

# Processi e Thread

- Concetto di processo
- Operazioni sui processi
- Stati dei processi
- Threads
- Schedulazione dei processi

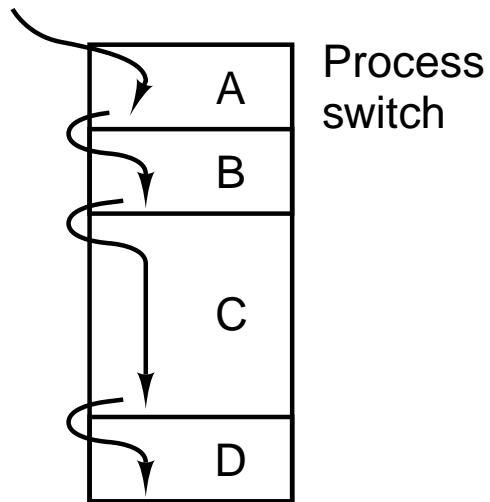
## Il Concetto di Processo

- Un sistema operativo esegue diversi programmi
  - nei sistemi batch – “jobs”
  - nei sistemi time-shared – “programmi utente” o “task”
- I libri usano i termini *job* e *processo* quasi come sinonimi
- Processo: programma in esecuzione. L'esecuzione è sequenziale.
- Un processo comprende anche tutte le risorse di cui necessita, tra cui:
  - programma
  - program counter
  - stack
  - sezione dati
  - dispositivi

# Multiprogrammazione

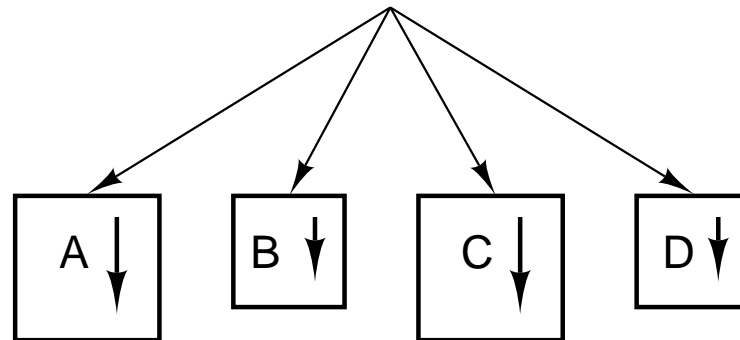
Multiprogrammazione: più processi in memoria, per tenere occupate le CPU.  
Time-sharing: le CPU vengono “multiplexate” tra più processi

One program counter

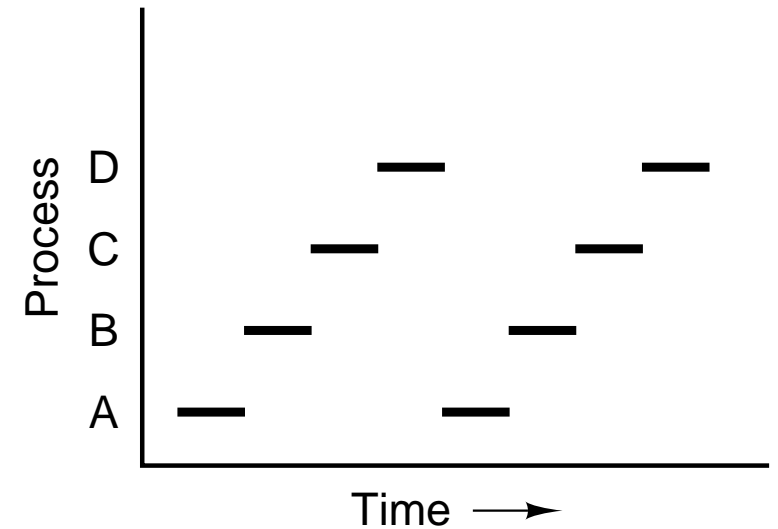


(a)

Four program counters



(b)



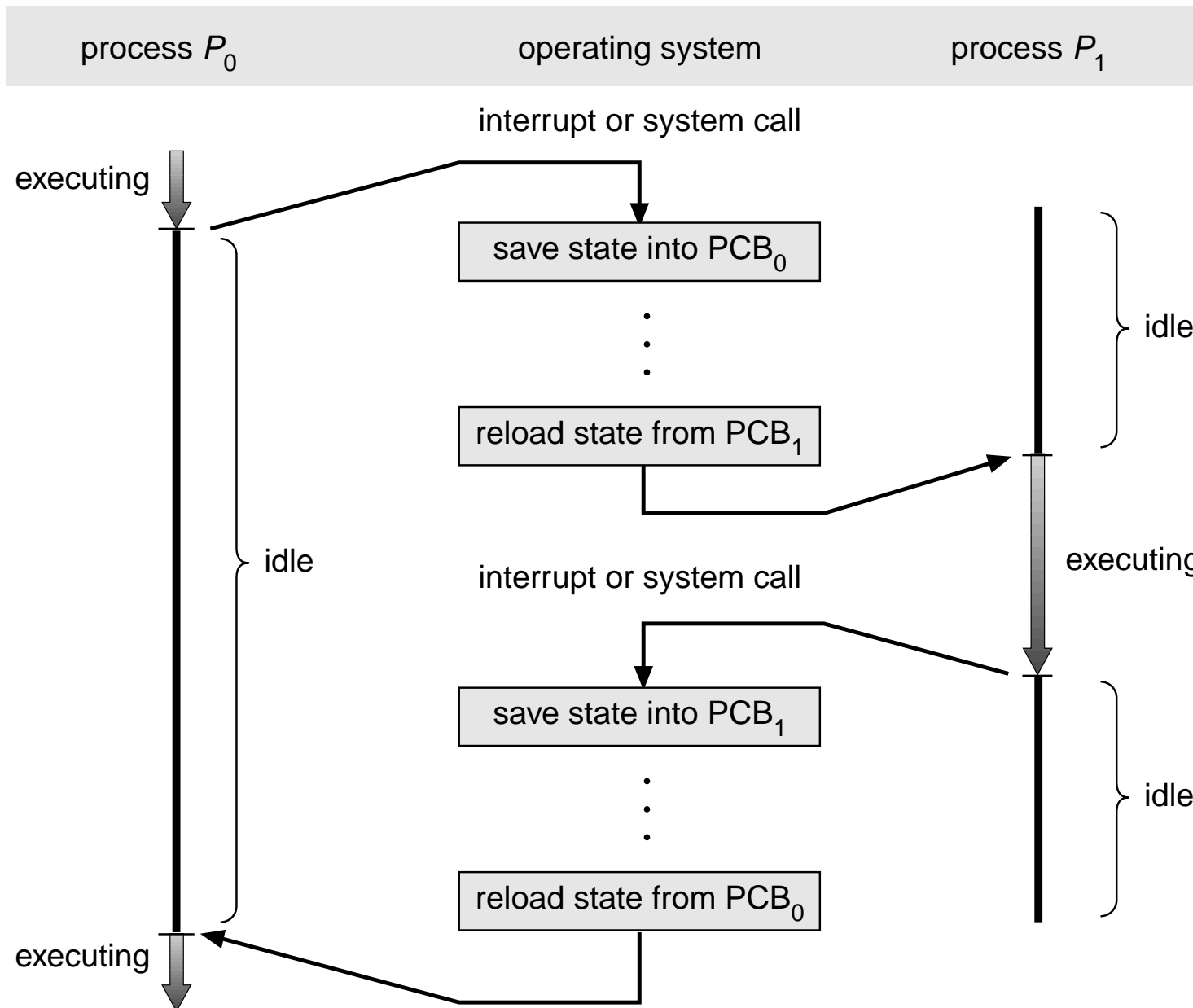
(c)

Switch causato da terminazione, prelazione, system-call bloccante.

## Switch di Contesto

- Quando la CPU passa ad un altro processo, il sistema deve salvare lo stato del vecchio processo e caricare quello del nuovo processo.
- Lo stato del processo viene salvato nel Process Control Block (PCB)
- Il tempo di *context-switch* porta un certo overhead; il sistema non fa un lavoro utile mentre passa di contesto
- Può essere un collo di bottiglia per sistemi operativi ad alto parallelismo (migliaia - decine di migliaia di thread).
- Il tempo impiegato per lo switch dipende dal supporto hardware

# Switch di contesto



# Creazione dei processi

- Quando viene creato un processo
  - Al boot del sistema (es. demoni di stampa, di rete, ecc.)
  - Su esecuzione di una system call apposita (es. tramite la system call `fork` di Unix)
  - Su richiesta da parte dell'utente
  - Inizio di un job batch

La generazione dei processi indica una naturale gerarchia, detta *albero di processi*.

- Esecuzione: Possibili alternative
  - Padre e figli sono in esecuzione concorrente
  - Il padre attende che i figli terminino per riprendere l'esecuzione

- Condivisione delle risorse: Possibili alternative
  - Padre e figli condividono le stesse risorse
  - I figli condividono un sottoinsieme delle risorse del padre
  - Padre e figli non condividono nessuna risorsa
  
- Spazio indirizzi: Possibili alternative
  - I figli duplicano quello del padre
  - I figli caricano sempre un programma

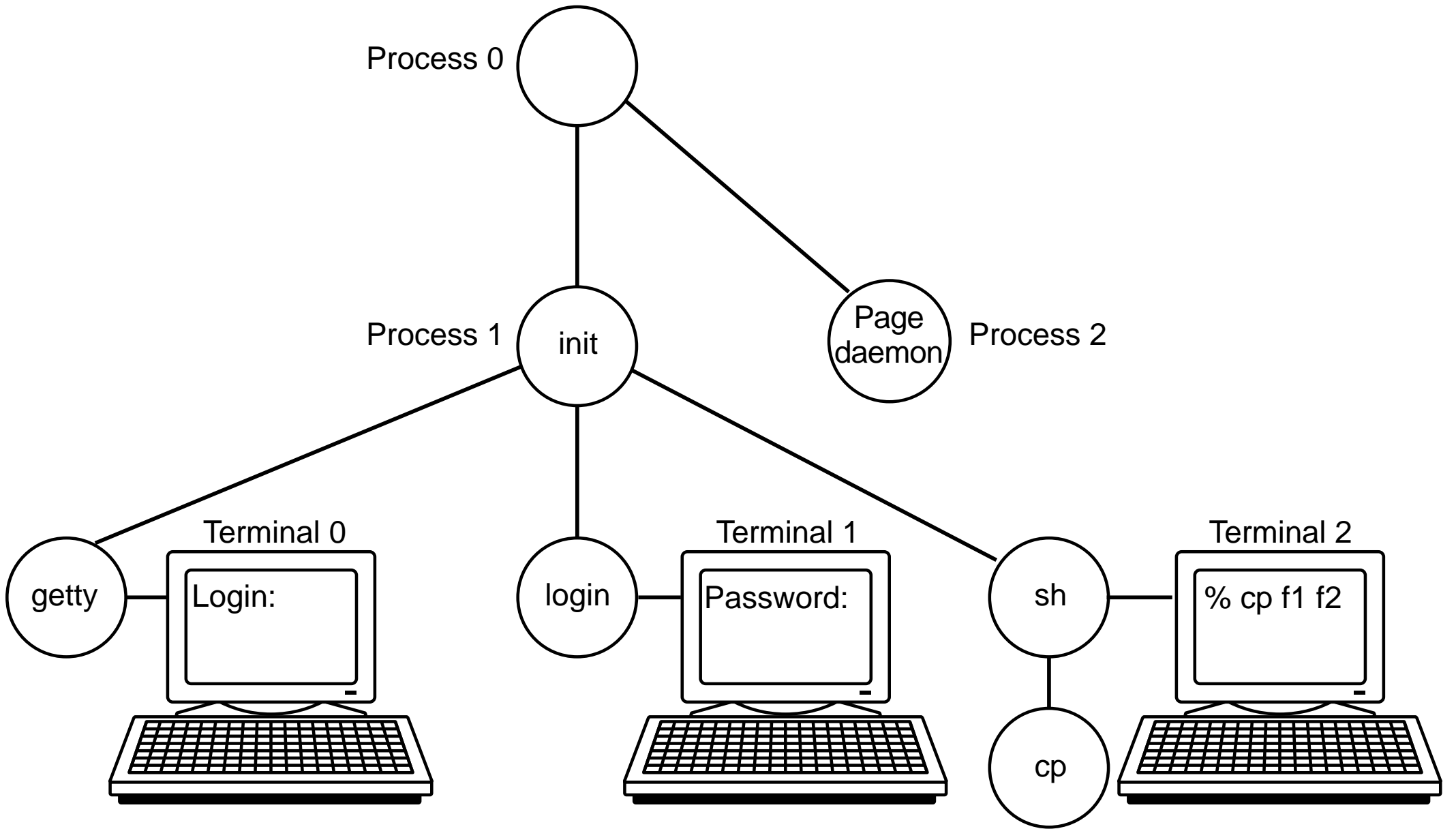
## Terminazione dei Processi

- Terminazione volontaria—normale o con errore. I dati di output vengono ricevuti dal processo padre (che li attendeva ad esempio invocando la system call **wait** in Unix).
- 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.

