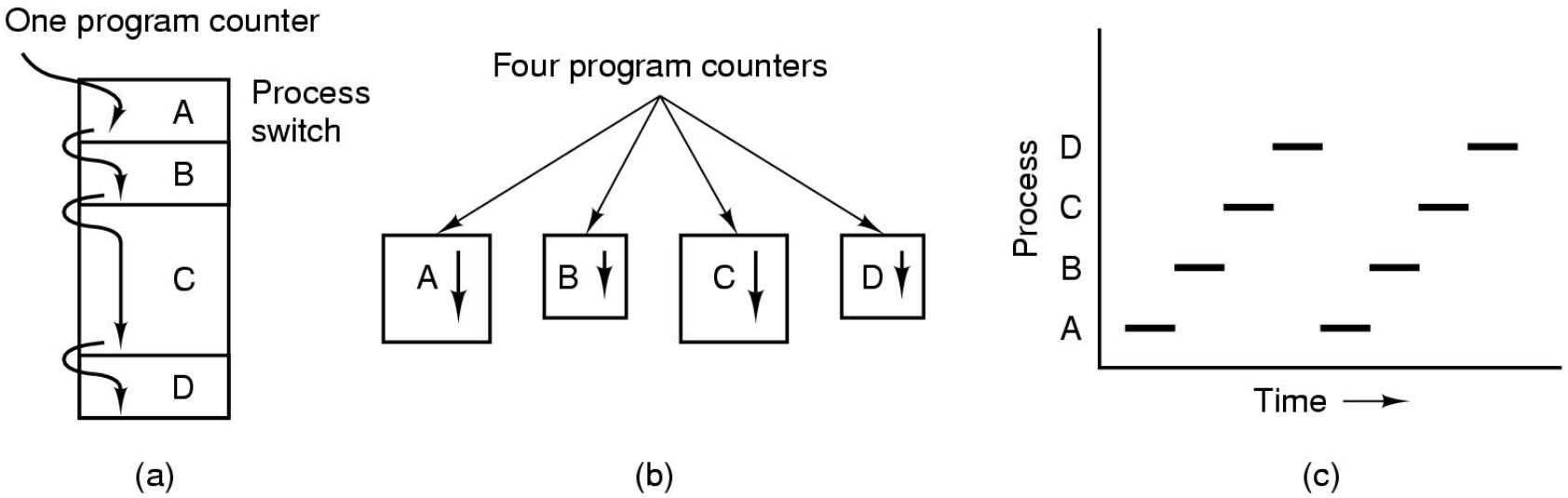
Processi e Thread

- Processi
- Thread
- Meccanismi di comunicazione fra processi (IPC)
- Problemi classici di IPC
- Scheduling
- Processi e thread in Unix
- Processi e thread in Windows

Processi

Il Modello a Processi



- *processo* : programma in esecuzione completo del suo *stato* (spazio di indirizzamento, registri, file aperti…)
- Modello concettuale: processi sequenziali indipendenti
- Solo un processo attivo ad ogni istante

Creazione di un processo

Principali eventi che causano la creazione di processi:

- inizializzazione del sistema
- esecuzione di una chiamata di sistema di creazione
- richiesta di creazione di un nuovo processo da parte dell'utente interattivo (es. invocazione di un comando, doppio click su un'icona)
- inizio di un job batch

ogni creazione di processo corrisponde a una chiamata di sistema

Creazione di un processo

- Unix (posix)
  - padre *fork*
  - figlio *exeve*
- *Windows (Win 32 API)*
  - *CreateProcess*

