Gestione dei processi di elaborazione

Processi

- Concetto di Processo
- Scheduling di Processi
- Operazioni su Processi
- Processi Cooperanti
- Concetto di Thread
- Modelli Multithread
- I thread in diversi S.O.

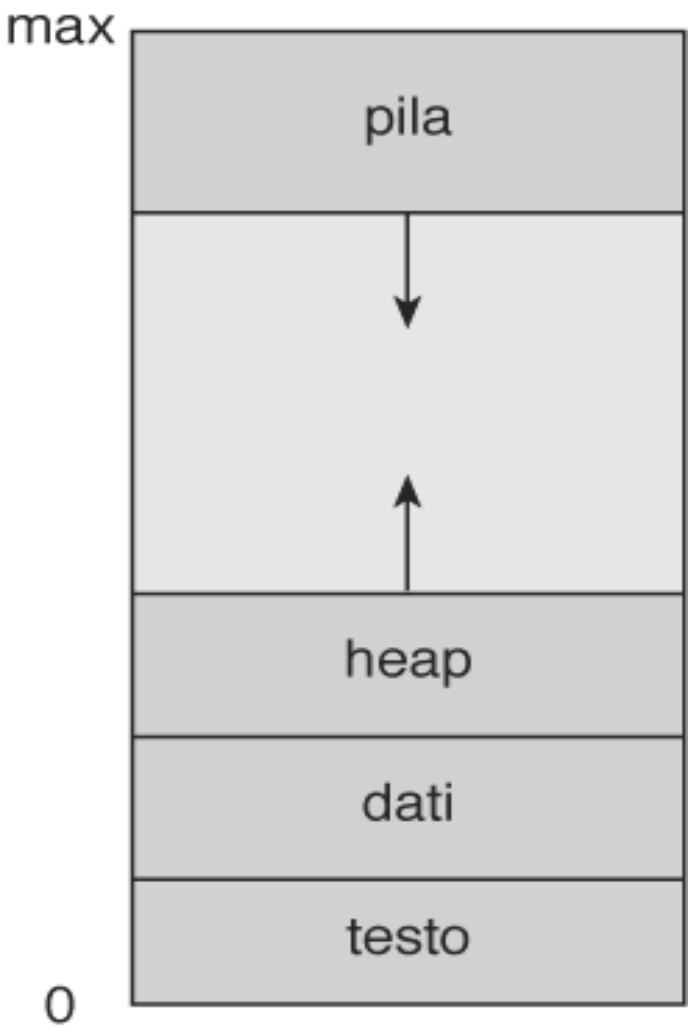
Concetto di Processo

- L'esecuzione di programmi ha diversi nomi in diversi contesti:
  - Sistemi Batch – **job**
  - Sistemi Time-sharing– **processo** o **task**
- I termini ***job*** e ***processo*** si usano spesso come sinonimi.

- **Processo** di elaborazione: un programma in esecuzione; l'esecuzione di un singolo processo avviene in maniera sequenziale.

- Un processo include:
  - sezione testo (codice),
  - il program counter,
  - lo stack,
  - la sezione dati.

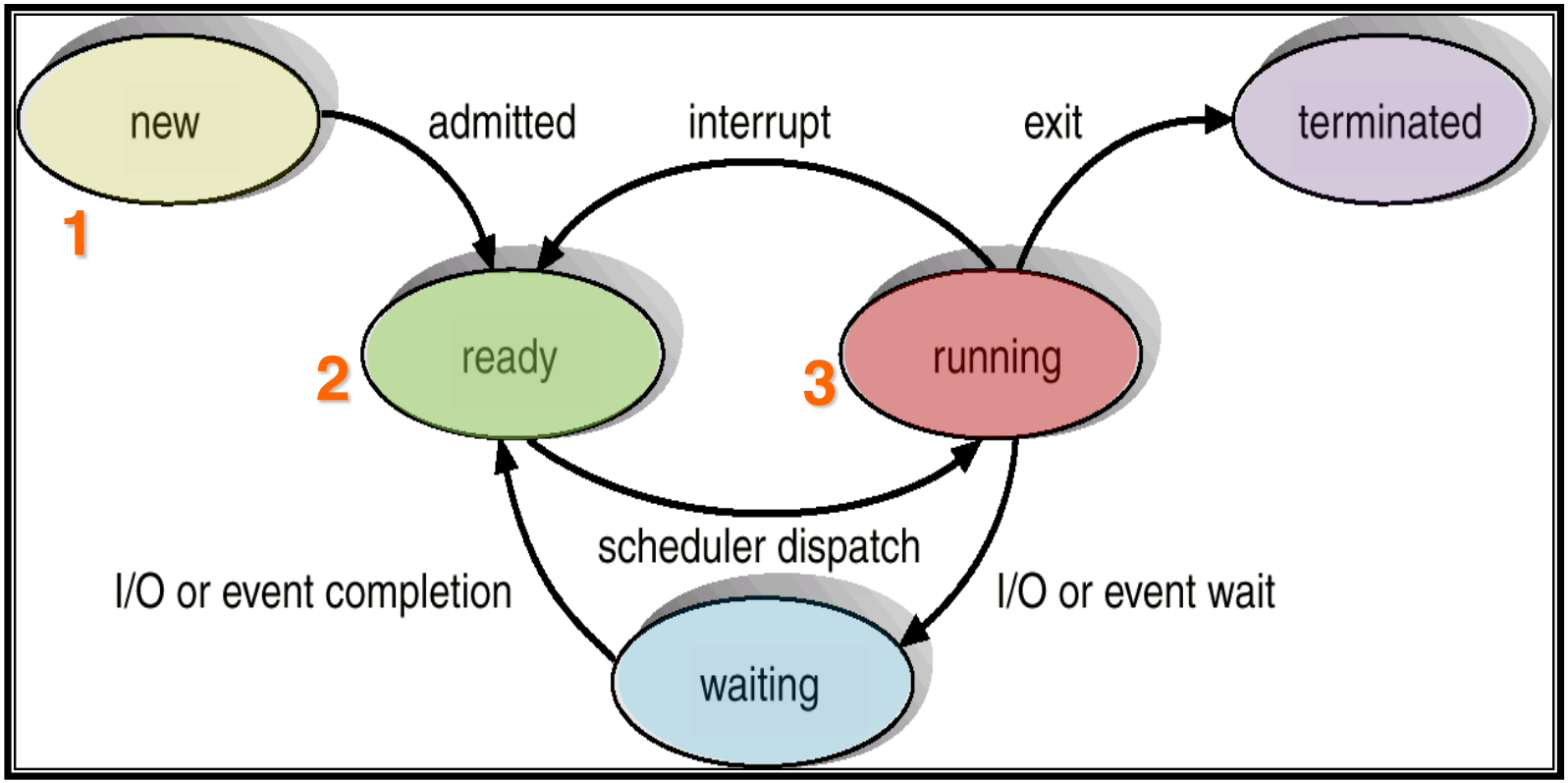
Processo in memoria



Stato del Processo

- Durante la sua esecuzione un processo cambia il proprio ***stato*** che può essere:
  - **new**: Il processo viene creato.
  - **running**: Il processo (le sue istruzioni) è in esecuzione.
  - **waiting**: Il processo è in attesa di un dato evento.
  - **ready**: Il processo è pronto per essere eseguito.
  - **terminated**: Il processo ha completato la sua esecuzione.