# Gerarchia dei processi

- In alcuni sistemi, i processi generati (*figli*) rimangono collegati al processo generatore (*parent, genitore*).
- Si formano “famiglie” di processi (*gruppi*)
- Utili per la comunicazione tra processi: e.g. i **segnali** possono essere mandati solo all'interno di un gruppo, o ad un intero gruppo.
- In UNIX: tutti i processi discendono da `init` (processo con identificatore `PID=1`). Se un processo genitore muore, il figlio viene ereditato da `init`. Un processo non può diseredare il figlio.
- In Windows non c'è gerarchia di processi; il task creatore ha un puntatore al figlio, che comunque può essere passato ad un altro processo.



# Stato del processo

Durante l'esecuzione, un processo cambia *stato*.

In *generale* si possono individuare i seguenti stati:

**new:** il processo è appena creato

**running:** istruzioni del programma vengono eseguite da una CPU.

**waiting:** il processo attende qualche evento

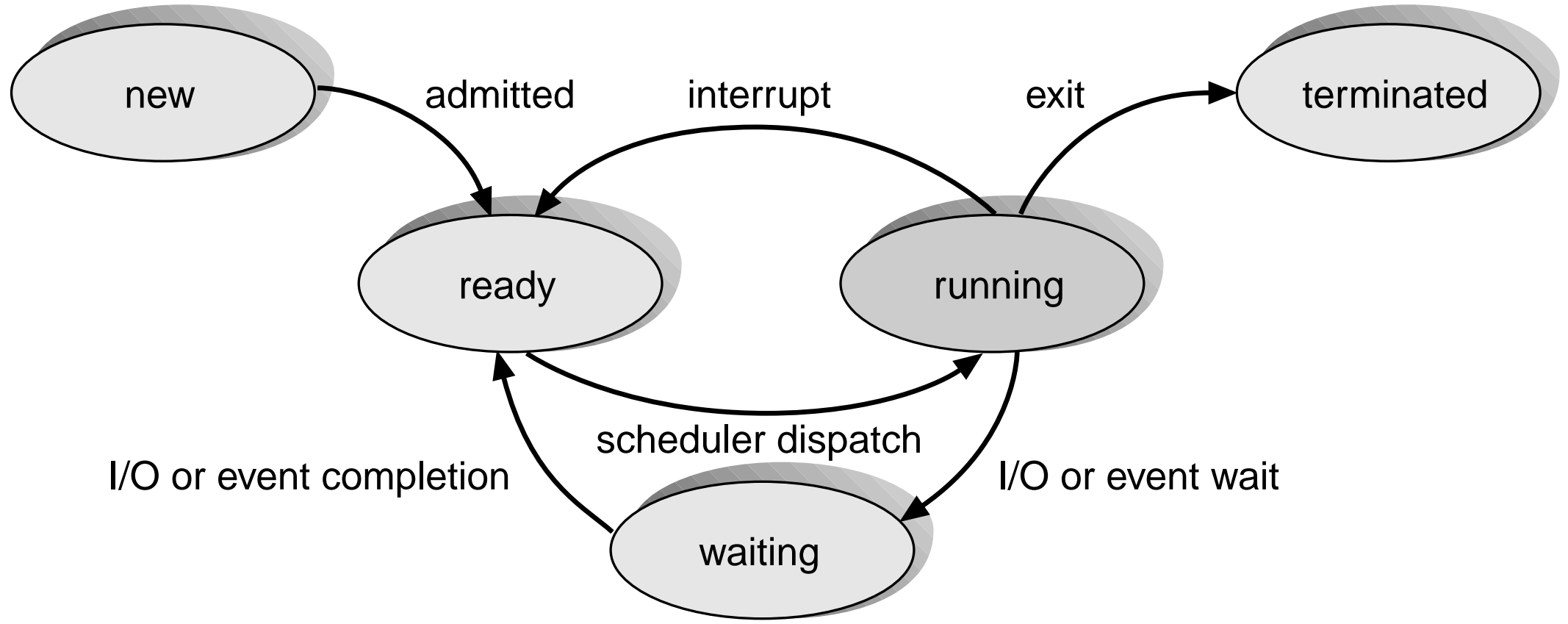
**ready:** il processo attende di essere assegnato ad un processore

**terminated:** il processo ha completato la sua esecuzione

Il passaggio da uno stato all'altro avviene in seguito a interruzioni, richieste di risorse non disponibili, selezione da parte dello scheduler, ...

$0 \leq n.$  processi in running  $\leq n.$  di processori nel sistema

# Diagramma degli stati



# Process Control Block (PCB)

Contiene le informazioni associate ad un processo

- Stato del processo
- Dati identificativi (del processo, dell'utente)
- Program counter
- Registri della CPU
- Informazioni per lo scheduling della CPU
- Informazioni per la gestione della memoria
- Informazioni di utilizzo risorse
  - tempo di CPU, memoria, file...
  - eventuali limiti (*quota*)
- Stato dei segnali (per la comunicazione interprocess)