Terminazione di un processo

Condizioni tipiche di terminazione

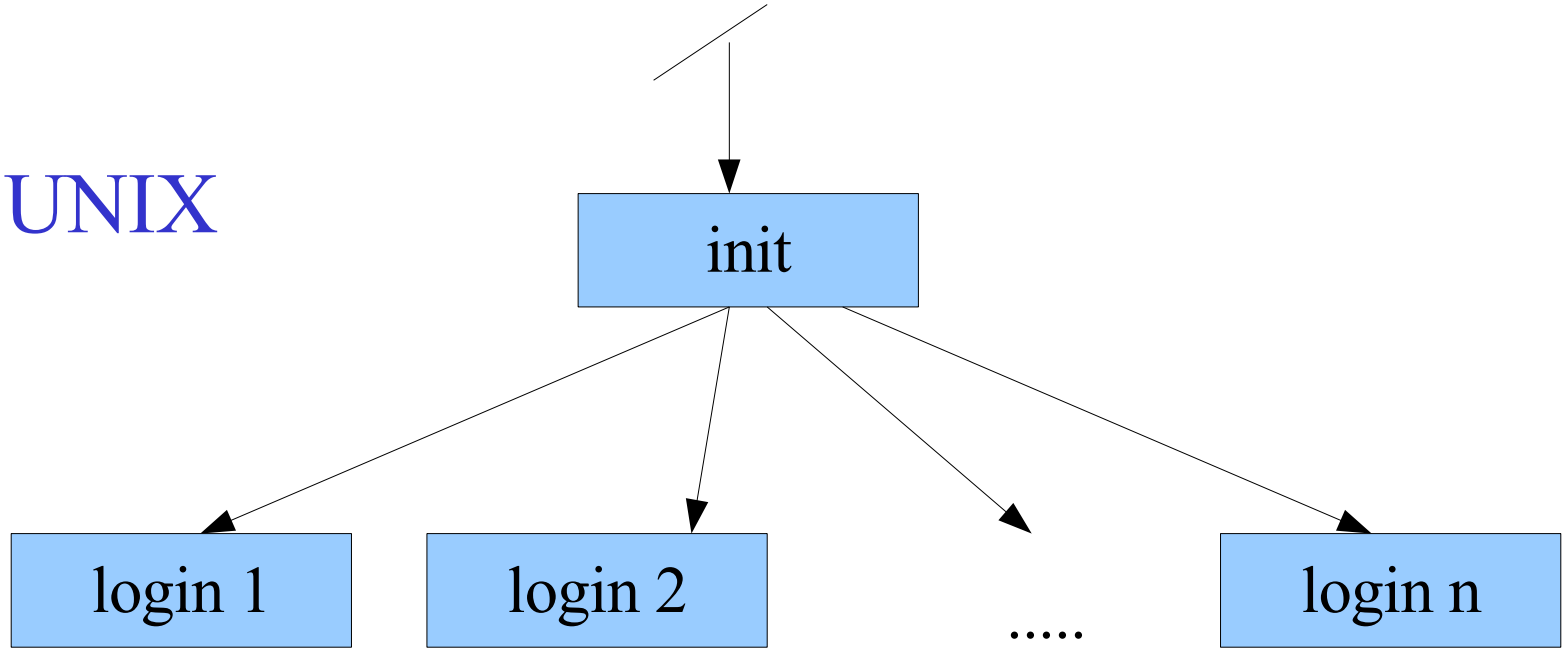
2. uscita normale (volontaria)
   - *exit* (Unix Posix)
   - *ExitProcess* (WIN 32)
3. uscita in presenza di errore (volontaria)
   - cc foo.c
4. fatal error (involontaria)
5. terminato da un altro processo (inv.)
   - *kill()* (UNIX Posix)
   - *TerminateProcess()* (Win32)

Gerarchie di Processi

- Un processo crea dei processi figli che a loro volta possono creare altri processi
- quello che si ottiene è una *gerarchia di processi*
  - UNIX inserisce tutti i processi di una gerarchia in un gruppo a parte (*process group*)
- Windows non ha il concetto di gerarchia di processi

Gerarchie di Processi

UNIX



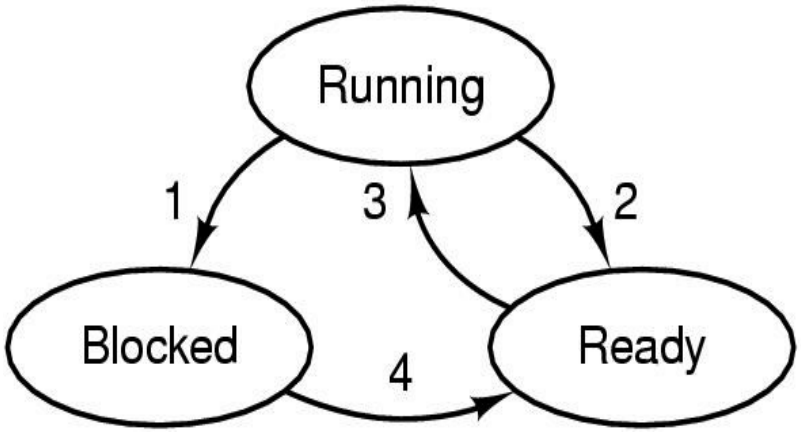
1 per terminale

pipe

> cat file1 file 2 | grep spezia

pipe

Stati di un processo (1)



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

- Possibili stati di un processo
  - in esecuzione (*running*), in attesa (*blocked*), pronto (*ready*)
- Transizioni di stato

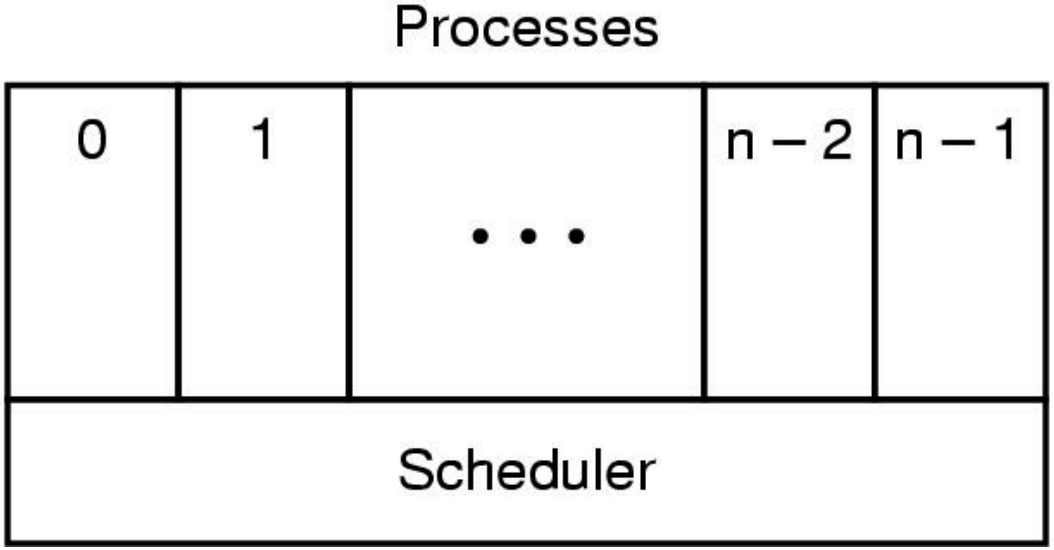
  (1) il processo si blocca in attesa di un'operazione di I/O

