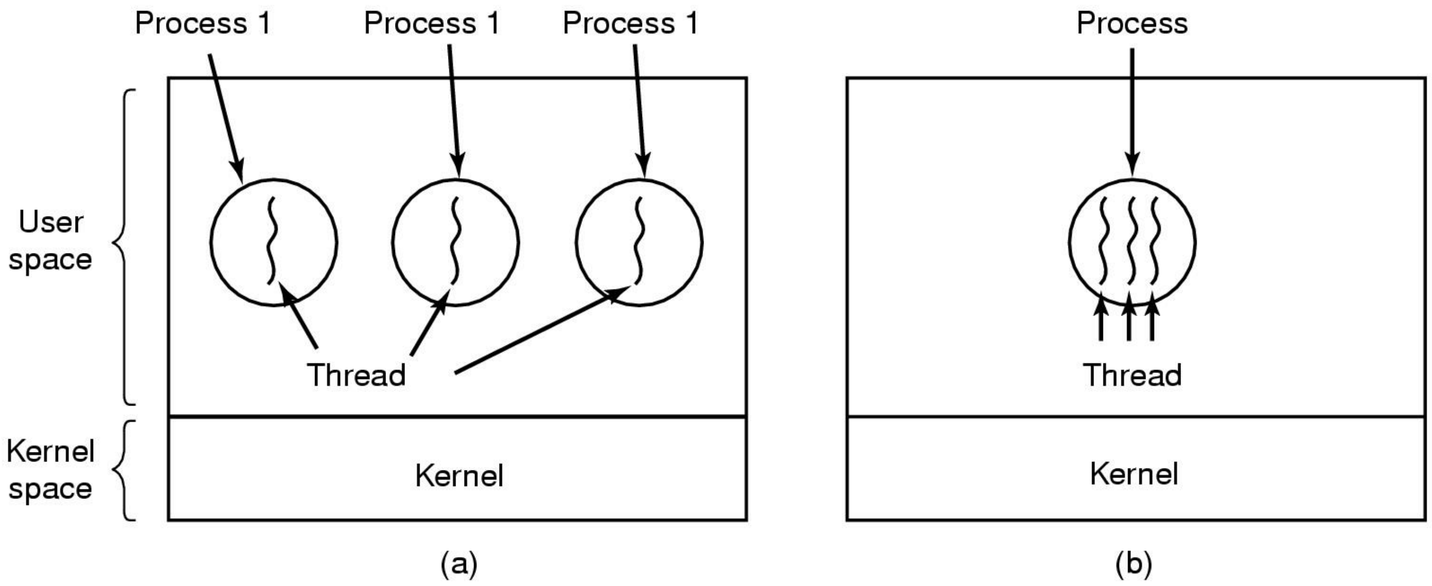
Thread Model

- Process model
  - Resource grouping
    - Address space, open files, child processes, accounting information, etc.
  - Thread (of execution)
    - Program counter, registers, stack
- Thread model
  - Allows multiple thread of executions to take place in the same process environment (multithreading)
  - Entity scheduled for execution on the CPU
  - Lightweight process