**Process management**

Registers

Program counter

Program status word

Stack pointer

Process state

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

**Memory management**

Pointer to text segment

Pointer to data segment

Pointer to stack segment

**File management**

Root directory

Working directory

File descriptors

User ID

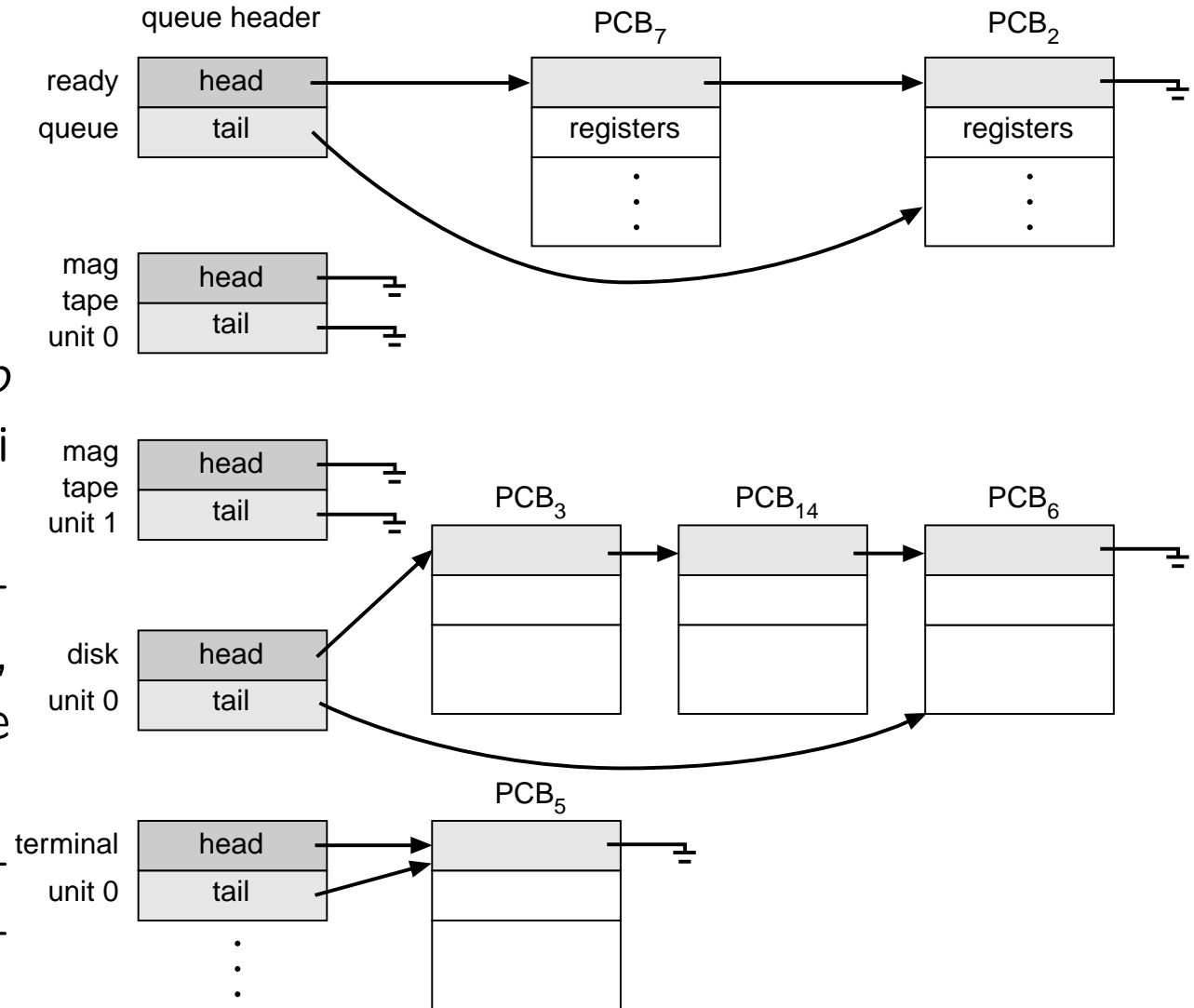
Group ID

## **Gestione di una interruzione hardware**

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

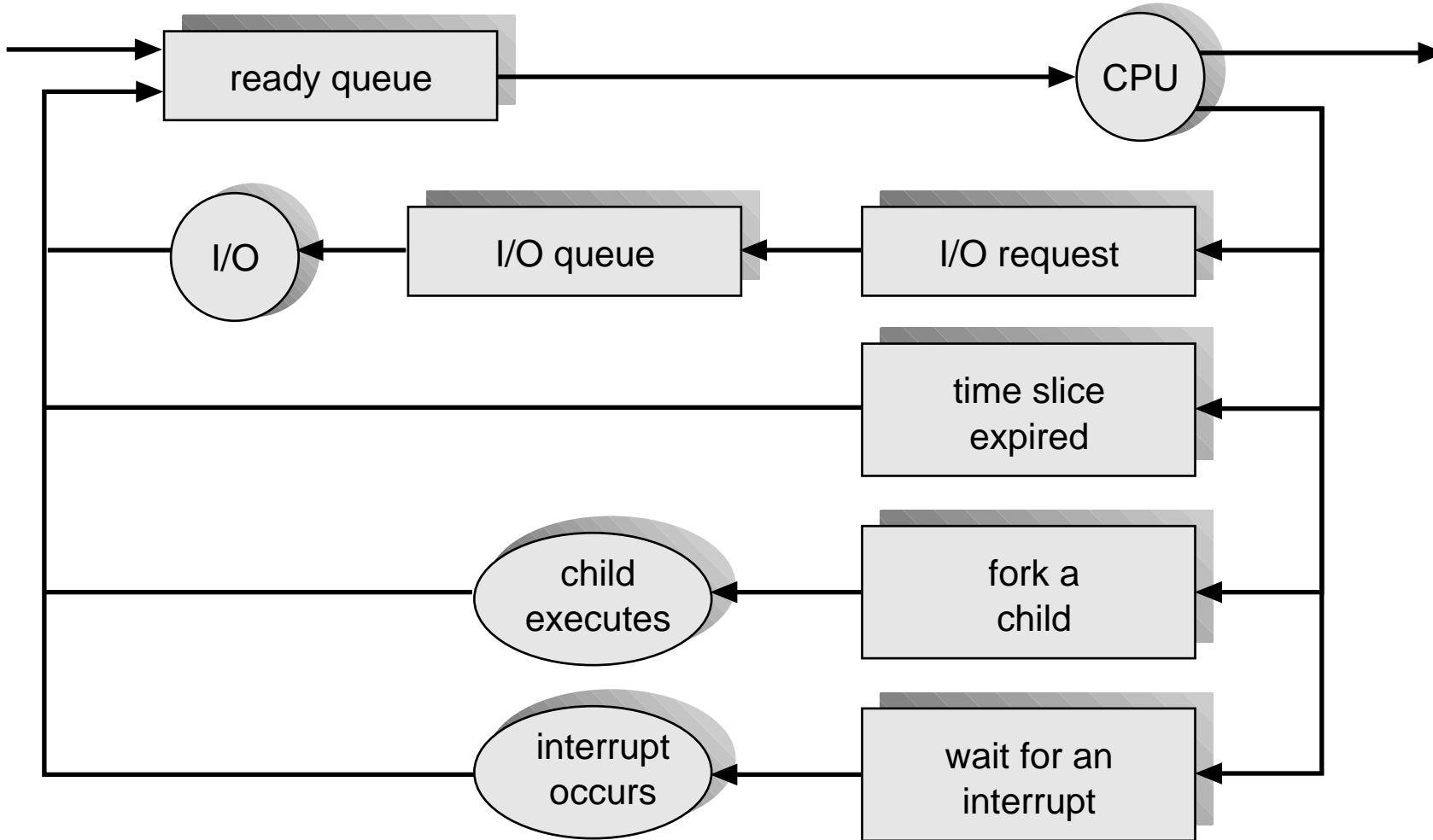
# Code di scheduling dei processi

- Coda dei processi (*Job queue*) – insieme di tutti i processi nel sistema
- *Ready queue* – processi residenti in memoria principale, pronti e in attesa di essere messi in esecuzione
- *Code dei dispositivi* – processi in attesa di un dispositivo di I/O.



# Migrazione dei processi tra le code

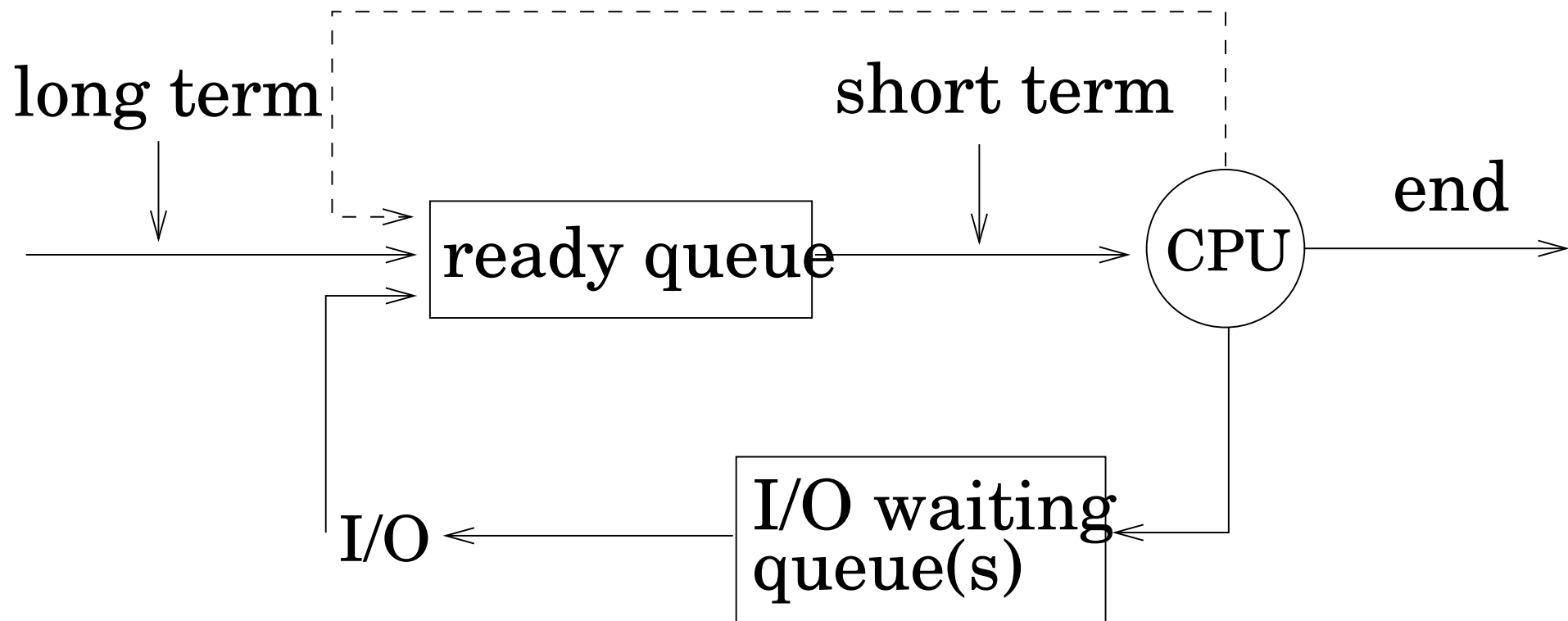
I processi, durante l'esecuzione, migrano da una coda all'altra



Gli *scheduler* scelgono quali processi passano da una coda all'altra.

# Gli Scheduler

- Lo *scheduler di lungo termine* (o *job scheduler*) seleziona i processi da portare nella ready queue.
- Lo *scheduler di breve termine* (o *CPU scheduler*) seleziona quali processi ready devono essere eseguiti, e quindi assegna la CPU.

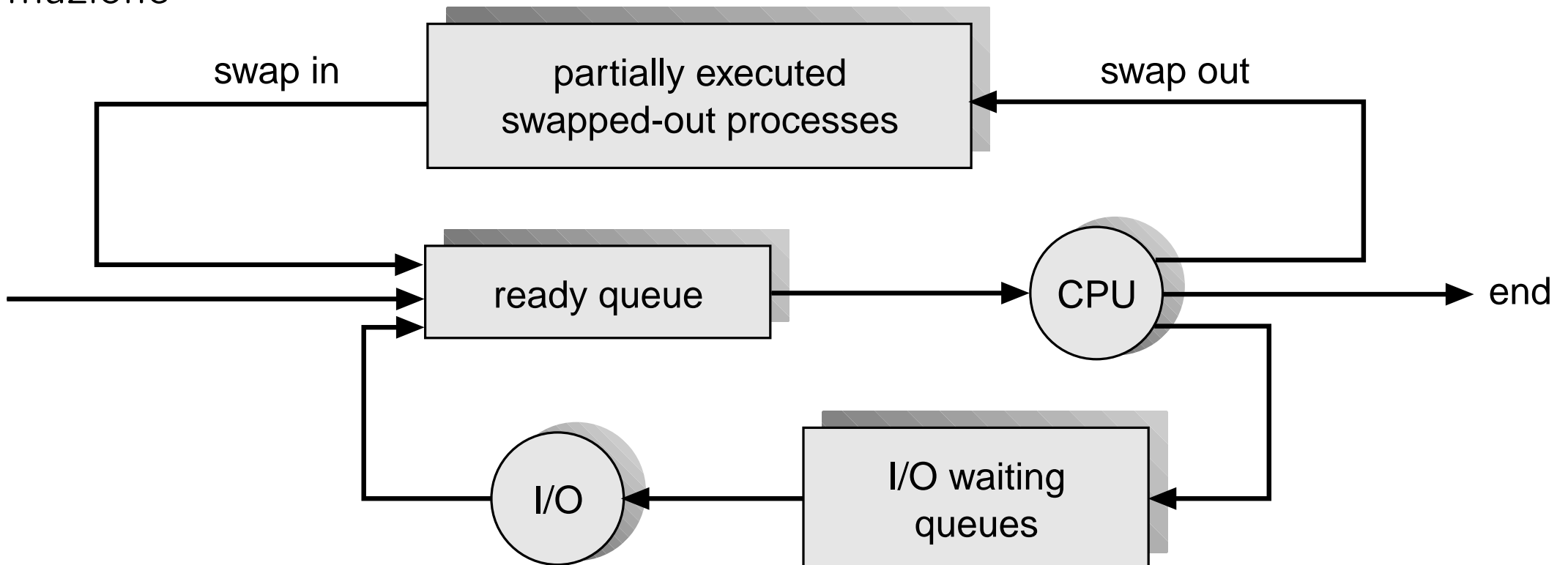


## Gli Scheduler (Cont.)

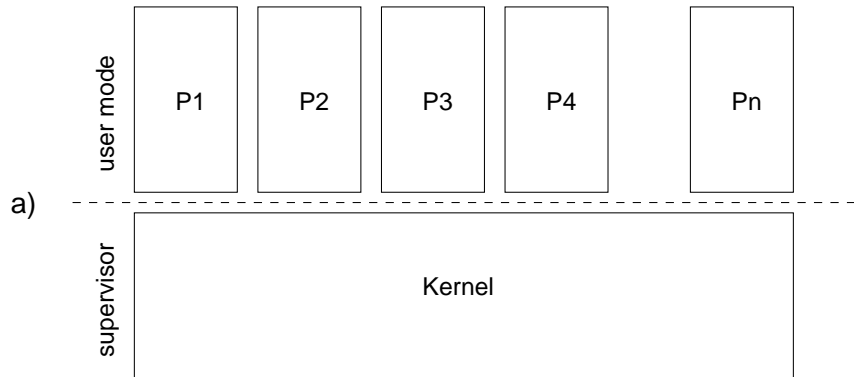
- Lo scheduler di breve termine è invocato molto frequentemente (decine di volte al secondo)  $\Rightarrow$  deve essere veloce
- Lo scheduler di lungo termine è invocato raramente (secondi, minuti)  $\Rightarrow$  può essere lento e sofisticato
- I processi possono essere descritti come
  - I/O-bound: lunghi periodi di I/O, brevi periodi di calcolo.
  - CPU-bound: lunghi periodi di intensiva computazione, pochi (possibilmente lunghi) cicli di I/O.
- Lo scheduler di lungo termine controlla il grado di multiprogrammazione e il *job mix*: un giusto equilibrio tra processi I/O e CPU bound.