  (2) lo scheduler decide di mandare in esecuzione un altro processo

  (3) lo scheduler decide di mandare in esecuzione questo processo

  (4) l'operazione di I/O è terminata ed il processo può continuare

Stati di un processo (2)



- Livello più basso di un OS a processi (*microkernel*)
  - gestisce interrupt, scheduling

Implementazione di processi (1)

Tabella dei processi:

- un array di strutture (record)
- una struttura per ogni processo (PCB, *process control block*)
- contiene tutte le informazioni sullo stato del processo diverse dal suo spazio di indirizzamento
  - PC, SP, PCW, registri generali
  - stato (pronto, bloccato …)
  - informazioni relative ai file
  - informazioni relative alla memoria occupata dal processo
  - altre informazioni dipendento dal particolare SO

Implementazione di processi (2)

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		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		

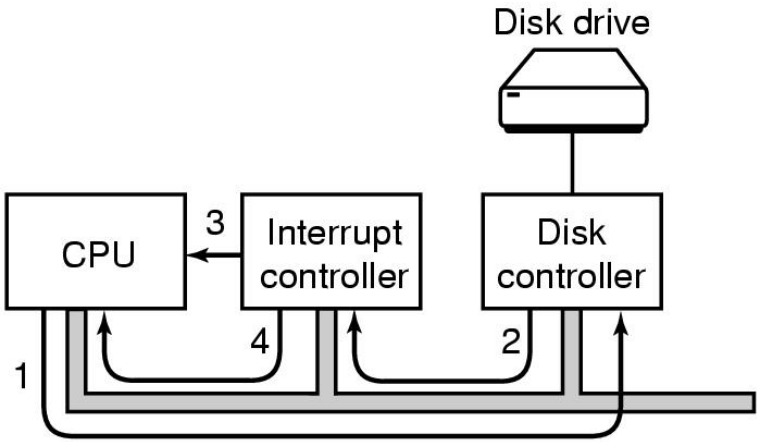
Tipici campi di un descrittore nella tebella dei processi

Implementazione dei processi (3)

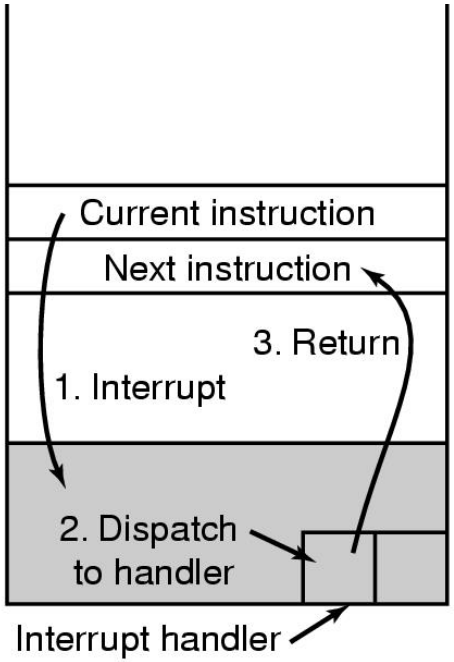
1. Salvataggio PC, PSW e qualche RG sulla pila (hw)
2. L'interrupt vector viene caricato nel PC (hw)
3. Salvataggio delle info sullo stack e di tutti i registri nella tabella dei processi (assembler)
4. L'SP punta a una nuova pila (assembler)
5. Esecuzione del gestore della interruzioni (C)
6. Esecuzione dello scheduler per decidere il nuovo processo da eseguire (C)
7. Caricamento dei registri e di tutte le informazioni relative al nuovo processo da eseguire (assembler)

Schema di cosa accade nei livelli bassi dell'OS quando viene rilevata una interruzione

Implementazione di processi (4)



(a)



(b)

Gestione di una interruzione

Thread of Execution
Filo di Esecuzione

Thread = 'light weight process'