The Thread Model



(a) Three processes each with one thread

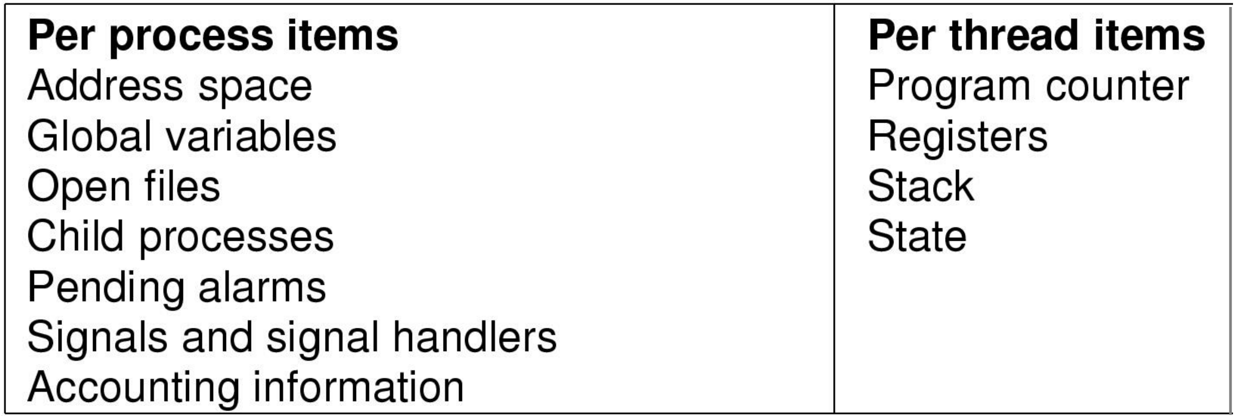
- The three processes are unrelated

(b) One process with three threads (sharing the same address space)

- The three threads are part of the same job and are closely cooperating with each other

The Thread Model

- No protection between threads
  - Multiple threads in a process cooperate.
- Per process items
  - shared by all threads in a process
- Per thread items
  - private to each thread

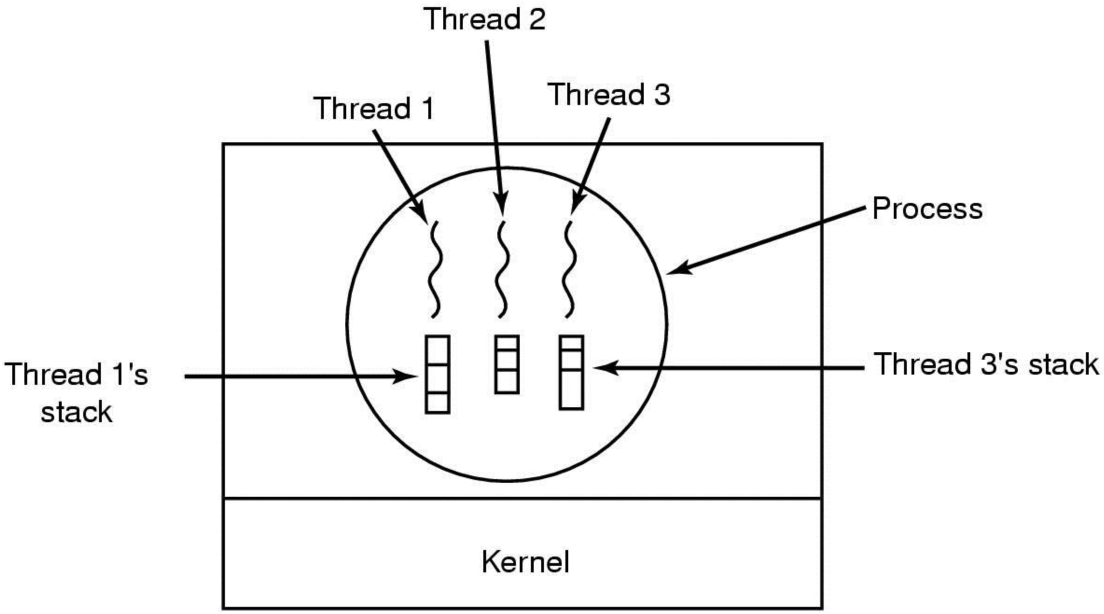


Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

- Multiple threads of execution share a set of resources so they can work together closely to perform some task.

The Thread Model

- Like a traditional process, a thread can be in any one of several states: running, blocked, ready.
- Each thread has its own stack
  - The stack contains one frame for each procedure called but not yet returned from. This frame contains the procedure’s local variables and the return address.
  - Each thread will generally call different procedures and a thus a different execution history.



