Processi e thread

Il sistema operativo realizza un certo numero di nuove astrazioni: il PROCESSO è una di queste

Una possibile definizione:

Un processo è una *attività di elaborazione guidata da un programma*; la sua velocità di esecuzione dipende da quanti processi condividono la stessa CPU e Memoria

Una visione più astratta: un processo è una attività di elaborazione su una CPU virtuale (completa di registri) e una memoria virtuale grande abbastanza per contenere i dati e il codice del programma associato al processo

In questa interpretazione la CPU virtuale funziona in modo "intermittente", dato che è in grado di portare davvero avanti la computazione solo quando ad essa corrispondono CPU e memoria *vere*.

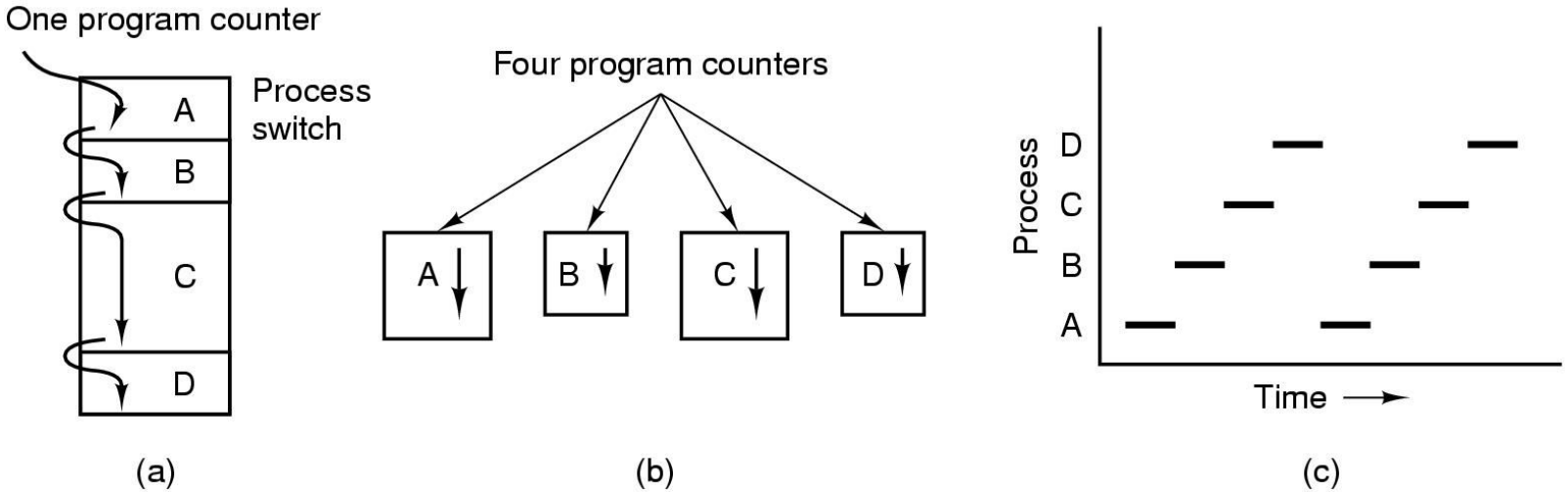
In ogni caso, un processo non coincide con un programma: in un dato momento ci possono essere 0, 1, 2,… processi che eseguono lo stesso programma.

Inoltre il programma di un processo può anche cambiare durante la vita di un processo (il processo chiede di cambiare programma; non si tratta del caso in cui il sistema operativo decide di assegnare la CPU un altro processo)

Il sistema operativo deve mantenere lo **stato** dell'elaborazione per ogni processo, *in questo caso* inteso come dati in memoria e valori dei registri. Nel caso di più processi che eseguono lo stesso programma, questo (essendo in sola lettura) può essere condiviso, i dati no

Multiprogrammazione nel modello a processi

Si usa il termine “pseudo-parallelismo” per indicare che l’evoluzione è apparentemente parallela, con una singola unità di elaborazione.



In un sistema con *n* unità di elaborazione (es. *cores*, ma le chiameremo impropriamente CPU) è effettivamente parallela fino al *grado n* (si richiede di eseguire *n* processi o threads), oltre quello, *m (>n)* processi/threads si dividono le *n* unità fornite dall’hw

a) Multiprogrammazione di 4 programmi

b) Modello concettuale di 4 processi sequenziali indipendenti

(c) Solo un processo è *running* in ogni istante (o solo *n*, con *n* CPU)

Velocità di esecuzione dei processi

In un sistema multiprogrammato (senza supporto al tempo reale), non si possono in generale dare garanzie sulla velocità dei processi basandosi su una stima del tempo di CPU necessario ad eseguire parti del codice del processo:

- né quella assoluta: non possiamo garantire che un processo arrivi ad un punto x del suo codice in un dato tempo (assoluto)

- né quella relativa: non possiamo garantire che il processo A raggiungerà un certo punto x della sua esecuzione prima che il processo B raggiunga il punto y della sua esecuzione

Questo perché:

- non si può prevedere sempre quali parti del codice verranno eseguite e quindi quanto tempo di CPU serve per arrivare a un dato punto
- mentre il processo sta eseguendo (anche con CPU multiple), può arrivare una interruzione e la CPU può essere assegnata ad un altro processo

Velocità di esecuzione dei processi

Garanzie sulla velocità assoluta possono essere fornite attraverso appositi meccanismi solo dai sistemi operativi real-time

- i vincoli possono essere di *hard* real-time, es. nel caso di processi per il controllo di impianti, veicoli, etc: non soddisfarli ha un costo non accettabile (l'impianto può esplodere, il veicolo non reagisce ai comandi umani o non viene comandato in tempo dal sistema di guida autonoma)
- o di *soft* real-time: è accettabile non soddisfarli ogni tanto (es. si perde qualche fotogramma in un video), il costo di non soddisfarli (es. fastidio per l'utente) cresce, anche non linearmente, con il numero di volte in cui non li si soddisfa

In altri sistemi, la velocità effettiva dei processi dipende dal criterio di assegnazione della/e CPU ai processi; possiamo solo aspettarci che su scala «lunga» i processi vadano avanti con velocità simile, ma senza garanzie

Garanzia sulla velocità relativa possono essere ottenute con meccanismi espliciti di sincronizzazione fra processi, che verranno introdotti più avanti

Creazione dei processi

Esempio di Unix: al momento del *bootstrap* viene creato un primo processo "init" di inizializzazione, il quale provvede a creare un processo di "login" ed un certo numero di processi che eseguono in "background" (es. uno per gestire le richieste di stampa), che vengono chiamati "daemon" e che intervengono quando si verificano determinati eventi.

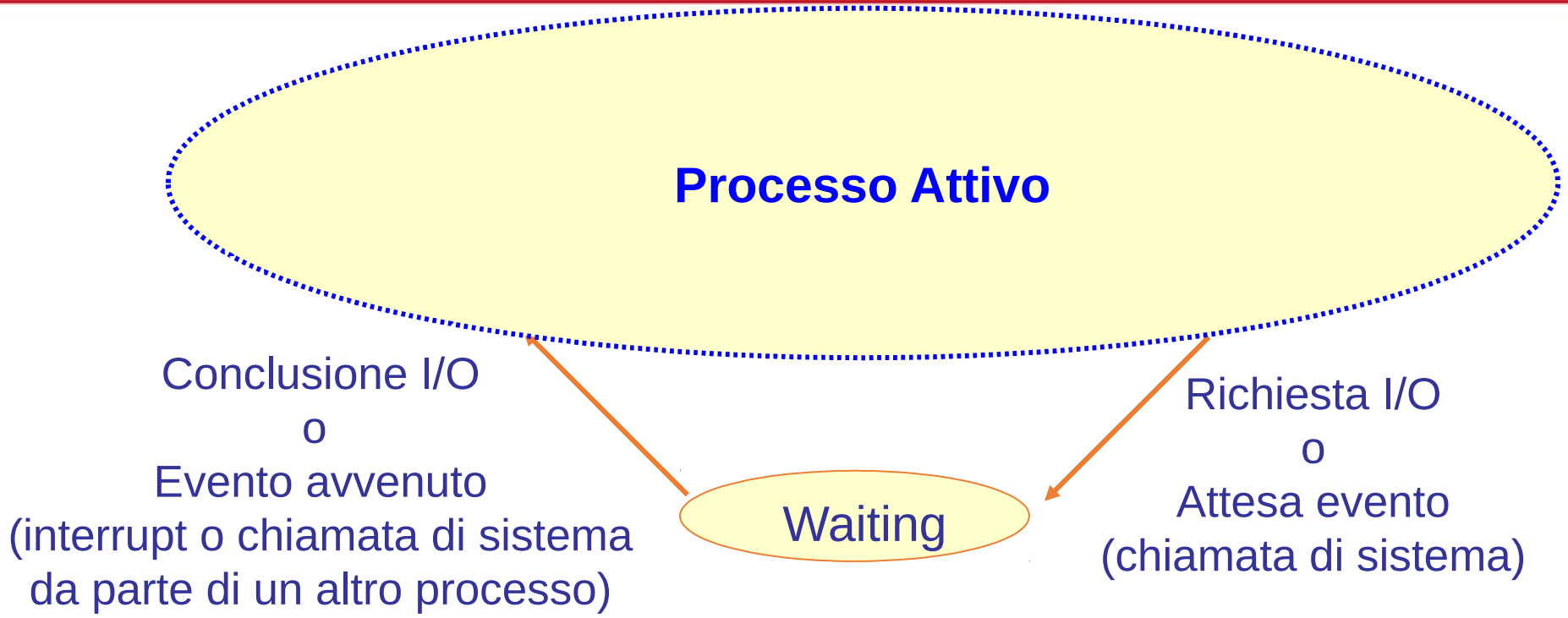
Quando un utente apre una finestra "terminale", o si collega in remoto, viene creato un nuovo processo, un interprete di comandi.
Se all'interprete si chiede di eseguire un programma (per esempio il gcc per compilare un programma in C), l'interprete crea un nuovo processo che eseguirà il programma in questione. Intanto l'interprete di comandi si metterà in attesa della *terminazione* di tale processo: quando questa si verifica, il processo "padre" riprende la sua esecuzione, chiedendo un nuovo comando all'utente
Analogamente, dal file manager di una interfaccia grafica, cliccando sull'icona di un file, tipicamente viene fatto partire un processo che esegue un programma associato al tipo di file

