THREADS

Module 2.08
COP4600 – Operating Systems

Richard Newman

Computer and Information Science and Engineering

University of Florida

WHAT IS A THREAD?

- Thread of execution
  - Sequence of instructions following logical control flow in program
  - Sequence of program counter values with execution environment
- Part of process that depends on particular execution history
  - Program counter and other registers
  - Stack
  - Dynamic data
  - Not text or constants

THE CLASSICAL THREAD MODEL (1)

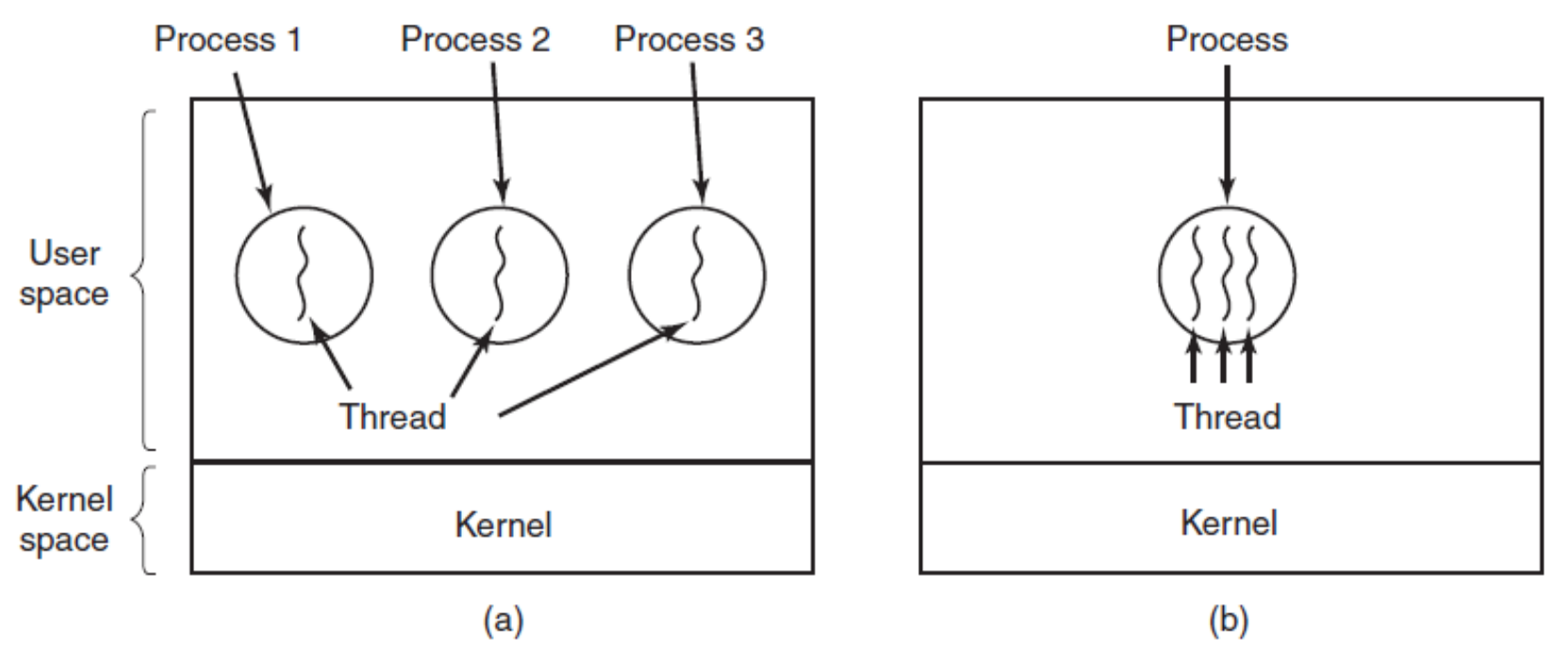


Figure 2-11. (a) Three processes each with one thread. (b) One process with three threads.