Thread Model

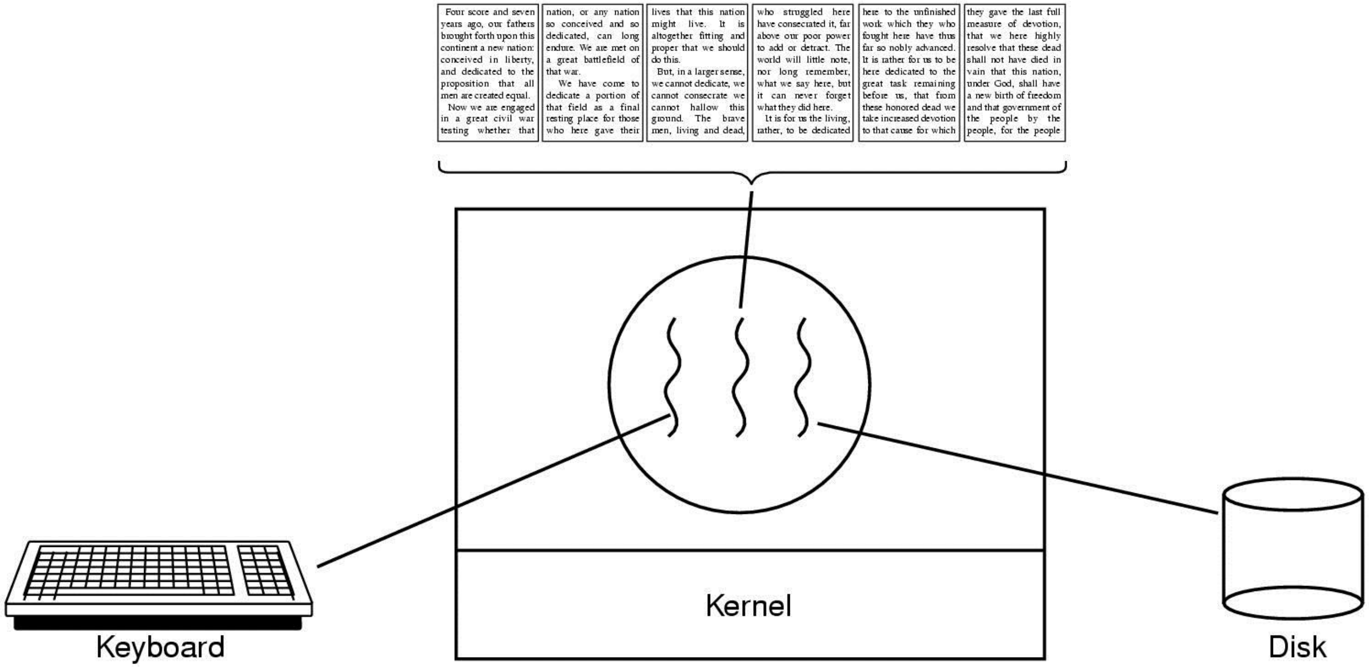
- Thread-related library procedures
  - *thread_create(name of a procedure for the new thread to run)*
    - In multithreading, processes normally start with a single thread present. The thread creates new threads by calling *thread_create*
  - *thread_exit*
    - When a thread had finished its work, it can exit by calling *thread_exit*
  - *thread_wait*
    - Blocks the calling thread until a specific thread has exited
  - *thread_yield*
    - Allows a thread to voluntarily give up the CPU to let another thread run
- Complications

Thread Usage

- Why use threads ?
  - In many applications, multiple activities are going on at once.
  - Threads are easier to create and destroy than processes.
  - Performance gain
  - Real parallelism is possible with multiple CPUs.

Thread Usage (1)

- A word processor with three threads
  - Interactive thread
  - Reformatting thread
  - Disk-backup thread



Four score and seven years ago, our fathers brought forth upon this continent a new nation: conceived in liberty, and dedicated to the proposition that all men are created equal.

Now we are engaged in a great civil war testing whether that nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battlefield of that war.