Terminazione dei processi

La terminazione di un processo può essere causata da diverse circostanze:

- Il programma da eseguire ha concluso correttamente la sua esecuzione
- Si è verificata una condizione di errore rilevata dal programma, e il programma stesso decide di terminare (per esempio un file da cui doveva prendere input non esiste)
- Si è verificata una condizione di errore imprevista, che ha causato una trap, gestita dal S.O. terminando il processo
- Il processo è stato “terminato” da un altro processo (attraverso una opportuna system call)

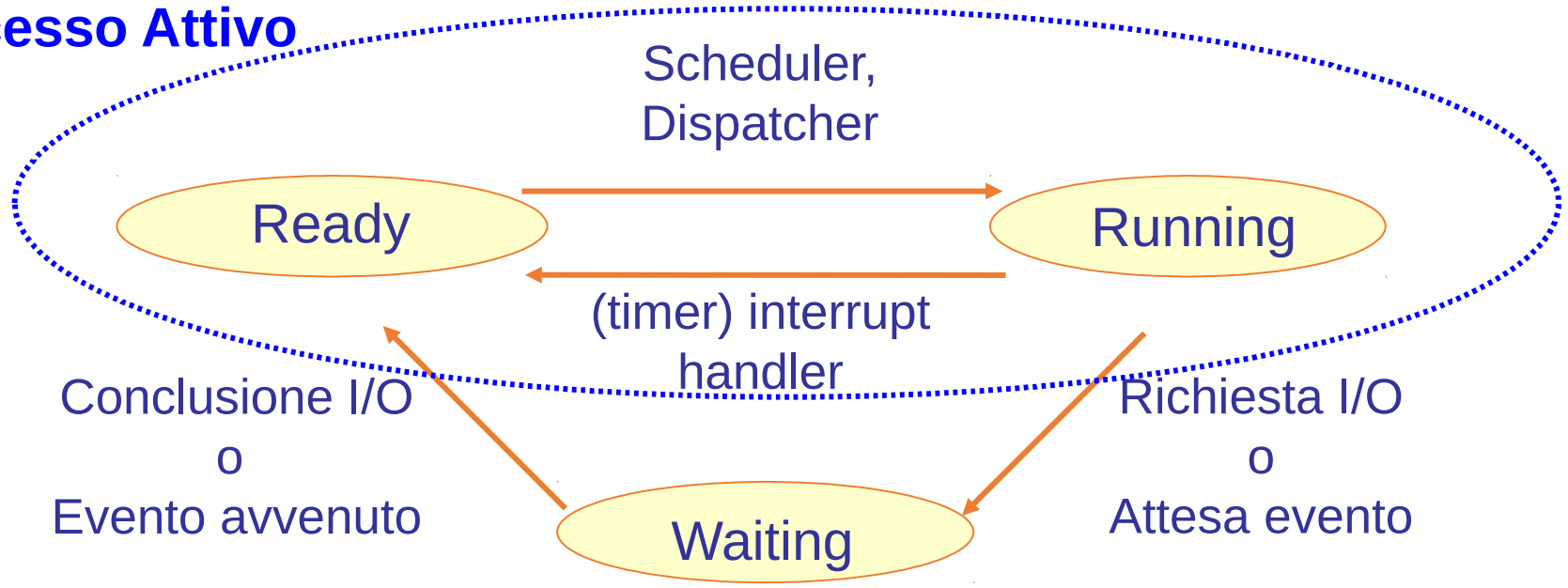
Stati di un processo



Attivo: la sua esecuzione può proseguire (è in un punto del suo programma in cui non sta attendendo eventi)

Waiting (o Bloccato): per esempio è in attesa di ricevere input da un dispositivo (*read* in Unix), in attesa della terminazione di un processo figlio (*wait* o *waitpid*), o in attesa che vengano immessi dati in una *pipe* (ancora *read*)

Processo attivo



Attivo: la sua esecuzione può proseguire

- **Running**: se ha anche la/una CPU, quindi è effettivamente in esecuzione
- **Ready**: pronto , ma la/le CPU è usata / sono usate da qualche altro processo

Scheduler e Dispatcher

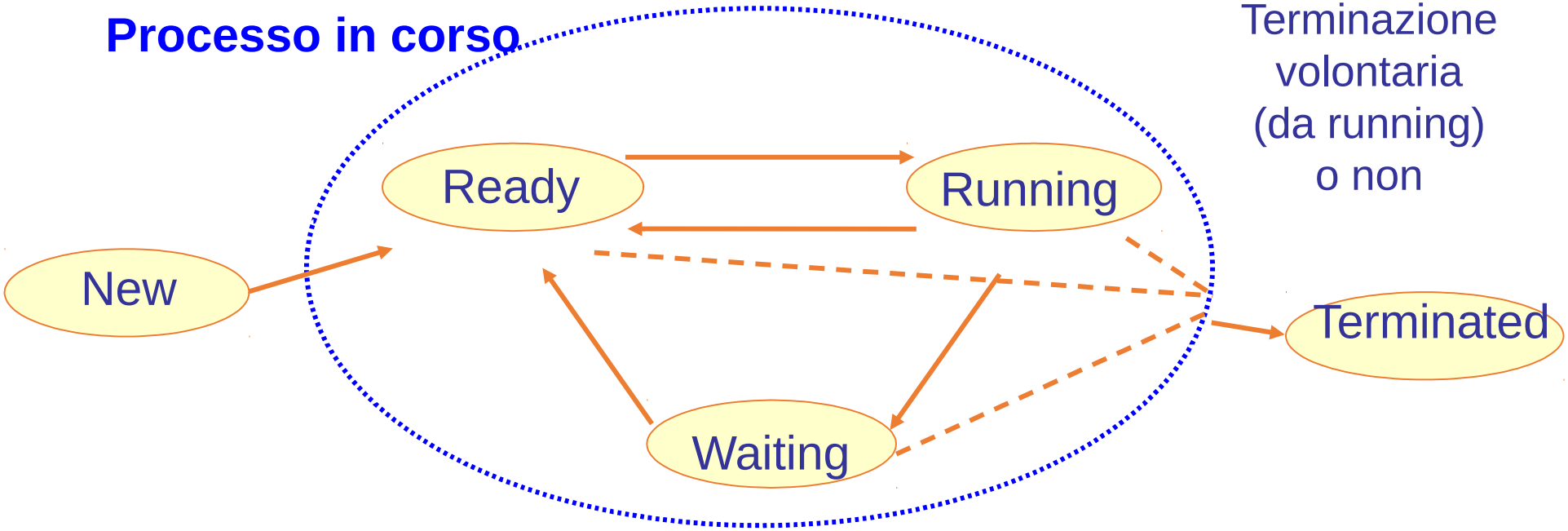
Introduciamo i seguenti termini:

Scheduler: il componente del sistema operativo che si occupa di scegliere quale processo deve diventare running tra tutti quelli attivi (la scelta viene attuata in base ad una *politica di scheduling*)

Dispatcher: il componente del sistema operativo che attua il *context switch* (caricando nei registri lo "stato" del processo scelto dallo scheduler per diventare running, e salvando lo stato del processo che smette di esere running; "stato" inteso come valori dei registri)

Questo è un esempio di separazione fra ***politiche*** e ***meccanismi*** , una buona caratteristica strutturale dei sistemi operativi che ne facilita la portabilità e la flessibilità. Qui il dispatcher realizza un meccanismo; lo scheduler una politica che utilizza il meccanismo.
Con questa separazione posso più facilmente:

- riusare un meccanismo (che in questo caso nasconde i dettagli dell'architettura) per politiche diverse,
- riusare una stessa politica su meccanismi diversi