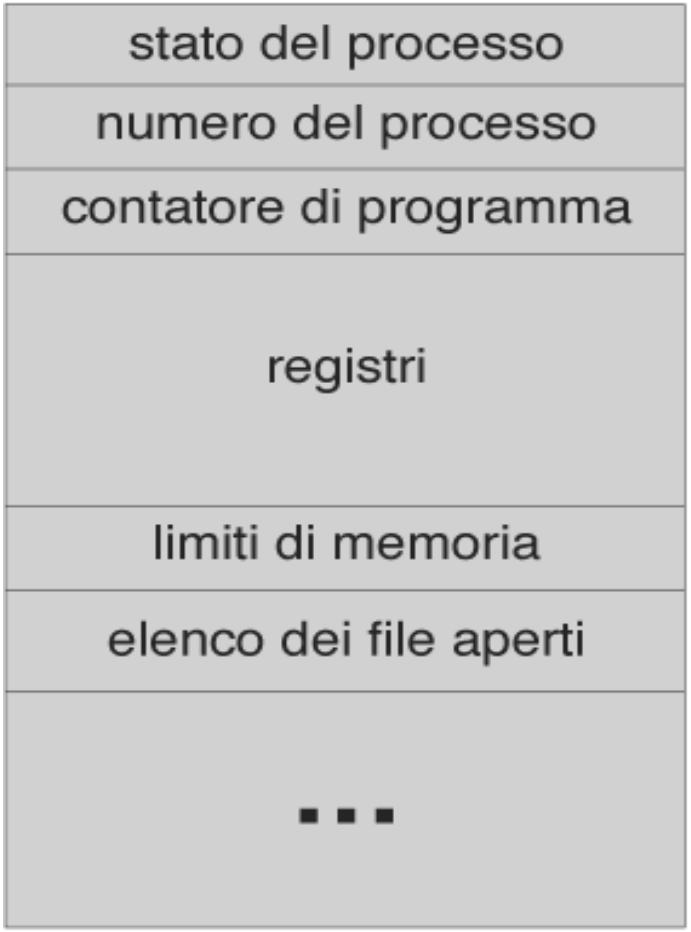
Diagramma di stato di un Processo



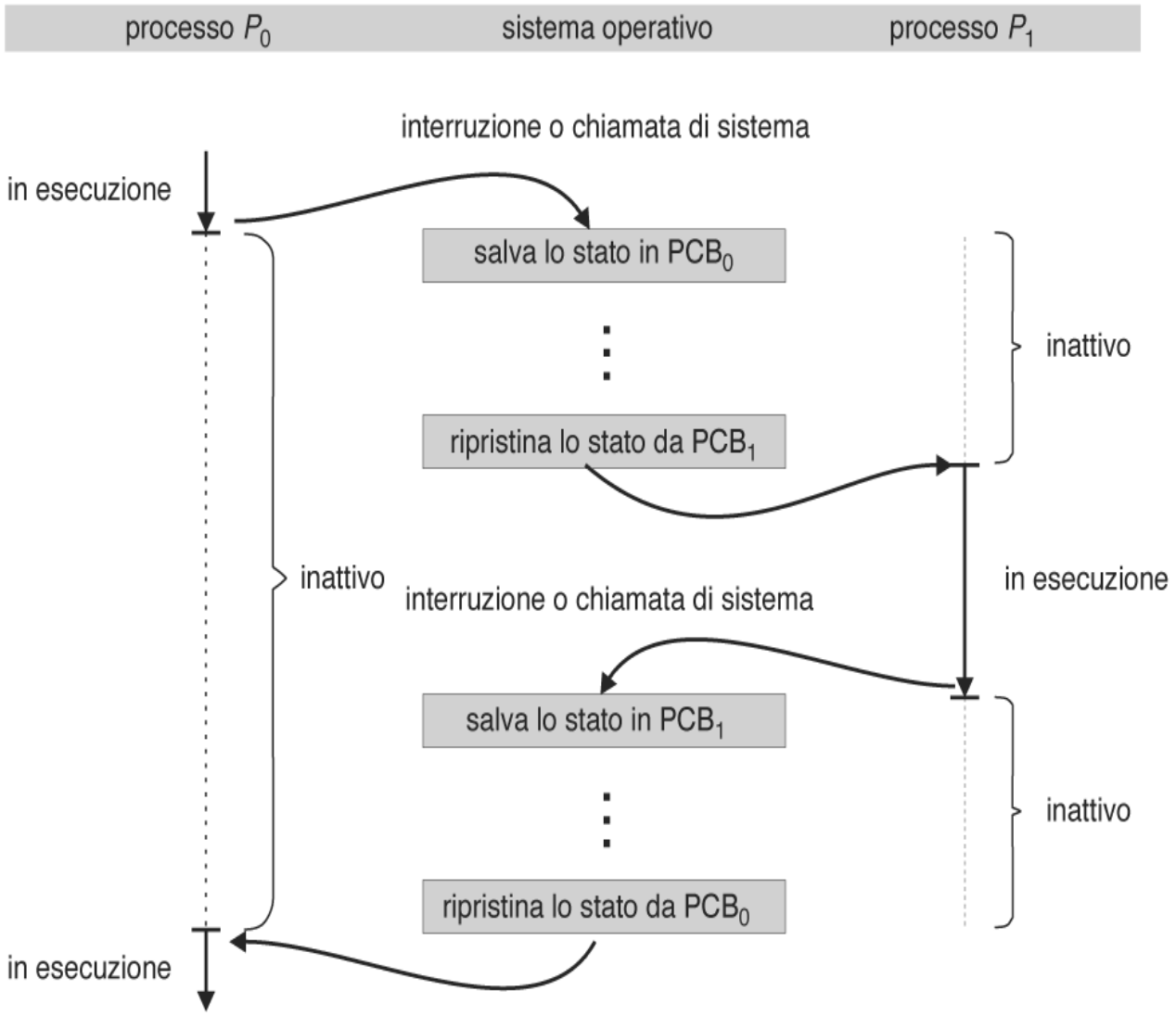
Process Control Block (PCB)

- Il PCB contiene l'informazione associata ad ogni processo:
  - stato del processo
  - program counter
  - registri della CPU
  - info sullo scheduling della CPU
  - informazioni di memory-management
  - informazioni di accounting
  - stato dell'I/O
  - ID del processo
  - ID dell'utente.

