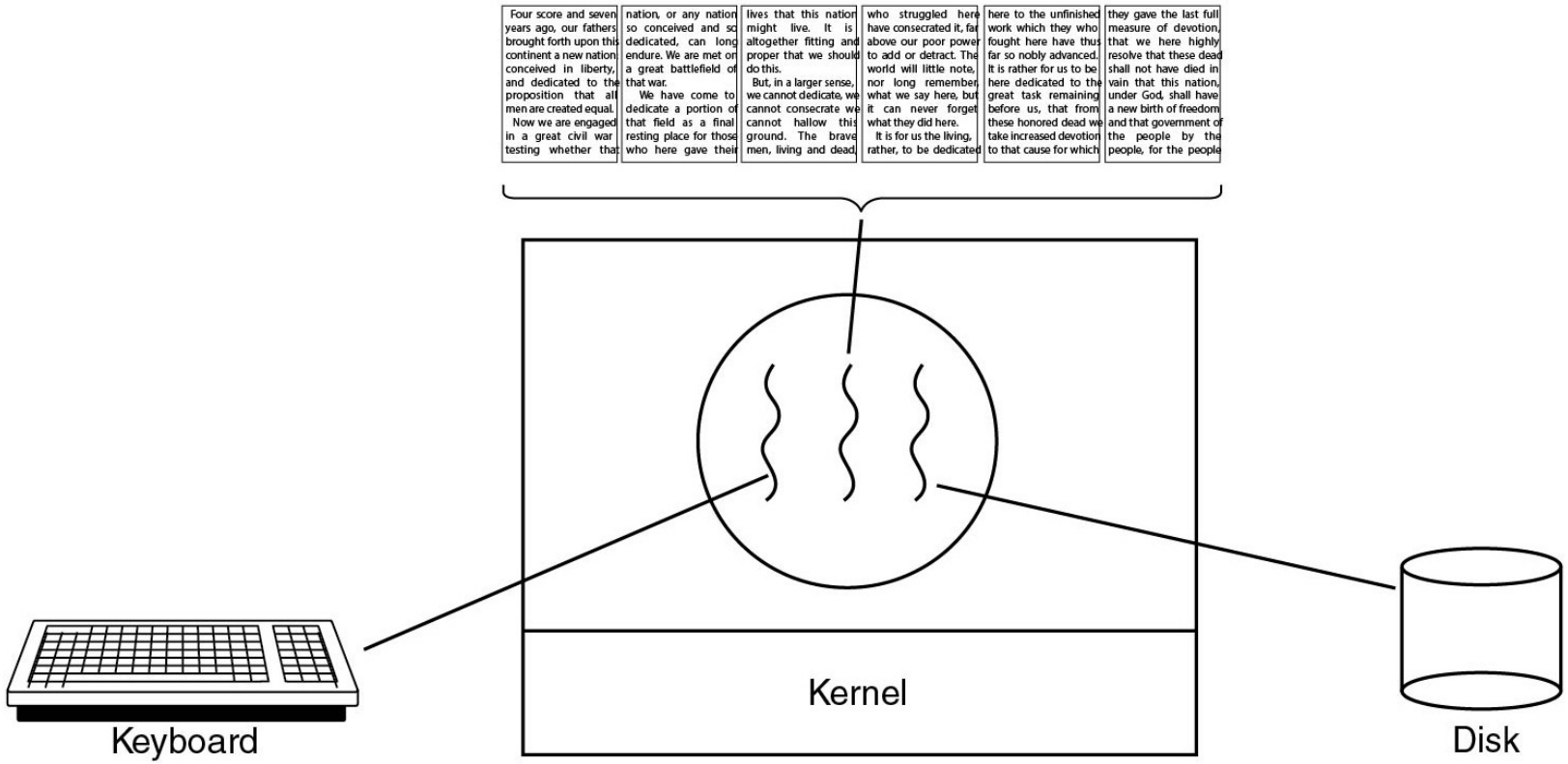
THREAD

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**

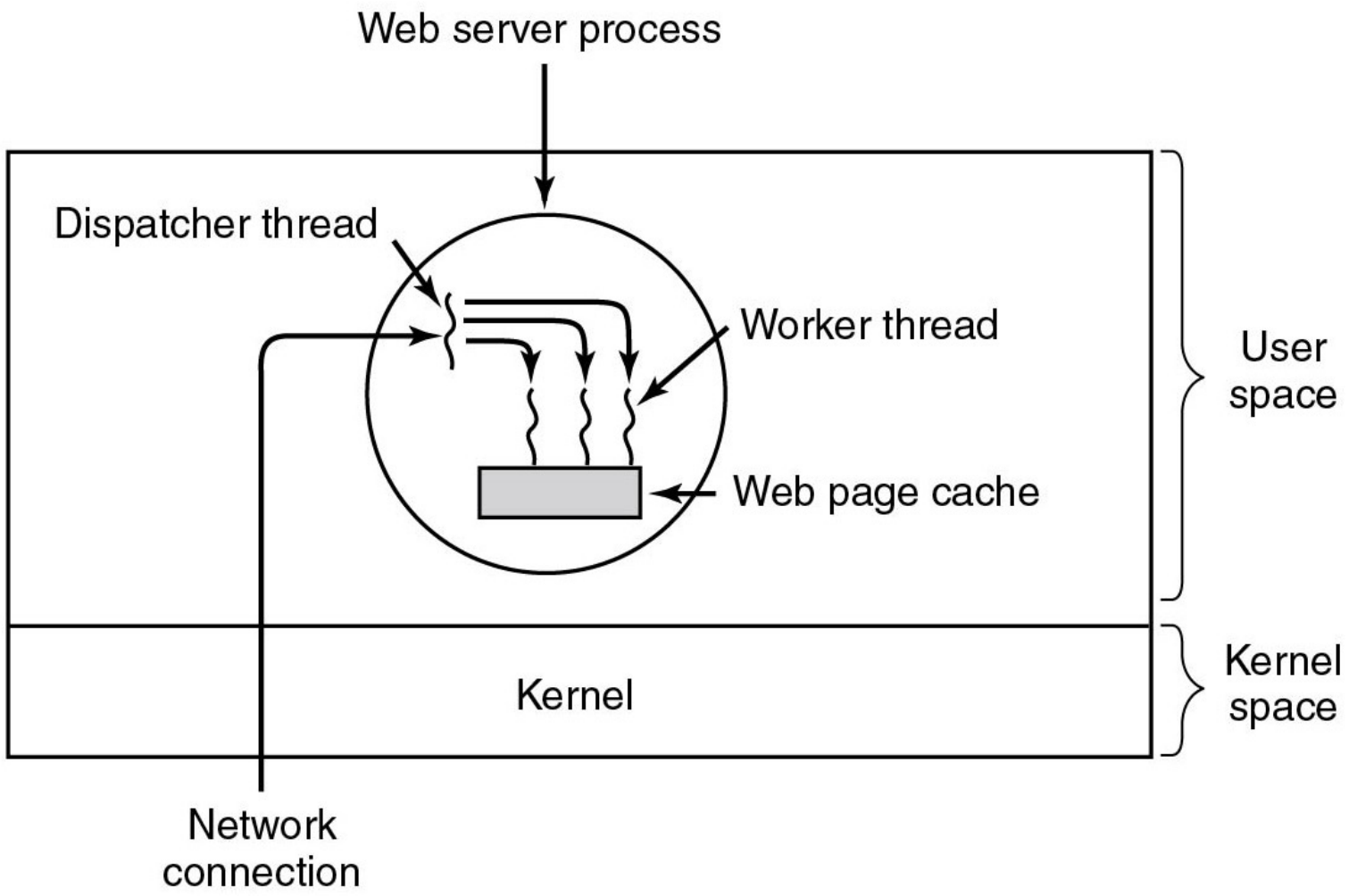
UTILIZZO DEI THREAD (1)



Un Word Processor con tre thread.

UTILIZZO DEI THREAD

A multithreaded Web server.