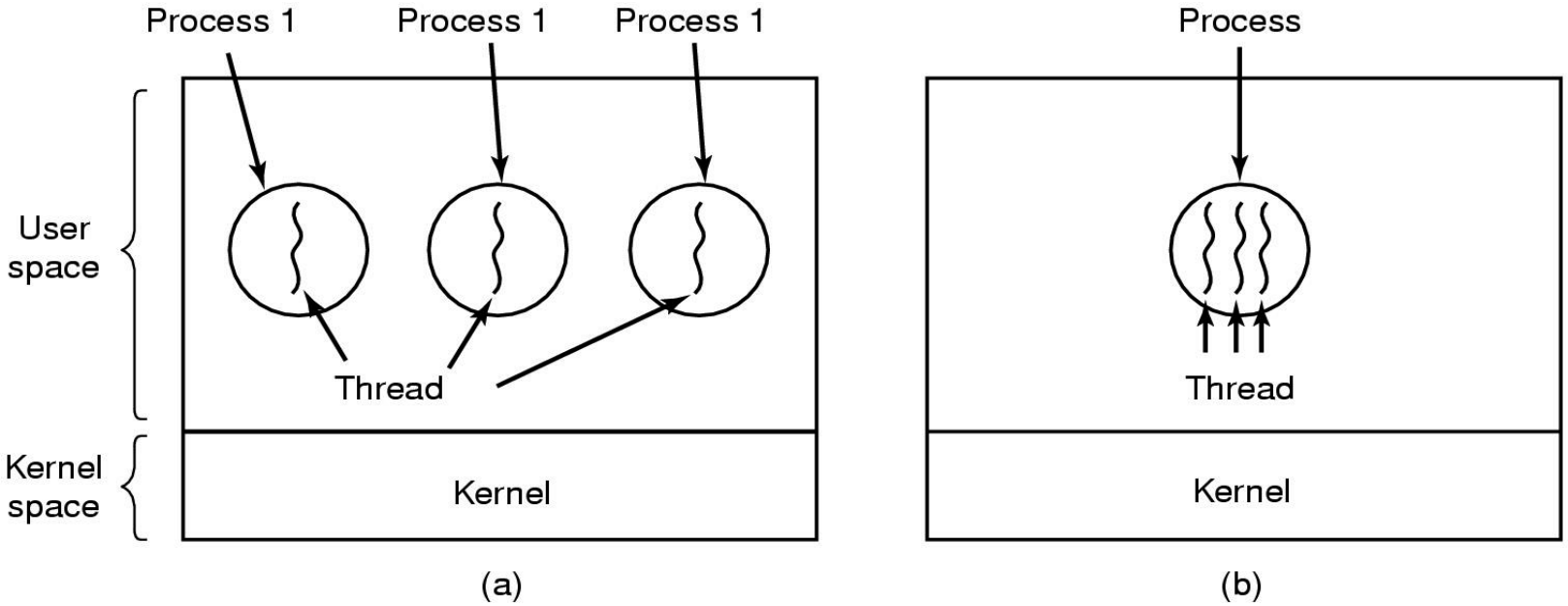
processo leggero, poco costoso

Thread

flusso di controllo all'interno di uno spazio di indirizzamento

Thread

Il Modello a Thread (1)



I 3 processi a singolo thread competono per:
- Memoria fisica
- dischi, stampanti
- altre risorse

I 3 thread dello stesso processo condividono:
- spazio di indirizzamento
- file aperti
- altre risorse

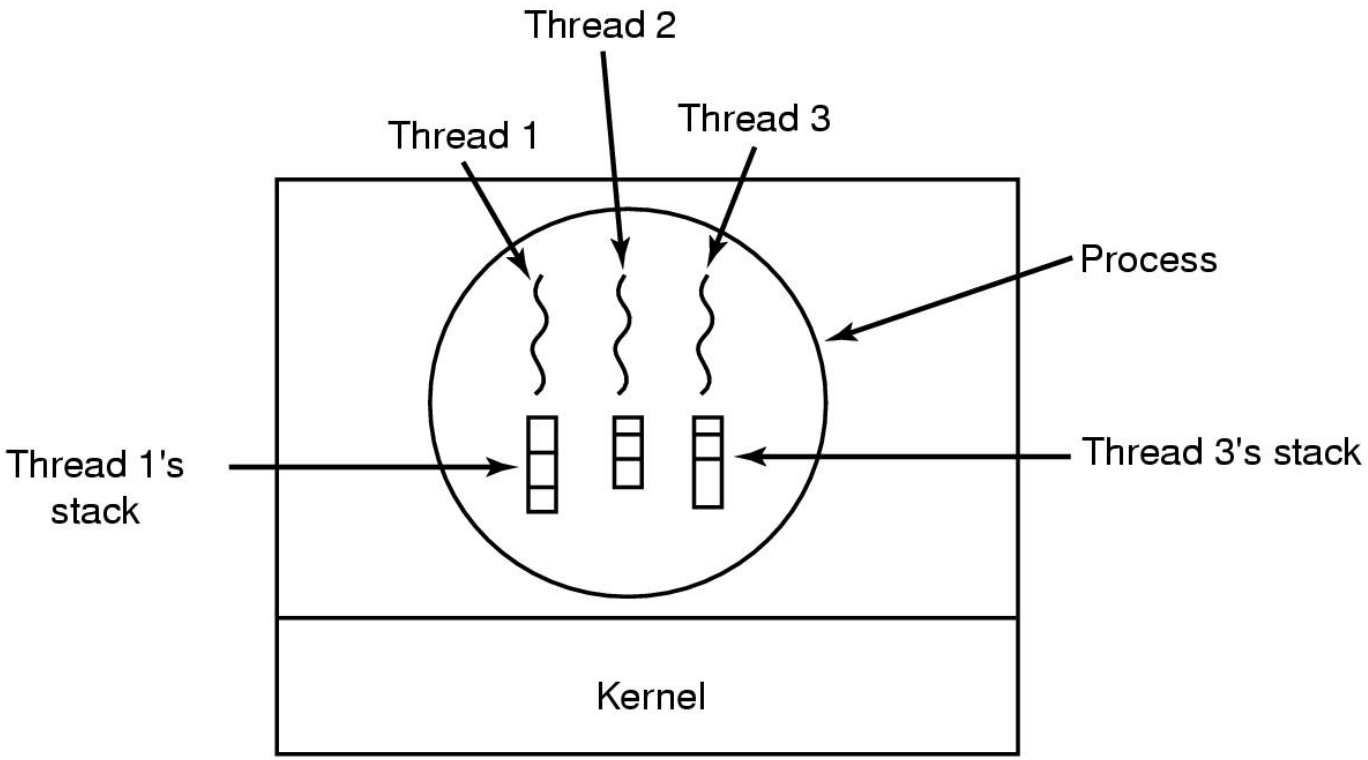
Il Modello a Thread (2)

