PROCESSI E THREAD

Danilo Croce

Ottobre 2024

PROCESSI

PROCESSI E THREAD

- Introduzione ai processi
  - Il modello di processo
  - Gestione dei processi
  - Stati di un processo
  - I thread
  - Gestione dei segnali
- IPC: Inter-Process Communication (Comunicazione tra processi)
  - Meccanismi IPC
  - Problemi classici di IPC
- Scheduling

PROCESSI E THREAD

- **Introduzione ai processi**
  - **Il modello di processo**
  - **Gestione dei processi**
  - **Stati di un processo**
  - **I thread**
  - **Gestione dei segnali**
- IPC: Inter-Process Communication (Comunicazione tra processi)
  - Meccanismi IPC
  - Problemi classici di IPC
- Scheduling

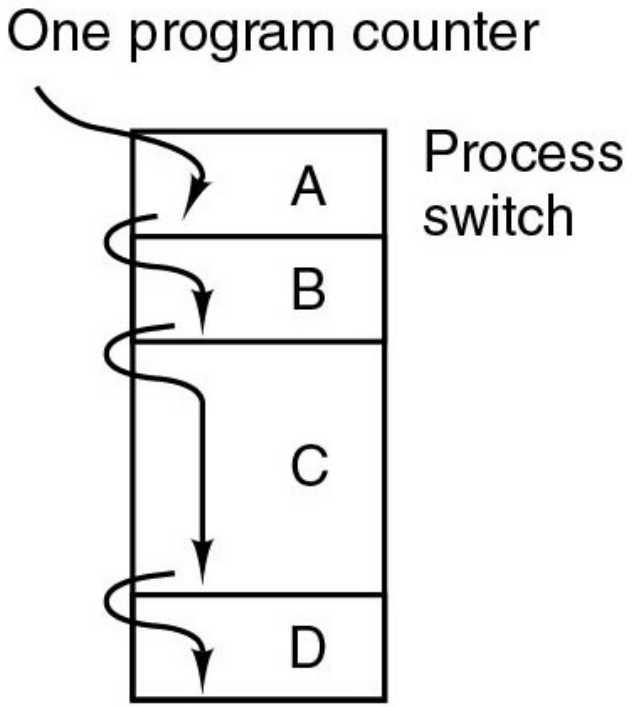
IL MODELLO DI UN PROCESSO

Processo = Programma in esecuzione

- Quanti processi per ogni programma?
- Un'**astrazione** fondamentale del sistema operativo
- Consente al sistema operativo di semplificare:
  - **Allocazione** delle risorse
  - Accounting (o "Contabilizzazione") delle risorse
  - Limitazione delle risorse
- Il **sistema operativo mantiene informazioni** sulle risorse e sullo stato interno di ogni singolo processo del sistema.

IL MODELLO DI UN PROCESSO (1)

- Contatore di programma singolo
- Ogni processo in un'unica posizione
- La CPU passa da un processo all'altro

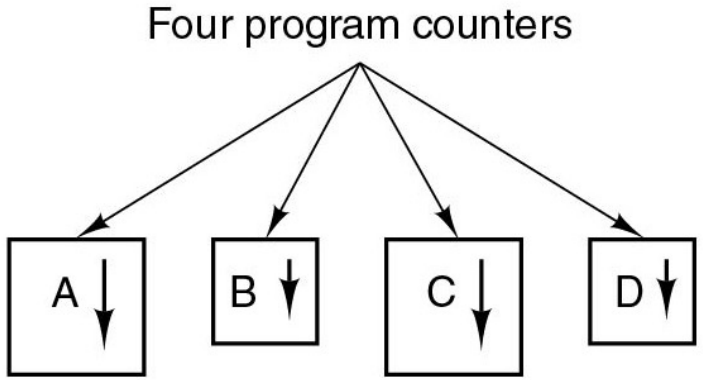


Schema di Multiprogrammazione di quattro programmi.

IL MODELLO DI UN PROCESSO (2)

- Ogni processo ha un **proprio flusso di controllo**
  - proprio contatore logico di programma

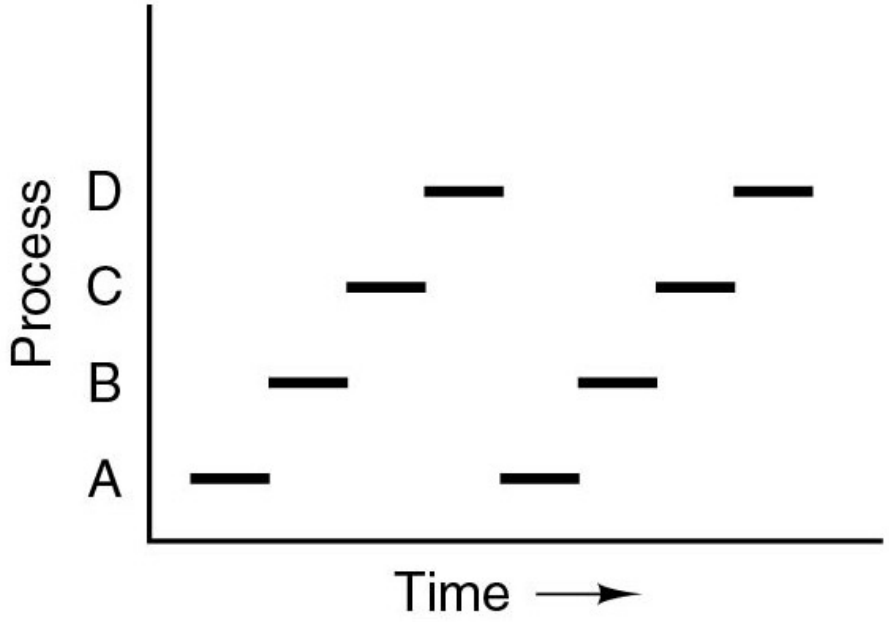
- Ogni volta che si passa da un processo all'altro, si salva il contatore di programma del primo processo e si ripristina il contatore di programma del secondo.



Modello concettuale di quattro processi indipendenti e sequenziali.