## Gli Schedulers (Cont.)

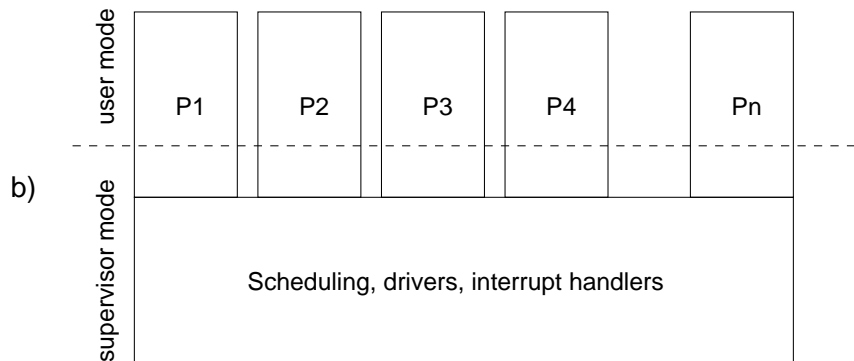
Alcuni sistemi hanno anche lo *scheduler di medio termine* (o *swap scheduler*) sospende temporaneamente i processi per abbassare il livello di multiprogrammazione



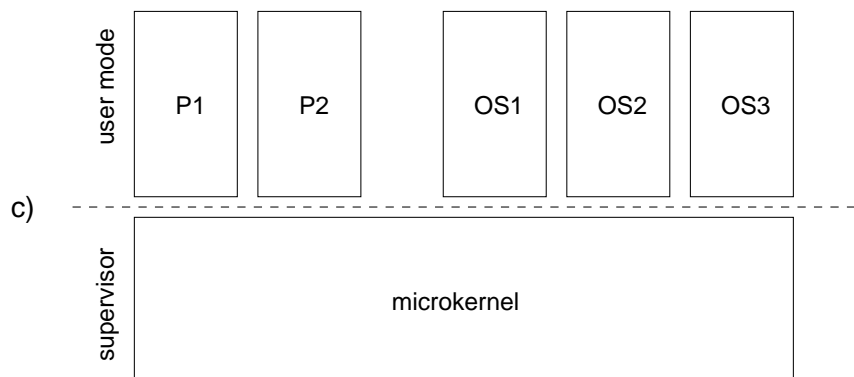
# Modelli di esecuzione dei processi



Esecuzione kernel separata dai processi utente.



Esecuzione kernel all'interno dei processi



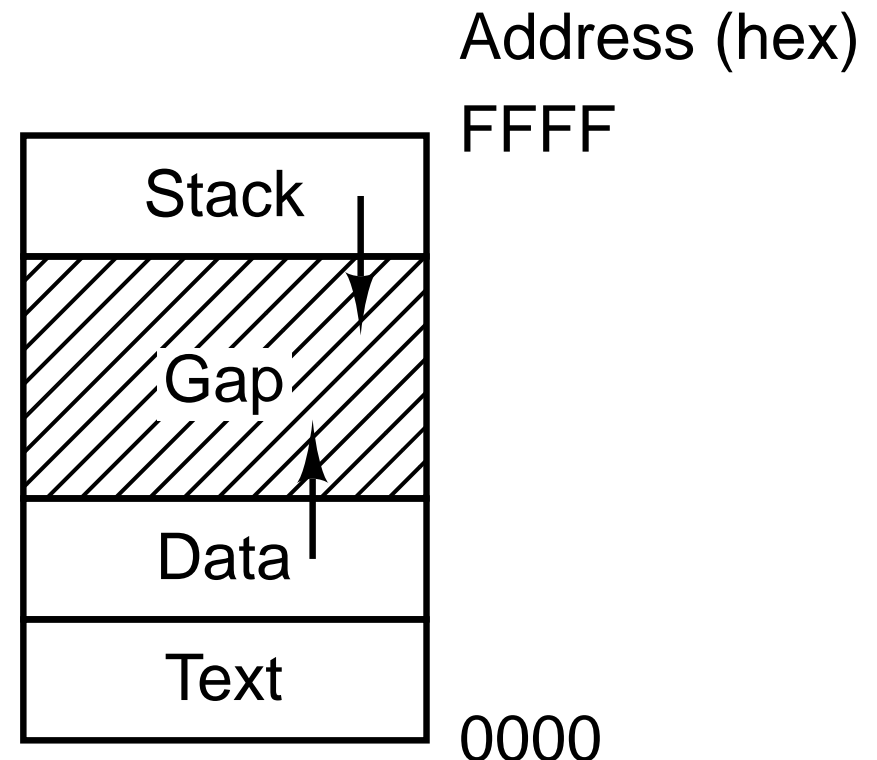
Stretto necessario all'interno del kernel; le decisioni vengono prese da processi di sistema.

## Esempio esteso: Processi in UNIX tradizionale

- Un *processo* è un programma in esecuzione + le sue risorse
- Identificato dal *process identifier (PID)*, un numero assegnato dal sistema.
- Ogni processo UNIX ha uno spazio indirizzi separato. Non vede le zone di memoria dedicati agli altri processi.

Un processo UNIX ha tre segmenti:

- Stack: Stack di attivazione delle subroutine. Cambia dinamicamente.
- Data: Contiene lo heap e i dati inizializzati al caricamento del programma. Cambia dinamicamente su richiesta esplicita del programma (es., con la **malloc**).
- Text: codice eseguibile. Non modificabile, protetto in scrittura.



## Alcune chiamate di sistema per gestione dei processi

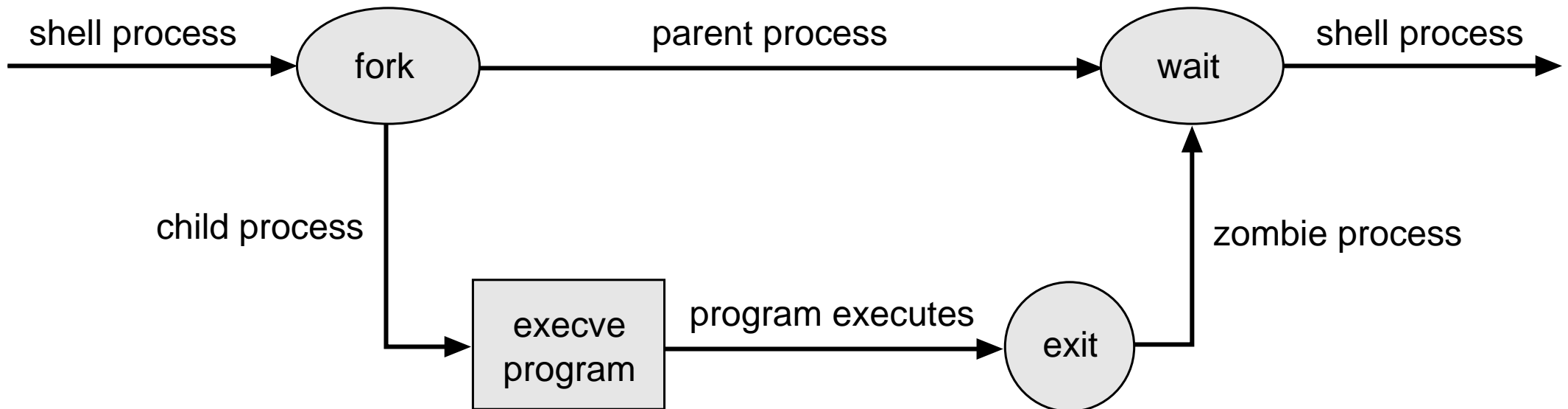
<b>System call</b>	<b>Description</b>
<code>pid = fork( )</code>	Create a child process identical to the parent
<code>pid = waitpid(pid, &amp;statloc, opts)</code>	Wait for a child to terminate
<code>s = execve(name, argv, envp)</code>	Replace a process' core image
<code>exit(status)</code>	Terminate process execution and return status
<code>s = sigaction(sig, &amp;act, &amp;oldact)</code>	Define action to take on signals
<code>s = sigreturn(&amp;context)</code>	Return from a signal
<code>s = sigprocmask(how, &amp;set, &amp;old)</code>	Examine or change the signal mask
<code>s = sigpending(set)</code>	Get the set of blocked signals
<code>s = sigsuspend(sigmask)</code>	Replace the signal mask and suspend the process
<code>s = kill(pid, sig)</code>	Send a signal to a process
<code>residual = alarm(seconds)</code>	Set the alarm clock
<code>s = pause( )</code>	Suspend the caller until the next signal

