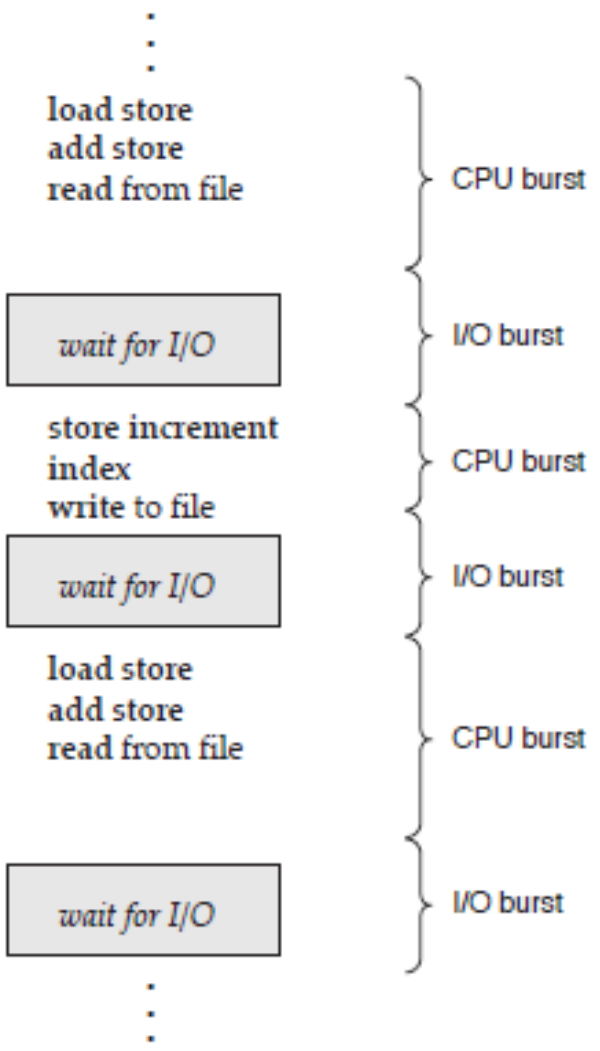
Le risorse del processo sono deallocate dal sistema operativo.

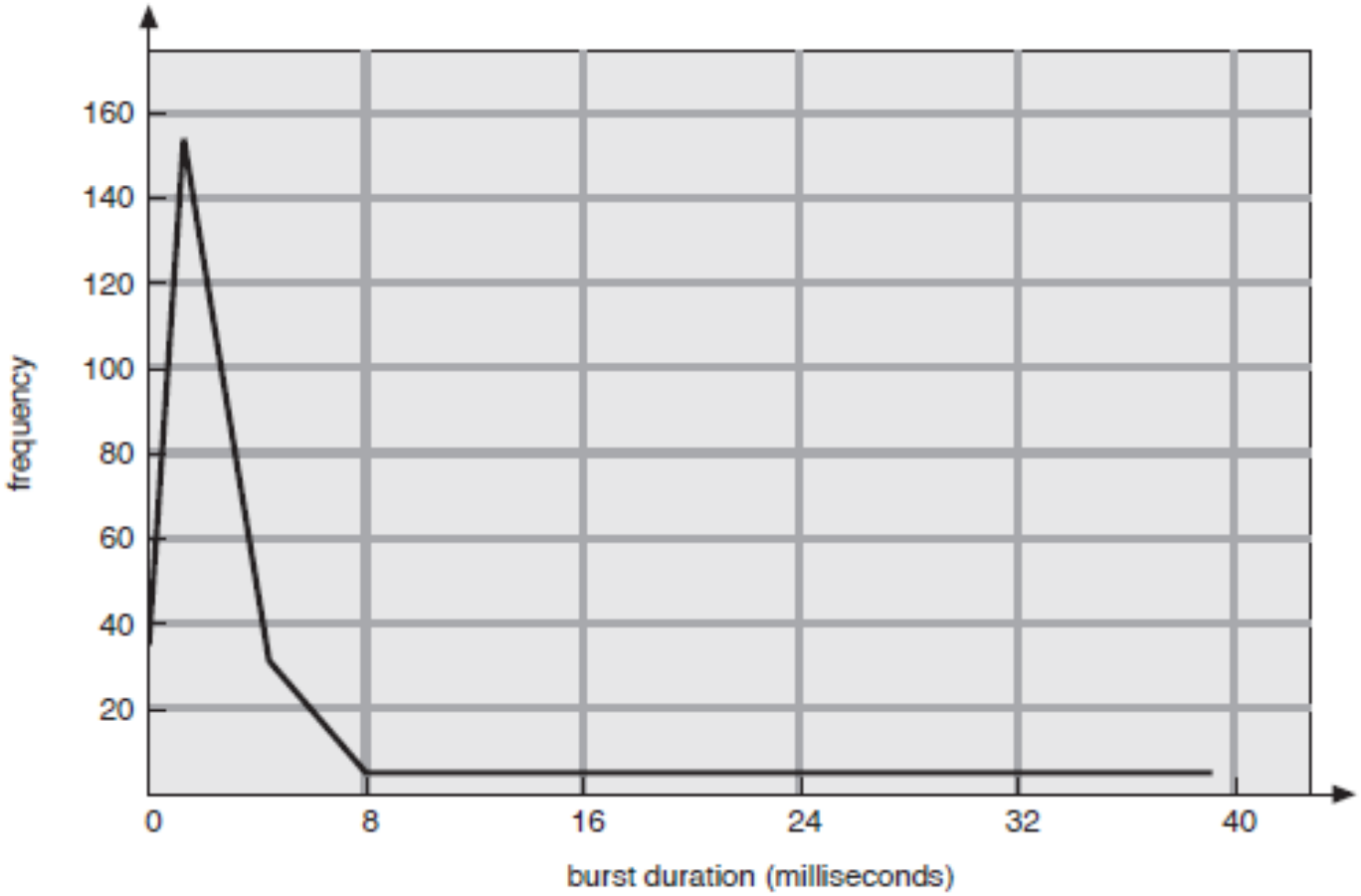
↳ Parent may terminate execution of children processes (**abort**).