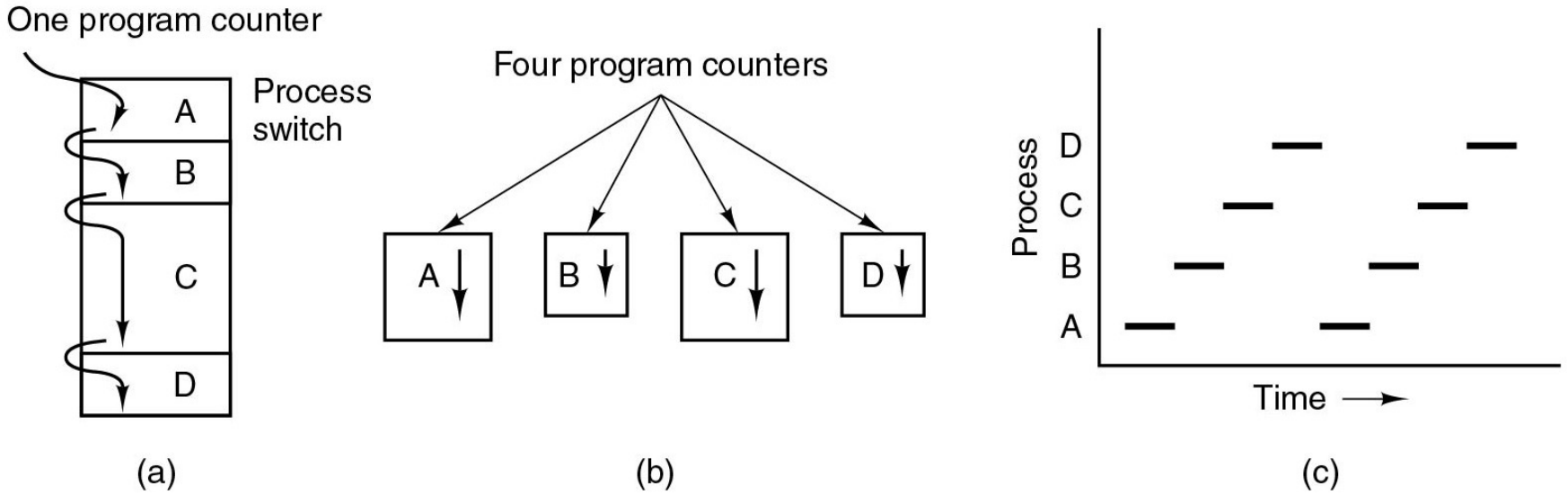
IL MODELLO DI UN PROCESSO (3)

- Tutti i processi progrediscono, ma solo uno è attivo in un dato momento.



È attivo solo un programma alla volta.

PROCESSI CONCORRENTI



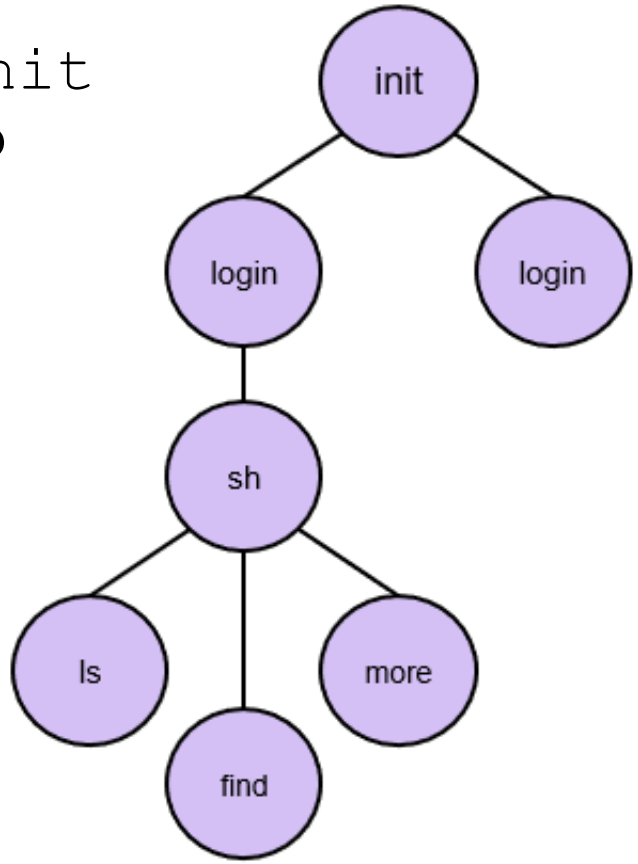
- In **linea di principio**, i processi multipli **sono reciprocamente indipendenti**
- Hanno bisogno di mezzi espliciti per interagire tra loro
- La CPU può essere assegnata a turno a diversi processi
- Il sistema operativo normalmente **non offre garanzie di tempistica o di ordine**

GERARCHIE DI PROCESSI

Il sistema operativo in genere crea solo un processo di `init`
- nei moderni sistemi init avvia `kthreadd` un processo per la gestione dei thread (see later)

I Sottoprocessi sono **creati in modo indipendente**:
- Un processo padre può creare un processo figlio
  - Ne consegue una struttura ad albero e gruppi di processi

- Ad esempio, la shell esegue i comandi:
  - `$ find /tmp & > t.log &`
  - `$ ls | more`



CREAZIONE DI PROCESSO

Quattro eventi principali che causano la creazione di processi:

1. **Inizializzazione** del sistema
2. Esecuzione di una **chiamata di sistema per la creazione** di un processo da parte di un processo in esecuzione (`fork()`)
3. **Richiesta dell'utente** di creare un nuovo processo
   - Esempio tramite bash
4. **Avvio di un lavoro** in modalità batch (o da bash ;-))

TERMINE DI UN PROCESSO

Condizioni tipiche che terminano un processo:

1. Uscita normale (volontaria).
2. Uscita a causa di un errore (volontaria).
3. Errore “fatale” (involontario).
4. Ucciso da un altro processo (involontario).