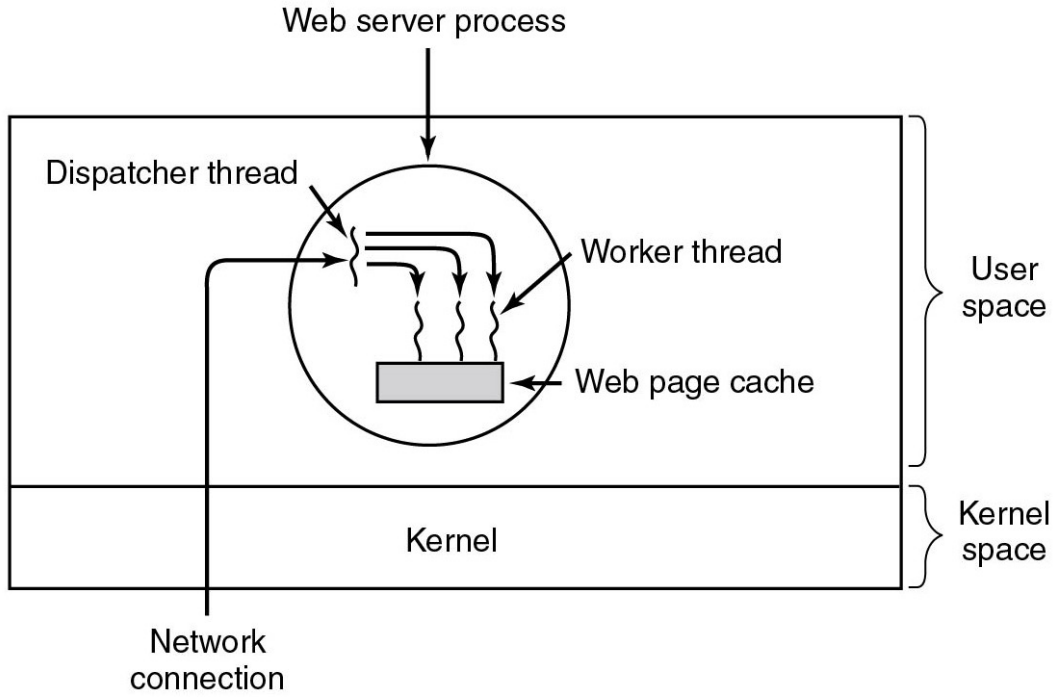
UTILIZZO DEI THREAD (2)

```
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)



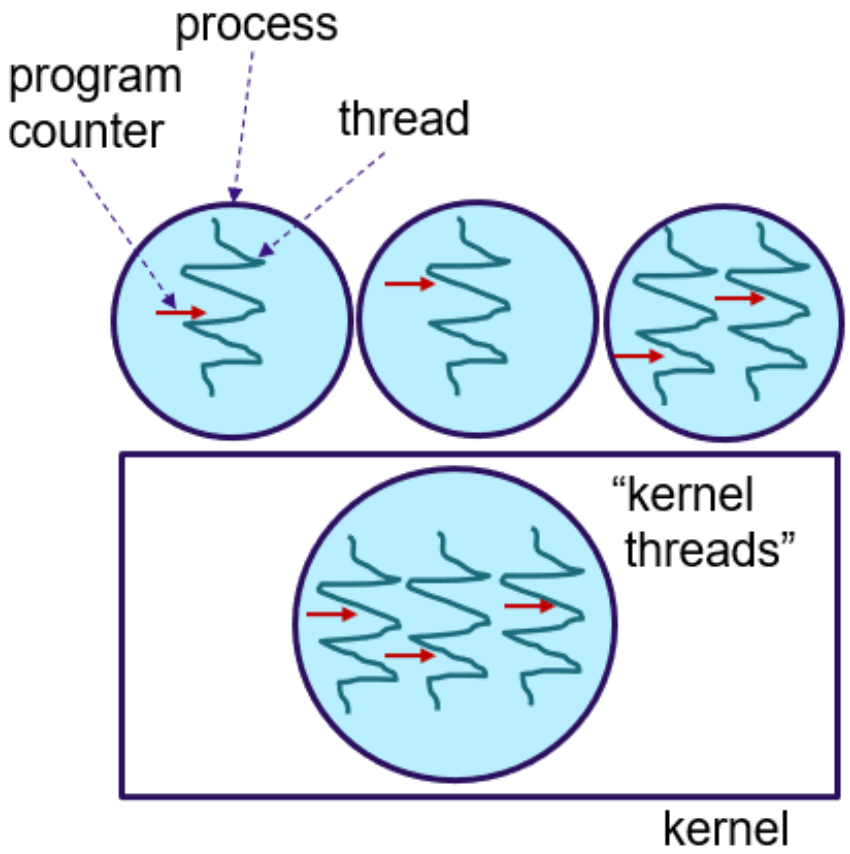
Schema di massima del codice della Figura

(a) Thread del dispatcher.