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	

- Oggetti condivisi da tutti i thread dello stesso processo
- Oggetti privati di un singolo thread

Il Modello a Thread (3)

thread_create
thread_exit
thread_wait
thread_yield



Ogni thread ha il suo stack privato

Perche' usare i thread ?

- processi paralleli che condividono lo spazio di indirizzamento dei dati
- la creazione di un thread e' “100” volte piu' veloce della creazione di un processo
- thread con calcolo intensivo

  thread con molto i/o
- sono utili su multi processori

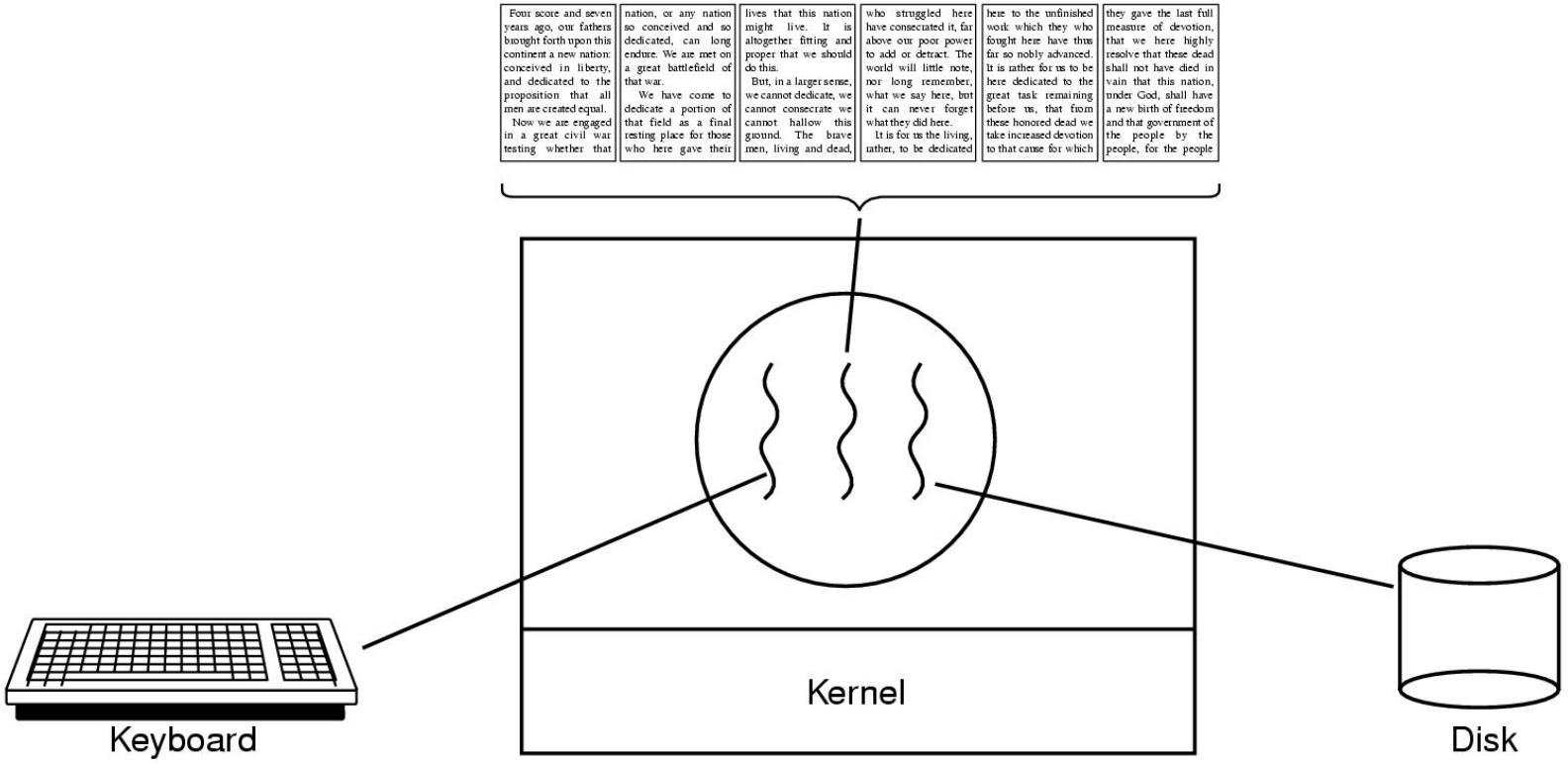
Uso dei Thread (1)