PROCESS MANAGEMENT

- `fork`: crea un nuovo processo
  - Il figlio è un clone "privato" del genitore
  - Condivide alcune risorse con il genitore
- `exec`: esegue di un nuovo processo
  - Utilizzato in combinazione con fork
- `exit`: causa la terminazione volontaria del processo
  - Lo "stato di uscita" viene restituito al processo "genitore"
- `kill`: invia un segnale a un processo (o a un gruppo)
  - Può causare la terminazione involontaria di un processo

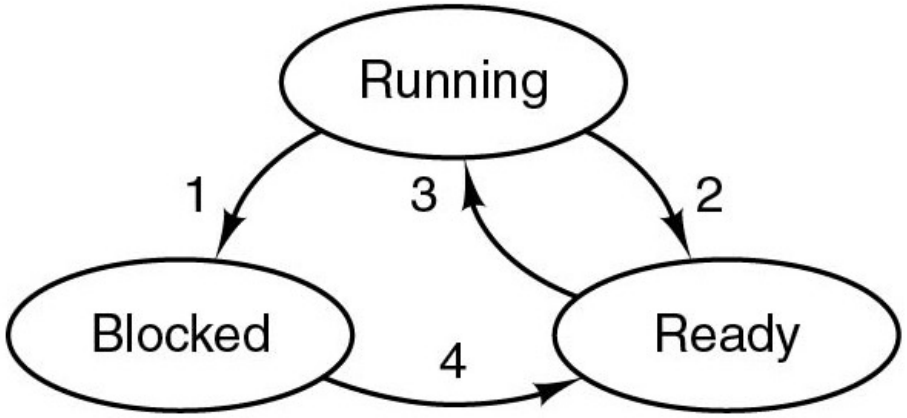
GLI STATI DI UN PROCESSO (1)

Tre stati in cui può trovarsi un processo:

1. **Running**/In esecuzione (sta effettivamente utilizzando la CPU in quel momento).
2. **Ready**/Pronto (eseguibile; temporaneamente fermato per consentire l'esecuzione di un altro processo).
3. **Blocked**/Bloccato (non può essere eseguito fino a quando non si verifica un evento esterno).

GLI STATI DI UN PROCESSO (2)

- Il sistema operativo alloca le risorse (ad esempio, la CPU) ai processi.
- Per allocare la CPU, il sistema operativo deve tenere traccia degli stati dei processi:
  - **Running/Blocked/Ready**
- Lo scheduler (de)assegna la CPU (vedi transizioni 2&3)



1. Il processo è in attesa di input
2. Lo scheduler sceglie un altro processo
3. Lo schedulatore sceglie questo processo
4. L'input diventa disponibile

INFORMAZIONI ASSOCIATE A UN PROCESSO

- ID (PID), Utente (UID), Gruppo (GID)
- Spazio degli indirizzi di memoria
- Registri hardware (ad esempio, il Program Counter)
- File aperti
- **Segnali** (Signal)
- **Interrupt**

- Queste informazioni sono memorizzate nella tabella dei processi del sistema operativo.

SIGNAL(S) VS INTERRUPT(S)

- "Signals" e "Interrupts" sono meccanismi utilizzati nei sistemi operativi e nelle applicazioni per gestire eventi asincroni

- **Interrupts**:
  - **Origine**: Dispositivi **hardware** (es. tastiera, disco rigido).
  - **Gestione**: Tramite routine di servizio di interrupt (ISR).
  - **Uso**: Comunicazione tra hardware e software; risposta pronta agli eventi hardware.
  - **Asincronia**: Si verificano in modo asincrono; gestiti immediatamente.
- **Signals**:
  - Origine: Eventi **software**; generati da un processo o dal SO.
  - **Gestione**: Gestori di segnali personalizzati o comportamento predefinito.
  - **Uso**: Gestione condizioni eccezionali nelle applicazioni.
  - **Asincronia**: Inviati asincronamente; possono essere gestiti in modo sincrono.

INTERRUPTS

- **Idea**: per deallocare la CPU a favore dello scheduler, ci si affida al supporto per la gestione degli interrupt fornito dall'hardware.
  - **Permette allo scheduler di ottenere periodicamente il controllo**, cioè …
  - … ogni volta che l'hardware genera un interrupt.

- **Interrupt vector:**
  - Associato a ciascun **dispositivo di I/O** e linea di interrupt
  - Parte della tabella dei descrittori di interrupt (IDT)
  - Contiene l'indirizzo iniziale di una procedura interna fornita dal sistema operativo
    - Gestore di Interrupt o interrupt handler che continua l'esecuzione

- Tipi di interruzione: sw, **dispositivo hw (async)**, eccezioni

IMPLEMENTAZIONE DEI PROCESSI

- Schema di ciò che fa il livello più basso del sistema operativo quando si verifica un'interruzione.

1. L'hardware impila il Program Counter e le altre informazioni del processo
2. L'hardware carica il nuovo contatore di programma dal vettore di interrupt.
3. La procedura in linguaggio assembly salva i registri.
4. La procedura in linguaggio assembly imposta un nuovo stack.
5. Il servizio di interrupt C viene eseguito (tipicamente legge e esegue il buffer dell'input).
6. Lo scheduler decide quale processo deve essere eseguito successivamente.
7. La procedura C ritorna al codice assembly.
8. La procedura in linguaggio assembly avvia il nuovo processo (corrente).

- **Ogni volta che si verifica un'interruzione, lo scheduler ottiene il controllo**
  => **agisce come mediatore**
- **Un processo non può cedere la CPU a un altro processo** (context switch) senza passare attraverso lo scheduler.

GESTIONE DEI SEGNALI (1 OF 2)

- **Tipi di Segnali**:
  - Hardware-induced (e.g., `SIGKILL`)
  - Software-induced (e.g., `SIGQUIT` or `SIGPIPE`)
- **Azioni possibili:**
  - `Term, Ign, Core, Stop, Cont`
  - Azione predefinita per ogni segnale, tipicamente sovrascrivibile
  - I segnali possono essere tipicamente bloccati e le azioni ritardate.
- **Gestione (catching) dei segnali:**
  - Il processo registra il gestore del segnale
  - Il sistema operativo invia il segnale e consente al processo di eseguire l'handler
  - Il contesto di esecuzione corrente deve essere salvato/ripristinato

GESTIRE IL SEGNALE INDOTTO DA CTRL+C

```
void signalHandler( int signum ) {
   printf ("Interrupt signal &d received\n", signum );
   // cleanup and terminate program
   exit(signum);
}
int main () {
   // register signal SIGINT and signal handler
   signal(SIGINT, signalHandler);
   while(1) {
      printf ("Going to sleep....\n");;
      sleep(1);
   }
   return 0;
}
```