- Child has exceeded allocated resources.
- Task assigned to child is no longer required.
- Parent is exiting.
  - Operating system does not allow child to continue if its parent terminates.
  - Cascading termination.

PROCESS EXECUTION

CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait.





SCHEDULERS

Long-term scheduler (or **job scheduler**) – selects which processes should be brought from the hold queue into the ready queue.

- Long-term scheduler is **invoked very infrequently** (seconds, minutes) ⇒ (may be slow).
- The long-term scheduler controls the *degree of multiprogramming.*

Short-term scheduler (or **CPU scheduler**) – selects which process should be executed next and allocates CPU.

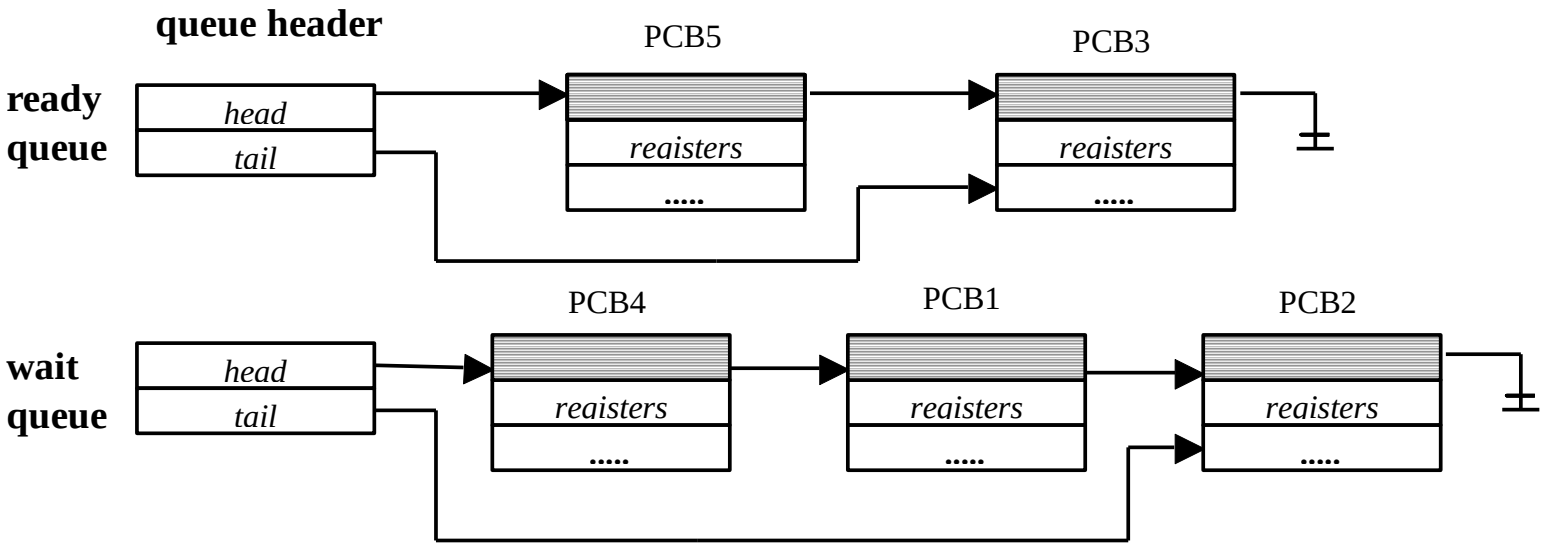
- Short-term scheduler is **invoked very frequently (milliseconds)** ⇒ (must be fast).
- Processes can be described as either:
  - *I/O-bound process* – spends more time doing I/O than computations, many short CPU bursts.
  - *CPU-bound process* – spends more time doing computations; few very long CPU bursts.