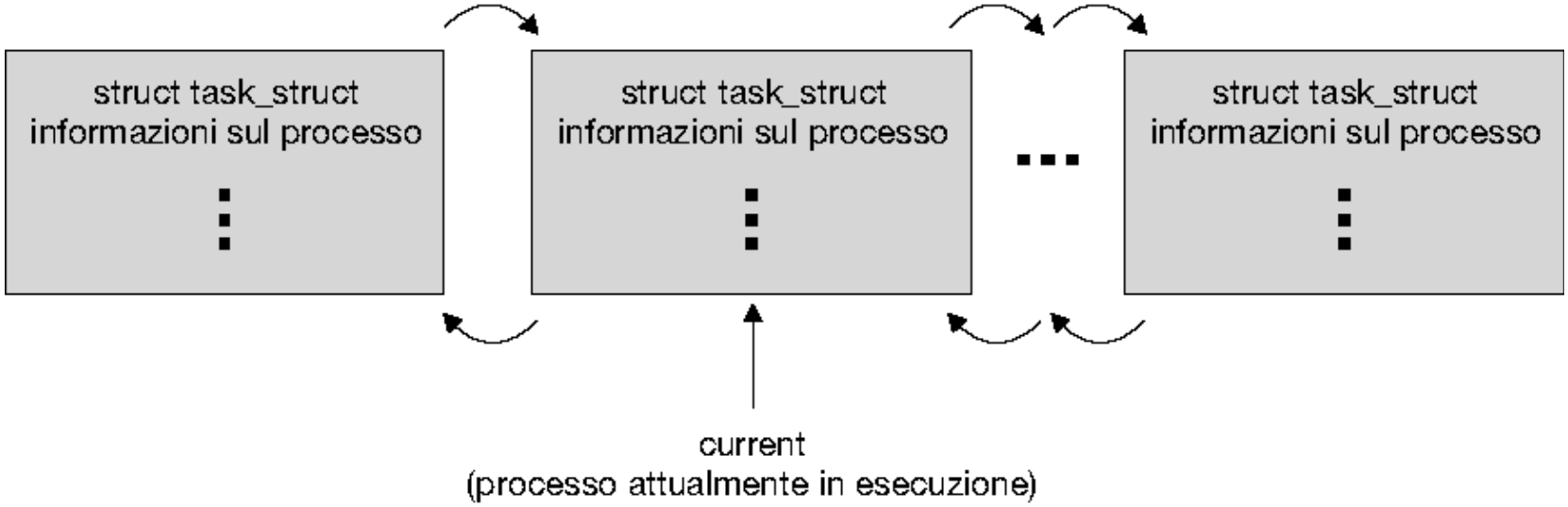
Blocco di controllo di un processo (PCB)



Commutazione della CPU tra due processi



Processi attivi di Linux



Struttura dati per la gestione del Blocco di controllo di un processo in Linux

- Lista bilinkata dei PCB (task_struct)
- Indirizzo del PCB del processo in esecuzione (current)
- Gestione efficiente e dinamica dei PCB.

Code di Scheduling

- **Coda dei processi** – l'insieme di tutti i processi nel sistema.
- **Ready queue** (coda dei processi pronti) – l'insieme dei processi in memoria centrale pronti per essere eseguiti.
- **Coda del dispositivo** – l'insieme dei processi in attesa di usare un dispositivo. (Più code)
- I processi passano da una coda all'altra mentre cambiano stato.

Ready Queue e code dei dispositivi di I/O

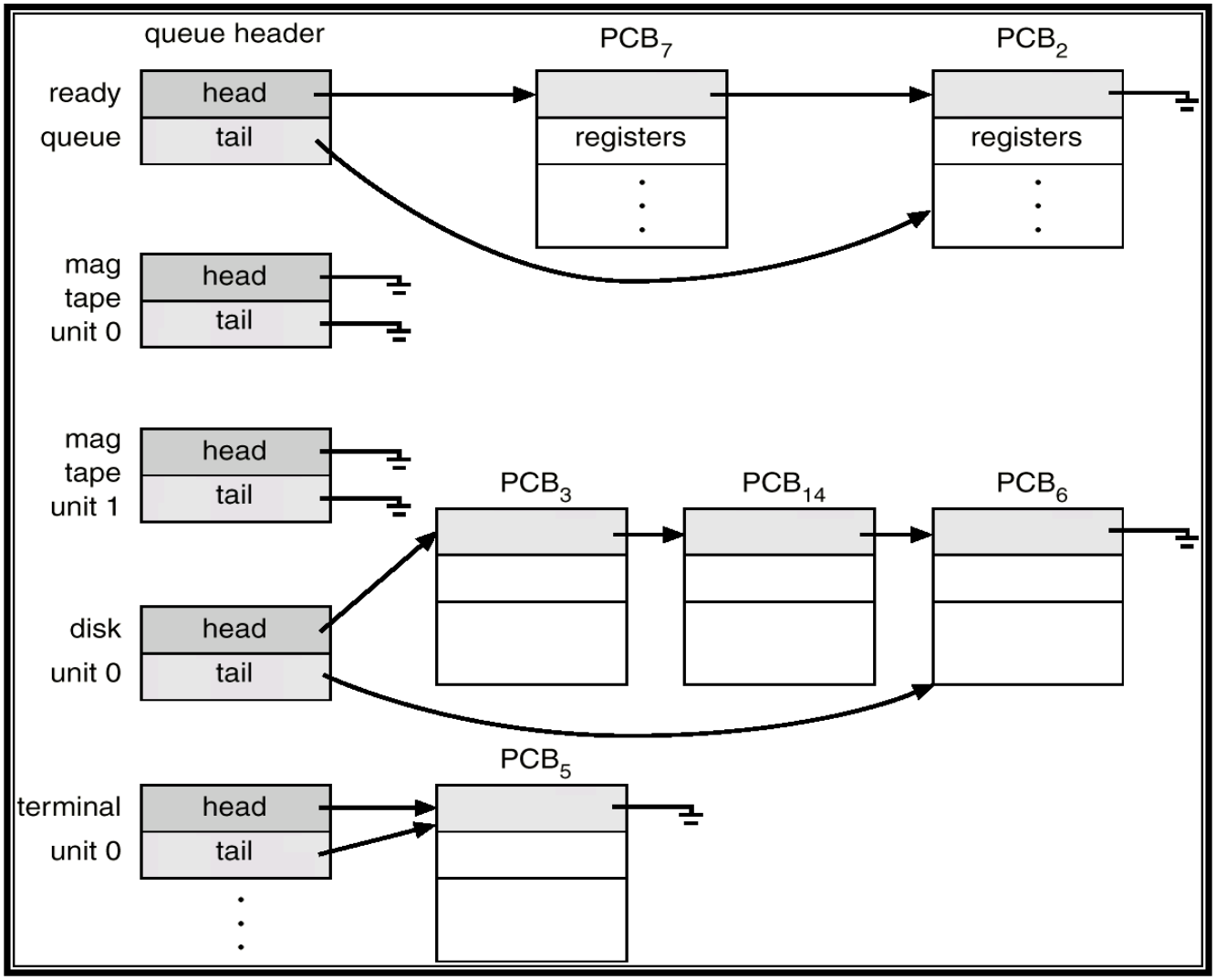
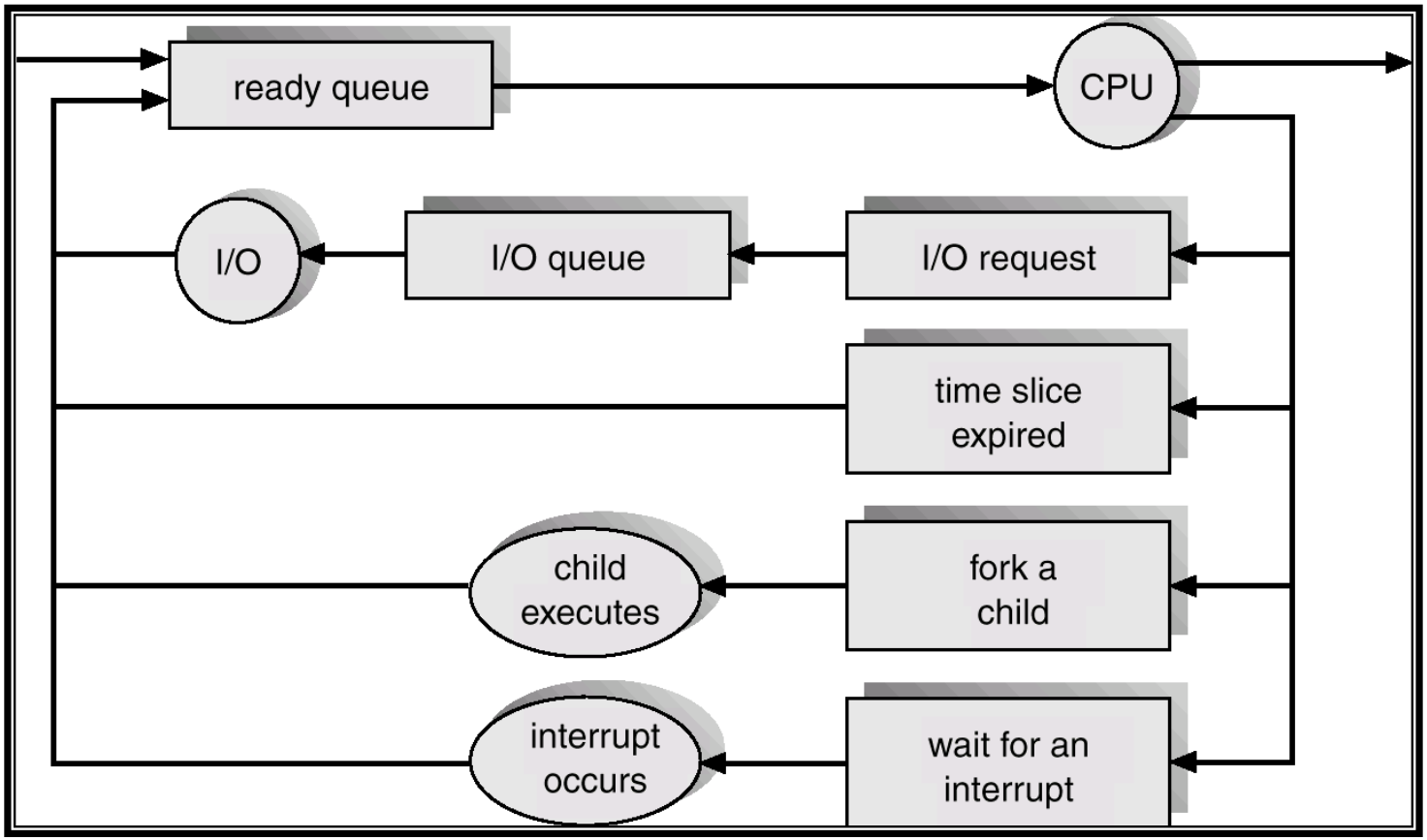