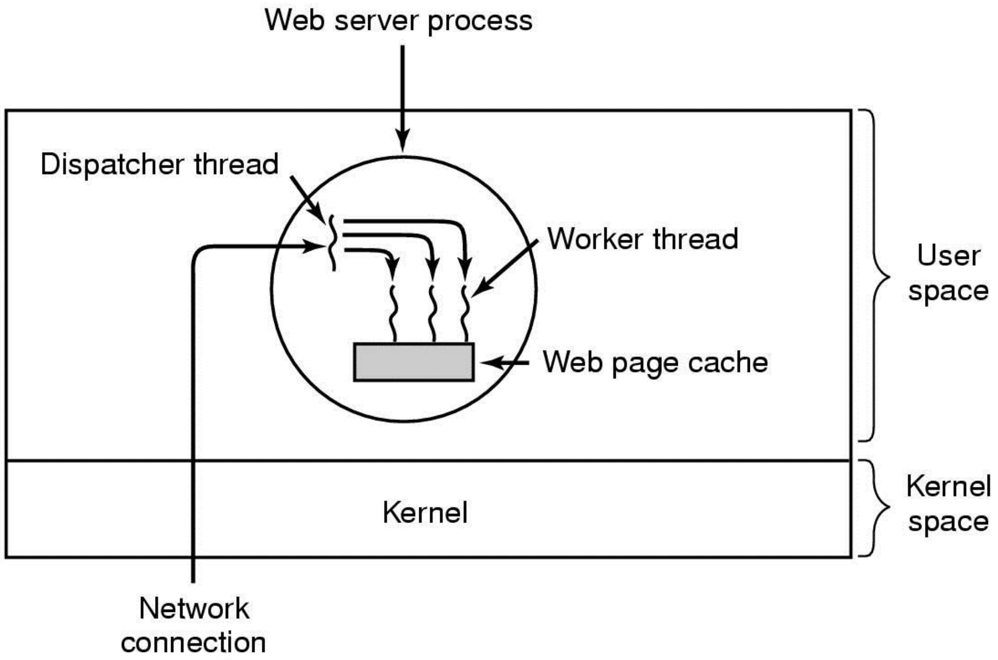
We have come to dedicate a portion of that field as a final resting place for those who here gave their lives that this nation might live. It is altogether fitting and proper that we should do this.

But, in a larger sense, we cannot dedicate, we cannot consecrate we cannot hallow this ground. The brave men, living and dead, who struggled here have consecrated it, far above our poor power to add or detract. The world will little note, nor long remember, what we say here, but it can never forget what they did here.

It is for us the living, rather, to be dedicated here to the unfinished work which they who fought here have thus far so nobly advanced. It is rather for us to be here dedicated to the great task remaining before us, that from these honored dead we take increased devotion to that cause for which they gave the last full measure of devotion, that we here highly resolve that these dead shall not have died in vain that this nation, under God, shall have a new birth of freedom and that government of the people by the people, for the people

Thread Usage (2)

- A multithreaded Web server



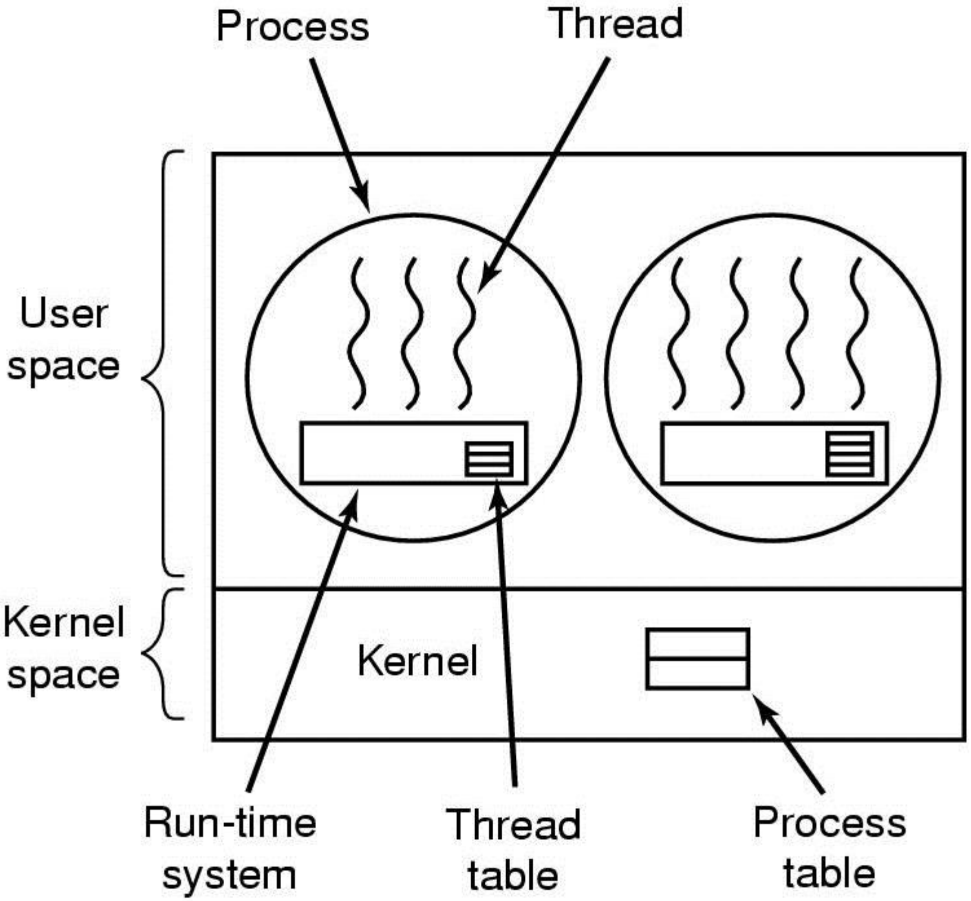
```
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
```