Applicazioni che :

- possono essere suddivise in più flussi di controllo
- interagiscono molto strettamente

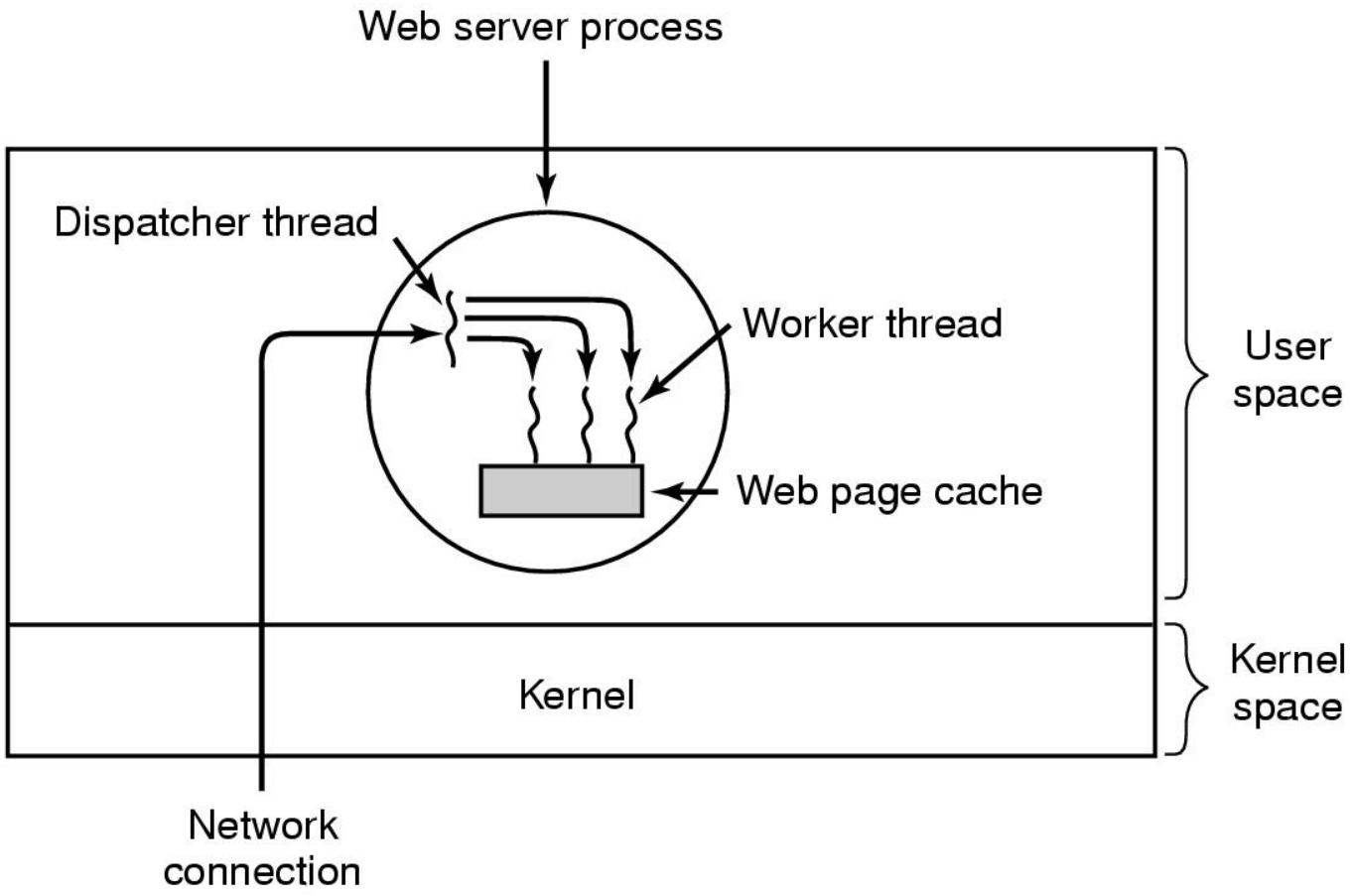
la condivisione dello spazio di indirizzamento e delle altre risorse permette di interagire senza pagare copie e cambi di contesto

Uso dei Thread (2)



Un word processor con 3 thread

Uso dei Thread (3)



Un Web Server con più thread (*multithreaded*)

Uso dei Thread (4)