Diagramma di accodamento



Tipi di scheduler

In un sistema possono esistere più scheduler (es. sistemi batch).

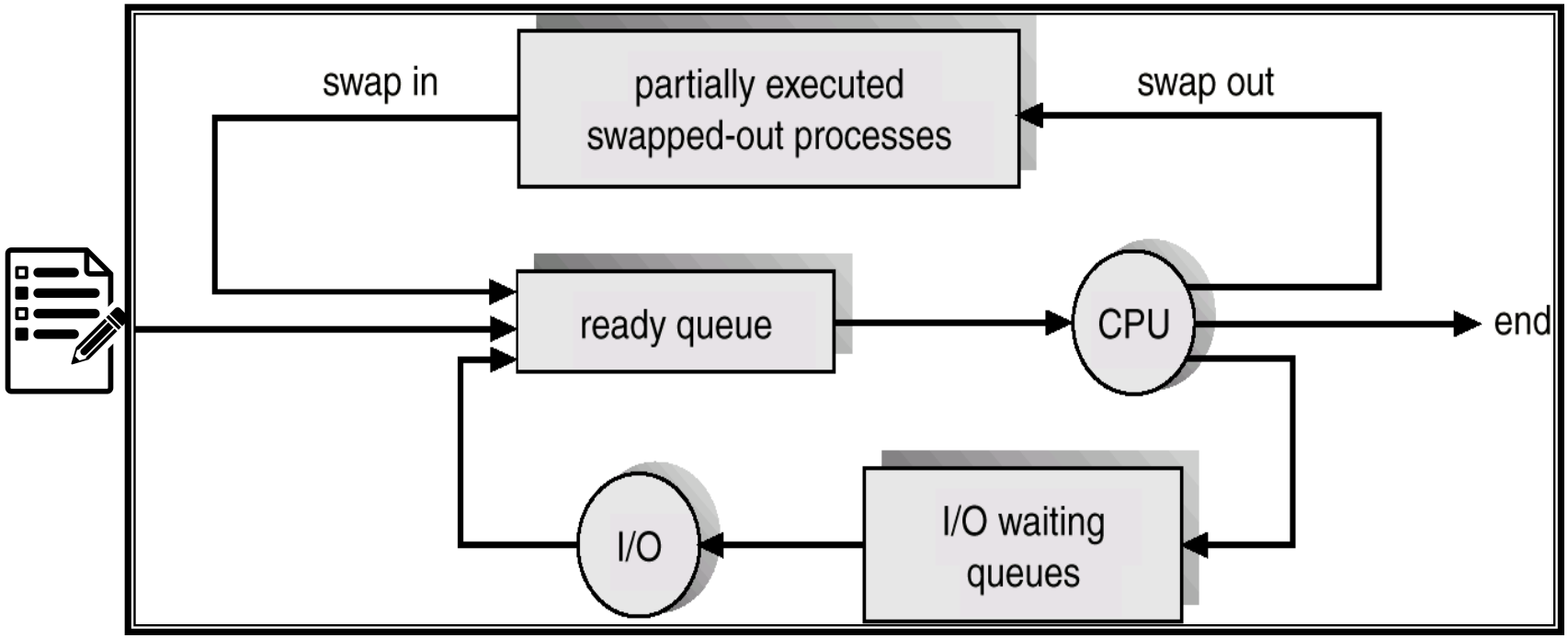
- **Scheduler a lungo termine** (o **job scheduler**) – seleziona i processi da inserire nella *ready queue* (la coda dei processi pronti).