(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)

Implementing Threads in User Space



A user-level threads package

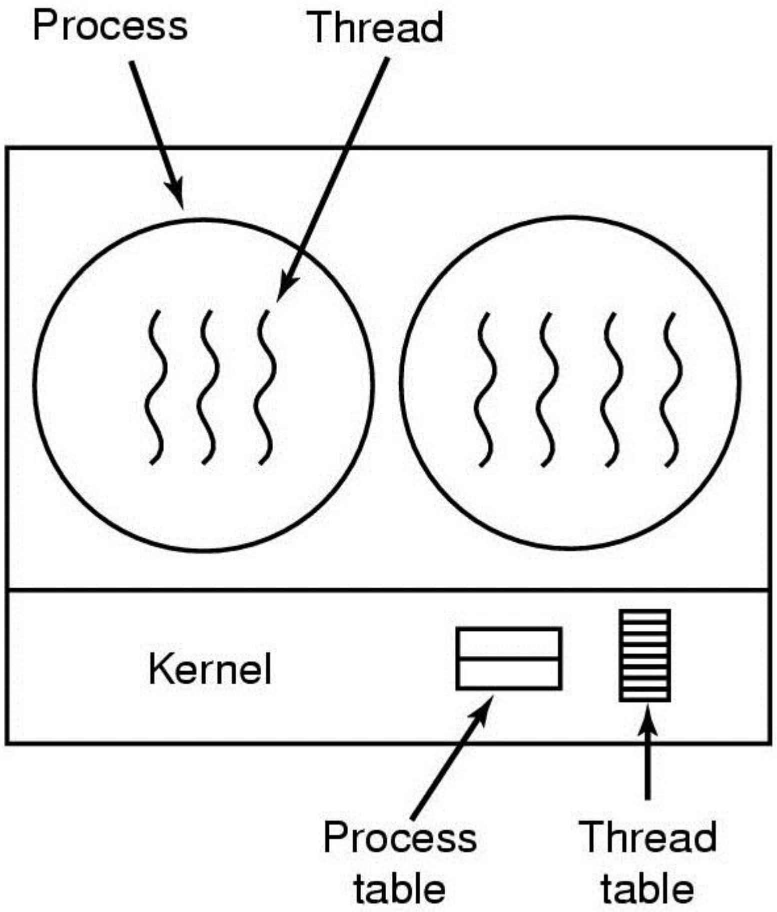
Adv. Of User-level Threads

- Can be implemented on an OS that does not support thread
- Thread switching is faster than the kernel-level thread.
- Allows each process to have its own customized scheduling algorithm
- Scales better

Disadv. Of User-level Threads

- How blocking system calls are implemented
  - A blocking system call could stop all the threads.
  - Could be changed to nonblocking call
  - Check in advance if a call will block: wrapper
- Giving up the CPU
  - Have the run-time system request a clock signal