SCHEDULING QUEUES

Each state in the state diagram, except for the run state, corresponds to a **queue**.

The **wait queue** sometimes is splitted in several queues.

Each queue is built up through a **linked list** of PCB.

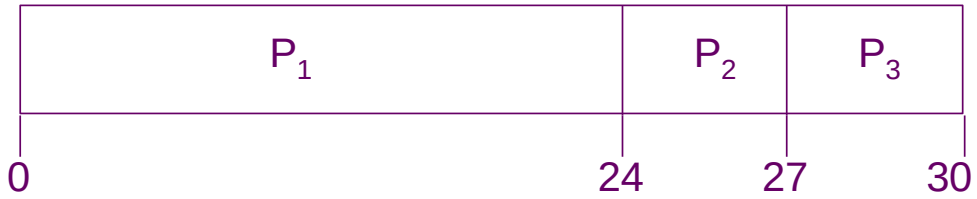


JOB SCHEDULER ALGORITHMS

First-Come, First-Served (FCFS) Scheduling

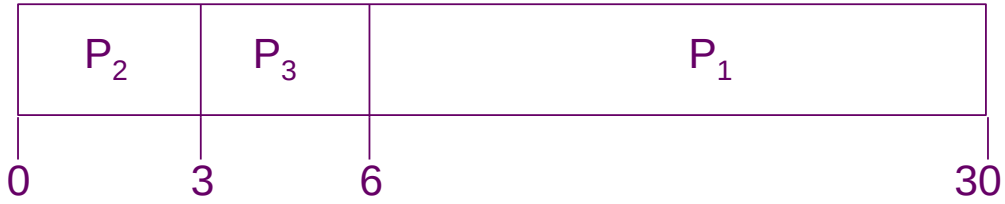
Process	Burst Time
P_1	24
P_2	3
P_3	3

Suppose that the processes arrive in the order: P_1 , P_2 , P_3. The Gantt Chart for the schedule is:



Waiting time for P_1 = 0; P_2 = 24; P_3 = 27 ***Average waiting time***: (0 + 24 + 27)/3 = 17

Suppose that the processes arrive in the order P_2 , P_3 , P_1 . The Gantt chart for the schedule is:



Waiting time for P_1 = 6; P_2 = 0; P_3 = 3 ***Average waiting time***: (6 + 0 + 3)/3 = 3

Much better than previous case. *Convoy effect* short process behind long process

JOB SCHEDULER ALGORITHMS

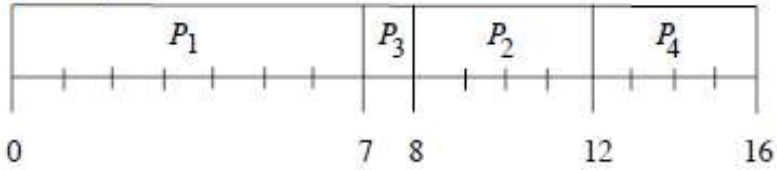
Shortest-Job-First (SJR) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- Two schemes:
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst.
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF).
- SJF is optimal – gives minimum average waiting time for a given set of processes.
- Risk: starvation

