Processes and Threads

Week 04 – Lecture 2

Threads

Team

Instructors

Giancarlo Succi

Joseph Brown

Teaching Assistants

Vladimir Ivanov

Stanislav Litvinov

Alexey Reznik

Munir Makhmutov

Hamna Aslam

Sources

- These slides have been adapted from the original slides of the adopted book:
  - Tanenbaum & Bo, Modern Operating Systems: 4th edition, 2013
    Prentice-Hall, Inc.

  and customised for the needs of this course.
- Additional input for the slides are detailed later

Threads

- In traditional OSs each process has an address space and a **single thread of control**
- In many situations, it is desirable to have **multiple threads of control** in the same address space running in quasi-parallel, as though they were separate processes (except for the shared address space)

Thread Usage (1)

- There are several reasons for having these miniprocesses, called threads:
  - in many applications, multiple activities are going on at once. Some of these may block from time to time. By decomposing an application into multiple sequential threads that run in quasi-parallel, the programming model becomes simpler. It's pretty much like having several "parallel" processes, only with threads it is possible for the parallel entities to share an address space and all of its data

Thread Usage (2)

- since they are lighter weight than processes, they are easier (i.e., faster) to create and destroy than processes
- when there is substantial computing and also substantial I/O, having threads allows these activities to overlap, thus speeding up the application
- threads are useful on systems with multiple CPUs providing real parallelism

Thread Usage (3)

- As an example, let's consider word processor (Fig. 2-7):
  - The first thread will interact with user, listening to the keyboard and mouse and responding to simple commands
  - The second thread will be responsible for changing formatting of pages that user does not see when user changes text on the current page
  - The third one will be responsible for autosaving
  - And so on…
- If the program was single-threaded, users would have to wait every time it would decide to save changes to a file

Thread Usage (4)