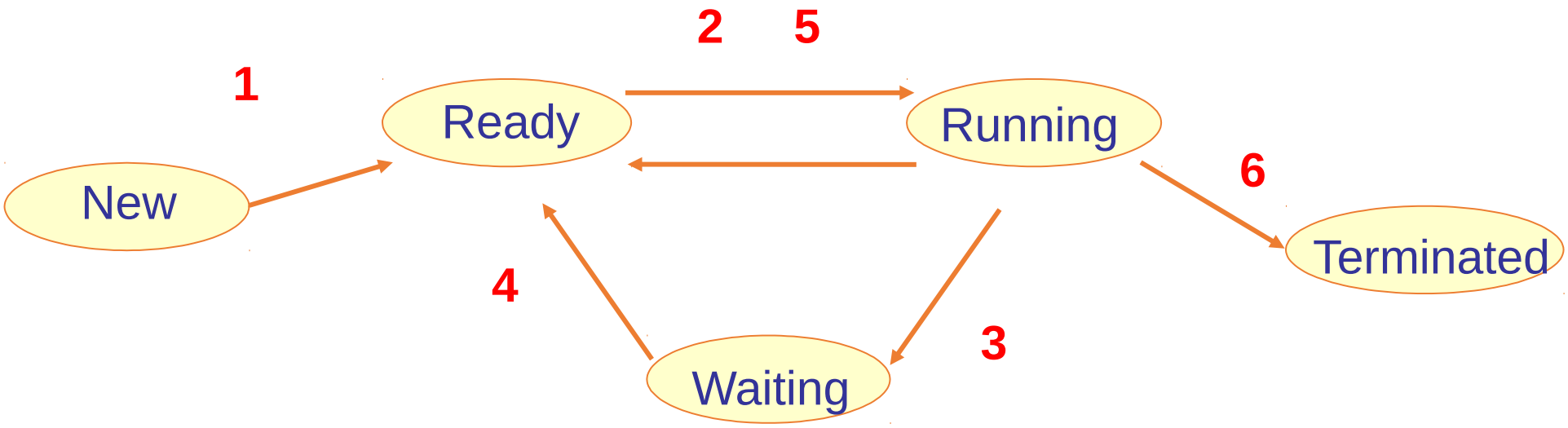
Nuovo: durante la creazione (inoltre potrebbe essere tenuto in "anticamera" prima di allocargli risorse, per limitare il livello di muiltiprogrammazione)

Terminato: il processo è terminato ma il s.o. ne tiene ancora traccia (può servire per informazioni sulla terminazione)

In corso: il processo è pronto per essere eseguito, in esecuzione o bloccato in attesa di un evento

Esempio: passaggi di stato di un processo P

Eventi	Stati del processo P dopo l'evento
1. creazione	new; ready
2. scelto dallo scheduler	running
3. P chiama read()	waiting
4. interrupt (concluso I/O)	ready
5. P scelto dallo scheduler	running
6. P esegue exit()	terminated



Strutture dati nel sistema operativo per la gestione dei processi

Il sistema operativo mantiene una struttura, chiamata Process Control Block (PCB) per ogni processo nel sistema. La tabella dei processi contiene l'insieme dei PCB di tutti i processi presenti nel sistema.

Contenuti tipici del PCB:

Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state: **Ready, Running, Waiting**		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Registri CPU

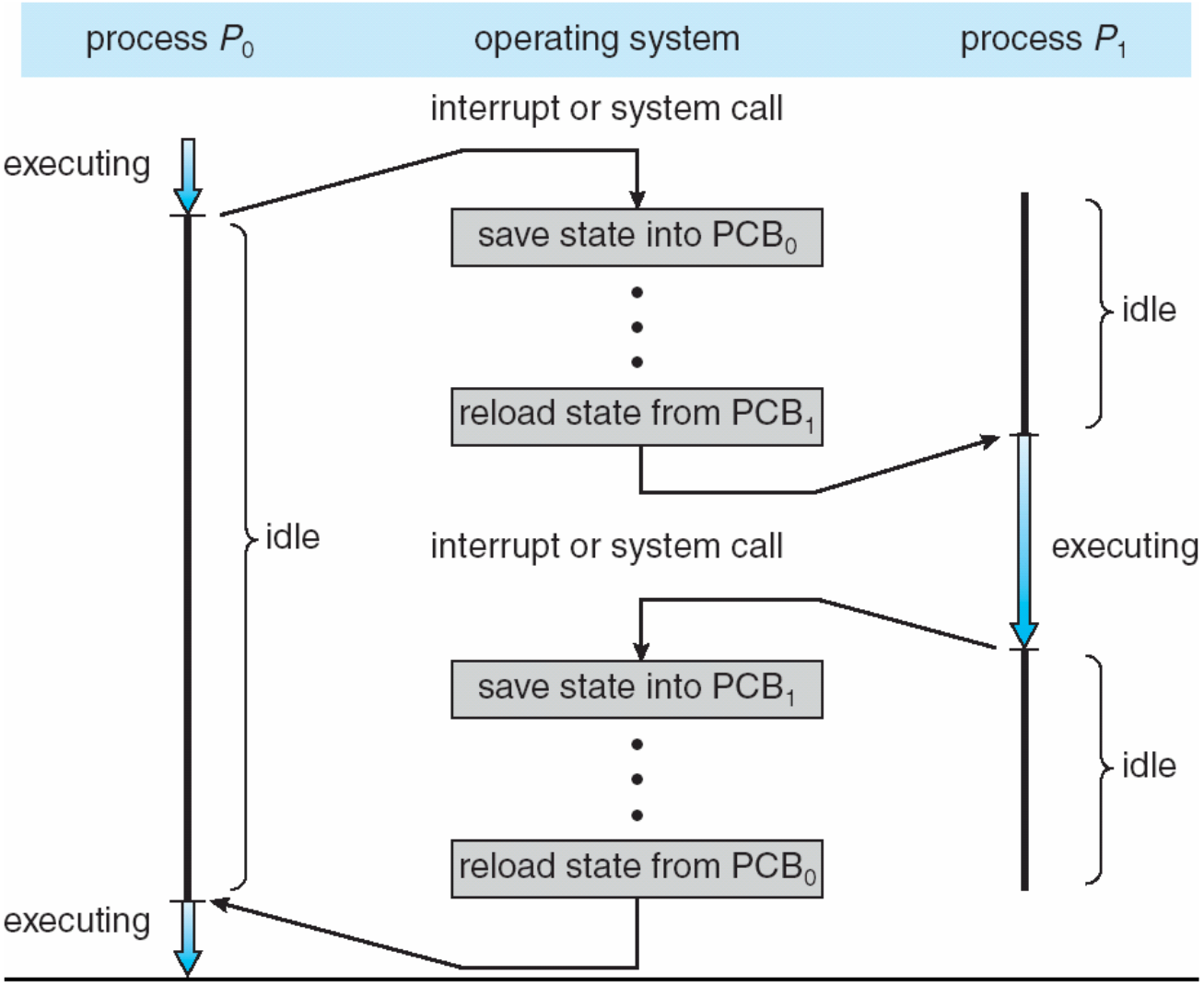
Dati per lo scheduler

Dati per accounting

Informazioni utili per protezione (controllo accesso ai file)