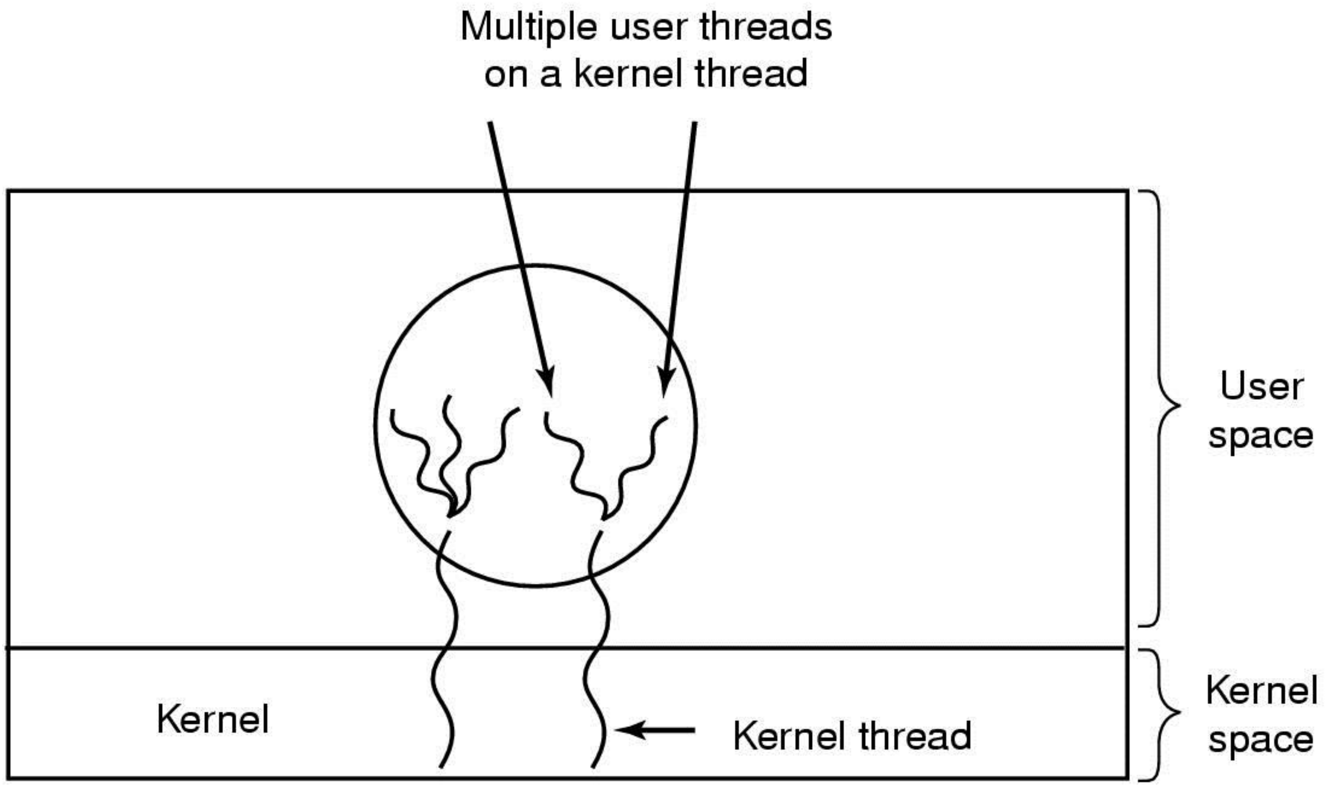
Implementing Threads in the Kernel



A threads package managed by the kernel:

Does not require any new, nonblocking system calls

Hybrid Implementations

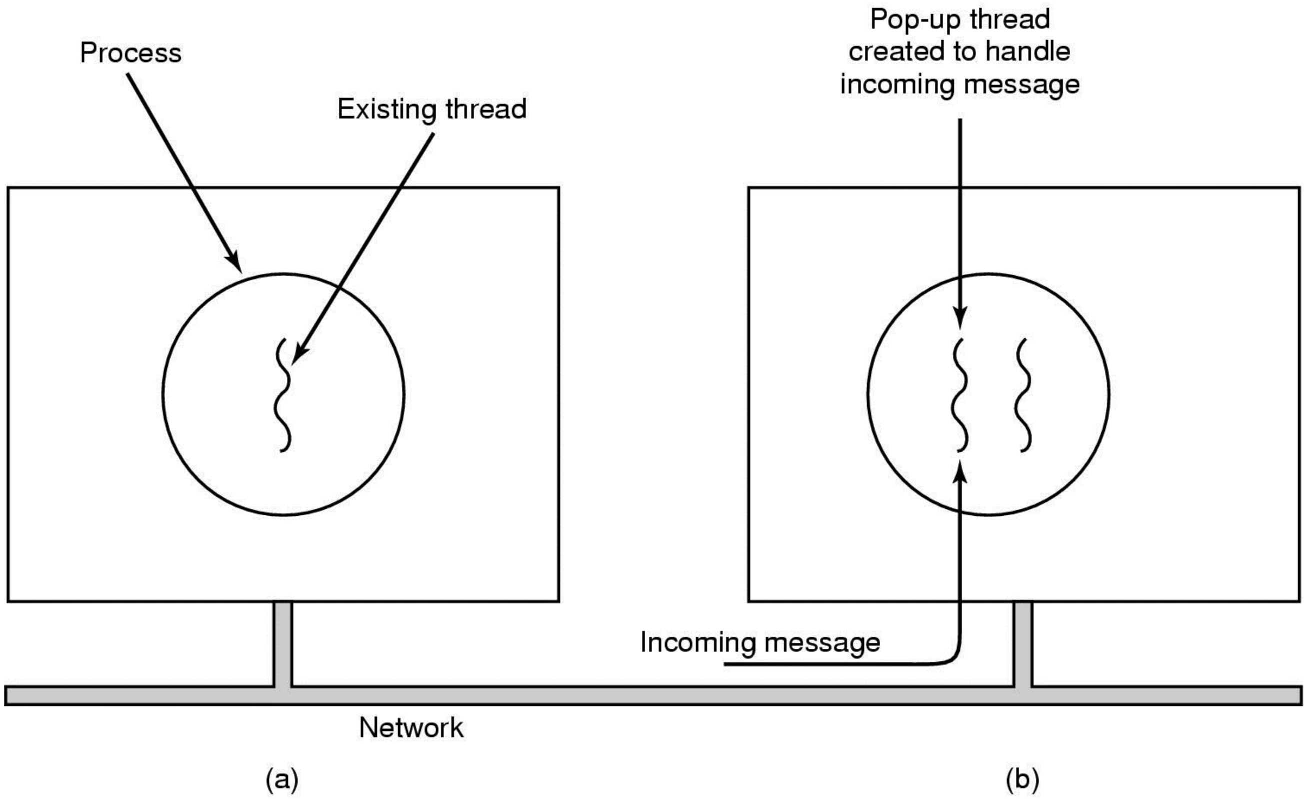


Multiplexing user-level threads onto kernel- level threads

Scheduler Activations

- Goal – mimic functionality of kernel threads
  - gain performance of user space threads
- Avoids unnecessary user/kernel transitions
- Kernel assigns virtual processors to each process
  - lets runtime system allocate threads to processors
- Problem:
  Fundamental reliance on kernel (lower layer)
  calling procedures in user space (higher layer)