MULTITHREADED PROCESS

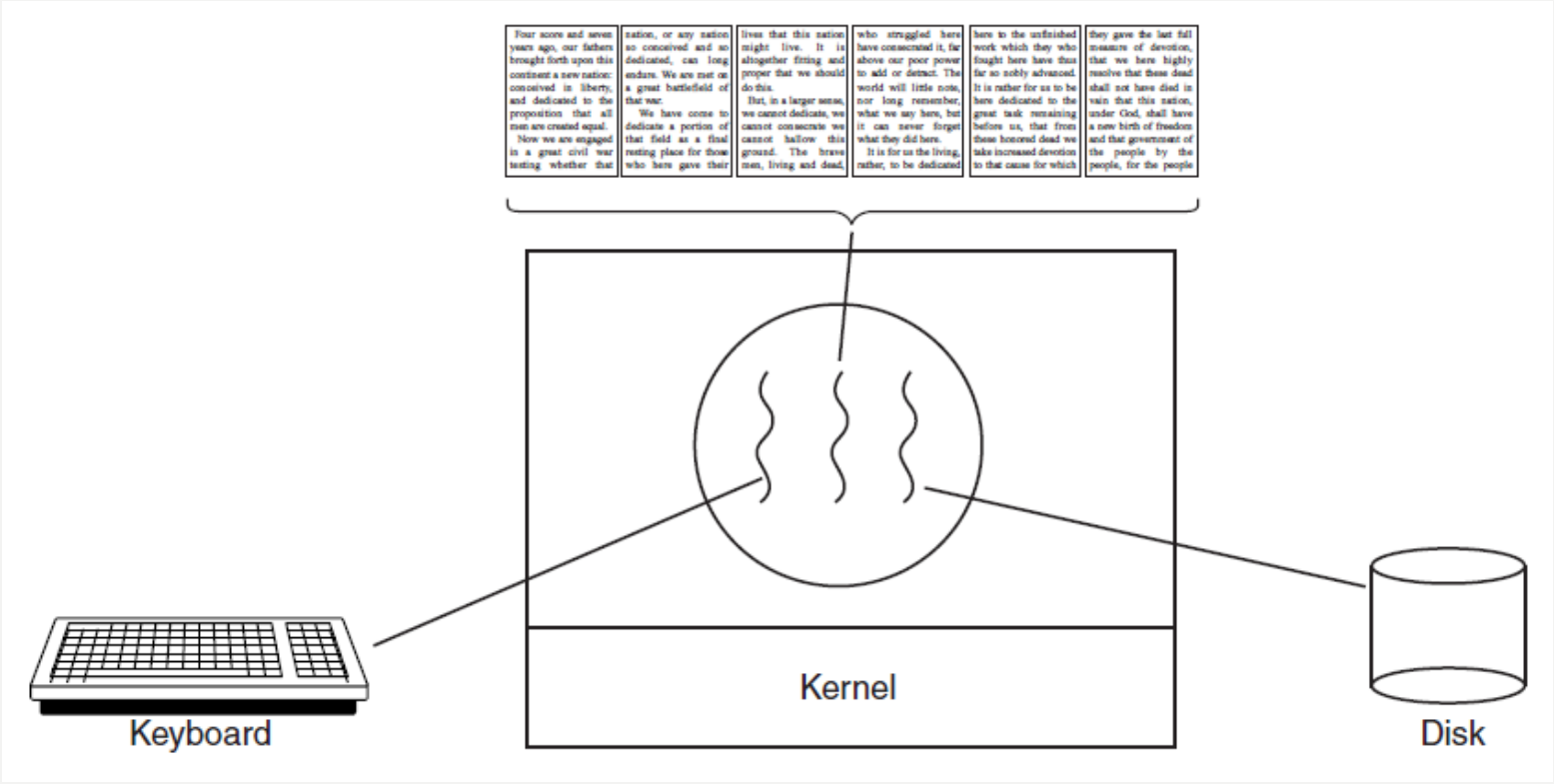


Figure 2-7. A word processor with three threads.

WHY THREADS?

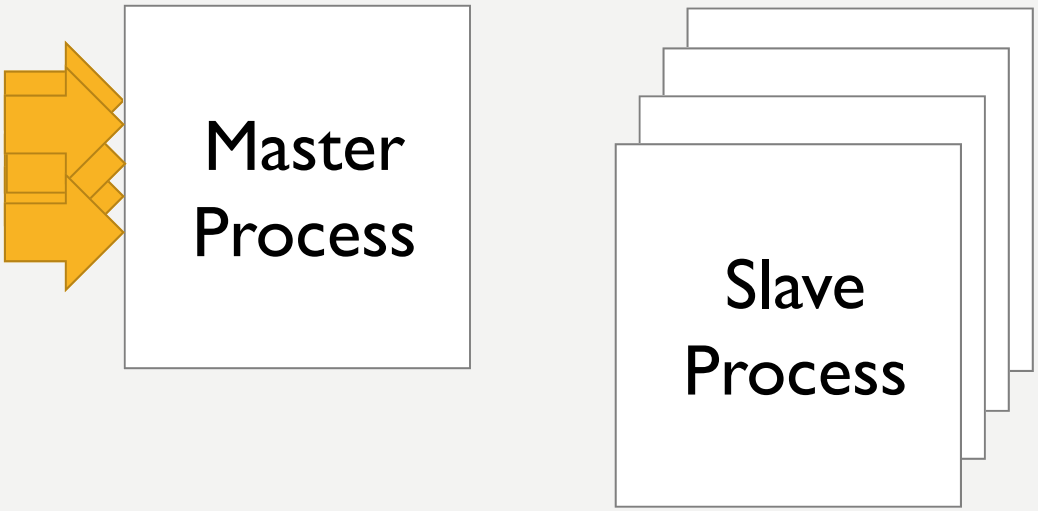
- Desirable to organize complex activities as collection of simpler ones
- Associate a thread with each simpler activity
  - Make it cheaper to create than full process (use same text segment)
  - Make it possible to share data (use same data segment)

SERVER APPROACHES

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts
Reactive Model	Single execution thread – schedule, not block
Distributed system	Distributed system – multiple processes

Figure 2-10. Three ways to construct a server. And two more.

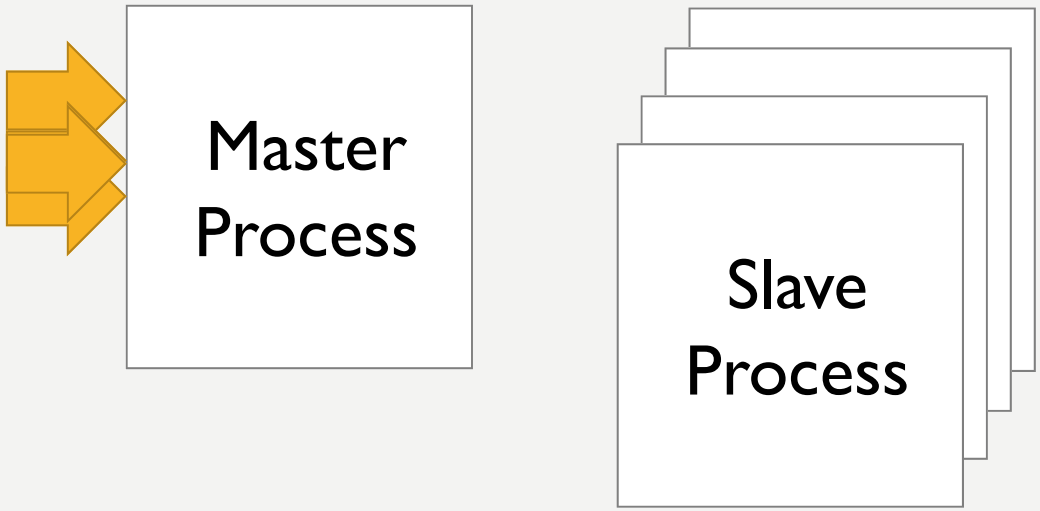
PROCESS POOL SERVER APPROACH



Pool is of fixed size
Idle slaves tie up resources
Can only handle that many requests at one time

Master creates a pool of slave processes
Master process fields requests
Sends request to slave process

DYNAMIC PROCESS POOL SERVER



Pool is of variable size
Pool can expand to handle many requests at one time

Master process fields requests
Master creates slave processes on demand
Sends request to newly created slave process