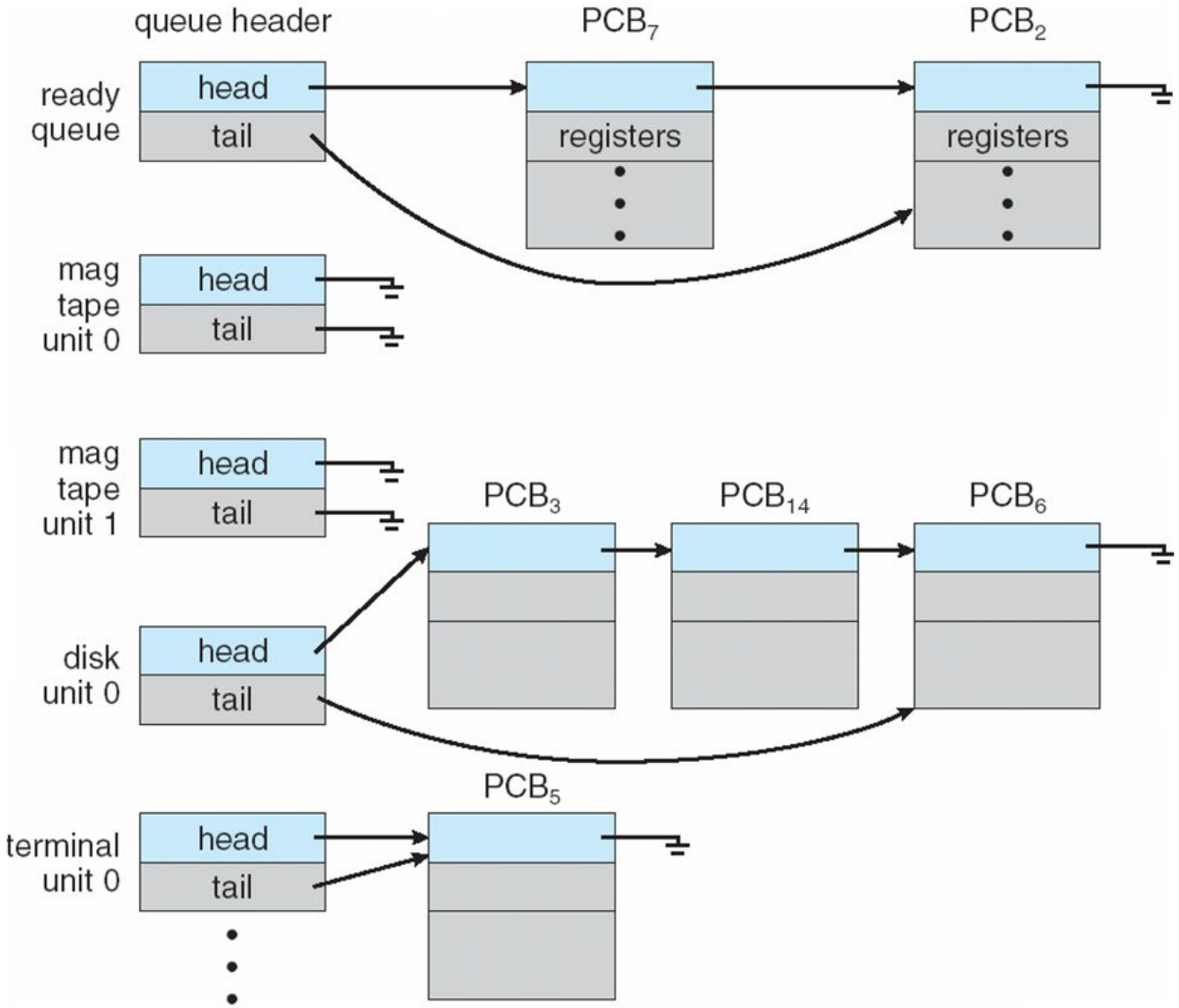
PCB e context switch

Lo stato dei registri viene salvato nel PCB e ripristinato in occasione del *context switch* (commutazione di contesto, la CPU passa da un processo a un altro):



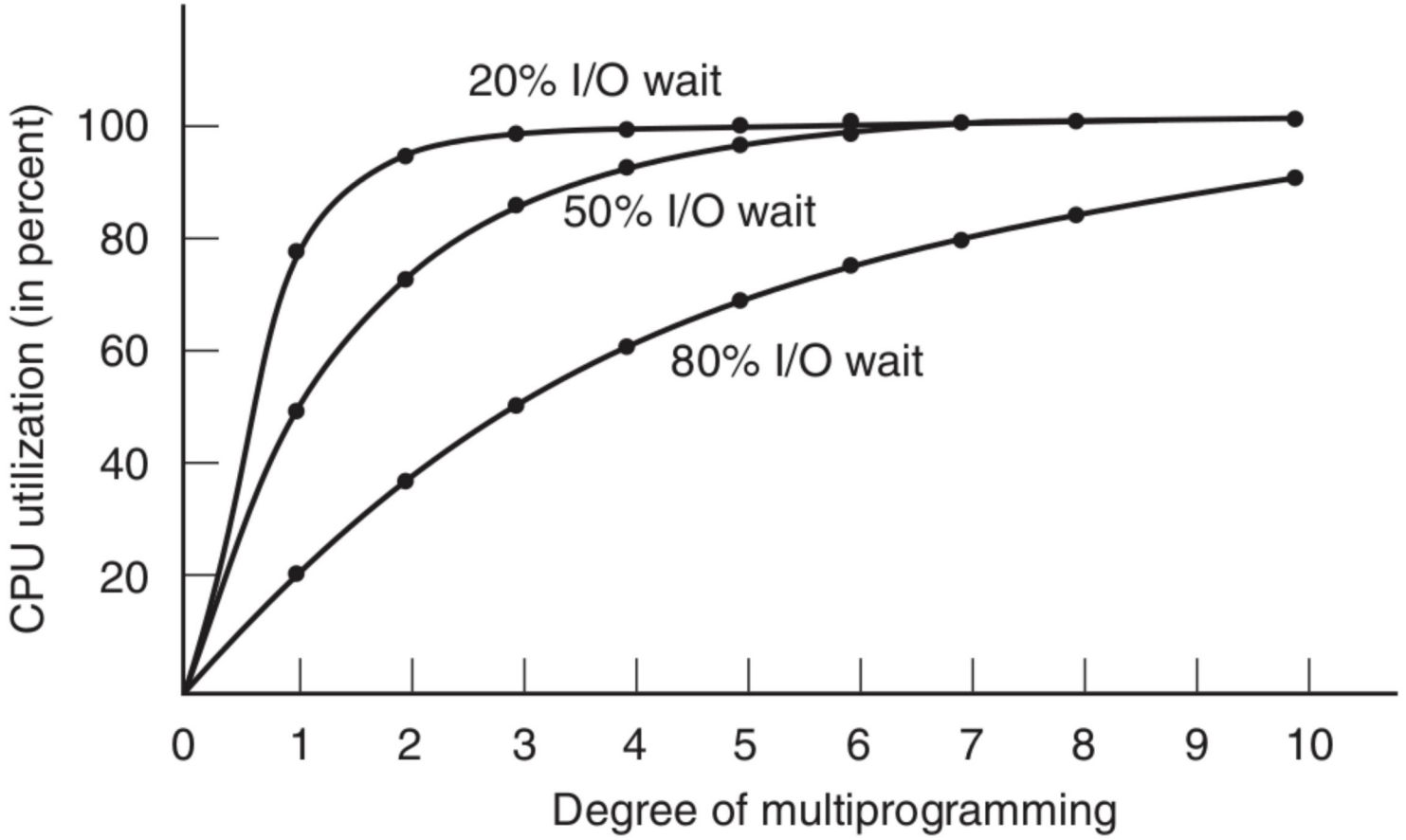
Code di processi

I vari PCB possono essere nodi di liste che costituiscono la coda dei processi pronti (*ready queue*) e le code di attesa di altri eventi

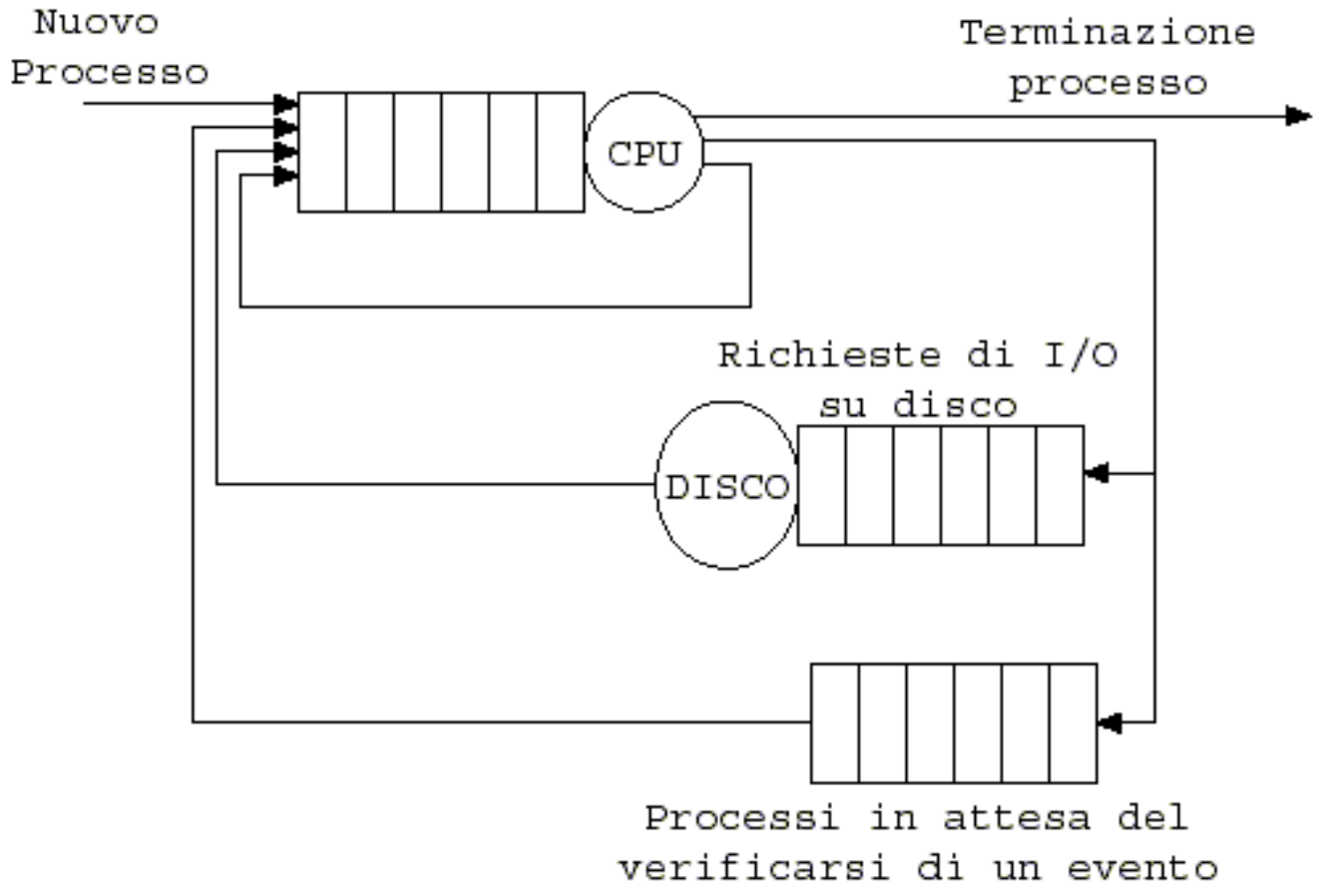


Modello utilizzazione CPU

$$\text{CPU utilization} = 1 - p^n$$



Il sistema di calcolo come sistema a code



Processi in competizione o cooperanti

L'introduzione dell'astrazione dei "***processi***" è stata inizialmente motivata dalla necessità di utilizzare al meglio le risorse disponibili, in particolare la CPU