- **Scheduler a breve termine** (o **CPU scheduler**) – seleziona tra i processi pronti quelli che devono essere eseguiti.

Scheduler a medio termine

- In alcuni sistemi time-sharing esiste uno **scheduler a medio termine** che gestisce i processi pronti in memoria centrale (**swapper**)
- In alcuni casi rimuove i processi dalla memoria (**swap-out**) per riportarli in memoria (**swap-in**) quando sarà possibile.
- Questo migliora l'utilizzo della memoria in caso di una alta richiesta di esecuzione di processi.

Scheduler a medio termine



Schedulers

- Lo scheduler a breve termine è invocato molto frequentemente (millisecondi) $\Rightarrow$ (deve essere veloce).
- Lo scheduler a lungo termine è invocato non molto spesso (secondi, minuti) $\Rightarrow$ (può essere lento).
  - Lo scheduler a lungo termine controlla il grado di multi-programmazione.
- I processi possono essere classificati come:
  - *processi* I/O-*bound* – basso uso della CPU e elevato uso dell'I/O.
  - *processi CPU-bound* – elevato uso della CPU e basso uso dell'I/O.

Context Switch

- **Context switch**: operazione di passaggio da un processo all'altro da parte della CPU.
- Il tempo impiegato per il context-switch è un costo: il sistema non effettua lavoro utile per nessun processo utente.
- Il tempo di context switch dipende dal supporto offerto dall'hardware.

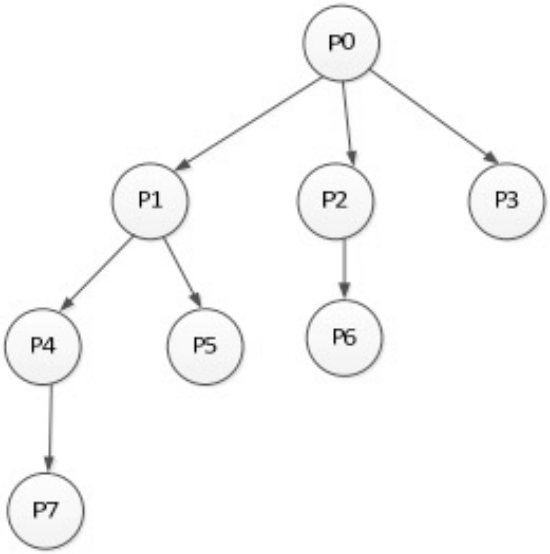
Operazioni sui processi: Creazione

- Un processo qualsiasi può creare altri processi come suoi figli i quali possono creare altri processi, e cosi via.
- **Il sistema operativo crea i processi utente come processi figli.**
- Possibile condivisione di risorse:
  - Processi padri e figli condividono tutte le risorse.
  - Un processo figlio condivide una parte delle risorse del padre.
  - Processi padri e figli non condividono risorse.
- Approcci di esecuzione:
  - Processi padri e figli eseguono concorrentemente.
  - Il padre rimane in attesa della terminazione dei figli.

Creazione

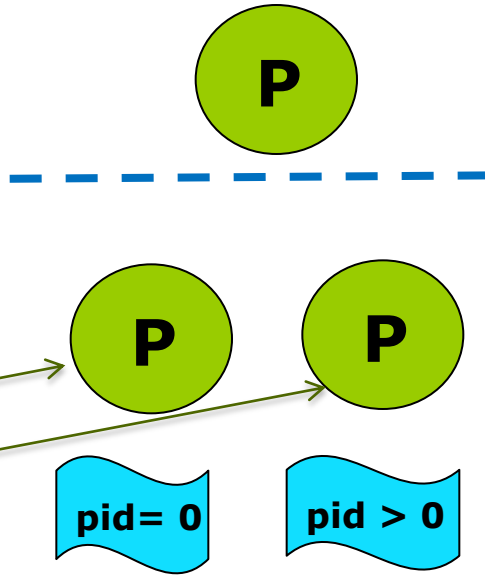
- Spazio di indirizzi:
  - Il processo figlio viene duplicato dal processo padre.
  - Il processo figlio ha un proprio codice.

- Esempio: UNIX
  - **fork**: system call che crea un nuovo processo.
  - **exec(nuovo prog):** system call usata dopo una fork per sostituire allo spazio di memoria di un processo un nuovo programma.

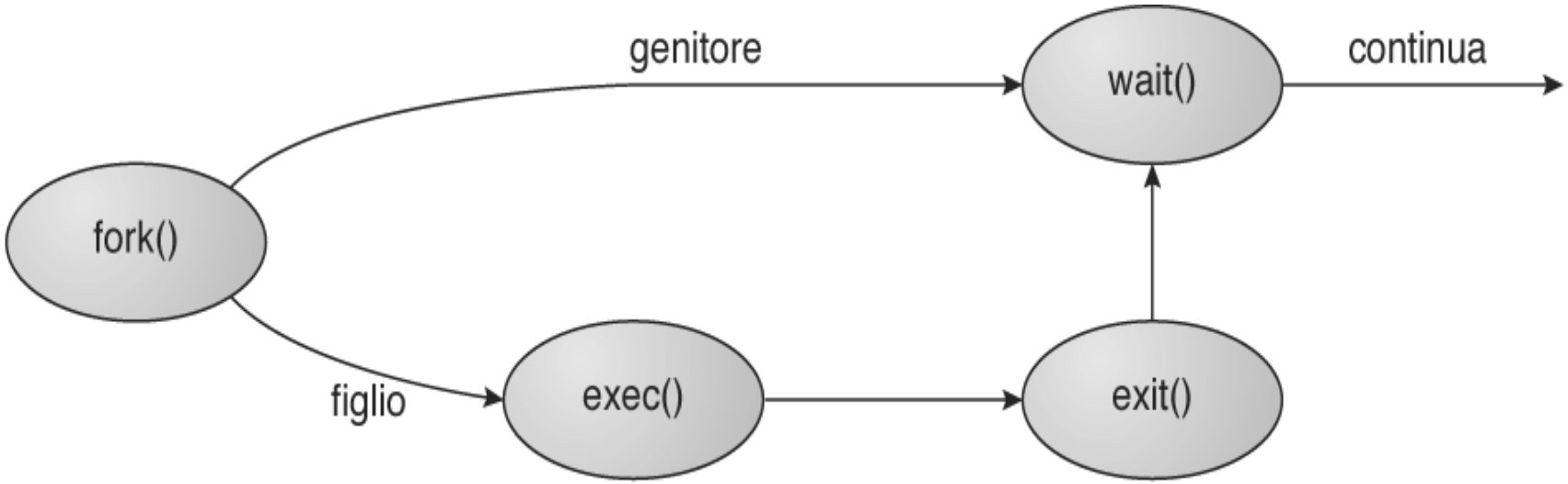


Creazione di un processo mediante fork()