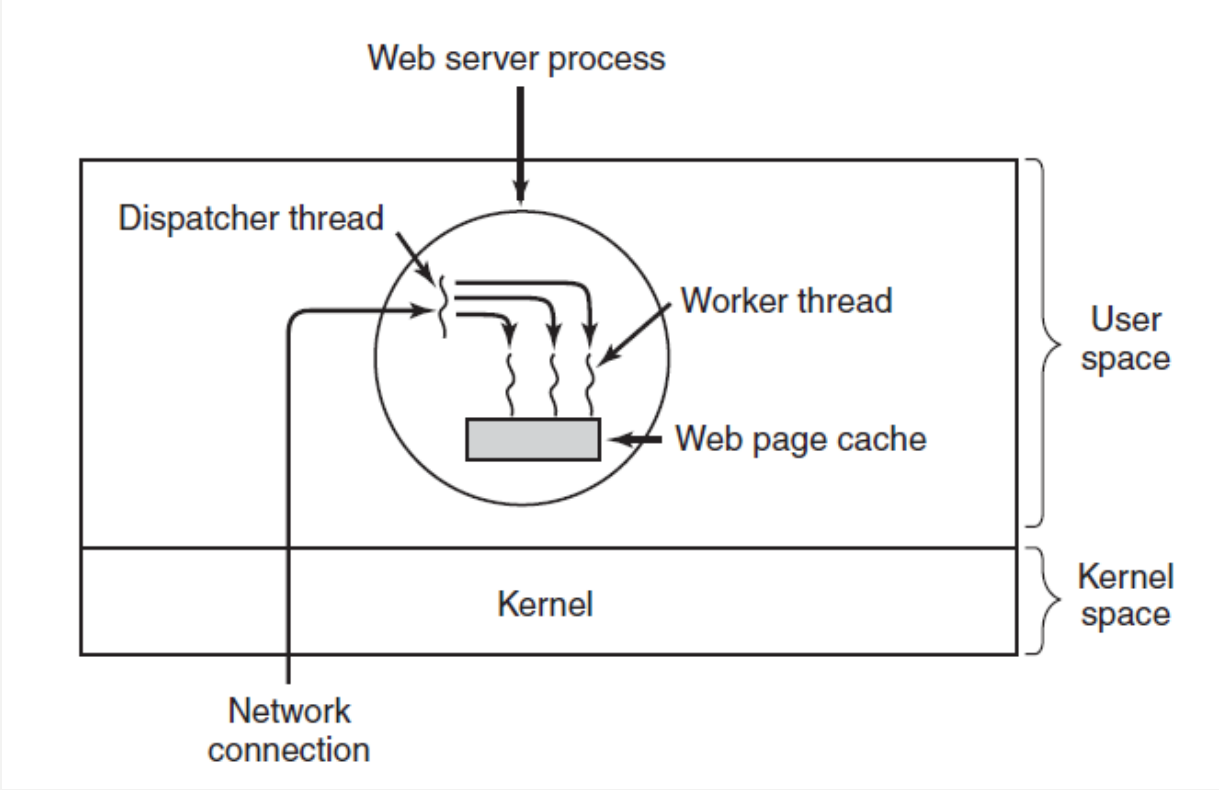
WHY THREADS?

- Desirable to organize complex activities as collection of simpler ones
- Associate a thread with each simpler activity
  - Make it cheaper to create than full process (use same text segment)
  - Make it possible to share data (use same data segment)
- Process creation is expensive
  - Must allocate space for process segments – text, data, stack
  - Must populate segments – copy from parent
    - Often share same code – why copy?
  - Often, child process is overwritten with new program
    - New text, data, stack, etc. – waste of time to copy parent segments!

WHY THREADS?

- Thread creation is cheap – called "lightweight processes"
  - Share most attributes in process table entry
    - Only a few needed per thread
  - Only allocate space and populate stack segment
    - Usually stack is pretty small
  - Share text and data segments
    - Can have private data segments if desired
- Thread destruction also inexpensive
  - Less to deallocate
- Allows shared memory
  - Shared data segment allows threads to communicate easily
  - Must coordinate access, though

MULTITHREADED SERVER



Each thread does one simple job

- Dispatcher: take incoming request, hand off to worker
- Worker: get request, service it

Figure 2-8. A multithreaded Web server.

MULTITHREADED SERVER

Dispatcher:

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

(a)