L'utente o gli utenti fanno partire: un elaboratore di testi, un browser, un lettore multimediale, etc:

- applicazioni progettate indipendentemente (nessuno si è messo a progettarne una apposita per il caso in cui l'utente vuole scrivere consultando il Web e/o sentendo musica)
- lanciate indipendentemente
- non devono condividere codice né dati
- vanno mandate avanti in (pseudo)parallelo

In questa luce, ***i processi sono tra loro in competizione*** per l'accesso a un insieme di risorse condivise

E il compito del Sistema Operativo è assicurare la ***non interferenza*** tra processi che si trovano *per caso* ad utilizzare le stesse risorse (hardware: CPU, memoria, dispositivi) del sistema di elaborazione

Processi in competizione o cooperanti

Un insieme di processi può anche ***cooperare*** per raggiungere un obiettivo: si possono progettare applicazioni come un insieme di processi cooperanti; in questo caso tipicamente condivideranno (per scelta) risorse software: programma, dati, file aperti --- oltre a quelle hardware

Il sistema operativo dovrà fornire dei meccanismi per permettere ai processi cooperanti di ***comunicare*** e ***sincronizzarsi.***

Vantaggio: permettere uno stile di programmazione modulare di attività in parte indipendenti; quando per portarne avanti una è necessario attendere un evento (es. I/O, o un risultato di una delle altre attività) si chiama una primitiva bloccante che sospende, se necessario, l'esecuzione fino al verificarsi di un evento:

⇒ *quando si arriva all'istruzione successiva, l'evento si è verificato*
(basta scrivere ***read(..) wait(..)*** *o simili; non: «****while*** *l'altra attività non ha finito...» il che oltretutto sprecherebbe CPU)*

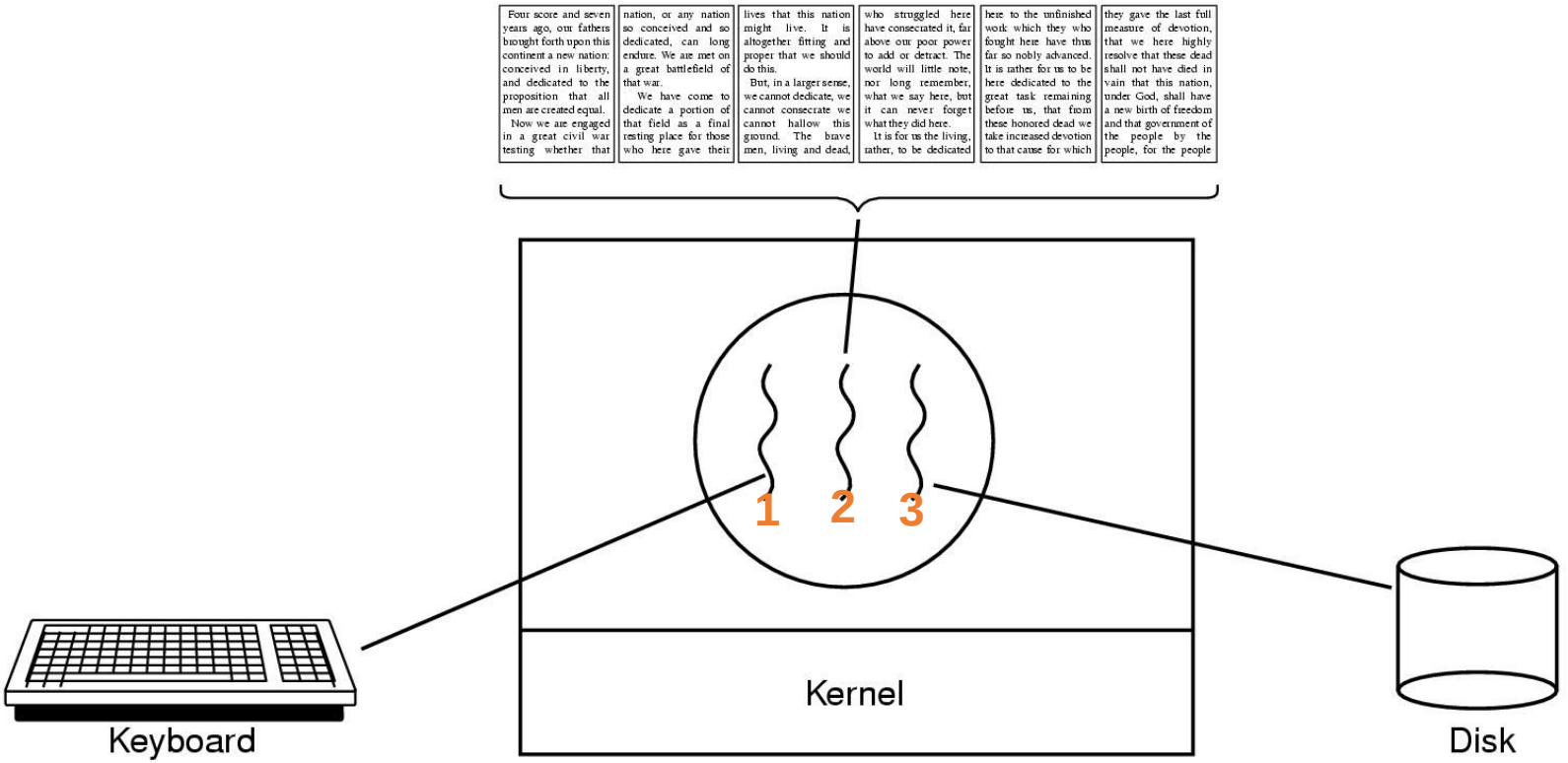
nel frattempo il S.O. potrà mandare avanti gli altri processi, preoccupandosi di riportare *pronti* quelli sospesi, quando l'evento occorre

Nel caso di attività cooperanti questo però è più **costoso** del necessario in termini di *context switch*: dato che alcune risorse (tipicamente di memoria) sono condivise, il contesto potrebbe non dover cambiare molto.

QUESTO HA PORTATO ALL'INTRODUZIONE DEI THREADS

(a volte anche detti ***processi leggeri***)

Esempio



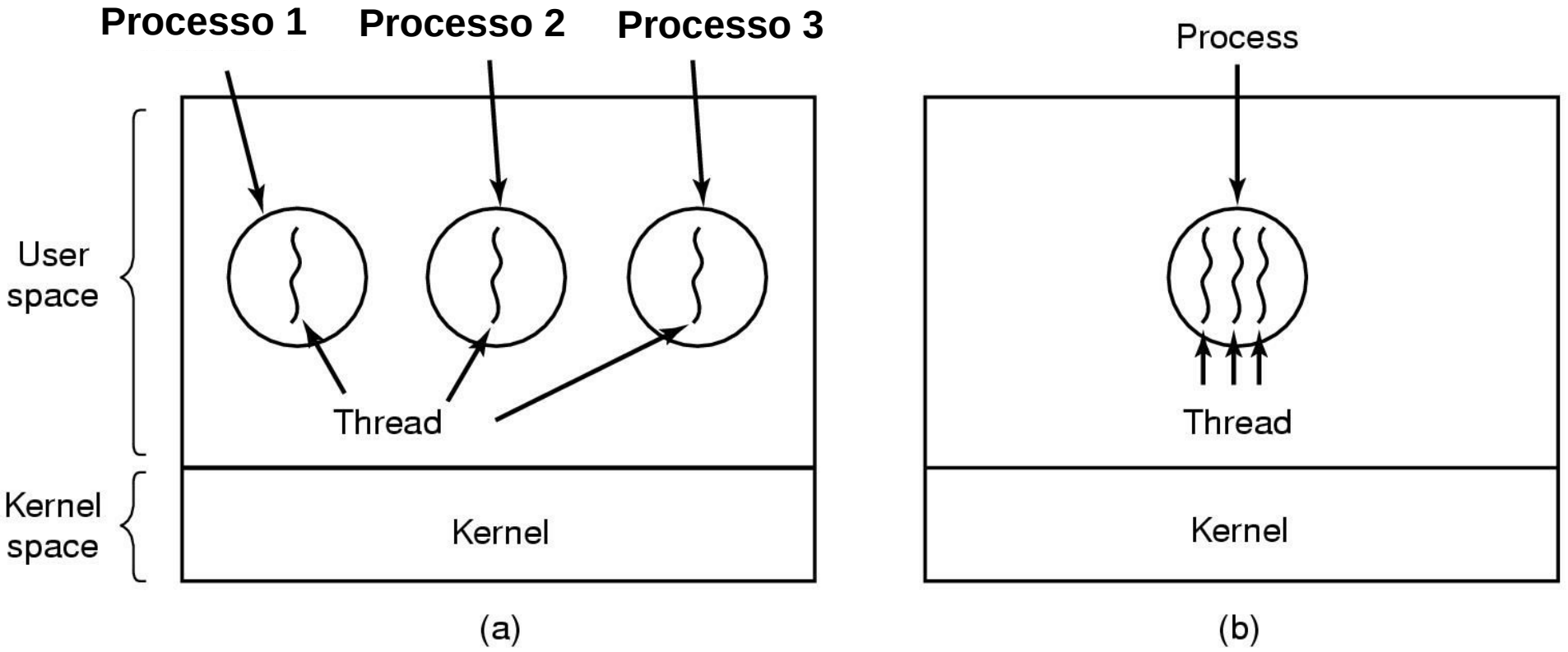
Un word processor di tipo WYSIWYG* realizzato mediante tre attività di elaborazione parzialmente indipendenti:

1. **accetta il testo inserito da tastiera e modifica la riga corrente**
2. **riformatta l'intero documento (es. spostando tutti i limiti di pagina) quando la modifica della riga corrente ha effetto sul resto del documento**
3. **salva periodicamente il documento su disco**

NB: 2 e 3 possono girare mentre 1 attende input da tastiera

* What You See Is What You Get ("quello che vedi è quello che è" o "ottieni quanto vedi")

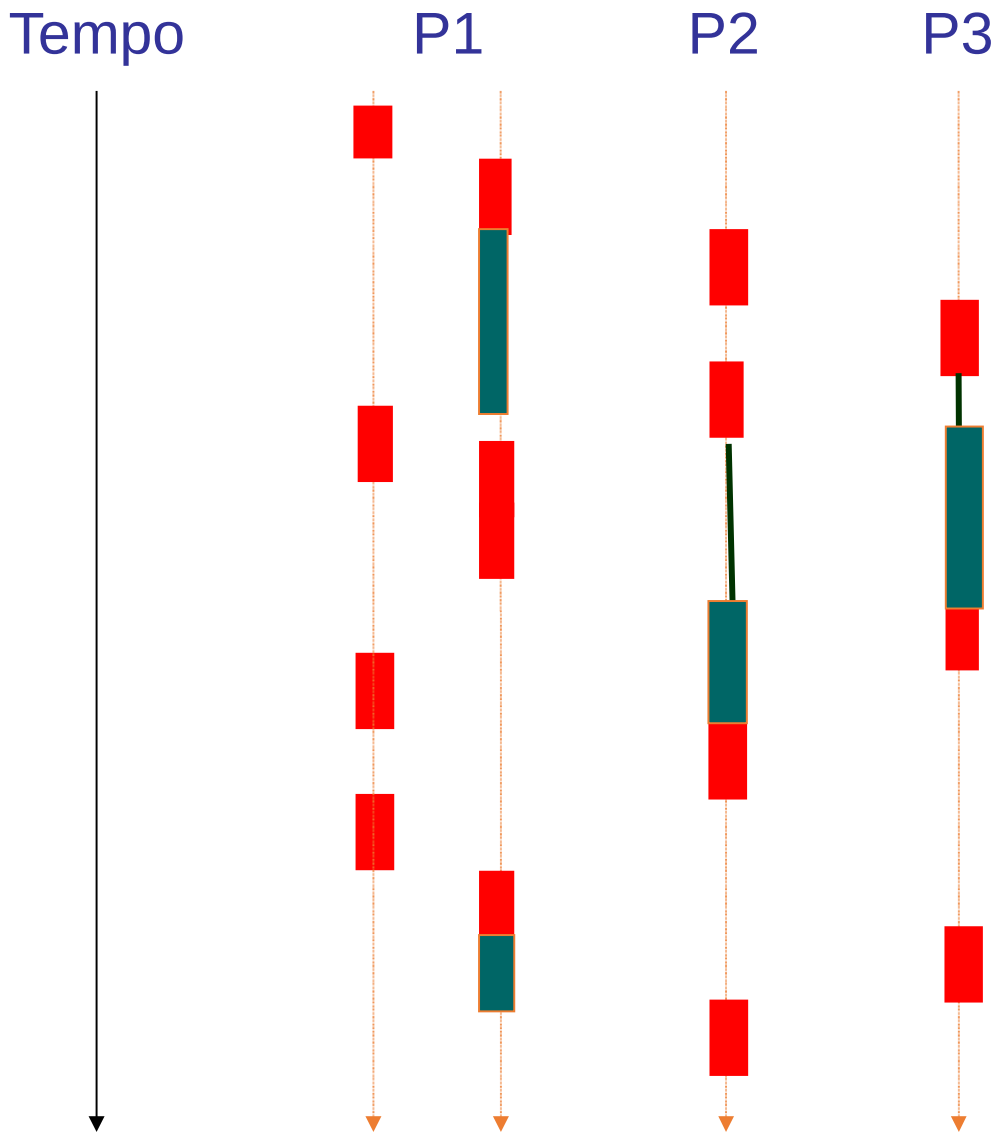
In un processo, più flussi di controllo (*threads*) che condividono la stessa memoria ed eseguono parti diverse del codice



(a) Tre processi, ciascuno con un thread (ognuno ha uno spazio di memoria separato)

(b) Un processo con tre threads (i tre threads condividono il codice e tutti i dati)

Processi multithreaded



La CPU viene assegnata dallo scheduling ai vari *thread*

I *thread* di un dato processo accedono alla *medesima area di memoria*

Indica chi è *running* in ogni istante

Indica per chi lavora il disco in ogni istante

Indica un processo in attesa di iniziare I/O

P1 contiene 2 thread, mentre P2 e P3 ne contengono 1 solo

programmazione con **primitive bloccanti** (come con processi multipli)**:**

es. server Web che riceve richieste, gestire una richiesta può richiedere lettura di informazioni dal disco (operazione relativamente lenta), non sempre se le pagine più richieste le teniamo in una *cache* in memoria

Soluzione più banale:

```
1.  while(TRUE)
        {
            leggi richiesta;
            esegui richiesta (con lettura bloccante, se da disco);
        }
```

Ma è inefficiente!

mentre avviene la lettura, vorremmo poter esaminare altre richieste

Non è il solito discorso di non lasciare la CPU inutilizzata?

Si e no: qui l'esigenza è all'interno di una singola applicazione; il S.O. farebbe girare il processo di un'*altra* applicazione, se ce n'è uno pronto

Perché sono utili i threads

altre 2 soluzioni:

2. con richiesta di lettura non bloccante e **thread singolo**:
   a) attendi o rileva il prossimo evento (arrivo di una nuova richiesta, o terminazione di una lettura da disco);
   b) se è la terminazione di una lettura, rispondi alla richiesta corrispondente
   c) se è una nuova richiesta, soddisfala se la pagina è in memoria, altrimenti richiedi la lettura con una operazione non bloccante;

   ma bisogna tener traccia delle richieste in corso di trattamento!

3. con **primitive bloccanti e thread multipli**:
   a) leggi richiesta;
   b) crea o risveglia un thread per gestirla;