Esempio di SJF Non-Preemptive

Processo	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

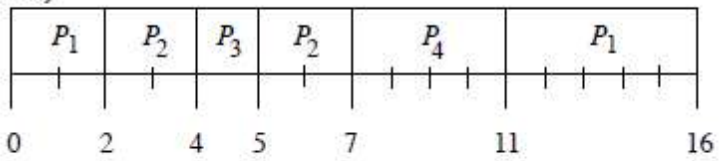


Tempo di attesa medio $= (0 + 6 + 3 + 7)/4 = 4$

Esempio di SJF Preemptive

Processo	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SRTF (preemptive)



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

CPU SCHEDULER

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state.
  2. Switches from running to ready state.
  3. Switches from waiting to ready.
  4. Terminates.
- Scheduling under 1 and 4 are ***nonpreemptive***.
- All other scheduling are ***preemptive***.

DISPATCHER

- ***Dispatcher module*** gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - Φ switching context
  - Φ switching the CPU from supervisor mode to user mode
  - Φ jumping to the proper location in the user program to restart that program

- ***Must be speedy***

- ***Dispatch latency*** – time it takes for the dispatcher to stop one process and start another running.

SCHEDULING CRITERIA

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process
- Waiting time – amount of time a process has been waiting in the ready queue
- Response time – amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing environment)

OPTIMIZATION CRITERIA

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

OBIETTIVI GENERALI DI UN ALGORITMO DI SCHEDULING

All systems

Fairness - giving each process a fair share of the CPU
Policy enforcement - seeing that stated policy is carried out
Balance - keeping all parts of the system busy

Batch systems

Throughput - maximize jobs per hour
Turnaround time - minimize time between submission and termination
CPU utilization - keep the CPU busy all the time

Interactive systems

Response time - respond to requests quickly
Proportionality - meet users' expectations

Real-time systems

Meeting deadlines - avoid losing data
Predictability - avoid quality degradation in multimedia systems

Nota: **in generale, non esiste soluzione ottima sotto tutti gli aspetti**

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;

- ✓ ***multilevel queue***
  vengono organizzate diverse code di attesa;

- ✓ ***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

Round Robin (RR)

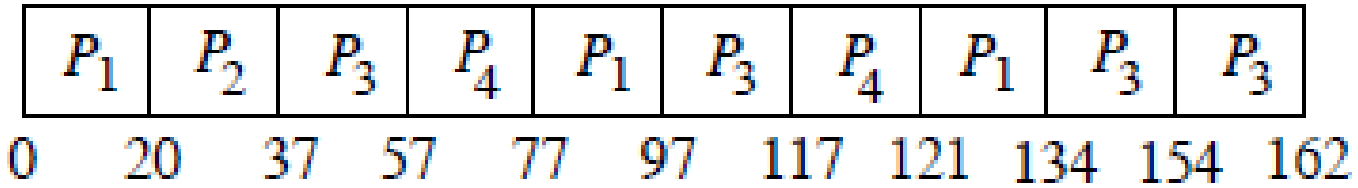
- Classical preemption algorithm for time-sharing systems: as FCFS but with quantized preemption.
- Each process gets a small unit of CPU time (*time slice*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
- Performance
  - *q* large $\Rightarrow$ FIFO
  - *q* small $\Rightarrow$ *q* must be large with respect to context switch, otherwise overhead is too high.
  - 80% of CPU bursts should be <= *q*

CPU SCHEDULER

Round Robin (RR) example with 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

CPU SCHEDULER

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

Static priority

- Priorities can be defined:
  - Internally, with parameters measured from the system on the single process (CPU time usage, opened files, memory, I/O usage, . . .)
  - Externally, through process relevance, process owner, economical cost, . . .)
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority).
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time.
- Problem ≡ Starvation – low priority processes may never execute, due to a continuous flow of higher priority processes.
- Solution ≡ Aging – as time progresses increase the priority of the unexecuted process.

CPU SCHEDULER

Multilevel Queue

- Ready queue is partitioned into separate queues:
  - system processes
  - foreground (interactive)
  - background (batch)
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues.
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS

CPU SCHEDULER

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 volte che il processo i-esimo ha completato l'uso del time slice (negli altri casi, il processo è andato nello stato di WAIT)
  - $n_i \leq N_i$
  - $0 \leq R_i \leq 1$;

CPU SCHEDULER

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 $R_i \rightarrow 0$.