GESTIONE DEI SEGNALI (2 OF 2)

- Il kernel invia un segnale
- Interrompe il codice in esecuzione
- Salva il contesto
- Esegue il codice di gestione del segnale
- Ripristina il contesto originale

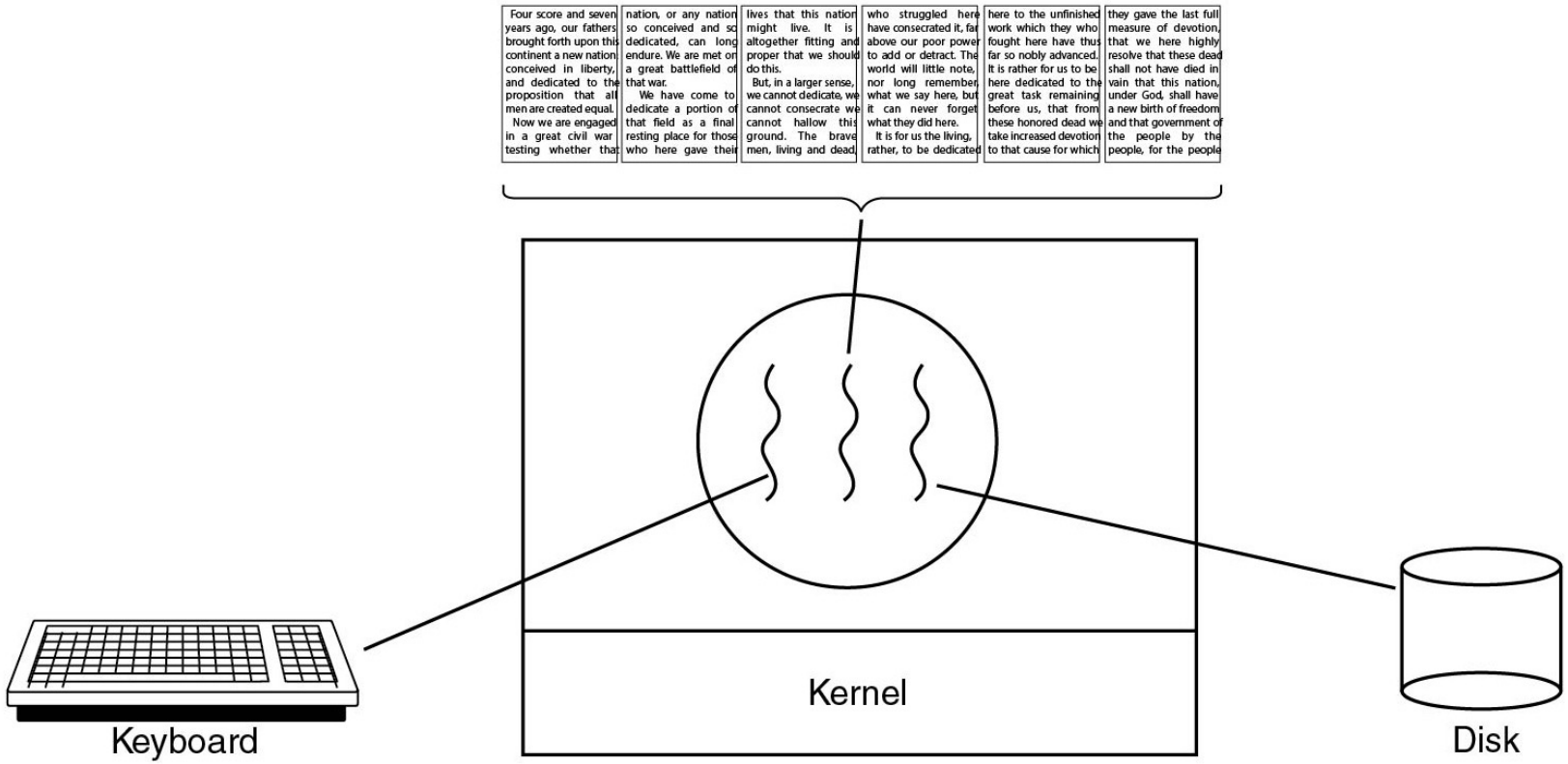
THREAD

THREAD

- Assunzione implicita finora:
  - 1 processo => 1 thread in esecuzione
- Multithreaded execution:
  - 1 processo => *N* thread in esecuzione

- Perché consentire più thread per processo?
  - **Lightweight processes** (Processi leggeri)
    - Consentire un parallelismo efficiente in termini di spazio e di tempo
  - **Una comunicazione e una sincronizzazione semplici**