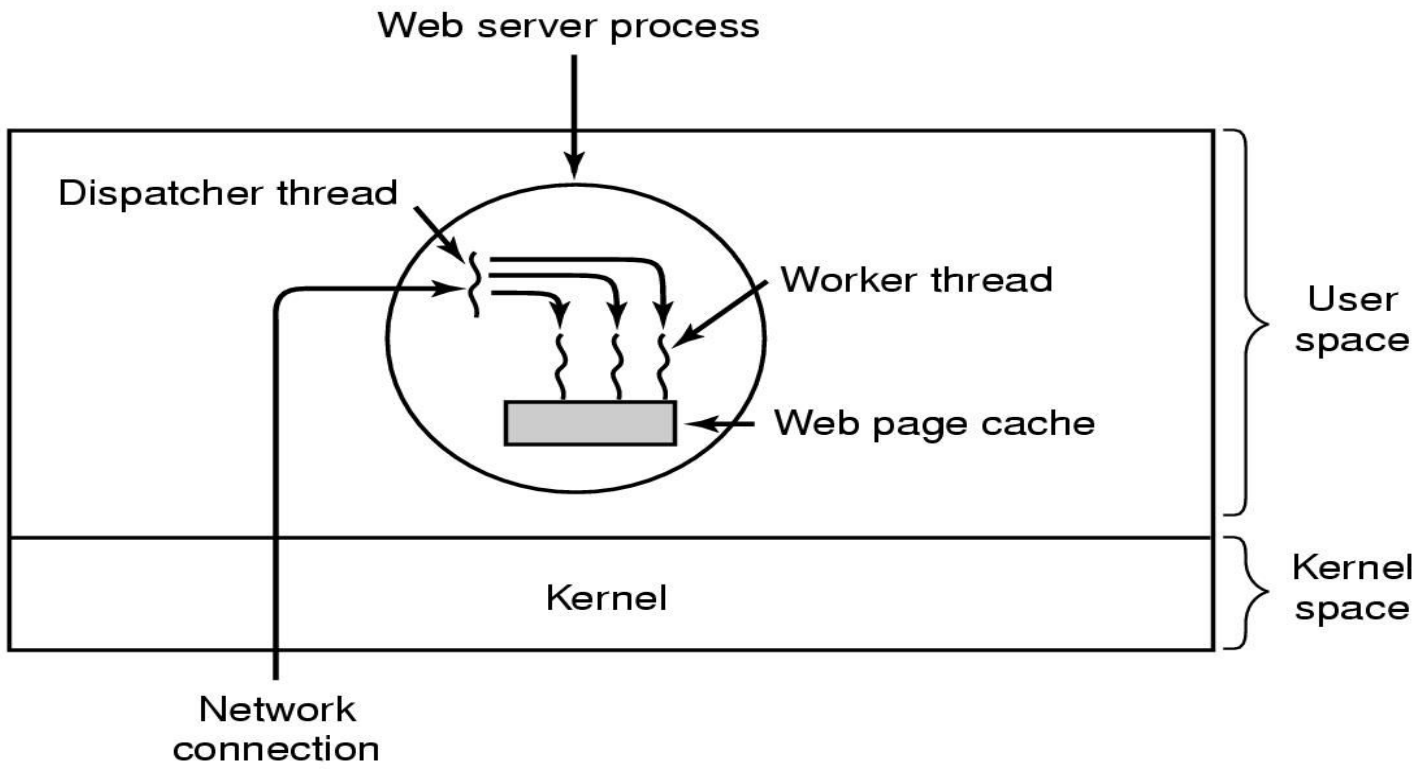
   E il codice del thread:

   c) effettua la lettura (bloccante se da disco)
   d) risponde alla richiesta

   per ogni richiesta in corso di trattamento, c'è un thread:

   è la gestione dei thread a tenere traccia per noi di queste richieste

Perché sono utili i threads




```
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
```

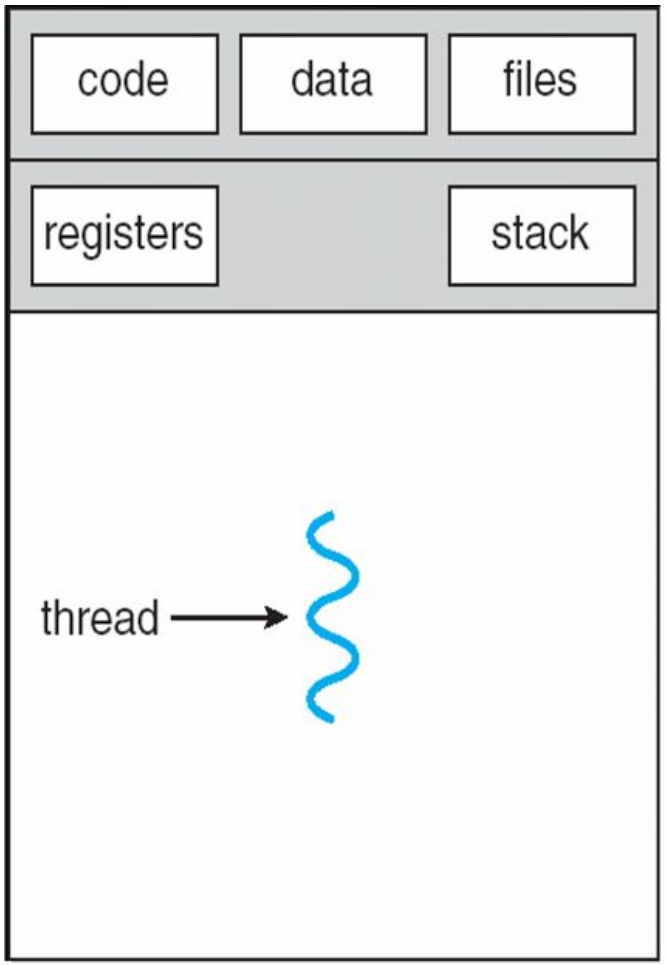
(a)