Pop-Up Threads

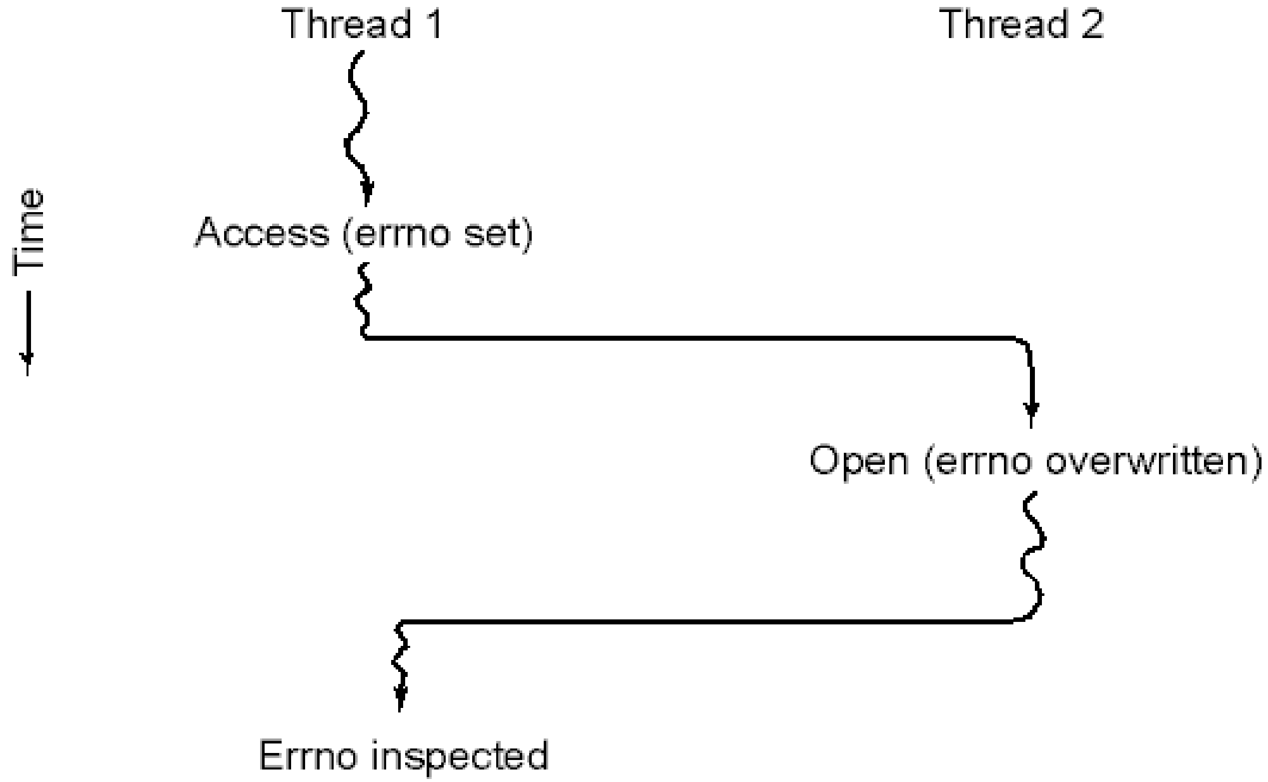


- Creation of a new thread when message arrives

  (a) before message arrives

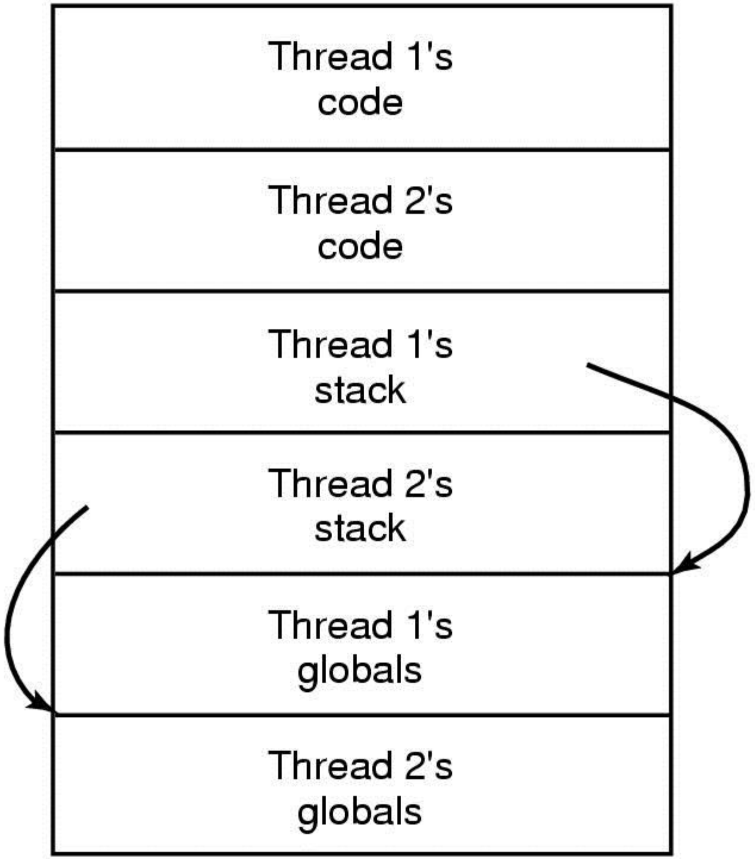
  (b) after message arrives

Making Single-Threaded Code Multithreaded (1)



Conflicts between threads over the use of a global variable

Making Single-Threaded Code Multithreaded (2)



Threads can have private global variables