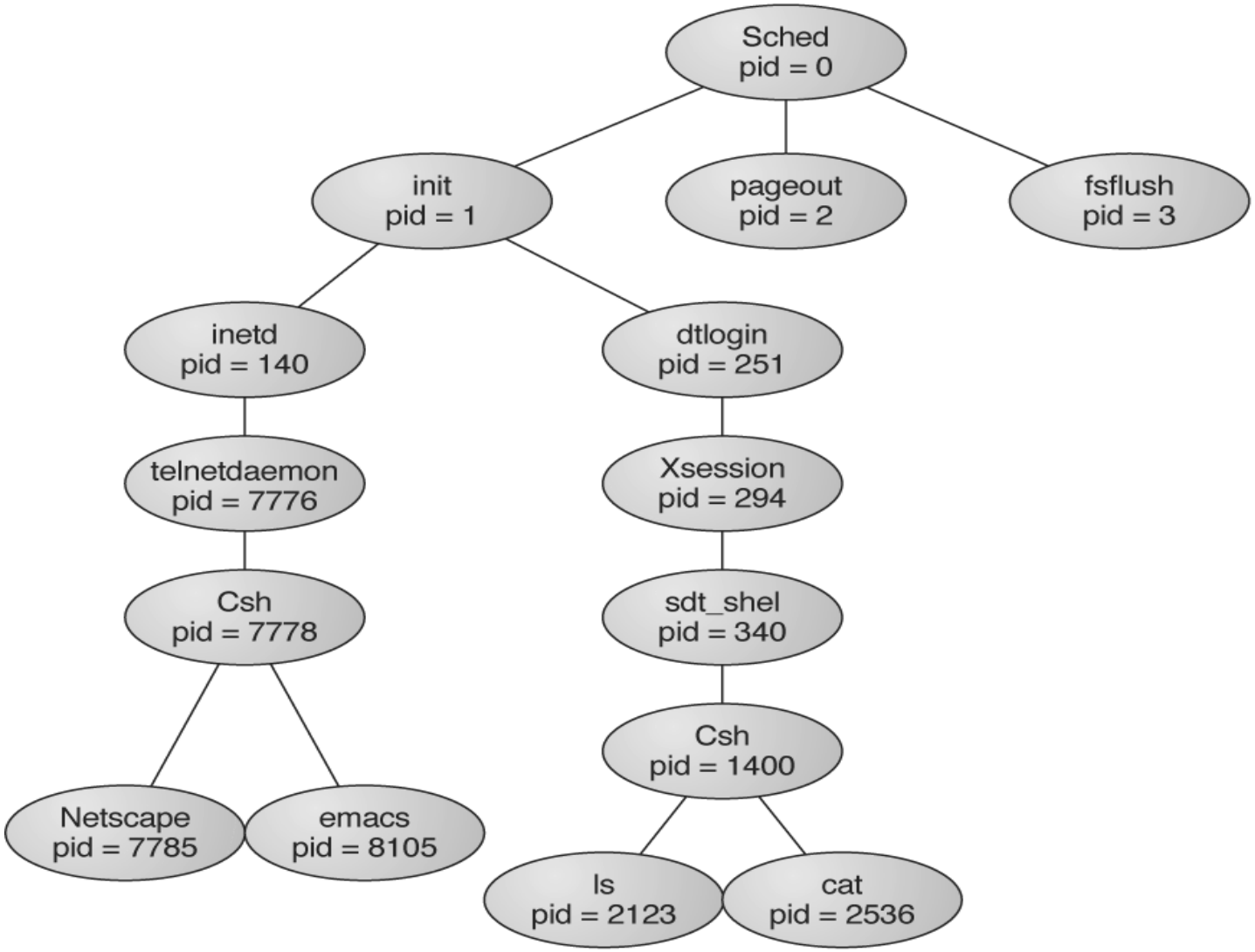
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

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

    if (pid < 0) { /* errore */
      fprintf(stderr, "generazione del nuovo processo fallita");
      return 1;
    }
    else if (pid == 0) { /* processo figlio */
      execlp("/bin/ls", "ls", NULL);
    }
    else {/* processo genitore */
      /* il genitore attende il completamento del figlio */
      wait(NULL);
      printf("il processo figlio ha terminato");
      }