Worker:

```
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)

Figure 2-9. A rough outline of the code for Fig. 2-8. (a) Dispatcher thread. (b) Worker thread.

POP-UP THREADS

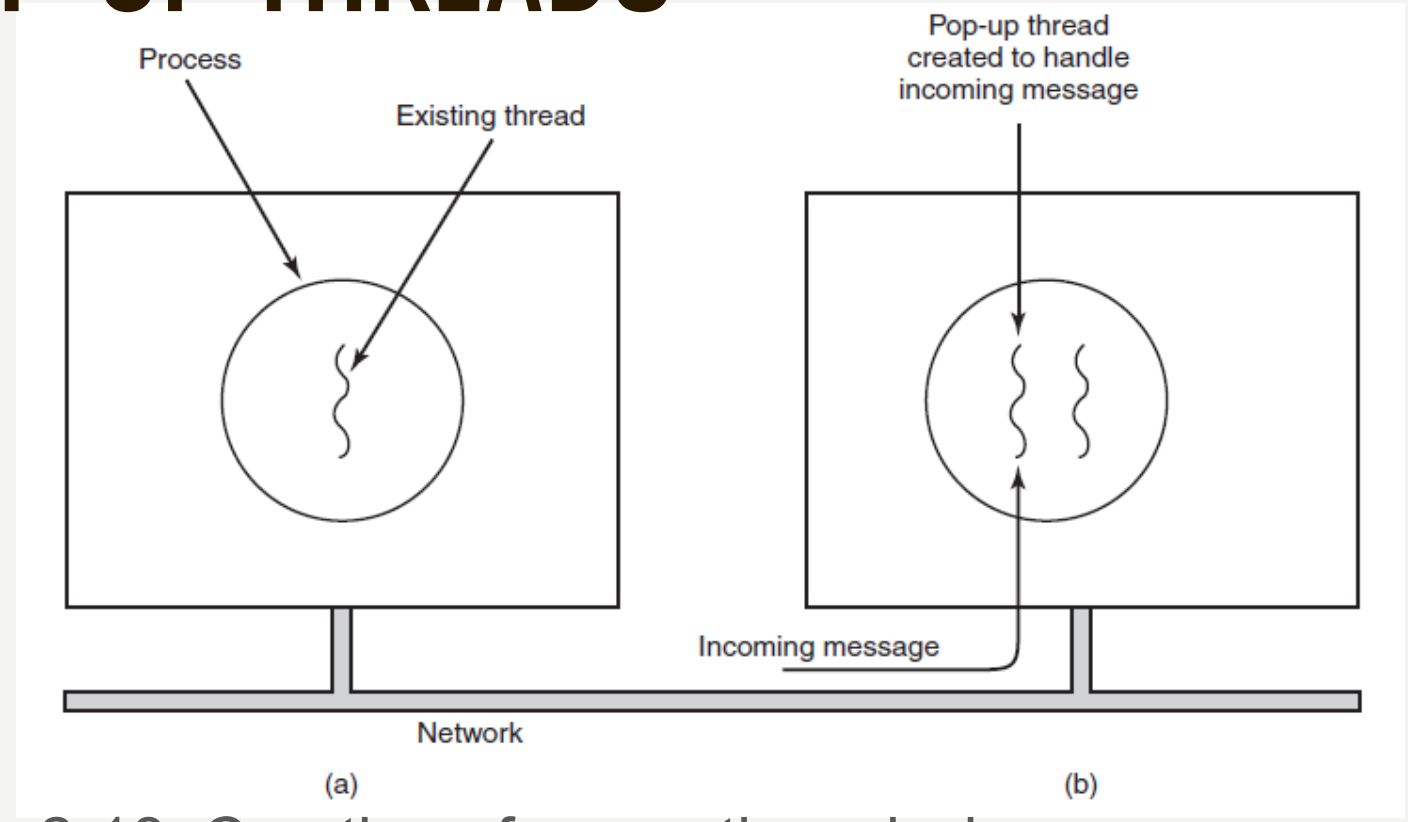


Figure 2-18. Creation of a new thread when a message arrives. (a) Before the message arrives. (b) After the message arrives.

THE CLASSICAL THREAD MODEL (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	

Figure 2-12. The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

THE CLASSICAL THREAD MODEL (3)

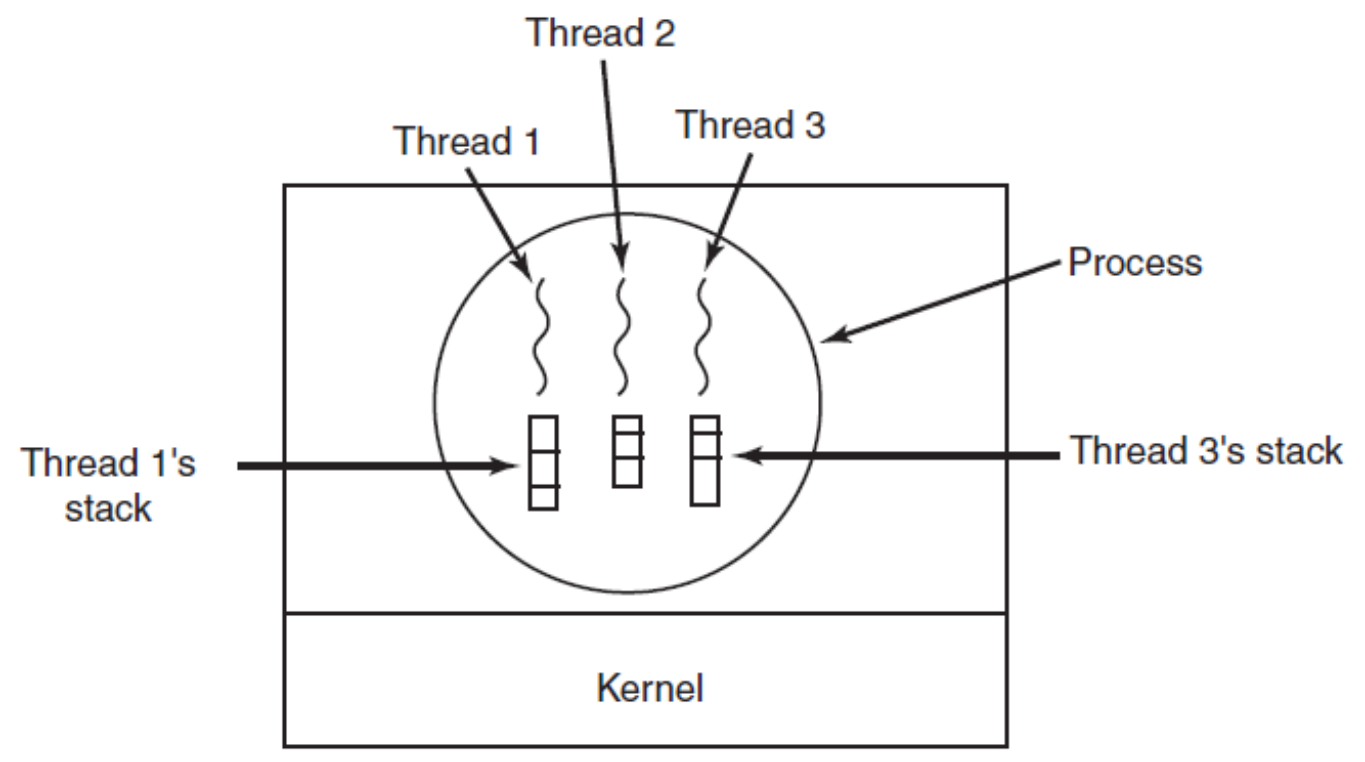


Figure 2-13. Each thread has its own stack.

USER-LEVEL VS. KERNEL THREADS

User-level threads