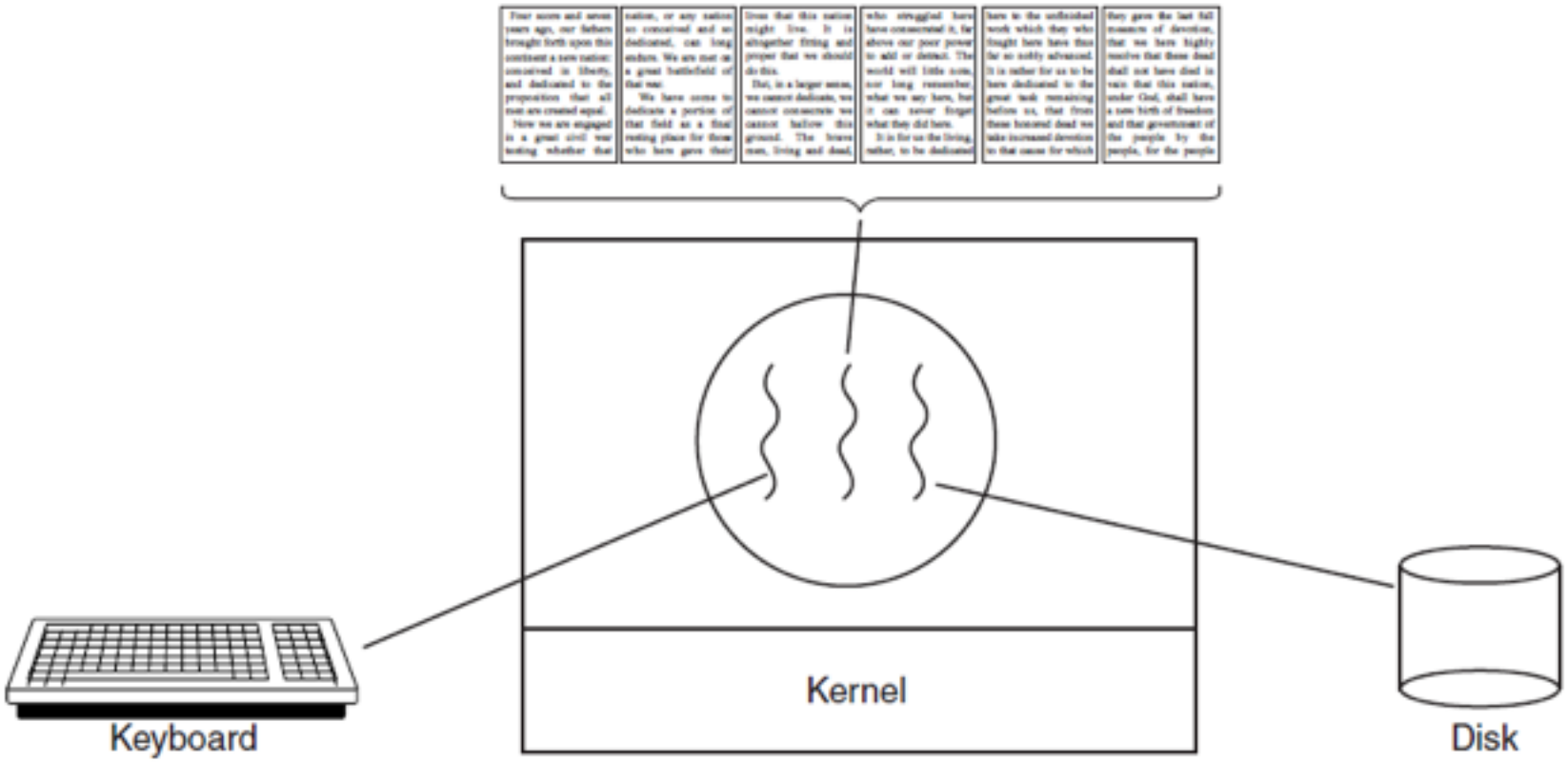

Figure 2-7. A word processor with three threads.

Thread Usage (5)

- Another example, a web-server (Fig. 2-8):
  - some of the most accessed web-pages are stored in **cache** in main memory
  - one thread, **the dispatcher**, reads incoming requests for work from the network
  - after examining the request, it chooses an idle (i.e., blocked) **worker thread** and hands it the request
  - the dispatcher then wakes up the sleeping worker, moving it from blocked state to ready state
  - if the request cannot be satisfied from the cache, to which all threads have access, worker starts a **read** operation to get the page from the disk and blocks until the disk operation completes
  - when the thread blocks on the disk operation, another thread is chosen to run (the dispatcher or another worker)

Thread Usage (6)