      return 0;
}
```



Creazione di un processo mediante fork()



Esempio di albero dei processi in Solaris



Terminazione di un processo

- Un processo esegue l'ultima istruzione e chiede al sistema operativo di terminare (**exit**).
  - risultati dal figlio al padre (tramite **wait**).
  - Le risorse del processo sono dealloccate dal sistema operativo.

- Un processo può eseguire la terminazione dei propri figli (tramite **abort**) perché:
  - Il processo figlio non è più utile.
  - il figlio ha usato risorse in eccesso.
  - Il processo padre termina.
    - Molti sistemi non permettono ai figli di eseguire quando il processo padre termina.
      - Terminazione a cascata.

Processi Indipendenti o Cooperanti

- I *processi indipendenti* non interagiscono con altri processi durante la loro esecuzione.

- *I processi cooperanti* influenzano o possono essere influenzati da altri processi. Il comportamento dipende anche dall'ambiente esterno.

- Vantaggi della cooperazione:
  - Condivisione dell'informazione
  - Velocità di esecuzione
  - Modularità
  - Distribuzione
  - Convenienza.

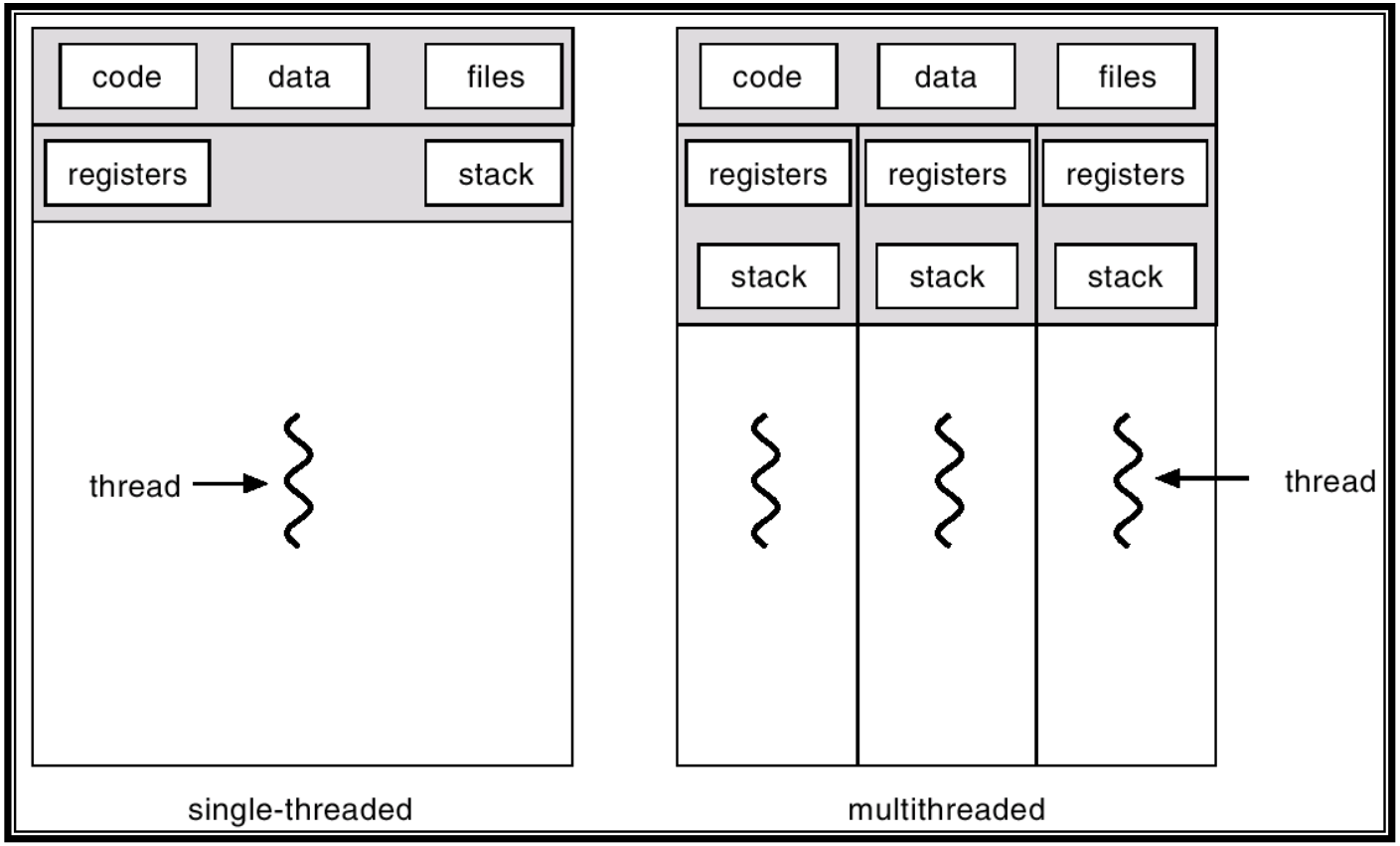
Processi Cooperanti

- *I processi cooperanti* possono interagire tramite:
  - Scambio esplicito di dati,
  - Sincronizzazione su un particolare evento.
  - Condivisione dell'informazione.
- I sistemi operativi offrono meccanismi per realizzare queste diverse forme di cooperazione.
- Ad esempio:
  - send e receive
  - semafori
  - monitor
  - chiamata di procedura remota
- Alcuni linguaggi offrono anche meccanismi di cooperazione (es: Java).

Thread

- Un **thread**, detto anche processo leggero, è una unità di esecuzione che consiste di un program counter, lo stack e un insieme di registi.
- Un thread condivide con altri thread la sezione codice, la sezione dati, e le risorse che servono per la loro esecuzione.
- Un insieme di thread associati prendono il nome di **task**.
- Un **processo** equivale ad un task con un unico thread.
- I thread rendono più efficiente l'esecuzione di attività che condividono lo stesso codice.

Task con thread singoli e multipli



Thread

- Il context switch tra thread è molto più veloce.
- Un sistema operativo composto da thread è più efficiente.
- **Con le CPU multi-core i programmi composti da più thread possono essere eseguiti sui diversi core in parallelo.**

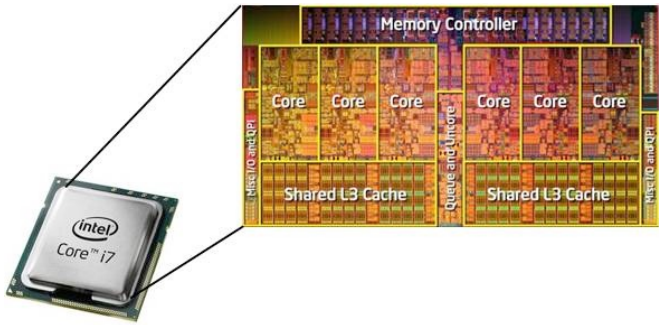
TUTTAVIA:

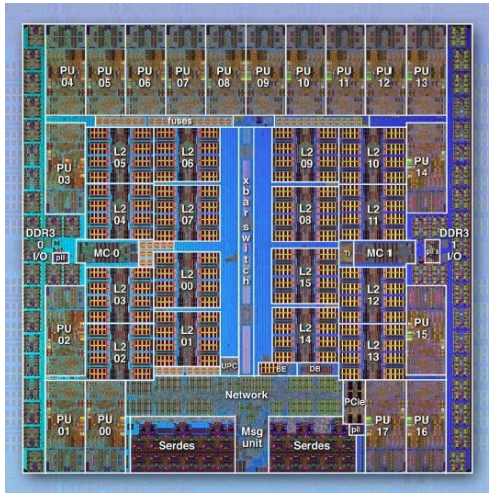
- I thread di uno stesso task non sono tra loro indipendenti perché condividono codice e dati.
- E' necessario che le operazioni non generino conflitti tra i diversi thread di un task.

Benefici

- Velocità di risposta
- Condivisione di risorse
- Economia di risorse
- Uso efficace delle architetture parallele

  (multiprocessore – es. dual/quad core)





Thread utente

- Generalmente esistono thread di utente (**user threads**) e thread di sistema (**kernel threads**)