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

- CPU scheduling more complex when multiple CPUs are available.
- *Homogeneous processors* within a multiprocessor: the next task can run on anyone of the processors. Anyway a task can be executed on a specific processor.
- *Load sharing*: all the processors select the processes from the same ready queue.
- *Asymmetric multiprocessing* – only one processor accesses the system data structures, alleviating the need for data sharing.
- *Symmetric multiprocessing* - the system data structures can be shared; needs special hardware and synchronization controls within the kernel.
- More details will be given later.

REAL-TIME SCHEDULING

- ***Hard real-time*** systems – required to complete a critical task within a precise and guaranteed amount of time.
- ***Firm real-time*** systems – the completion time violation makes the results usefulness to decrease as more as the the time delay increases.
- ***Soft real-time*** systems – requires that critical processes receive priority over less fortunate ones.

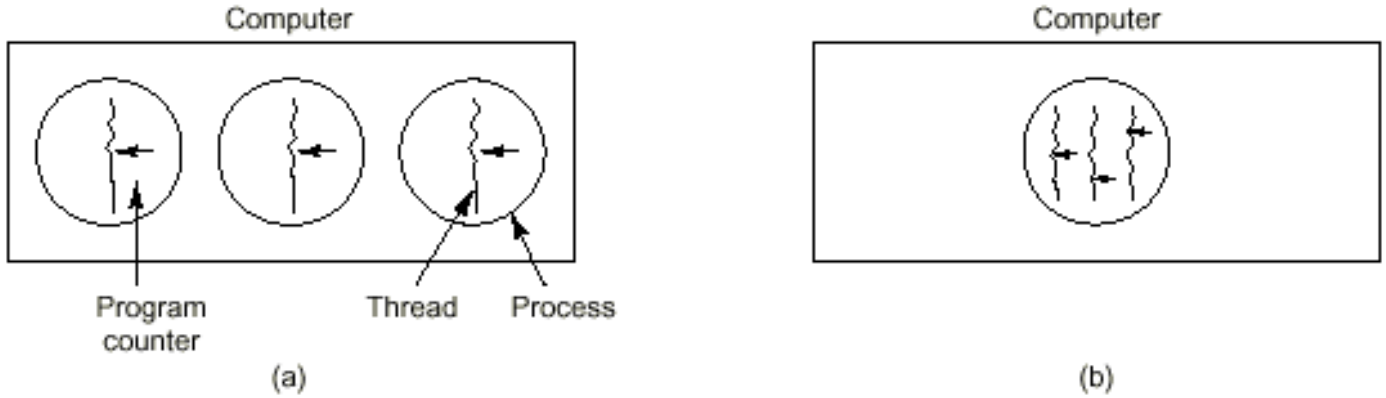
More details will be given later.

THREADS

Φ un processo è un programma la cui esecuzione si svolge seguendo un singolo flusso di controllo (thread) ⇨ *figura (a);*

Φ se un processo ha più di un flusso di controllo, esso si dice che è costituito da più thread (multi-thread);