Dispatcher thread

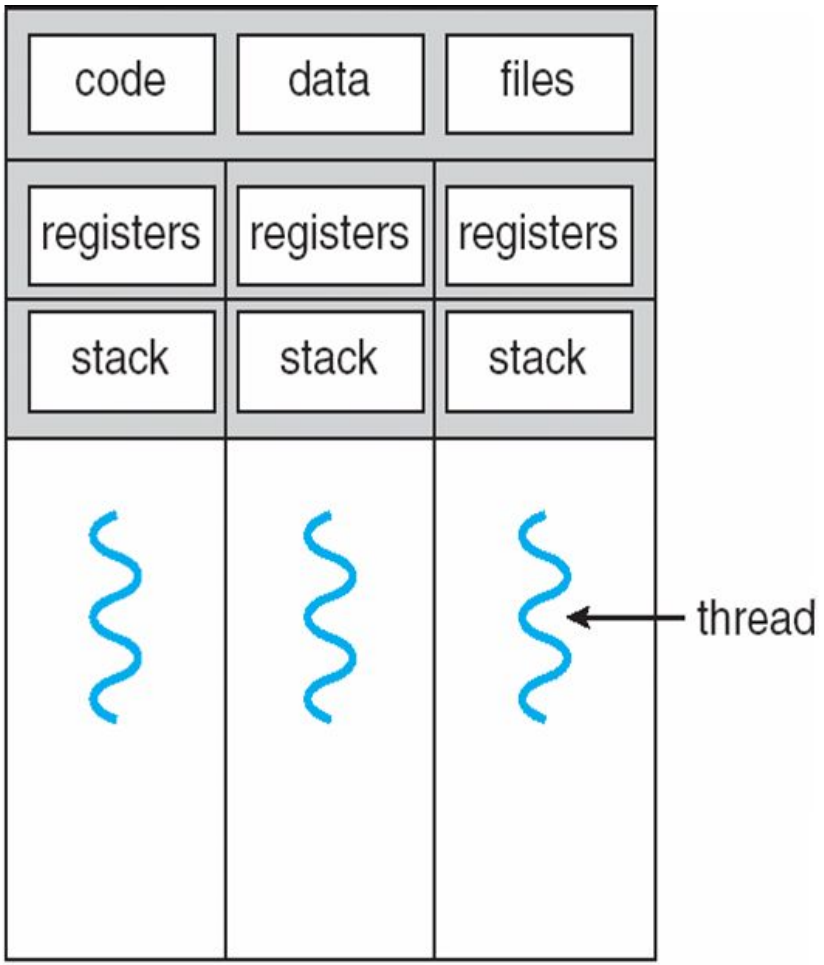
```
while (TRUE) {
  wait_for_work(&buf)
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page)
      read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

(b)

Worker thread



single-threaded process



multithreaded process

Informazioni mantenute per gestire processi multithreaded

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Ogni thread ha un suo stato (ready, running, waiting), i suoi registri di CPU incluso il program counter, stack pointer, ecc., *il suo stack*

Lo spazio di indirizzamento è condiviso, NON VI È PROTEZIONE FRA THREAD, perché sono progettati per cooperare e condividere risorse – ma errori di programmazione possono avere effetti più gravi ed essere più difficili da scoprire rispetto al caso di thread singolo, - i thread si potrebbero alterare i dati (per esempio quelli sullo stack) a vicenda...

Perché sono utili i threads

- parallelismo (su architetture multiprocessore/multicore/...) all'interno di una singola applicazione

  come per applicazione con processi multipli

- sovrapposizione (anche su processore singolo) di I/O e computazione per i thread di una singola applicazione

  come per applicazione con processi multipli

- condivisione di risorse fra attività cooperanti in una applicazione

  più semplice rispetto ad applicazione con processi multipli

- creazione di un nuovo thread, switch fra thread

  meno costoso rispetto a processi

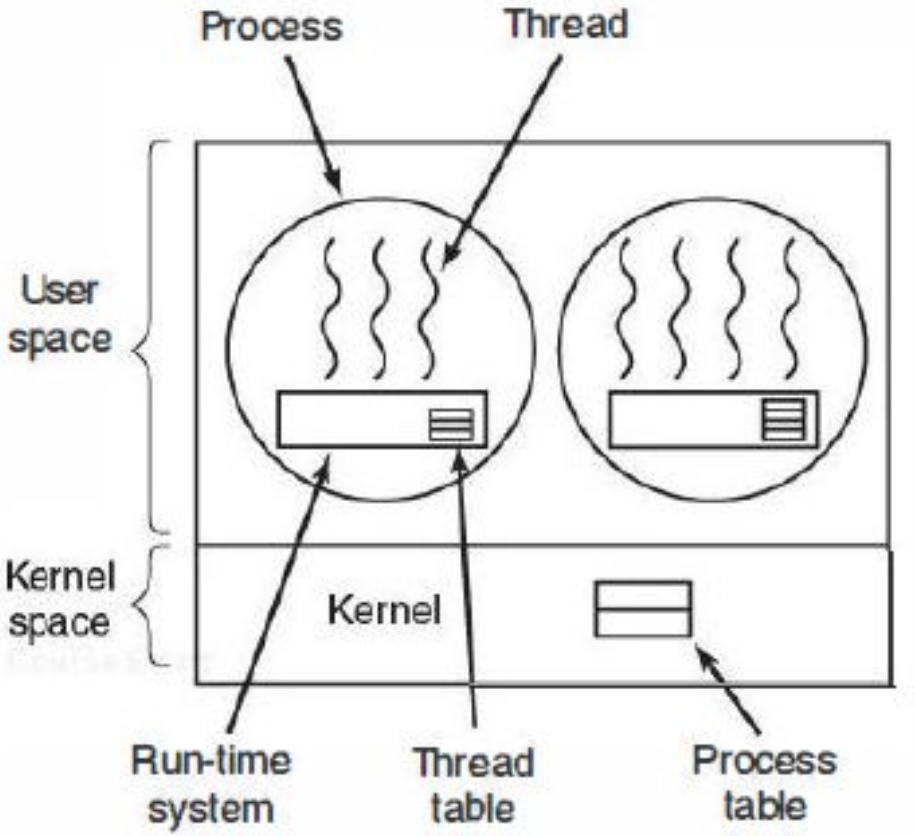
Thread a livello utente e thread a livello kernel

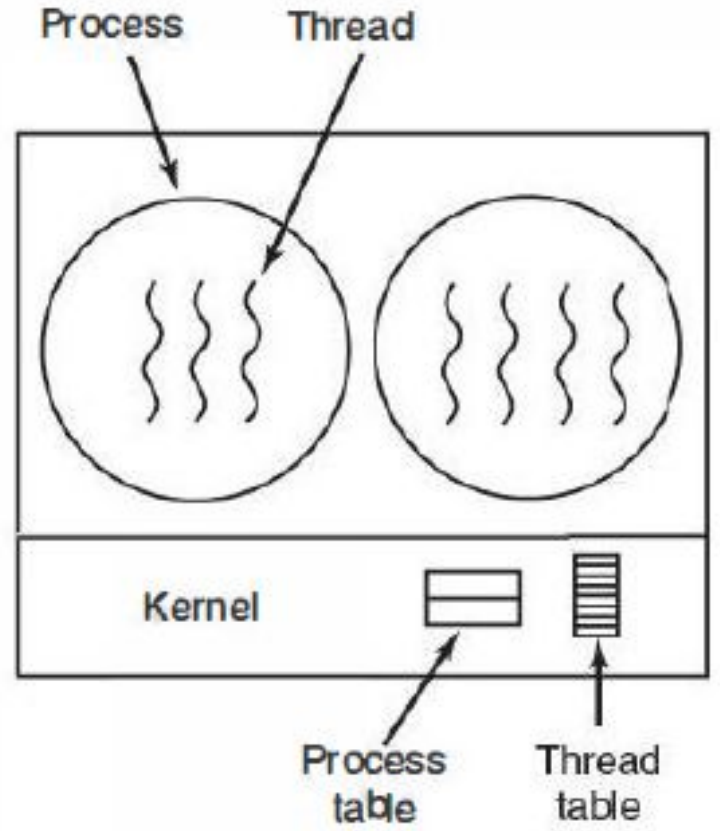
I thread possono essere implementati:

1) a livello utente
2) a livello kernel

Nel caso 1 il S.O. non è a conoscenza dell'esistenza dei thread, che vengono gestiti completamente in modalità utente – cioè non si richiede l'intervento del sistema operativo per creare, terminare, schedulare per l'esecuzione, sospendere o risvegliare threads
Il sistema di gestione dei thread è costituito da una *libreria di funzioni*
Lo scheduler della CPU la assegna ai processi, il sistema di gestione, per ogni processo P che usa thread multipli, all'interno dello spazio di indirizzi di P mantiene una *tabella dei thread* con il loro stato, ed assegna il tempo di CPU concesso a P ai suoi thread pronti.

Thread a livello utente e thread a livello kernel

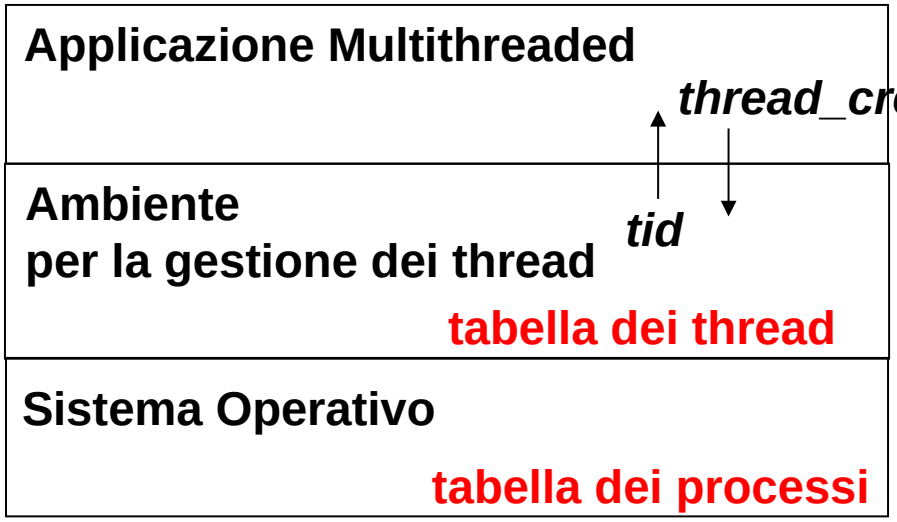




Thread a livello utente: vantaggi e svantaggi

Vantaggi

- Tempo di *context switch* (fra thread di uno stesso processo) molto basso (non c'è il passaggio a kernel mode)
- Può essere implementato al di sopra di qualsiasi sistema operativo
- Si possono attuare politiche di scheduling ad-hoc per una applicazione



Svantaggi

- È complesso fare in modo che quando un thread ha bisogno di eseguire una system call bloccante non vengano bloccati anche gli altri thread nello stesso processo (perché il S.O. memorizzerebbe nel PCB del processo "stato = bloccato" senza assegnargli tempo di CPU finché la system call bloccante non viene servita).

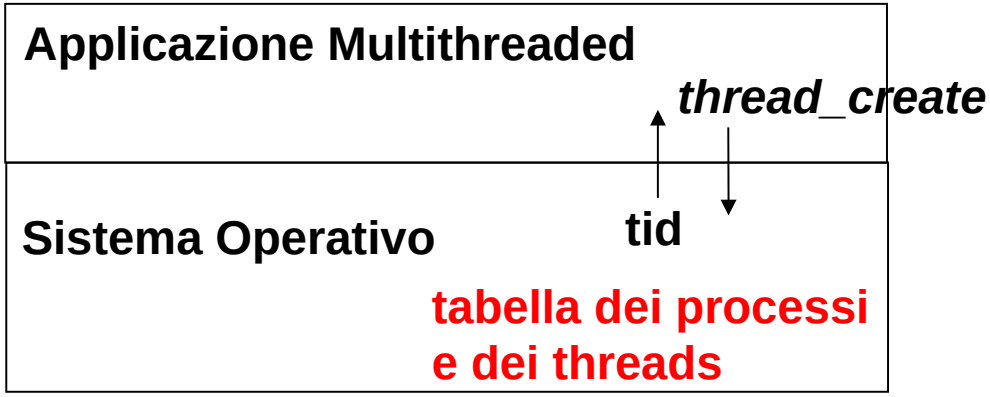
Se ne deve occupare la libreria delle chiamate di sistema (non si vogliono cambiare le chiamate di sistema stesse)

- Non potrà mai esserci più di un thread running per processo anche se il sistema possiede molte CPU
- I thread non sono automaticamente in time sharing: se serve qualcosa del genere, vanno inserite nei progammi esplicite chiamate di *thread_yield* con cui un thread permette al sistema di gestione di assegnare la CPU ad un altro

Thread a livello kernel: vantaggi e svantaggi

Vantaggi

- In un sistema con molte CPU è possibile avere più di un thread running per processo
- La gestione di system call bloccanti non pone alcun problema: il sistema è sempre in grado di schedulare i thread pronti ad eseguire anche se uno o più altri thread nello stesso processo sono bloccati in attesa di un evento.



Svantaggi

- Il tempo di context switch fra thread di uno stesso processo, anche se meno costoso di un context switch fra processi diversi, è comunque un po' costoso perché richiede il passaggio a modalità kernel
- Il sistema operativo deve predisporre strutture dati per gestire tutti i thread: questo pone limitazioni sul numero massimo di thread che possono essere attivati all'interno di ciascun processo e complessivamente nel sistema