## 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 */
```

## Esempio: ciclo fork/wait di una shell

```
while (1) {  
  read_command(commands, parameters);  
  if (fork() != 0) { /* parent code */  
    waitpid(-1, &status, 0);  
  } else { /* child code */  
    execve(command, parameters, NULL);  
  }  
}
```



## Gestione e implementazione dei processi in UNIX

- In UNIX, l'utente può creare e manipolare direttamente più processi
- I processi sono rappresentati da *process control block*
  - Il PCB di ogni processo è memorizzato in parte nel kernel (*process structure, text structure*), in parte nello spazio di memoria del processo (*user structure*)
  - L'informazione in questi blocchi di controllo è usata dal kernel per il controllo dei processi e per lo scheduling.

# Process Control Blocks

- La struttura base più importante è la *process structure*: contiene
  - stato del processo
  - puntatori alla memoria (segmenti, u-structure, text structure)
  - identificatori del processo
  - identificatori dell'utente
  - informazioni di scheduling (e.g., priorità)
  - segnali non gestiti
- La *text structure*
  - è sempre residente in memoria
  - memorizza quanti processi stanno usando il segmento text (permette quindi condivisioni del codice)
  - contiene dati relativi alla gestione della memoria virtuale per il text

## Process Control Block (Cont.)

- Le informazioni sul processo che sono richieste solo quando il processo è residente sono mantenute nella *user structure* (o *u structure*).

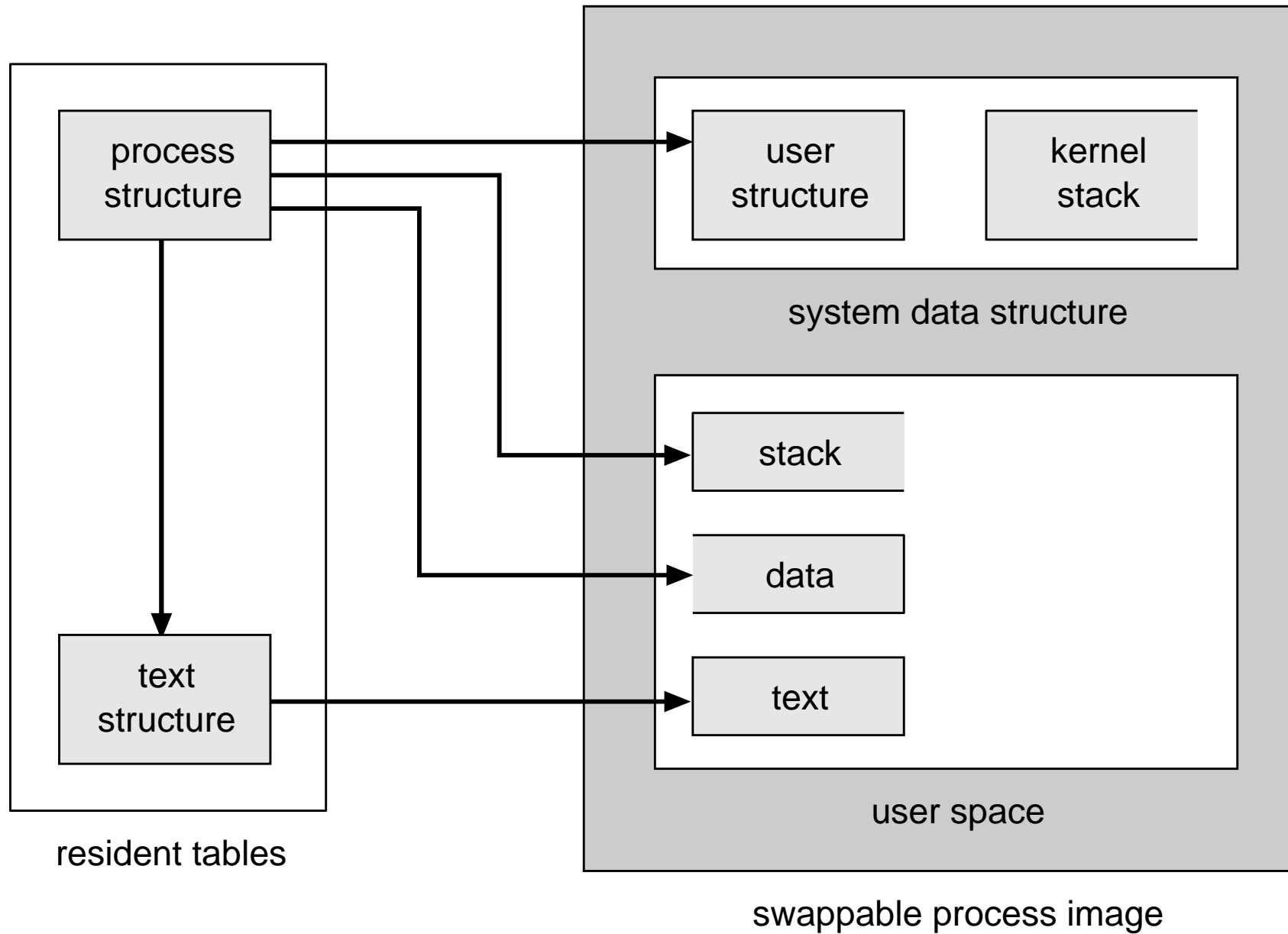
Fa parte dello spazio indirizzi modo user, read-only (ma scrivibile dal kernel) e contiene (tra l'altro)

- identificatore utente e gruppo
- risultati/errori delle system call
- tabella dei file aperti
- limiti del processo

## Segmenti dei dati di sistema

- La maggior parte della computazione viene eseguita in user mode; le system call vengono eseguite in modo di sistema (o supervisore).
- Le due fasi di un processo non si sovrappongono mai: un processo si trova sempre in una o l'altra fase.
- Per l'esecuzione in modo kernel, il processo usa uno stack separato (*kernel stack*), invece di quello del modo utente.
- Kernel stack + u structure = *system data segment* del processo

# Parti e strutture di un processo



## Creazione di un processo

- La **fork** alloca una nuova process structure per il processo figlio
  - nuove tabelle per la gestione della memoria virtuale
  - nuova memoria viene allocata per i segmenti dati e stack
  - i segmenti dati e stack e la user structure vengono copiati ⇒ vengono preservati i file aperti, UID e GID, gestione segnali, etc.
  - il text segment viene condiviso, puntando alla stessa text structure