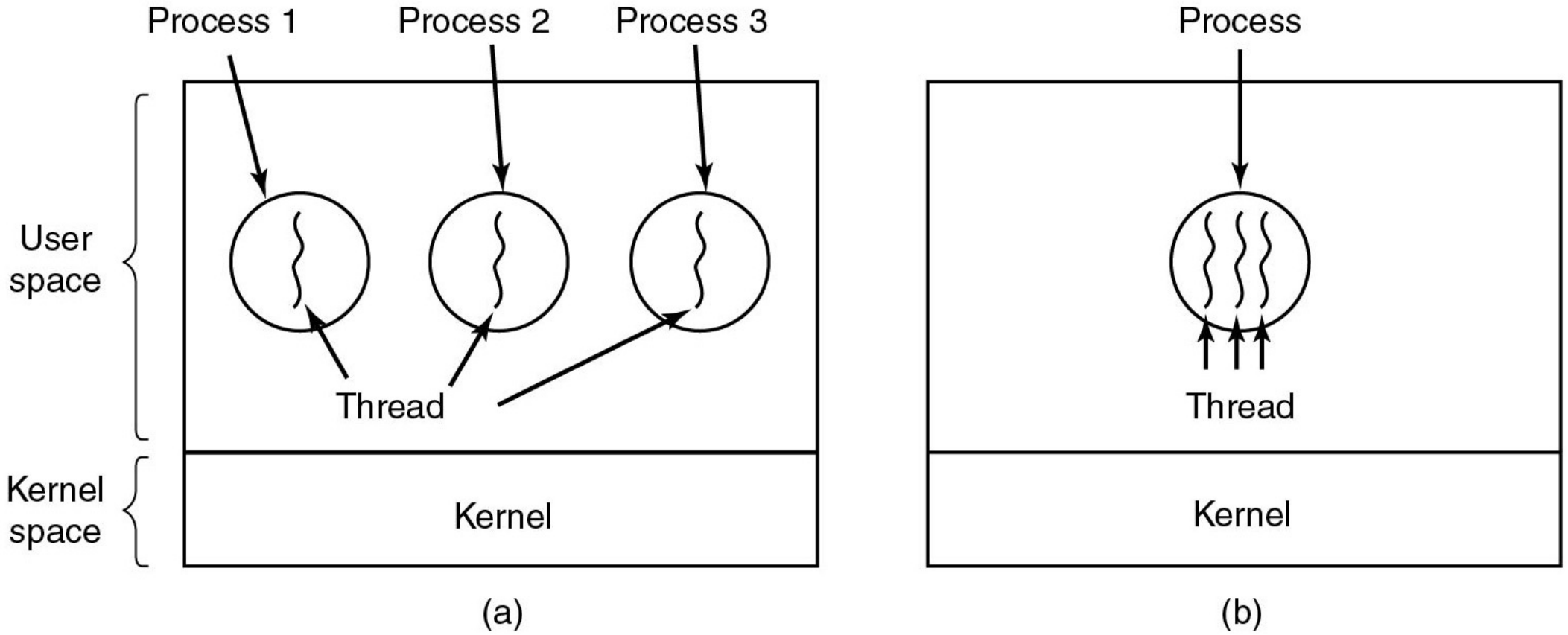
(b) Thread di lavoro.