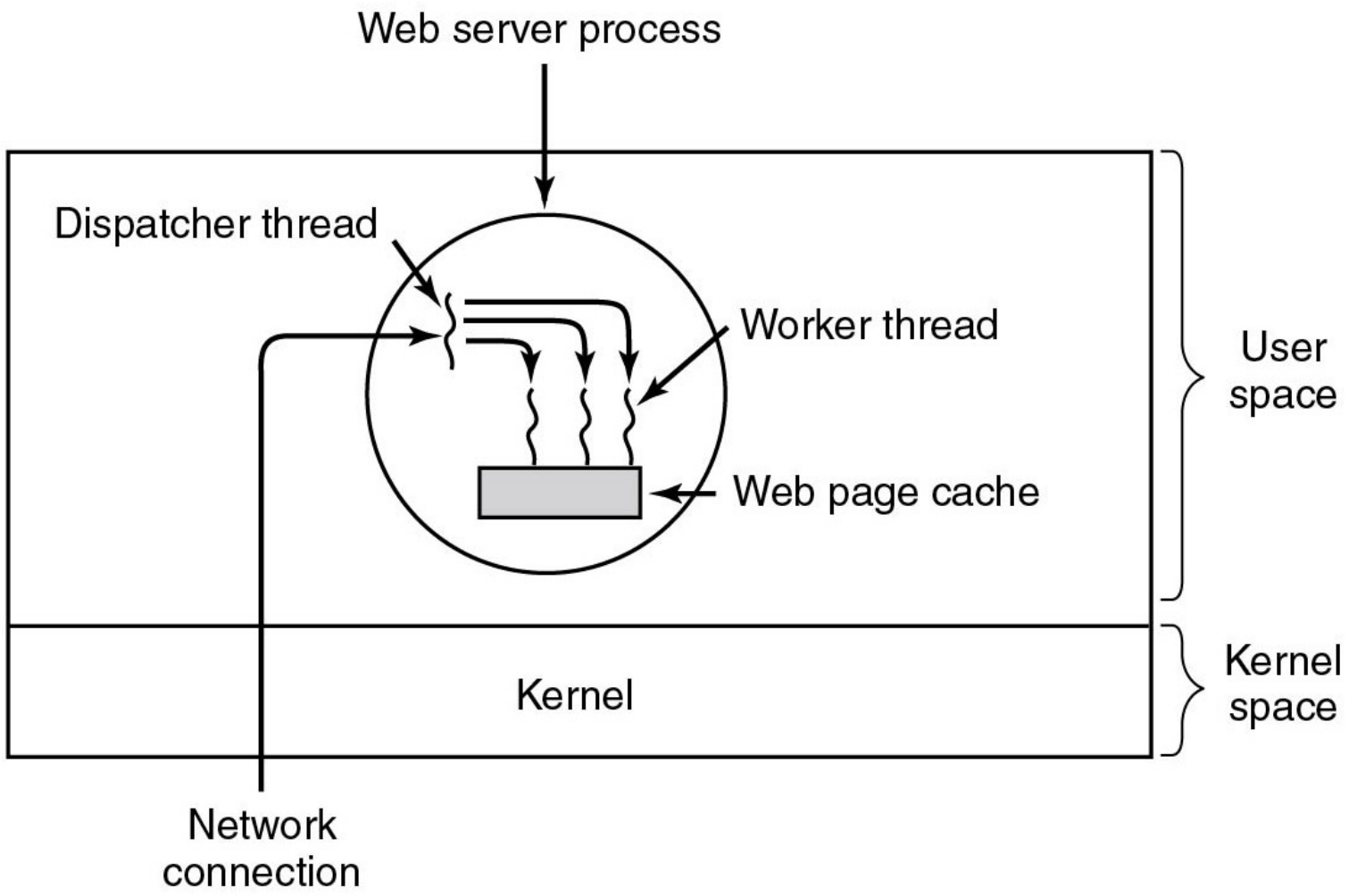
UTILIZZO DEI THREAD (1)



Un Word Processor con tre thread.

UTILIZZO DEI THREAD

A multithreaded Web server.