- La **execve** non crea nessun nuovo processo: semplicemente, i segment dati e stack vengono rimpiazzati

# Threads

## Dai processi. . .

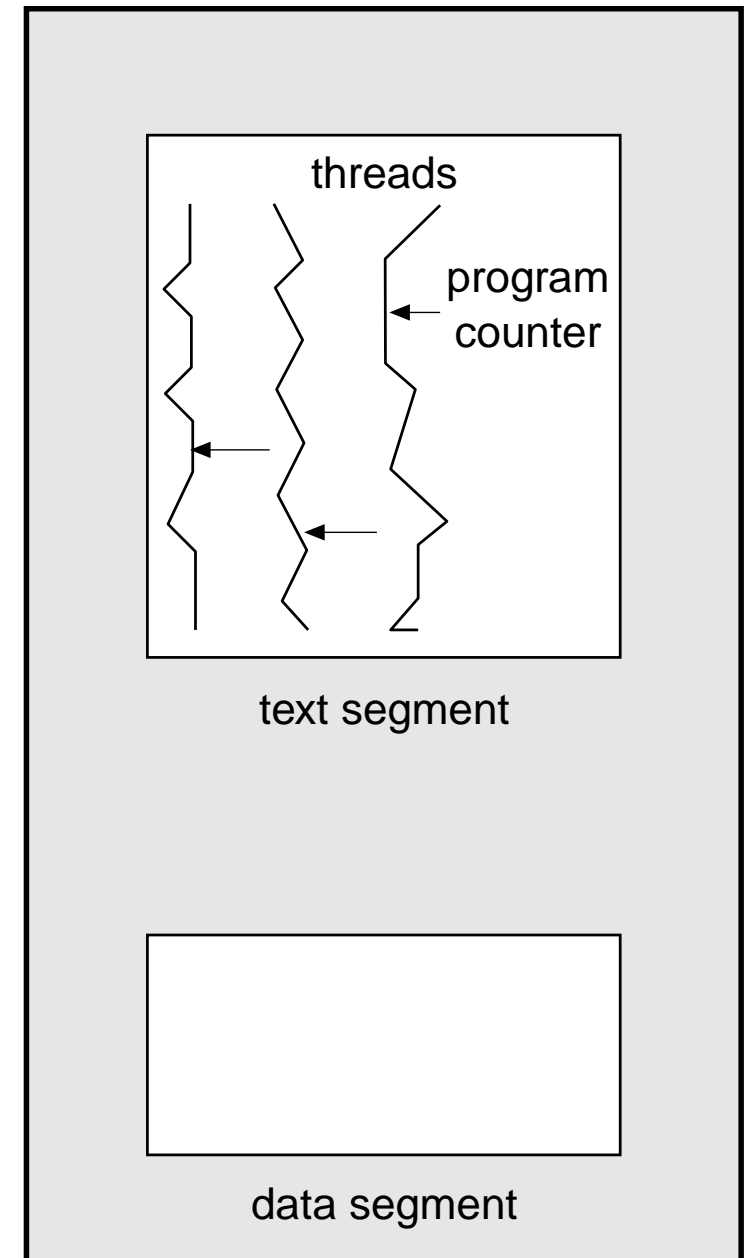
I processi finora studiati incorporano due caratteristiche:

- Unità di allocazione risorse: codice eseguibile, dati allocati staticamente (variabili globali) ed esplicitamente (heap), risorse mantenute dal kernel (file, I/O, workind dir), controlli di accesso . . .
- Unità di esecuzione: un percorso di esecuzione attraverso uno o più programmi: stack di attivazione (variabili locali), stato (running, ready, waiting, . . .), priorità, parametri di scheduling, . . .

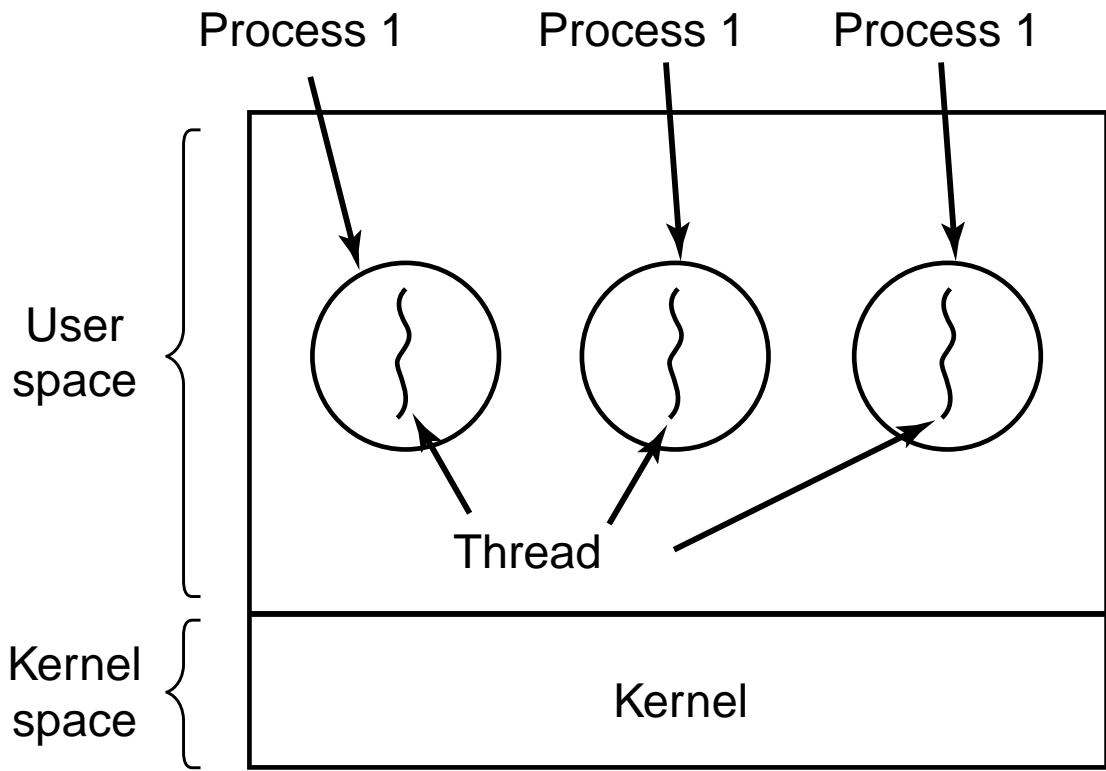
Queste due componenti sono in realtà *indipendenti*

## ... ai thread

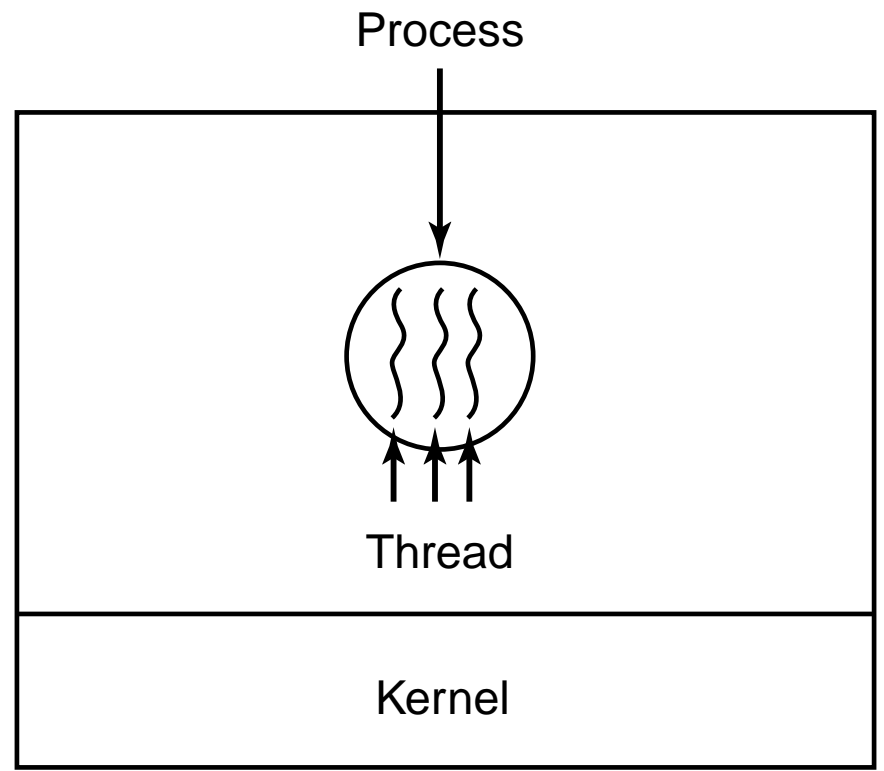
- Un *thread* (o *processo leggero*, *lightweight process*) è una unità di esecuzione:
  - program counter, insieme registri
  - stack del processore
  - stato di esecuzione
- Un thread condivide con i thread suoi pari *task* una unità di allocazione risorse:
  - il codice eseguibile
  - i dati
  - le risorse richieste al sistema operativo
- un *task* = una unità di risorse + i thread che vi accedono



# Esempi di thread

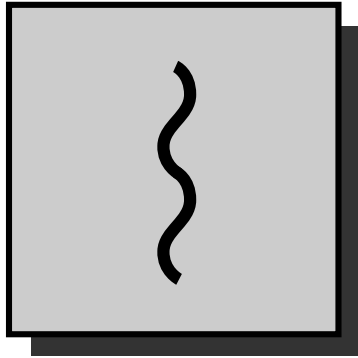


(a)

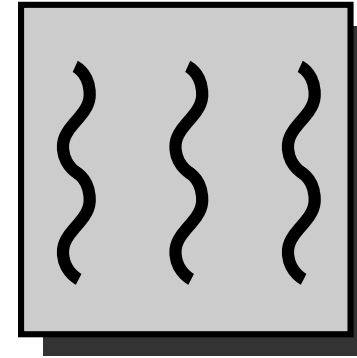


(b)

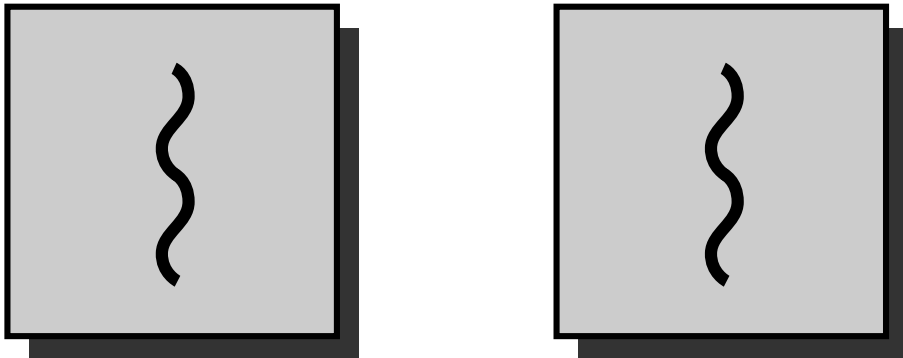
# Processi e Thread: quattro possibili scenari



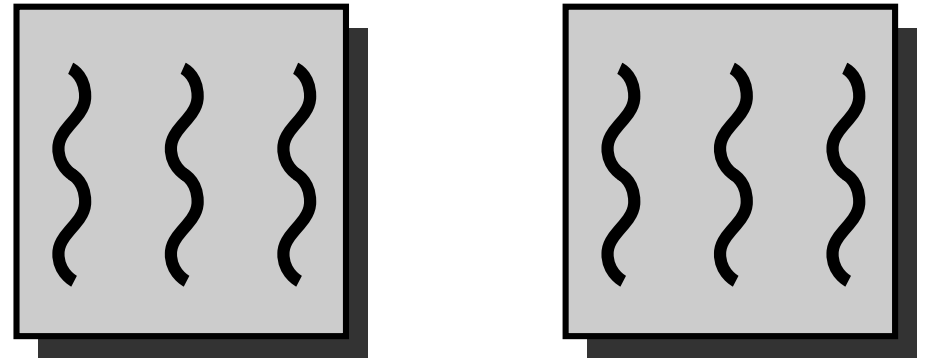
**one process  
one thread**



**one process  
multiple threads**



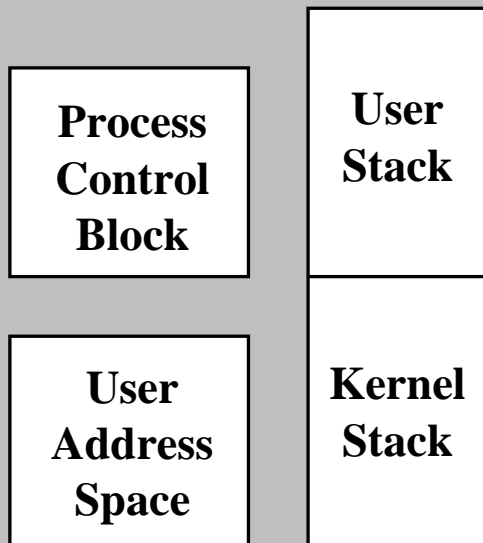
**multiple processes  
one thread per process**



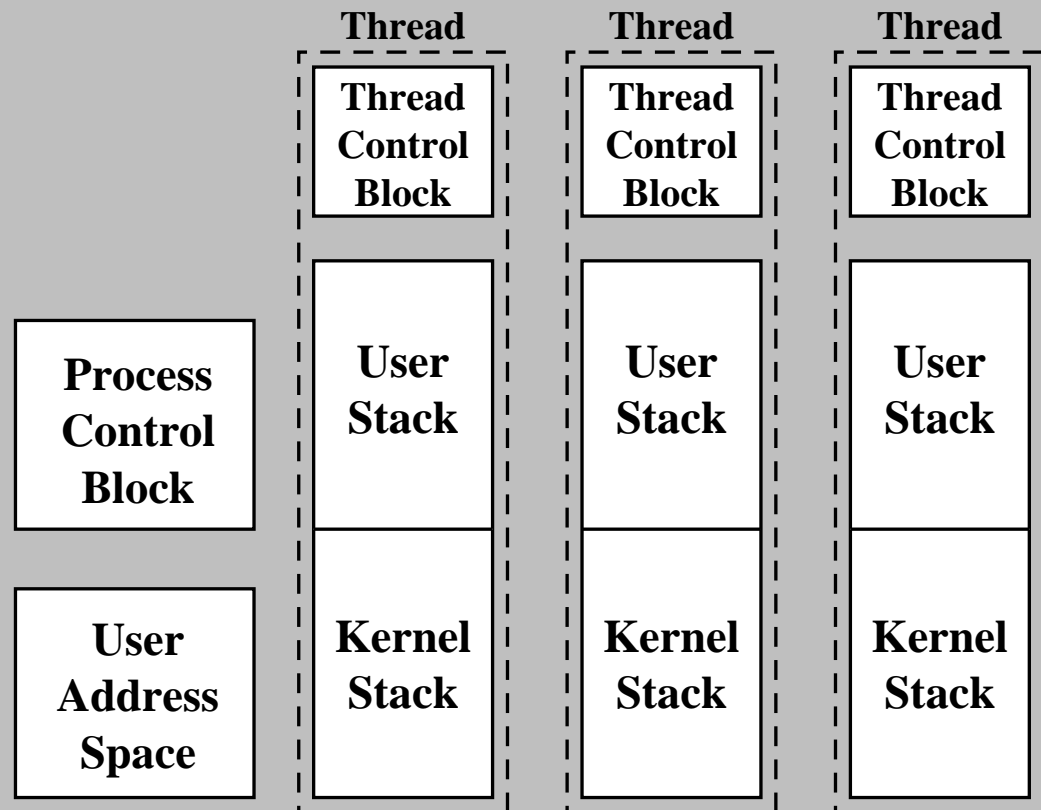
**multiple processes  
multiple threads per process**

# Modello multithread dei processi

## Single-Threaded Process Model



## Multithreaded Process Model



## Risorse condivise e private dei thread

Tutti i thread di un processo accedono alle stesse risorse condivise

### **Per process items**

- Address space
- Global variables
- Open files
- Child processes
- Pending alarms
- Signals and signal handlers
- Accounting information

### **Per thread items**

- Program counter
- Registers
- Stack
- State

## Condivisione di risorse tra i thread

- Vantaggi: maggiore efficienza
  - Creare e cancellare thread è più veloce (100–1000 volte): meno informazione da duplicare/creare/cancellare (e a volte non serve la system call)
  - Lo scheduling tra thread dello stesso processo è molto più veloce che tra processi
  - Cooperazione di più thread nello stesso task porta maggiore throughput e performance  
(es: in un file server multithread, mentre un thread è bloccato in attesa di I/O, un secondo thread può essere in esecuzione e servire un altro client)