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

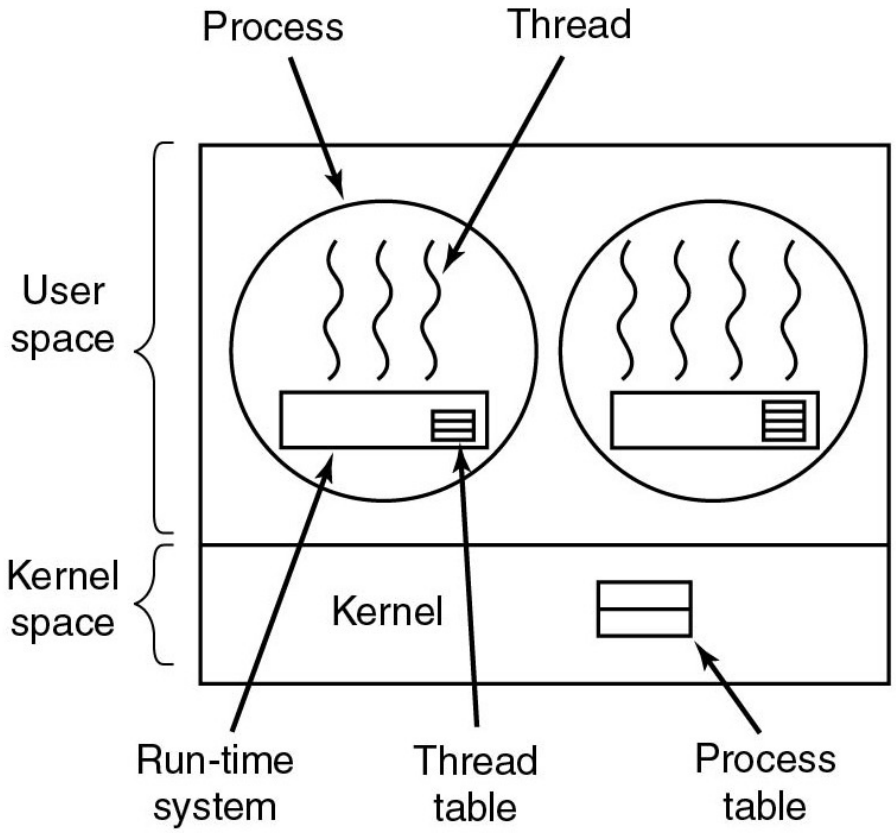
```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#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;
}
```

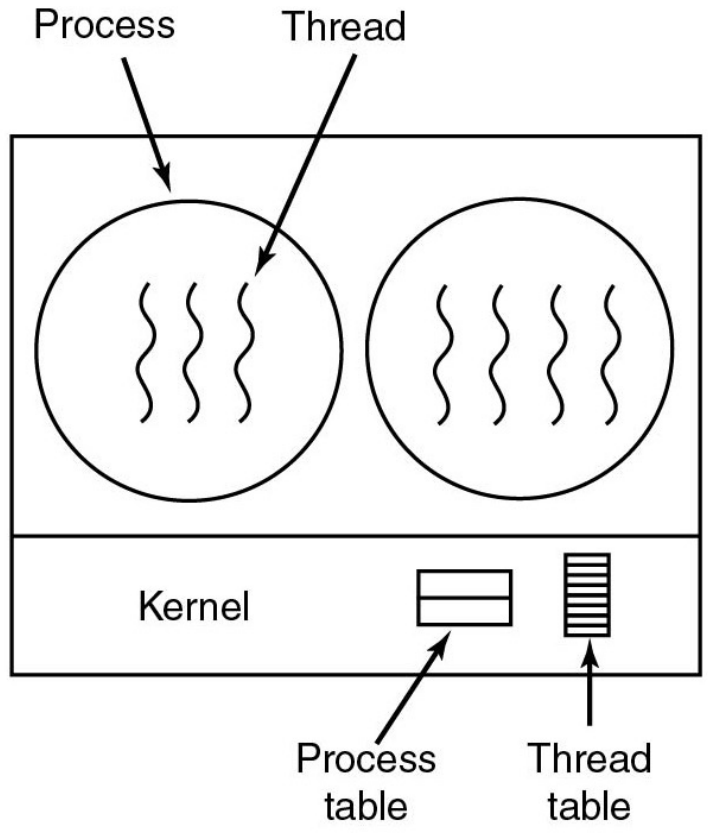
What will the output be?