- Nei thread di utente la gestione è fatta tramite una libreria di thread.
- I thread utente sono implementati sopra il kernel.
- Esempi
  - POSIX *Pthreads*
  - Mach *C-threads*
  - Solaris *threads*

Kernel Threads

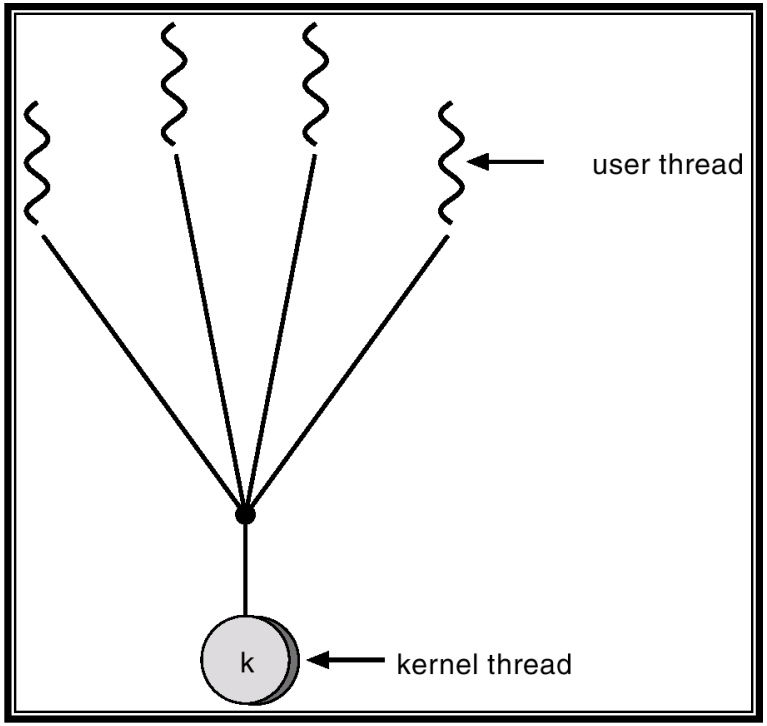
- I thread di sistema sono implementati e gestiti dal kernel.
- La gestione dei thread del kernel è più flessibile.
- Esempi
  - Windows
  - Solaris
  - Tru64 UNIX
  - Linux.

Modelli di Multithreading

- Alcuni S.O. implementano sia thread di sistema che thread di utente.
- Questo genere differenti modelli di gestione dei thread:
  - Molti-ad-Uno
  - Uno-ad-Uno
  - Molti-a-Molti.

Modello Molti-ad-Uno

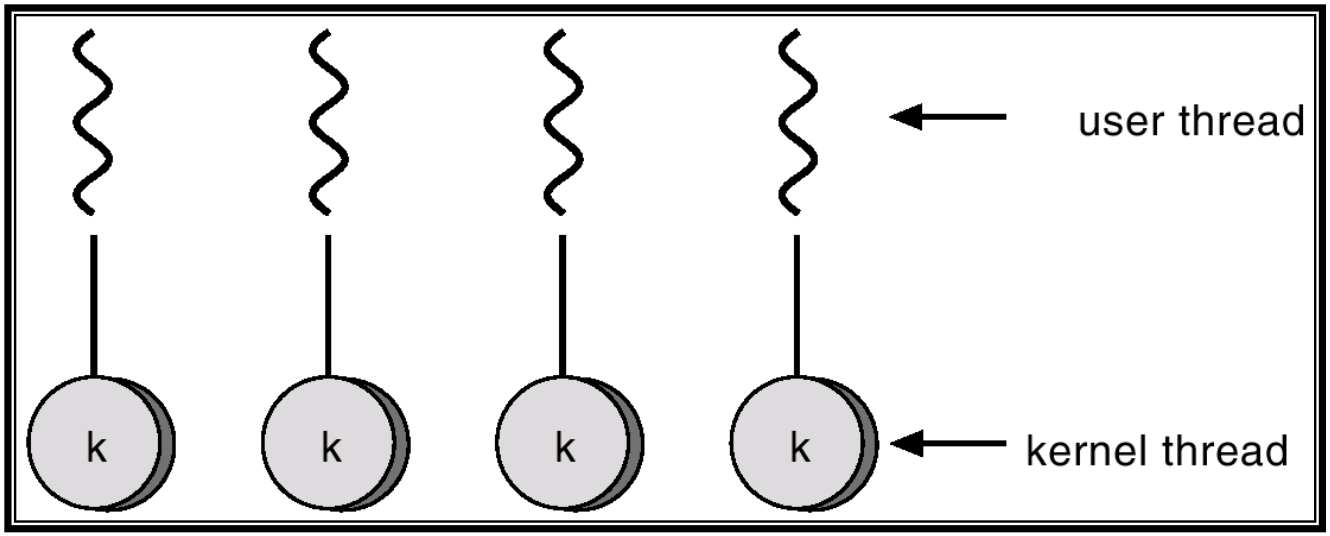
- Più user thread sono mappati su un singolo kernel thread.
- Usato nei sistemi che non supportano kernel threads.



Modello Uno-ad-Uno

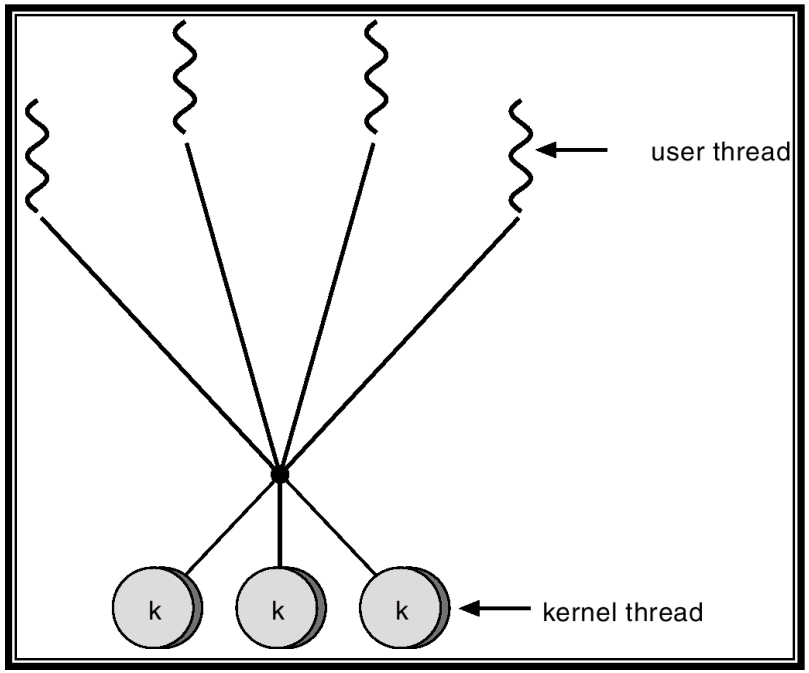
- Ogni user thread è associato ad un kernel thread.
- Esempi:

  - Windows
  - OS/2, Solaris.



Modello Molti-a-Molti

- Molti user thread possono essere associati a diversi kernel threads.
- Permette al sistema operativo di creare un numero sufficiente kernel thread.
- Esempi: Solaris (prima della versione 9) e Windows NT/2000 con i *ThreadFiber* package



Pthreads

- Pthread è un modello basato sull'API standard POSIX (IEEE 1003.1c) per la creazione e la sincronizzazione di thread.
- Le API specificano il comportamento della libreria dei thread, ma non sono una sua implementazione.
- Esempi di primitive:

  pthread_create(), pthread_join(), pthread_exit()
- Usato in diverse versioni di UNIX.

Thread di Windows

- Ogni thread utente è associato ad un thread del kernel (modello uno-ad-uno).

- Ogni thread contiene

  - un identificatore del thread

  - un insieme di registri

  - uno stack utente e uno stack kernel

  - un'area di memoria privata del thread

- Con i *ThreadFiber* Windows fornisce anche un modello molti-a-molti.

Thread di Linux

- Linux usa il termine *task* per indicare sia *processi* sia *thread*.
- Tramite la system call **fork()** si possono creare i processi.
- Per la creazione di un thread definisce la system call **clone(param)**.
- **clone(param)** permette ad un task figlio di condividere lo spazio di indirizzi del task genitore. È una variante delle system call **fork()**.
- Tramite un insieme di flag è possibile specificare il livello di condivisione tra i task padre e figlio.

Thread Java

- Java offre la possibilità di usare i thread che possono essere implementati :
  - Estendendo, tramite una sottoclasse, la classe **Thread**.

```
public class HelloThread extends Thread {
   public void run() {
      System.out.println("Hello from a thread!");
   }

   public static void main(String args[]) {
       (new HelloThread()).start();
   }
}
```

```
    }
}
```

Diagramma di stato di un Thread Java