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

(a)

```
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)

- Possibile struttura del codice del web server
  (a) dispatcher thread
  (b) worker thread

Implementazione dei Thread

I thread hanno gli stessi stati dei processi

Possono essere realizzati :

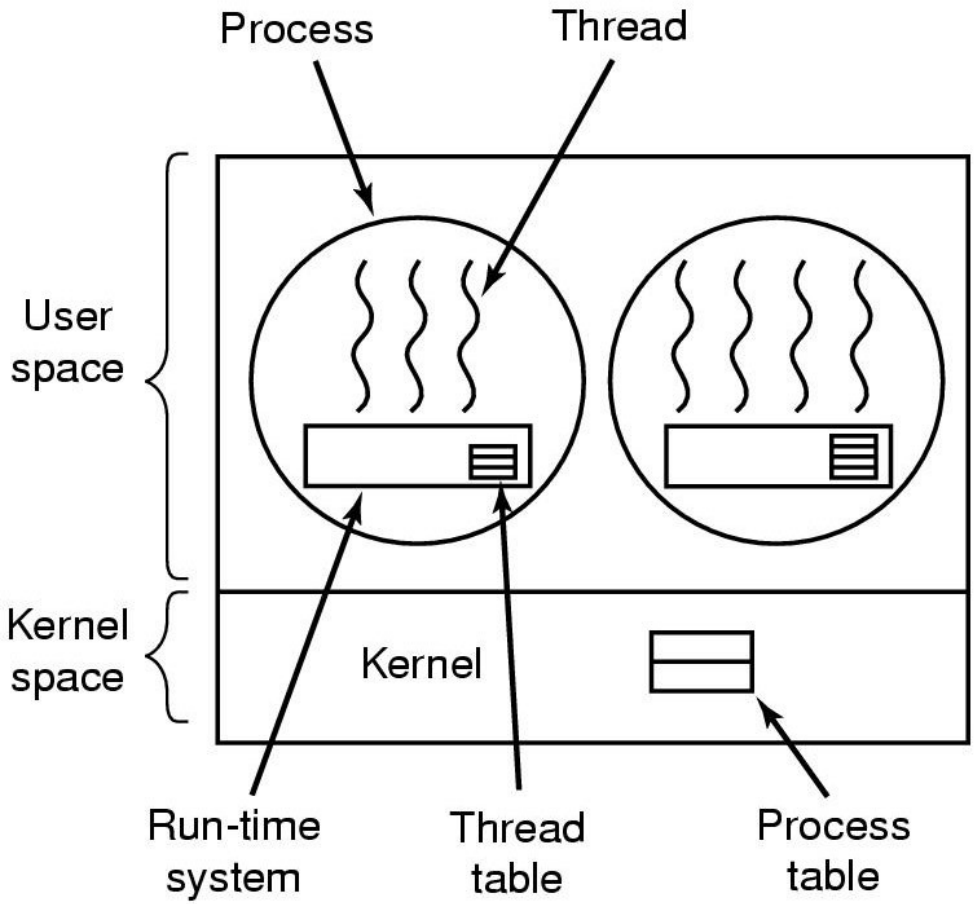
- da librerie che girano interamente in modo utente (*user-level* thread)
- nel SO (*kernel-level* thread)

Implementazione dei Thread in spazio utente (*user–level* thread) (1)

- Realizzati da una librerie di procedure che girano in modo utente
  - thread_create(), thread _exit(), thread_wait()...
- Ogni processo ha una *thread table* gestita dal run time support della libreria
- I thread devono rilasciare esplicitamente la CPU per permettere allo scheduler di esuguire un altro thread
  - thread _yeld ()
- Problema : che accade quando una system call si blocca?