Φ 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);*



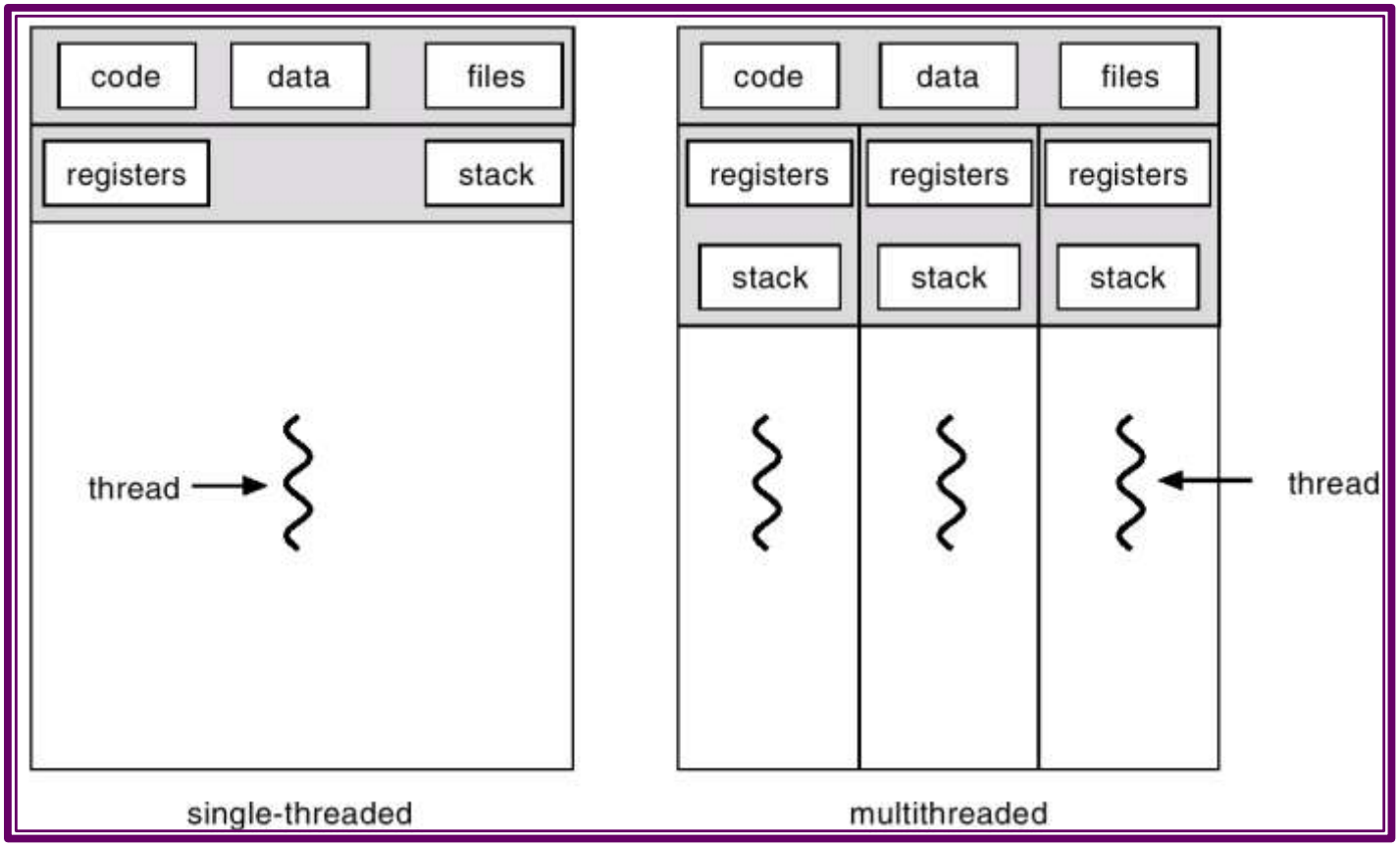
Φ tutti i thread di uno stesso processo condividono:

- la sezione del codice,
- la sezione dei dati,
- altre risorse (file utilizzati, segnali ricevuti, ecc.);

Φ ad ogni singolo thread sono associati:

- un identificatore,
- un program counter,
- un insieme di registri,
- uno 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

I tipi di realizzazione

Gestione dei thread fatta a run-time da una libreria (thread package): per creare un thread e passare il controllo ad un thread tramite uno scheduler, per salvare il contesto di un thread e ripristinare quello di un altro thread.

- 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; esso si occupa del cambio dei thread senza chiamare il SO;
- Un sistema multithread perde il controllo della CPU solo quando richiede un servizio del SO; l'intero sistema multithread 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 multithread, 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.
- Esempi: POSIX *Pthreads,* Mach *C-threads,* Solaris *threads*

- Supported by the **Kernel**: Esempi: Windows 95/98/NT/2000, Solaris, Tru64 UNIX, BeOS, Linux
- Implemented by the **combination 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 di UNIX, 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.

THREADS

Politiche di scheduling dei thread

- Round robin;
- Priorità statica.

↳ Si può realizzare un sistema multithread con una certa politica di scheduling ed un altro sistema multithread 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.