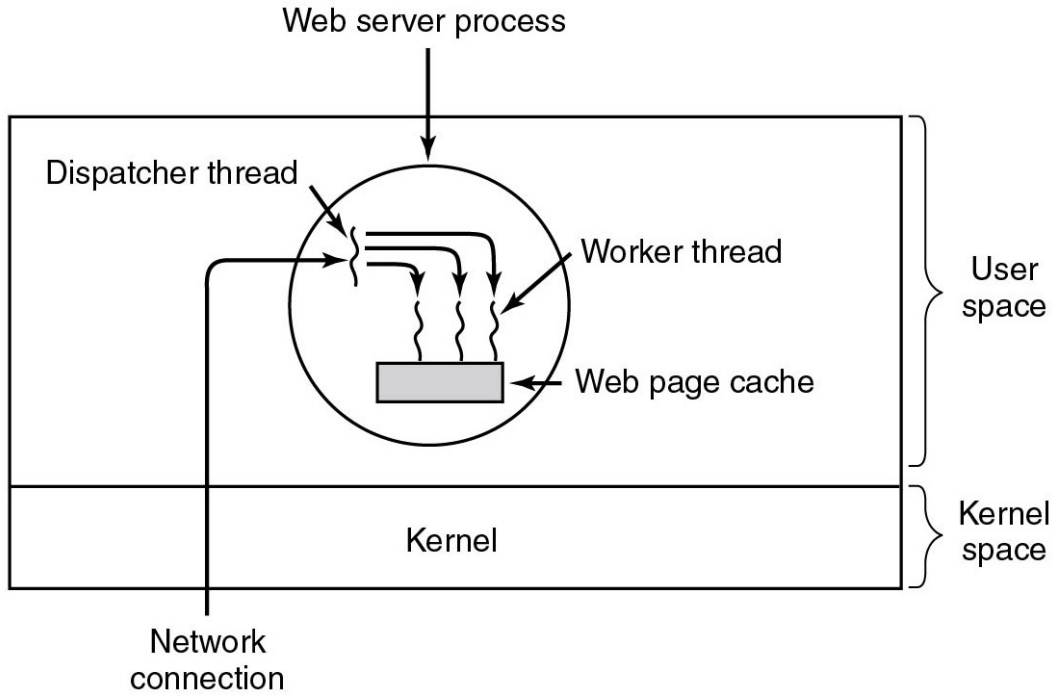
UTILIZZO DEI THREAD (2)

```
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)



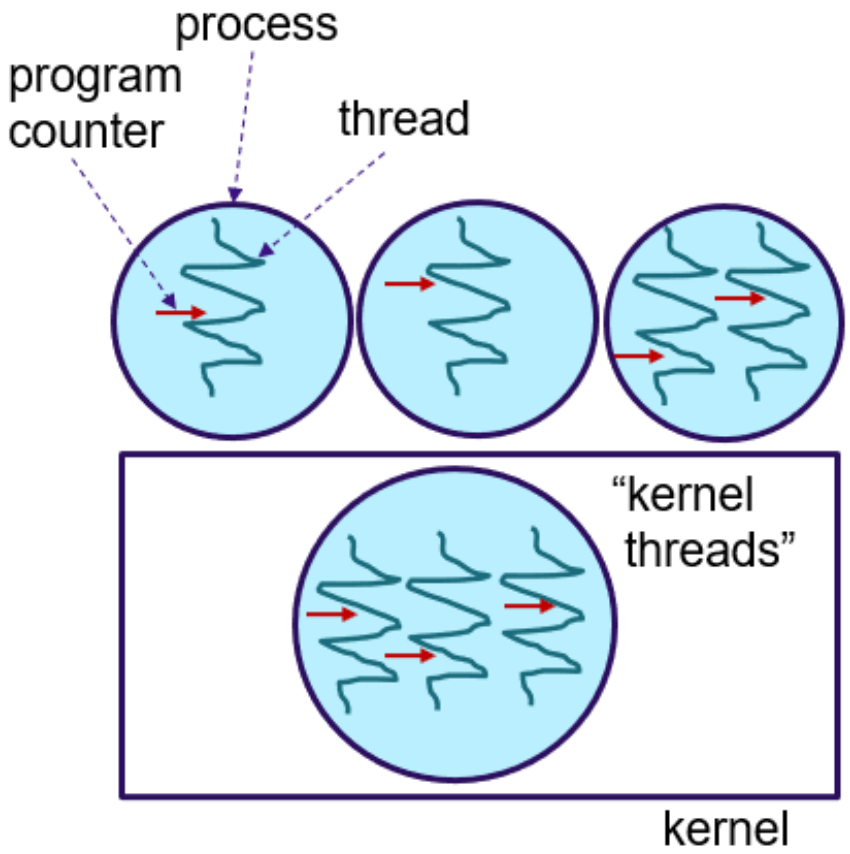
Schema di massima del codice della Figura

(a) Thread del dispatcher.

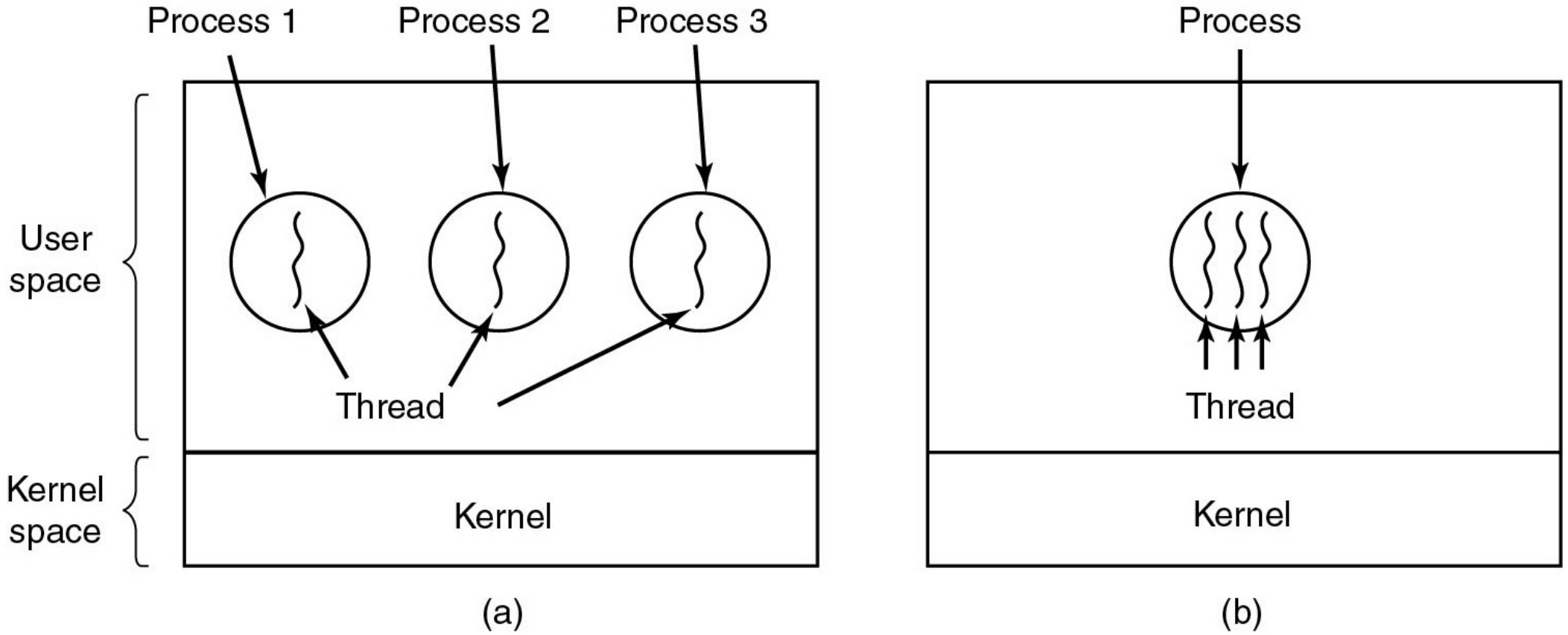
(b) Thread di lavoro.

THREAD E PROCESSI

- I thread **risiedono nello stesso spazio degli indirizzi di un singolo processo**.
- Tutti gli **scambi** di informazioni avvengono **tramite dati condivisi** tra i thread
  - I thread si sincronizzano tramite semplici primitive
- Ogni thread ha
  - il **proprio stack**
  - i **propri registri hardware**
  - il **proprio stato**.
- Tabella/interrupt dei thread:
  - una tabella/ interrupt di processo più leggera
- Ciascun **thread può chiamare qualsiasi chiamata** di sistema supportata dal sistema operativo **per conto del processo** a cui appartiene



IL MODELLO DI THREAD CLASSICO (1)



(a) Tre processi con un thread ciascuno. (b) Un processo con tre thread.

IL MODELLO DI THREAD CLASSICO (2)

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

elementi condivisi da tutti i thread di un processo.

elementi privati di ciascun thread.

I THREAD IN POSIX (1 OF 2)

Thread call	Description
`pthread_create`	Crea un nuovo thread
`pthread_exit`	Termina il thread chiamante
`pthread_join`	Attende l' "*uscita*" di uno specifico thread
`pthread_yield`	Rilascia la CPU per consentire l'esecuzione di un altro thread
`pthread_attr_init`	Crea e inizializza la struttura di attributi di un thread
`pthread_attr_destroy`	Rimuove la struttura di attributi di un thread

- Alcune chiamate di funzione di `Pthreads`.

PTHREADS

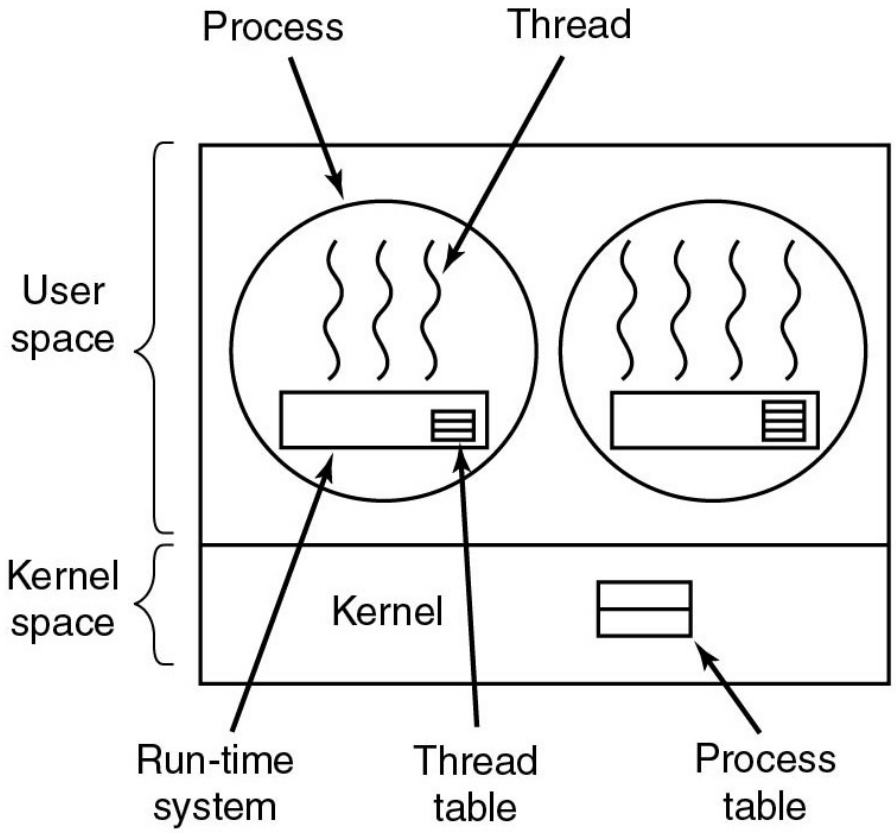
```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_THREADS  10
void * print_hello_world(void * tid)
{
  printf("Hello World. Greetings from thread %d\n", tid);
  pthread_exit(NULL);
}
int main(int argc, char * argv[])
{
  pthread_t threads[NUMBER_OF_THREADS];
  int status, i;
  for(i=0; i < NUMBER_OF_THREADS; i++) {
    status = pthread_create(&threads[i], NULL, print_hello_world, (void * )i);
    if (status != 0) {
      exit(-1);
    }