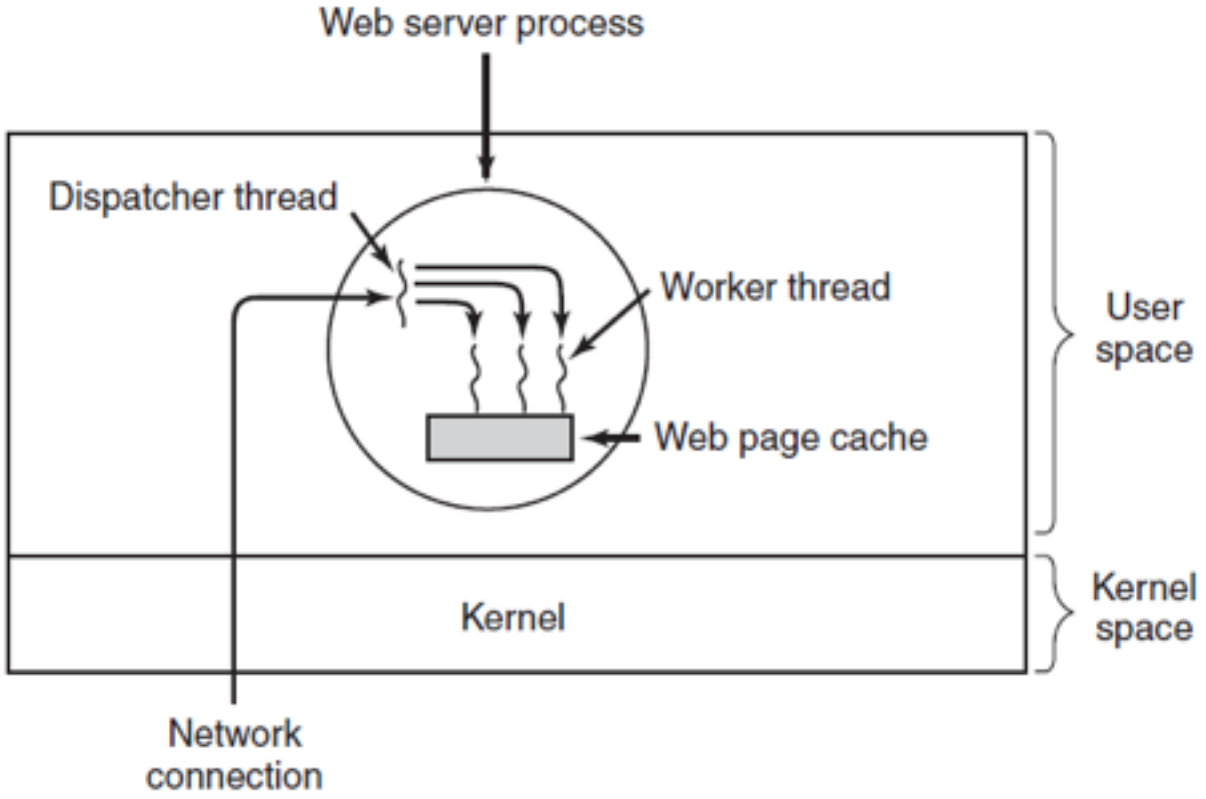

Figure 2-8. A multithreaded Web server.

Thread Usage (7)

- This model allows the server to be written as a collection of sequential threads (Fig. 2-9)
- The dispatcher's program consists of an infinite loop for getting a work request and handing it off to a worker
- Each worker's code consists of an infinite loop consisting of accepting a request from the dispatcher and checking the cache to see if the page is present
  - If so, it is returned to the client, and the worker blocks waiting for a new request
  - If not, it gets the page from the disk, returns it to the client, and blocks waiting for a new request