Implementazione dei Thread in spazio utente (*user–level* thread) (2)

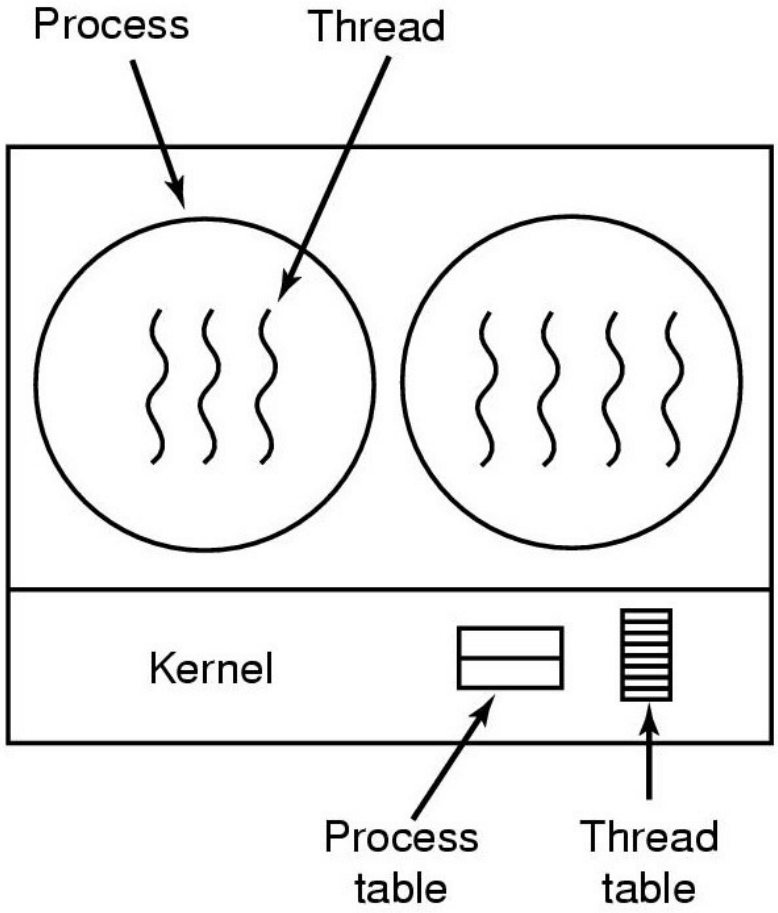
thread_create
thread_exit
thread_wait
thread_yield



Implementazione dei thread nel kernel (*kernel–level* thread) (1)

- Thread table unica (nel kernel)
- Le primitive che lavorano sui thread sono system call
  - thread_create(), thread _exit(), thread_wait()...
- Non è necessario che un thread rilasci esplicitamente la CPU
- Le system call possono bloccarsi senza bloccare tutti i thread di quel processo

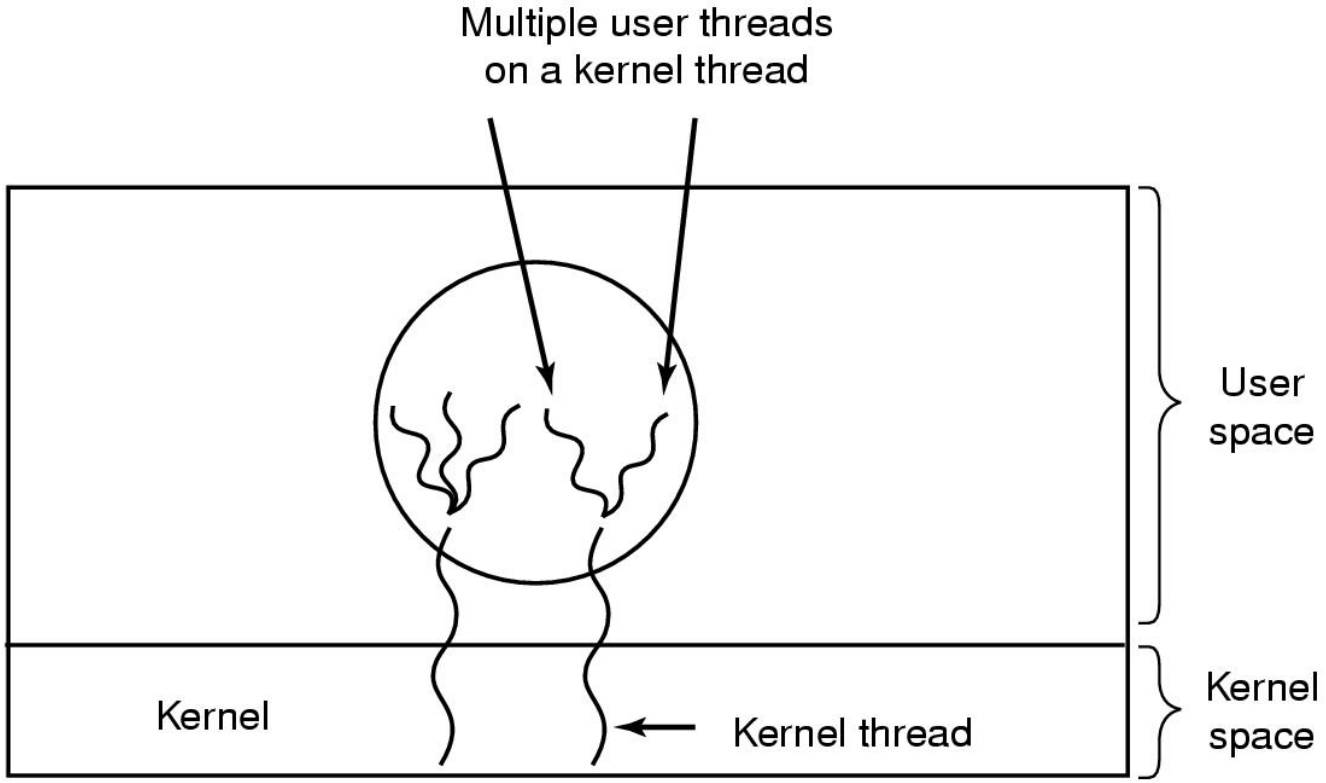
Implementazione dei thread nel kernel (*kernel–level* thread) (2)



User–level thread vs *kernel-level* thread

- Thread switch molto veloce
- Scheduling “personalizzato”, dipendente dall’applicazione
- Eseguibili su un SO che supporta solo i processi
- Gestione problematica delle system call bloccanti

Implementazioni ibride



Più *user-level* thread vengono eseguiti dallo stesso *kernel-level* thread