  }
  return 0;
}
```

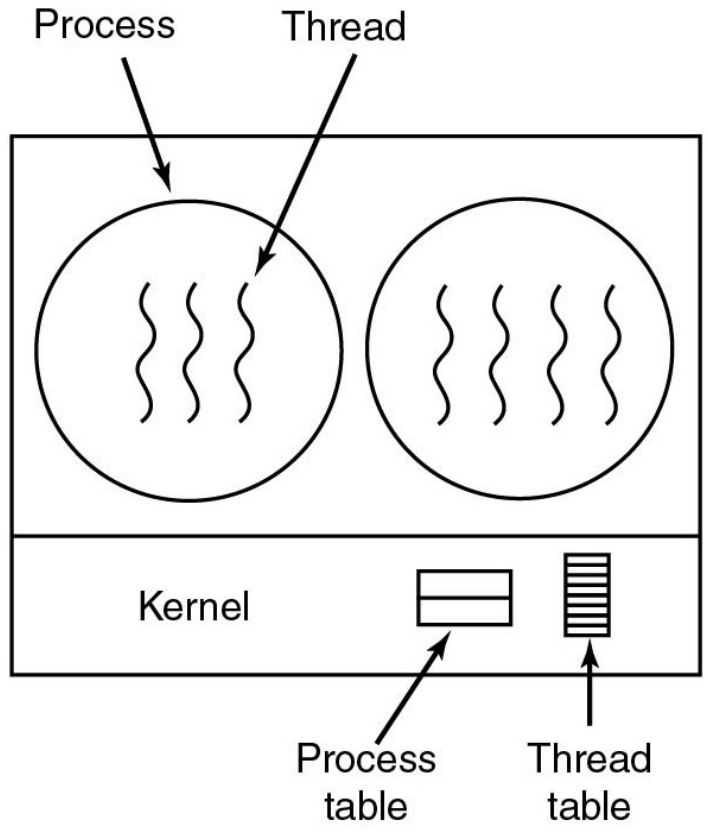
What will the output be?

IMPLEMENTAZIONE DEI THREAD NELLO SPAZIO UTENTE

Esistono tre luoghi di implementazione dei *thread*:

1. **nello spazio utente**
2. **nel kernel**
3. **un'implementazione ibrida.**





(a) Un package di thread a livello utente. (b) Un package di thread gestito dal kernel.

IMPLEMENTAZIONE DEI THREAD NELLO SPAZIO UTENTE PRO

- I **thread nello spazio utente** **sono gestiti dal kernel** come processi ordinari a singolo thread.
  - **possono essere eseguiti su sistemi operativi che non supportano** direttamente i thread.
  - sono **gestiti tramite una libreria**.
- Ogni processo che usa thread a livello utente **necessita di una propria tabella** dei thread per tracciare lo stato e altre proprietà dei suoi thread.
- L'interruzione e il cambio tra thread a livello utente **non richiedono un cambiamento di contesto** completo.
  - **No *trap*** ☺
  - **Sono molto più veloci** rispetto alle operazioni nel kernel
- Offrono l'abilità di **personalizzare l'algoritmo di scheduling** per ogni processo e una maggiore scalabilità.

IMPLEMENTAZIONE DEI THREAD NELLO SPAZIO UTENTE CONTRA

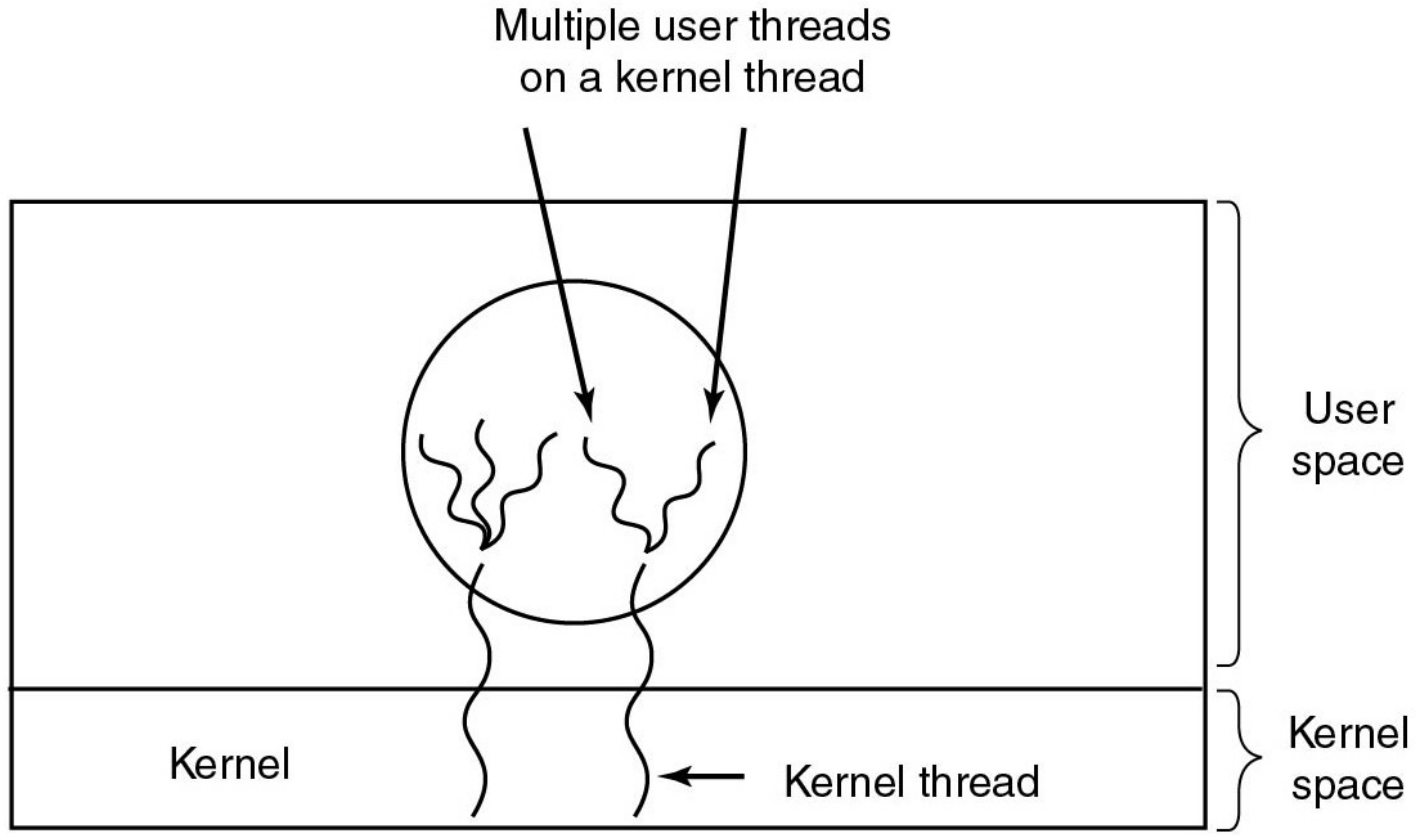
- Tuttavia, ci sono **problemi** con le chiamate di **sistema bloccanti**
  - se **un thread fa una chiamata che lo blocca, tutti gli altri thread nel processo vengono fermati**.
  - Gli **errori di pagina**, dove un programma accede a memoria non presente, **possono bloccare l'intero processo** quando sono causati da un thread a livello utente.
- I thread nello spazio utente **non hanno interrupt del clock**, rendendo impossibile uno scheduling di tipo round-robin (*see next lessons*).
- Sebbene i thread a livello utente siano più veloci e flessibili, **sono meno adatti per applicazioni in cui i thread si bloccano frequentemente**, come i web server multithread.
  - I thread a livello utente possono fermarsi completamente se un singolo thread effettua una chiamata di sistema bloccante, influenzando tutti gli altri thread nel processo.

IMPLEMENTAZIONE DEI THREAD NELLO SPAZIO KERNEL