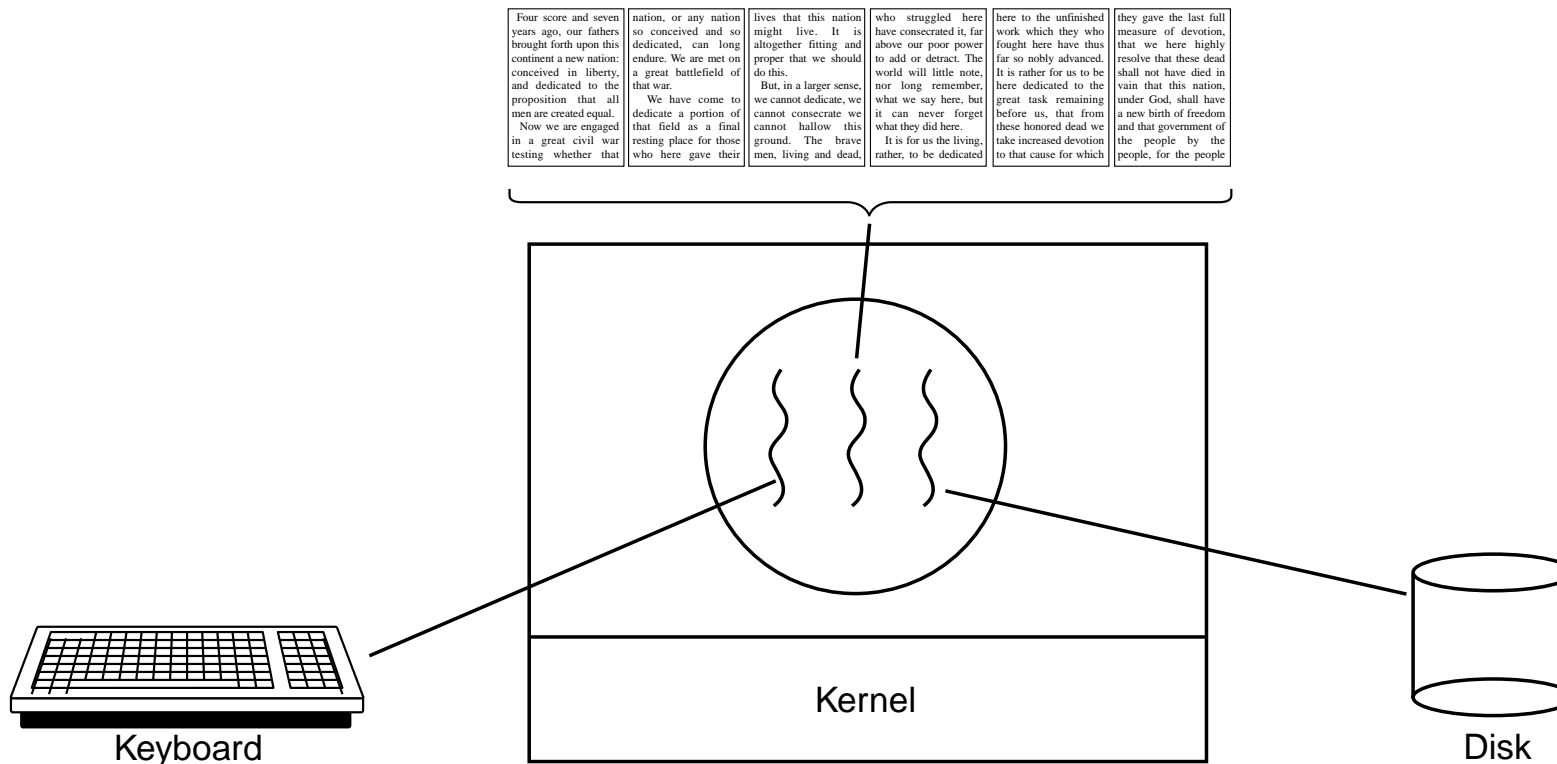
## Condivisione di risorse tra thread (Cont.)

- Svantaggi:
  - Maggiore complessità di progettazione e programmazione
    - \* i processi devono essere “pensati” paralleli
    - \* minore information hiding
    - \* sincronizzazione tra i thread
    - \* gestione dello scheduling tra i thread può essere demandato all’utente
  - Inadatto per situazioni in cui i dati devono essere protetti
- Ottimi per processi cooperanti che devono condividere strutture dati o comunicare (e.g., produttore–consumatore, server, ...): la comunicazione non coinvolge il kernel

# Esempi di applicazioni multithread

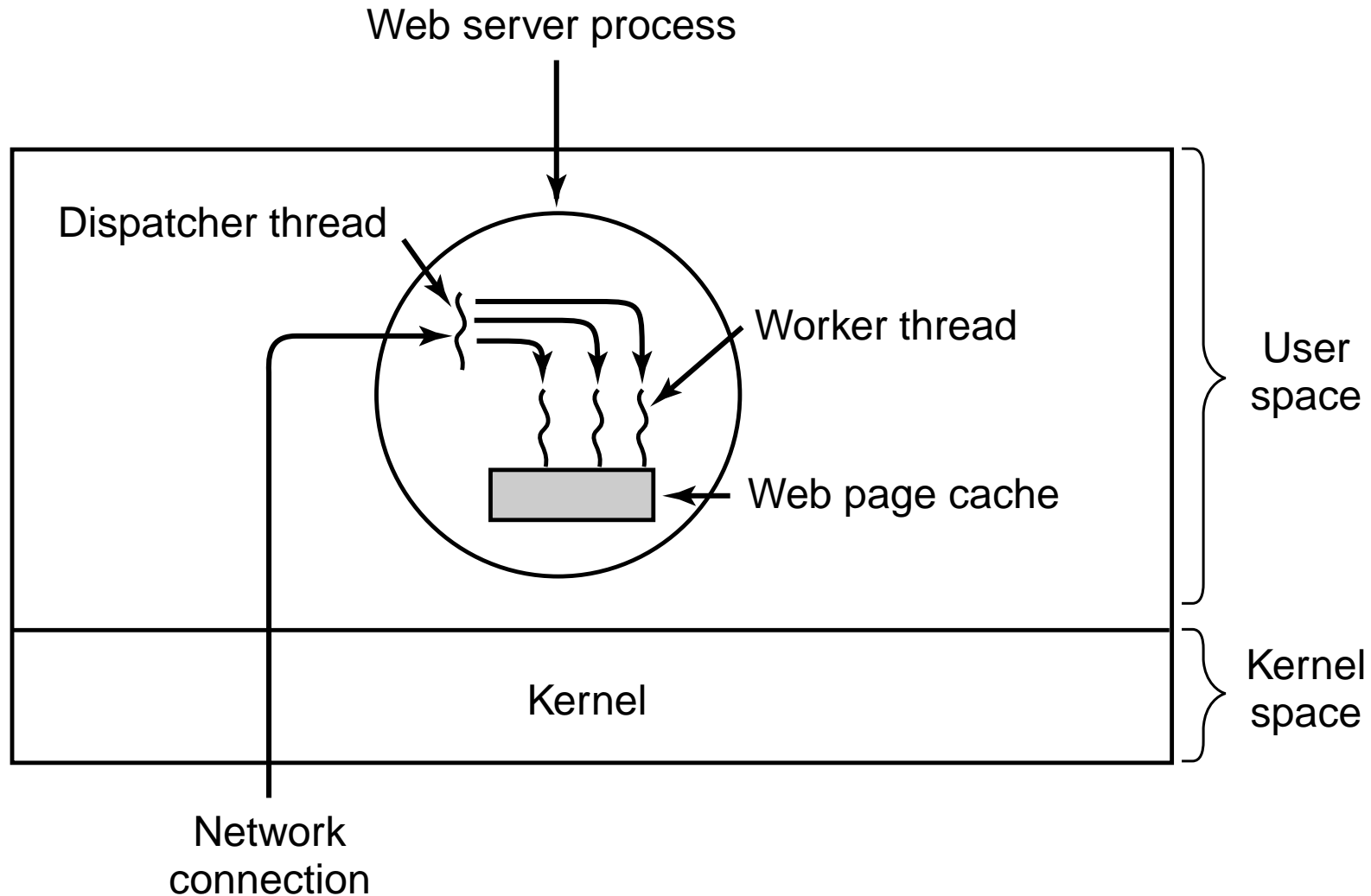
**Lavoro foreground/background:** mentre un thread gestisce l'I/O con l'utente, altri thread operano sui dati in background. Spreadsheets (ricalcolo automatico), word processor (reimpaginazione, controllo ortografico, . . .)

**Elaborazione asincrona:** operazioni asincrone possono essere implementate come thread. Es: salvataggio automatico.



## Esempi di applicazioni multithread (cont.)

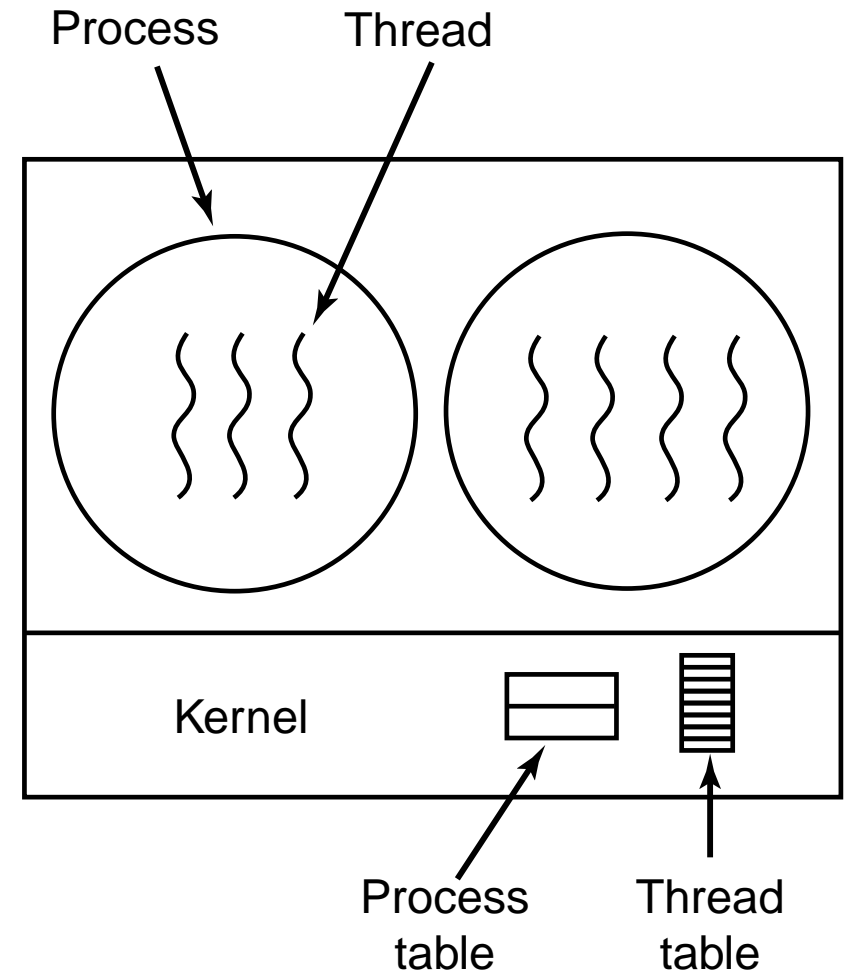
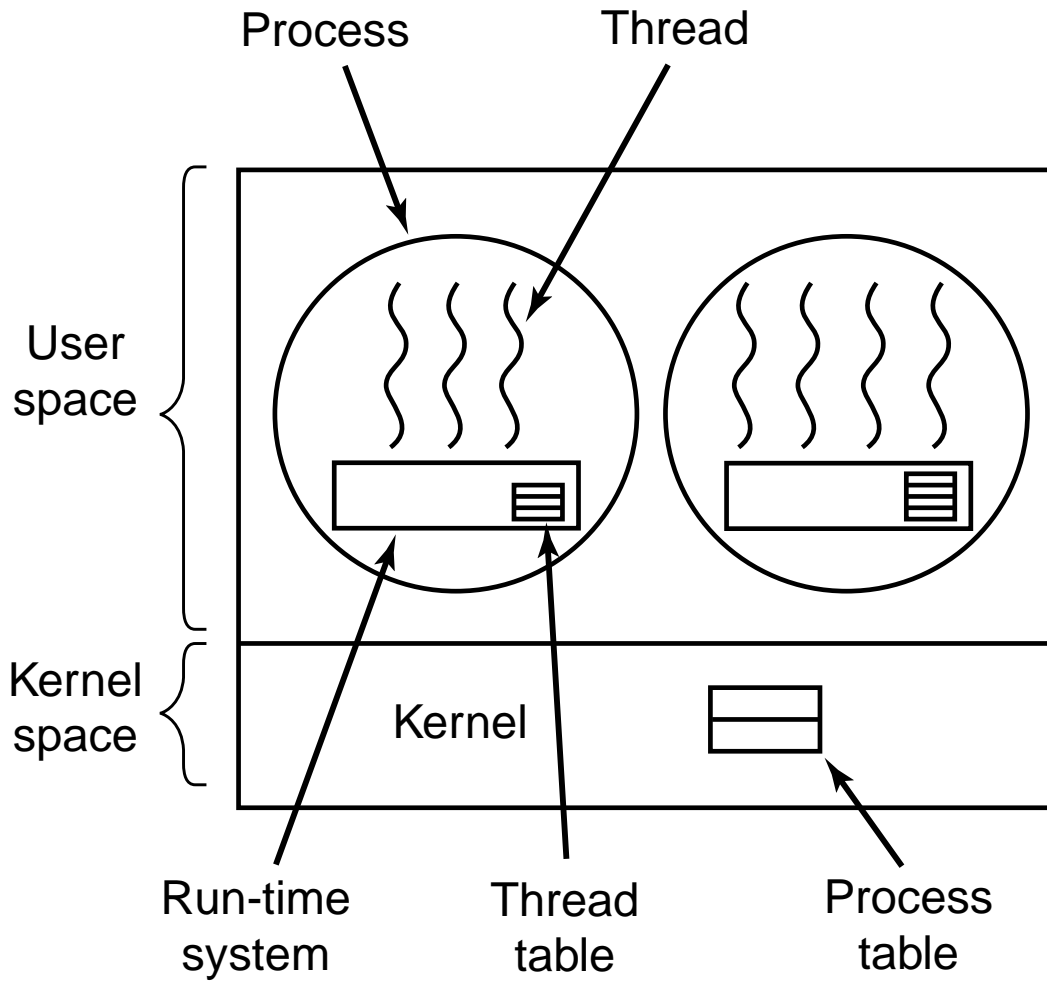
**Task intrinsecamente paralleli:** vengono implementati ed eseguiti più efficientemente con i thread. Es: file/http/dbms/ftp server, ...



## Stati e operazioni sui thread

- Stati: *running*, *ready*, *blocked*. Non ha senso “swapped” o “suspended”
- Operazioni sui thread:
  - creazione (spawn):** un nuovo thread viene creato all'interno di un processo (`thread_create`), con un proprio punto d'inizio, stack, ...
  - blocco:** un thread si ferma, e l'esecuzione passa ad un altro thread/processo. Può essere volontario (`thread_yield`) o su richiesta di un evento;
  - sblocco:** quando avviene l'evento, il thread passa dallo stato “blocked” al “ready”
  - cancellazione:** il thread chiede di essere cancellato (`thread_exit`); il suo stack e le copie dei registri vengono deallocati.
- Meccanismi per la sincronizzazione tra i thread (semafori, `thread_wait`): indispensabili per l'accesso concorrente ai dati in comune

# Implementazioni dei thread: Livello utente vs Livello Kernel



# User Level Thread

**User-level thread (ULT):** stack, program counter, e operazioni su thread sono implementati in librerie a livello utente.

Vantaggi:

- efficiente: non c'è il costo della system call
- semplici da implementare su sistemi preesistenti
- portabile: possono soddisfare lo standard POSIX 1003.1c (pthread)
- lo scheduling può essere studiato specificatamente per l'applicazione

## User Level Thread (Cont.)

Svantaggi:

- non c'è scheduling automatico tra i thread