THREADS

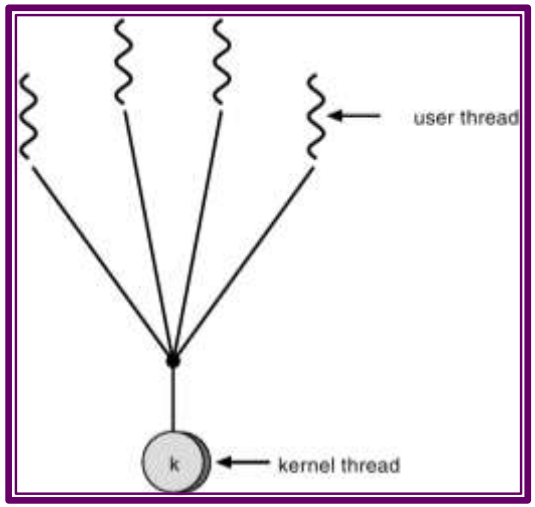
I vantaggi

- Responsiveness
- Resource Sharing
- Economy
- Utilization of Multi-Processors Architectures

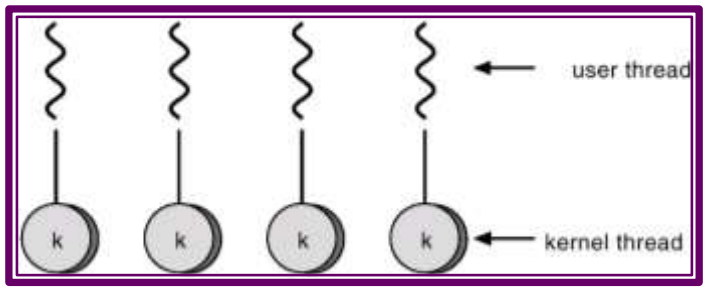
Rinunciando all'atomicità dei processi (attraverso la loro decomposizione in thread) si ha un elevato guadagno di tempo, poiché si evitano i continui cambi di contesto computazionale.

MULTITHREADING MODELS

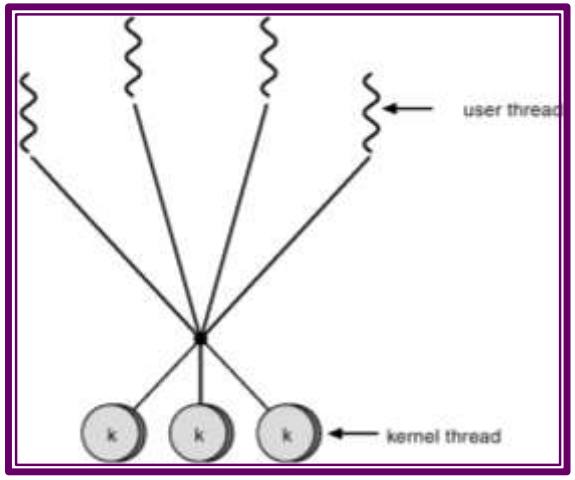
- ***Many-to-One:*** **Many user-level threads mapped to single kernel process (thread).**
  Prima di sospendersi un thread avvia il successore. La commutazione è veloce perché avviene nello stesso spazio utente. Però se un thread si blocca, il kernel blocca l'intero processo.



One-to-One: **Each user-level thread maps to one kernel thread.**
Se uno user thread si blocca, si blocca solo lui e inoltre più thread possono usare più processori.
Inconveniente: overhead per creazione kernel thread



- ***Many-to-Many:*** **Allows many user level threads to be mapped to many kernel threads.**
  Non ha gli svantaggi dell'uno-a-uno e del molti-a-uno.



THREADS ORGANIZATIONS

L'organizzazione Dispatcher/Worker

- multithreading utilizzato in Java
- Prevede un thread principale (dispatcher) 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.

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.

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.

THREADING ISSUES

Semantics of fork()

- una fork in un thread può generare un processo con tutti i thread duplicati oppure con il solo thread che ha emesso la fork.

exec() system calls

- il programma specificato come parametro sostituirà il processo, con tutti i suoi thread, a cui appartiene il thread chiamante.

Thread cancellation

- può essere immediata o differita.

Signal handling in multithread

- the signal is delivered to:
  - the thread to which the signal applies
  - every thread in the process
  - certain threads in the process
  - a specific thread to receive all signals for the process.

Thread pools in a web server

- create a number of threads at process startup and place them into a pool. When the server receives a request, it passes the request to one available thread from the pool.

Thread specific data

- used when each thread needs its own copy of data, instead of sharing the data with other threads belonging to the process.

JAVA THREADS

- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

Java threads are managed by the 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.