- Il **kernel** che gestisce i thread **elimina la necessità di un sistema run-time** per processo.
  - La tabella dei thread del kernel conserva informazioni simili a quelle dei thread a livello utente.
- Le **chiamate che potrebbero bloccare un thread** vengono implementate come chiamate di sistema
  - **hanno costi più elevati** rispetto alle chiamate di procedura dei sistemi run-time
  - Se un thread si blocca, il kernel può eseguire un altro thread, sia dello stesso processo sia di un altro
- Alcuni sistemi "riciclano" i thread per ridurre i costi, invece che terminarli
- Se un thread **causa un errore di** pagina, il **kernel verifica la disponibilità di altri thread eseguibili** nel processo e può eseguire uno di essi.
- La programmazione con **thread richiede cautela per evitare errori.**

IMPLEMENTAZIONI IBRIDE

- Alcuni sistemi effettuno il ***multiplexing* dei thread utente sui thread del kernel.**
  - Combinano i vantaggi dei due approcci
- I **programmatori decidono quanti thread del kernel utilizzare** e quanti thread utente multiplexare,
  - Maggiore flessibilità.
- Il kernel è consapevole solo dei thread del kernel…
- … ma ogni thread del kernel può gestire più thread a livello utente



Multiplexing dei thread a livello utente su quelli a livello kernel.

THREADS: PROBLEMI APERTI

- Molte **procedure di libreria possono causare conflitti** se un thread sovrascrive dati cruciali per un altro, *esempio*:
  - l'invio di un messaggio sulla rete potrebbe essere programmato assemblando il messaggio in un buffer fisso nella libreria e poi eseguendo una trap nel kernel per spedirlo
  - che cosa accade se un thread ha preparato il suo messaggio nel buffer e poi un interrupt del clock forza uno scambio con un secondo thread, che sovrascrive immediatamente il buffer con un suo messaggio?
- L'implementazione di wrappers (impostare un bit per segnalare che la libreria è in uso) può evitare conflitti, ma limita il parallelismo.
- La gestione dei segnali è complicate
  - alcuni sono specifici per un thread, mentre altri no.
  - decidere chi deve gestire questi segnali e come gestire conflitti tra thread può essere sfidante.