- Implemented with library
  - Not part of OS
  - Very portable
- Blocking system calls block all threads
- All threads in one process must share single CPU
- Allows user-defined policies

Kernel threads

- Has OS support
- Blocking calls only block the calling thread
- Threads in same process can be scheduled on different CPUs – parallel execution
- Thread management calls more expensive – system calls vs. procedure calls

USER-LEVEL VS. KERNEL THREADS

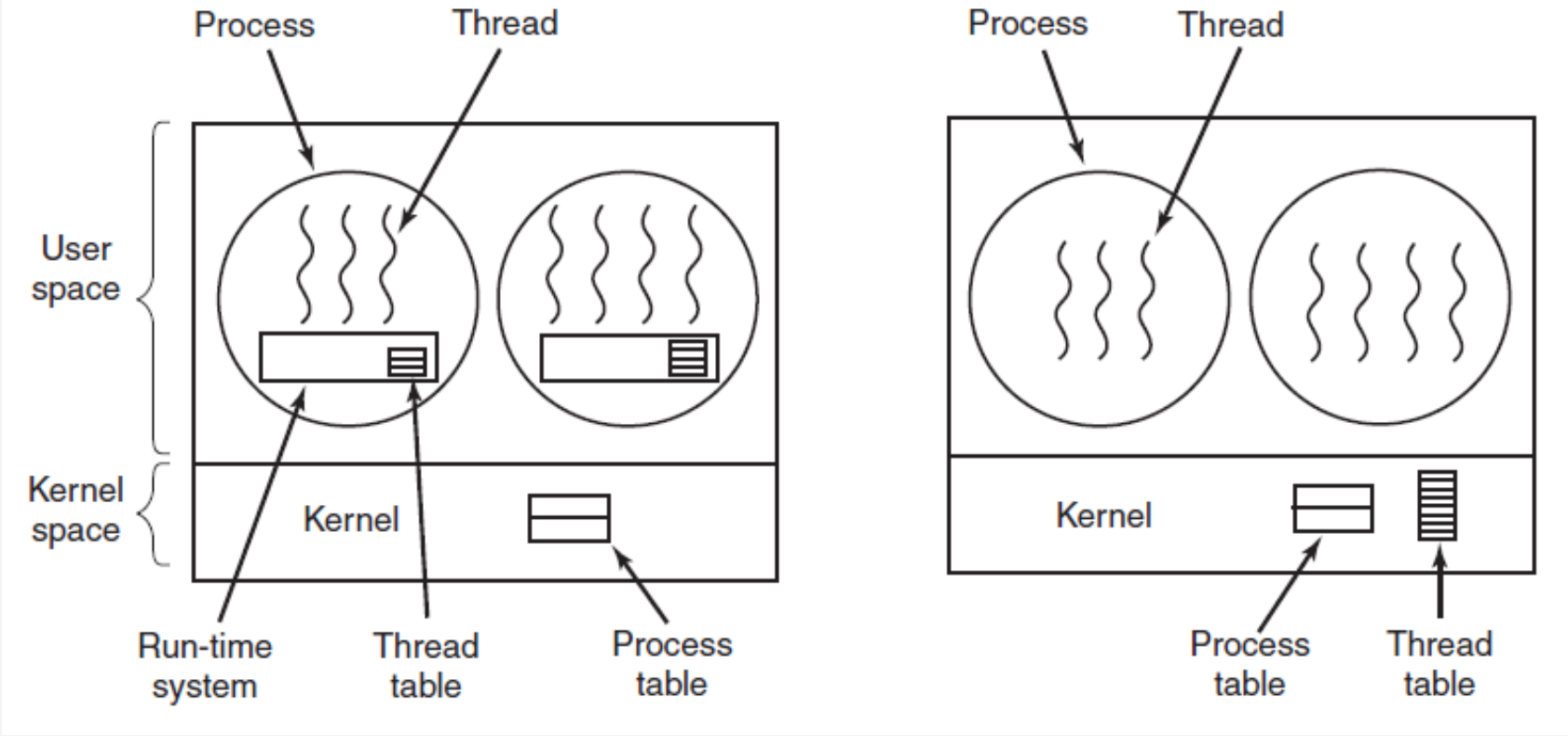


Figure 2-16. (a) A user-level threads package. (b) A threads package managed by the kernel.

HYBRID IMPLEMENTATIONS

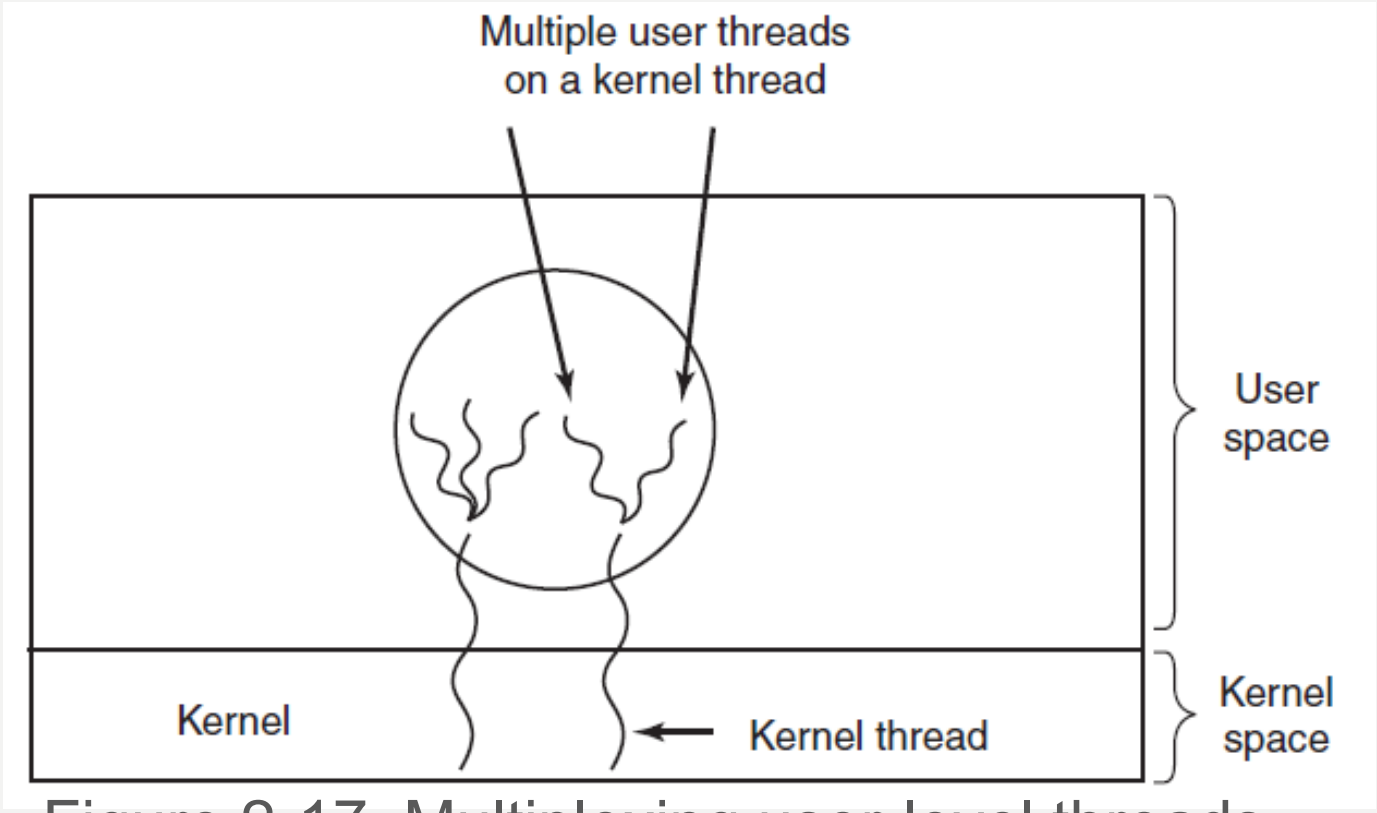
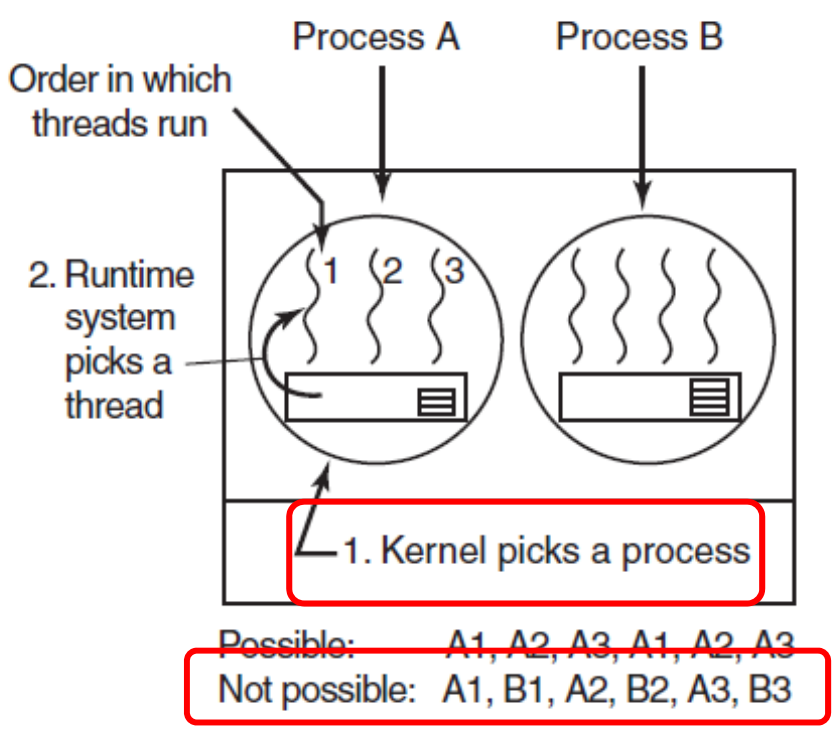