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

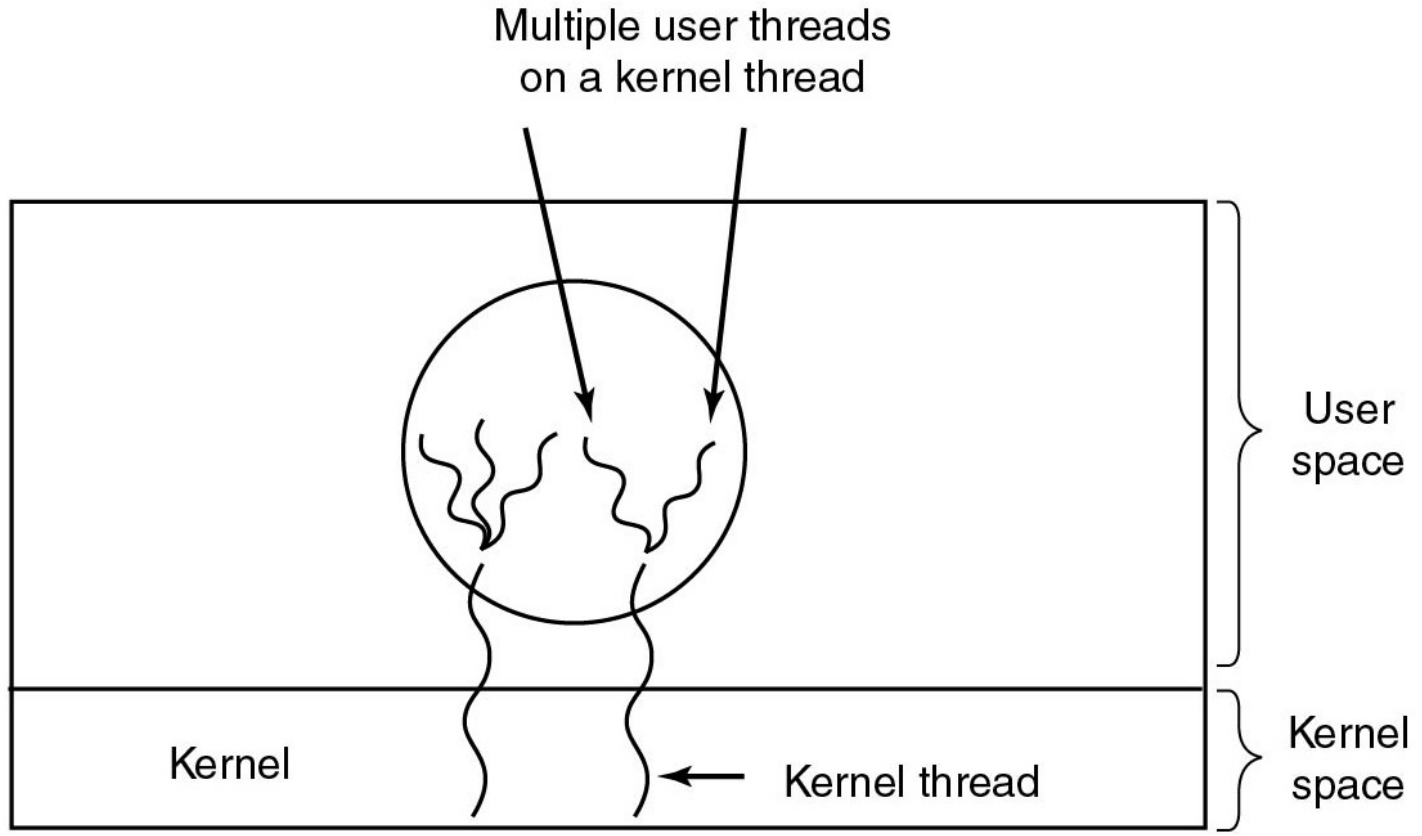
- Tuttavia, ci sono **problemi** con le chiamate di **sistema bloccanti**
  - se **un thread fa una chiamata che lo blocca, tutti gli altri thread nel processo vengono fermati**.
  - Gli **errori di pagina**, dove un programma accede a memoria non presente, **possono bloccare l'intero processo** quando 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.