  - non c'è prelazione dei thread: se un thread non passa il controllo esplicitamente monopolizza la CPU (all'interno del processo)
  - system call bloccanti bloccano tutti i thread del processo: devono essere sostituite con delle routine di libreria, che blocchino solo il thread se i dati non sono pronti (*jacketing*).
- L'accesso al kernel è sequenziale
- Non sfrutta sistemi multiprocessore
- Poco utile per processi I/O bound, come file server

Esempi: thread CMU, Mac OS  $\leq 9$ , alcune implementazioni dei thread POSIX

# Kernel Level Thread

**Kernel-level thread (KLT):** il kernel gestisce direttamente i thread. Le operazioni sono ottenute attraverso system call. Vantaggi:

- lo scheduling del kernel è per thread, non per processo  $\Rightarrow$  un thread che si blocca non blocca l'intero processo
- Utile per i processi I/O bound e sistemi multiprocessor

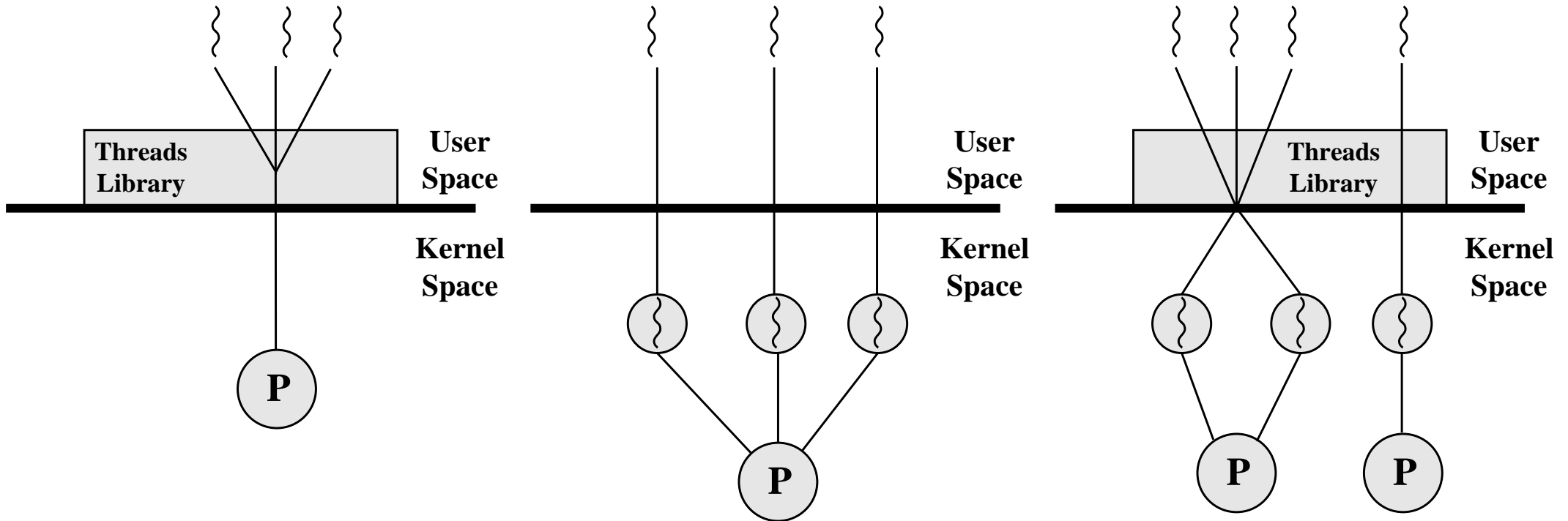
Svantaggi:

- meno efficiente: costo della system call per ogni operazione sui thread
- necessita l'aggiunta e la riscrittura di system call dei kernel preesistenti
- meno portabile
- la politica di scheduling è fissata dal kernel e non può essere modificata

Esempi: molti Unix moderni, OS/2, Mach.

# Implementazioni ibride ULT/KLT

**Sistemi ibridi:** permettono sia thread livello utente che kernel.



(a) Pure user-level

(b) Pure kernel-level

(c) Combined



## Implementazioni ibride (cont.)

Vantaggi:

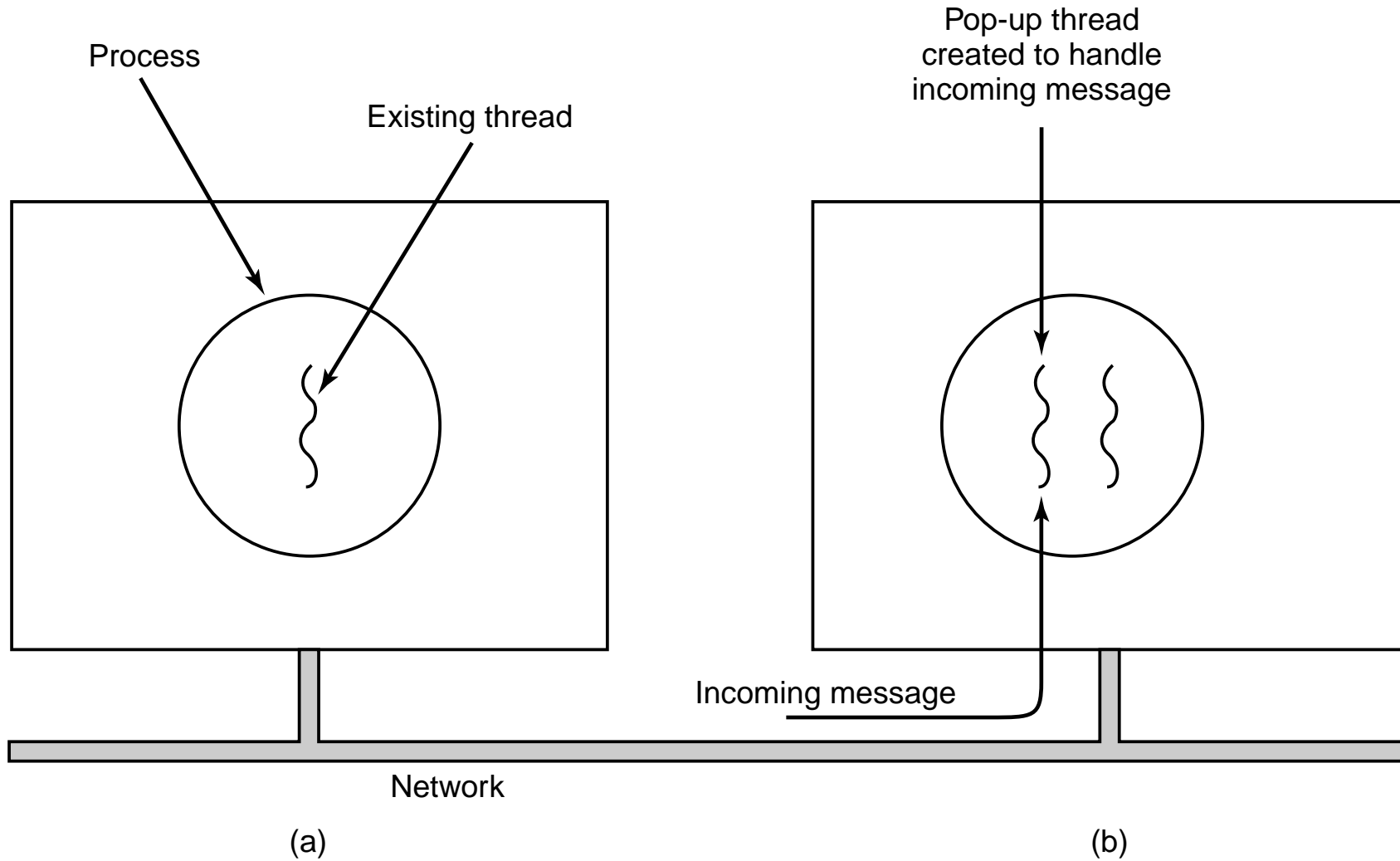
- tutti quelli dei ULT e KLT
- alta flessibilità: il programmatore può scegliere di volta in volta il tipo di thread che meglio si adatta

Svantaggio: portabilità

Es: Solaris 2 (thread/pthread e LWP), Linux (pthread e cloni), Mac OS X, Windows NT, ...

# Thread pop-up

- I thread *pop-up* sono thread creati in modo asincrono da eventi esterni.



(a) prima; (b) dopo aver ricevuto un messaggio esterno da gestire

- Molto utili in contesti distribuiti, e per servizio a eventi esterni
- Bassi tempi di latenza (creazione rapida)
- Complicazioni: dove eseguirli?
  - in user space: safe, ma in quale processo? uno nuovo? crearlo costa...
  - in kernel space: veloce, semplice, ma delicato (thread bacati possono fare grossi danni)
- Implementato in Solaris

# Processi e Thread di Windows 2000

Nel gergo Windows:

**Job:** collezione di processi che condividono quota e limiti

**Processo:** Dominio di allocazione risorse (ID di processo, token di accesso, handle per gli oggetti che usa). Creato con `CreateProcess` con un thread, poi ne può allocare altri.

**Thread:** entità schedulata dal kernel. Alterna il modo user e modo kernel. Doppio stack. Creato con `CreateThread`.

**Fibra (thread leggero):** thread a livello utente. Invisibili al kernel.

# Job, processi e thread in Windows 2000