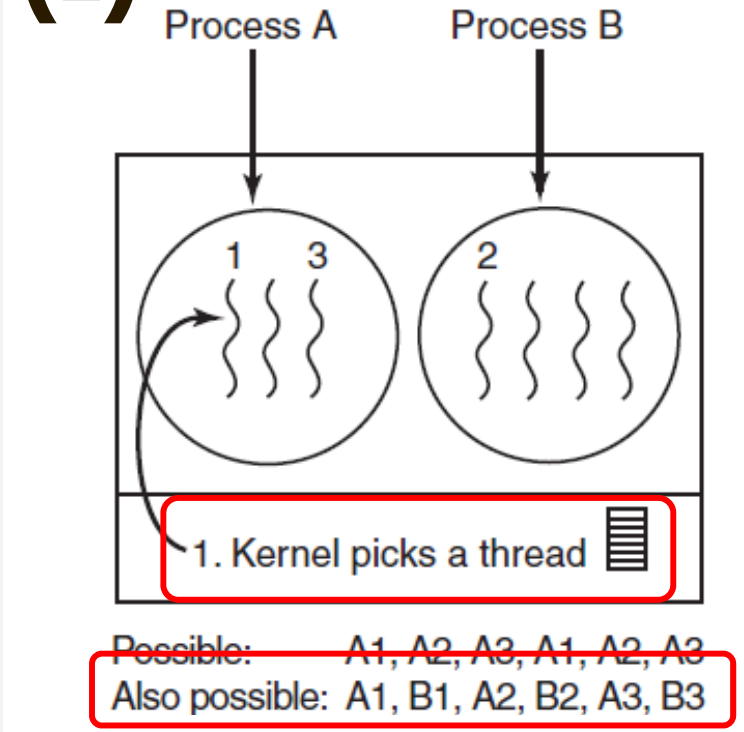


Figure 2-17. Multiplexing user-level threads onto kernel-level threads.

THREAD SCHEDULING (1)



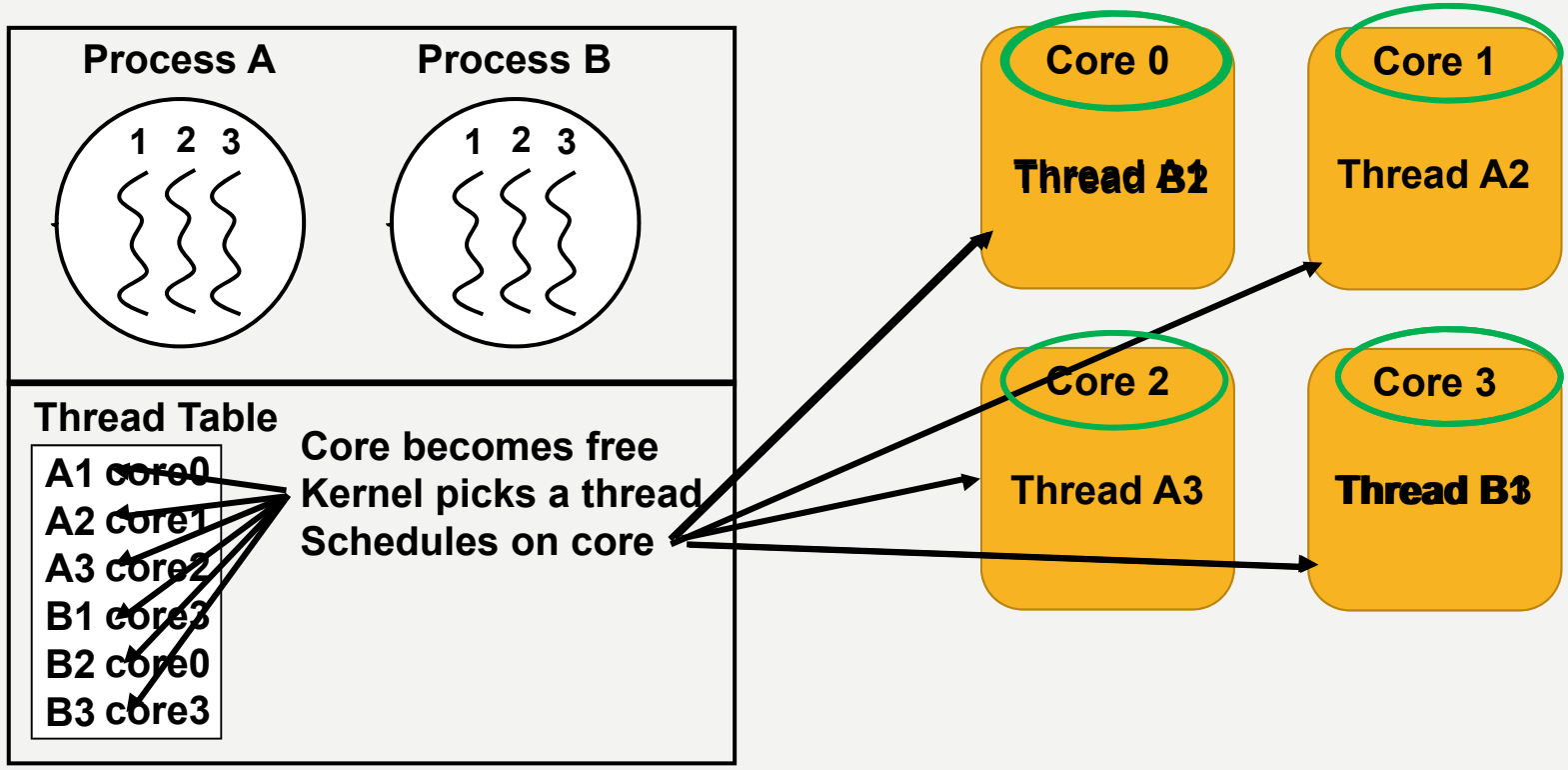
(a) User-level threads



(b) Kernel threads

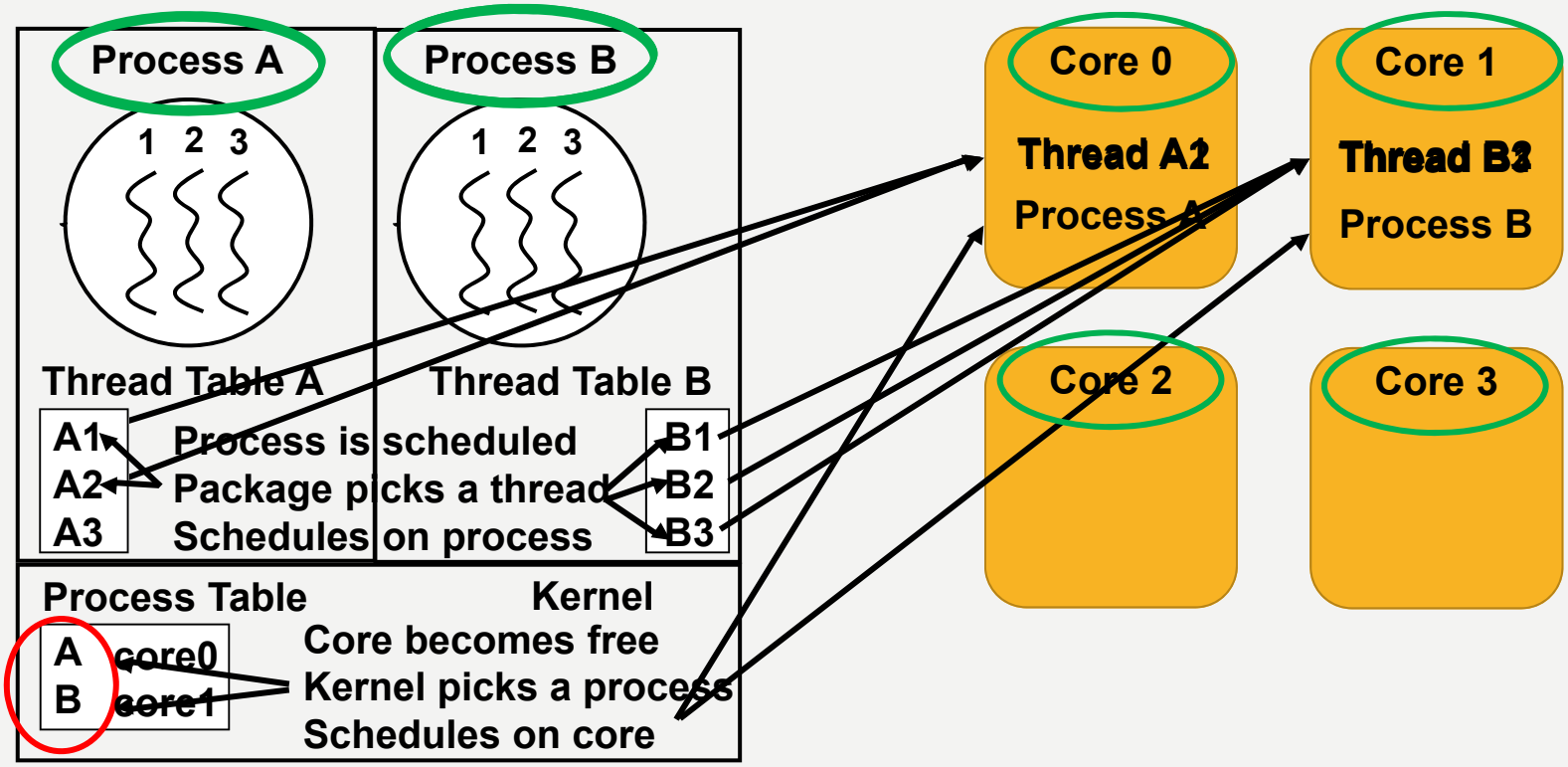
Figure 2-44. Scheduling of user-level threads with a 50-msec process quantum and threads that run 5 msec per CPU burst.

THREAD SCHEDULING (2)



Possible scheduling of kernel-level threads on multiple cores

THREAD SCHEDULING (3)



Possible scheduling of user-level threads on multiple cores

POSIX THREADS (1)

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Figure 2-14. Some of the Pthreads function calls.

POSIX THREADS (2)

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

#define NUMBER_OF_THREADS     10

void *print_hello_world(void *tid)
{
        /* This function prints the thread's identifier and then exits. */
        printf("Hello World. Greetings from thread %d\n", tid);
        pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
        /* The main program creates 10 threads and then exits. */
        pthread_t threads[NUMBER_OF_THREADS];
        int status, i;

        for(i=0; i < NUMBER_OF_THREADS; i++) {
                printf("Main here. Creating thread %d\n", i);
                status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);
```

Figure 2-15. An example program using threads.

POSIX THREADS (3)

```
        int status, i;

        for(i=0; i < NUMBER_OF_THREADS; i++) {
                printf("Main here. Creating thread %d\n", i);
                status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

                if (status != 0) {
                        printf("Oops. pthread_create returned error code %d\n", status);
                        exit(-1);
                }
        }
        exit(NULL);
}
```

Figure 2-15. An example program using threads.

MAKING SINGLE-THREADED CODE MULTITHREADED (1)

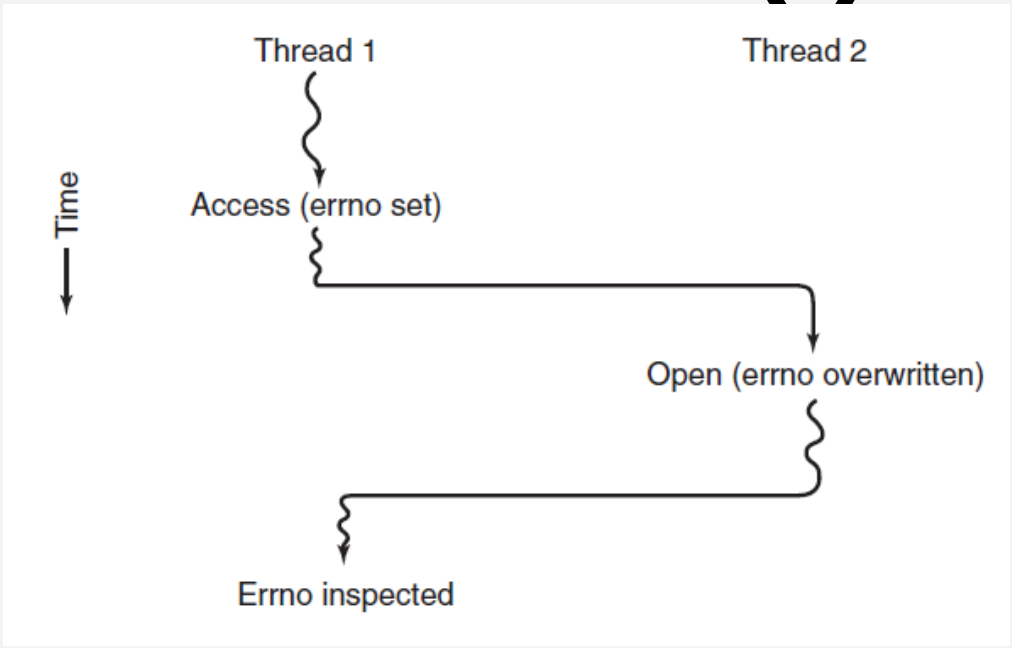


Figure 2-19. Conflicts between threads over the use of a global variable.

MAKING SINGLE-THREADED CODE MULTITHREADED (2)

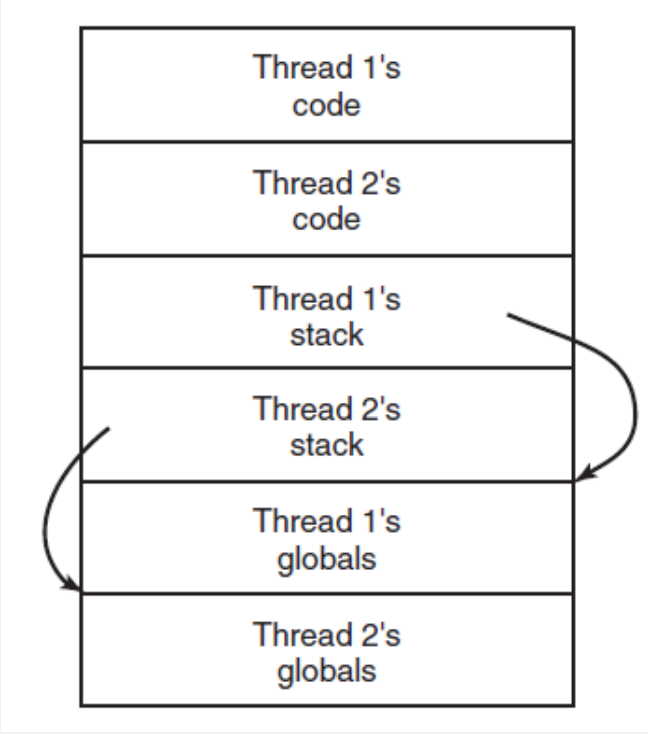


Figure 2-20. Threads can have private global variables.

SUMMARY

- What is a thread
  - Execution sequence within a process
- Why threads were developed
  - Efficiency – lightweight processes
- User-level vs. kernel threads
  - Overhead, portability, blocking calls, scheduling
- Thread programming
  - Posix pthreads package