Thread Usage (8)

```
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)

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

Thread Usage (9)

- There are three ways to construct the web-server:
  - Using threads as it was just described
  - Operating as a single thread. All the requests would be processed sequentially. CPU would be idle while the web-server would be waiting for the disk
  - Using nonblocking system calls. When a page has to be read from disk, the server **records the state of the current request in a table**

Thread Usage (10)

- (nonblocking system calls, cont-ed):
  - then it gets the next event which may either be a request for new work or a reply from the disk about a previous operation.
  - If it is new work, that work is started. If it is a reply from the disk, the relevant information is fetched from the table and the reply processed.
  - A design in which each computation has a saved state, and there exists some set of events that can occur to change the state, is called a **finite-state machine**

Thread Usage (11)

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

Figure 2-10. Three ways to construct a server.

The Classical Thread Model (1)

- The process model is based on two independent concepts: resource grouping and execution
- Sometimes it is useful to separate them; this is where threads come in
- First we will look at the classical thread model; after that we will examine the Linux thread model, which blurs the line between processes and threads

The Classical Thread Model (2)

- A process has an address space containing program text and data, as well as other resources
- These resources may include:
  - open files
  - child processes
  - pending alarms
  - signal handlers
  - accounting information, and more
- By putting them together in the form of a process, they can be managed more easily

The Classical Thread Model (3)

- The other concept a process has is a **thread of execution** or just **thread**
- The thread has:
  - a program counter that keeps track of which instruction to execute next
  - registers, which hold its current working variables
  - a stack, which contains the execution history, with one frame for each procedure called but not yet returned from
- Although a thread must execute in some process, the thread and its process are different concepts and can be treated separately
- Processes are used to group resources together
- Threads are the entities scheduled for execution on the CPU

The Classical Thread Model (4)

- Because threads have some of the properties of processes, they are sometimes called **lightweight processes**
- The term **multithreading** is also used to describe the situation of allowing multiple threads in the same process
- Some CPUs have direct hardware support for multithreading and allow thread switches to happen in nanoseconds

The Classical Thread Model (5)

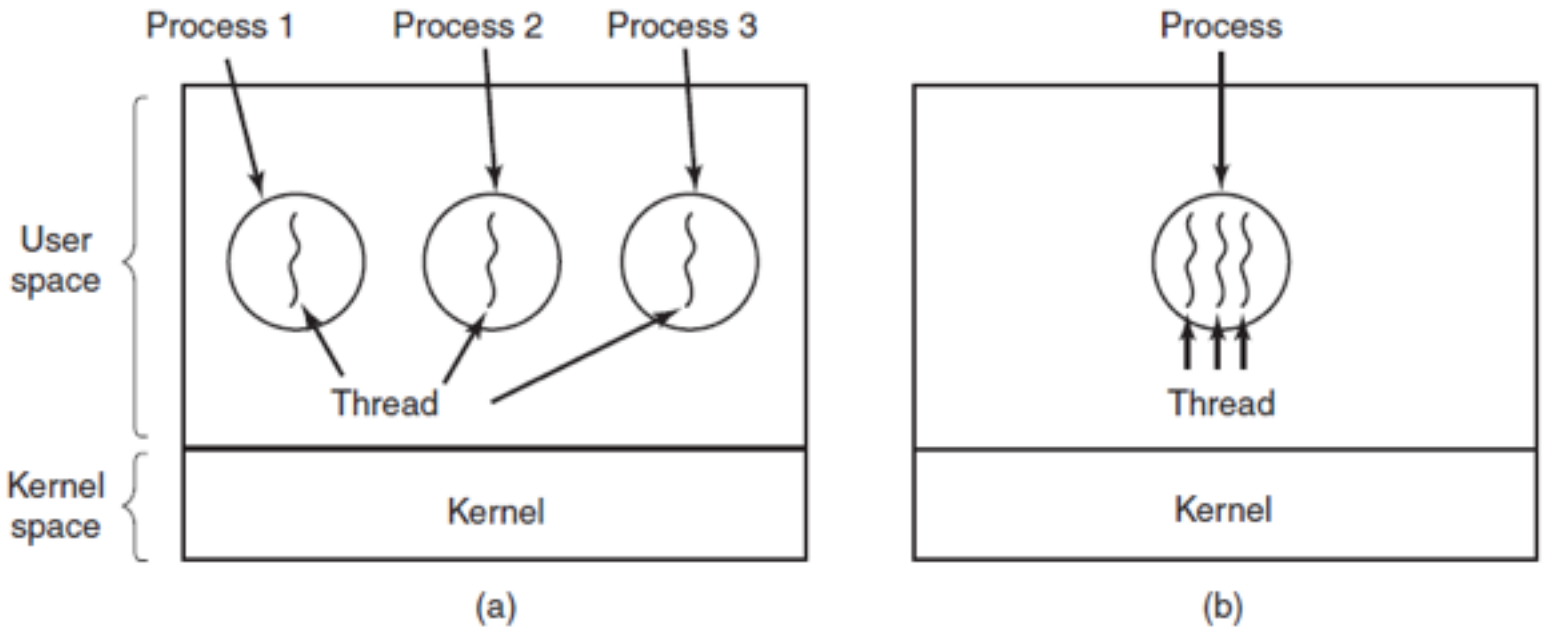

Figure 2-11. (a) Three processes each with one thread. (b) One process with three threads.

The Classical Thread Model (6)

- Different threads in a process are not as independent as different processes
- All threads have exactly the same address space and share the same global variables
- Since every thread can access every memory address within the process' address space, one thread can read, write, or even wipe out another thread's stack
- A process is always owned by a single user, who has presumably created multiple threads so that they can cooperate, not fight.
- All the threads can share the same set of open files, child processes, alarms, and signals, an so on (Fig. 2-12)

The Classical Thread Model (7)

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 (8)

- Like a traditional single-threaded process, a thread can be in any one of several states:
  - A **running** thread currently has the CPU and is active
  - A **blocked** thread is waiting for some event to unblock it
  - A **ready** thread is scheduled to run and will as soon as its turn comes up
  - Thread can be also **terminated**
- The transitions between thread states are the same as those between process states

The Classical Thread Model (9)

- Each thread has its own stack (Fig. 2-13) that contains one frame for each procedure called but not yet returned from
- A frame contains the procedure's local variables and the return address to use when the procedure call has finished
- For example, if procedure *X* calls procedure *Y* and *Y* calls procedure *Z*, then while *Z* is executing, the frames for *X*, *Y*, and *Z* will all be on the stack

The Classical Thread Model (10)

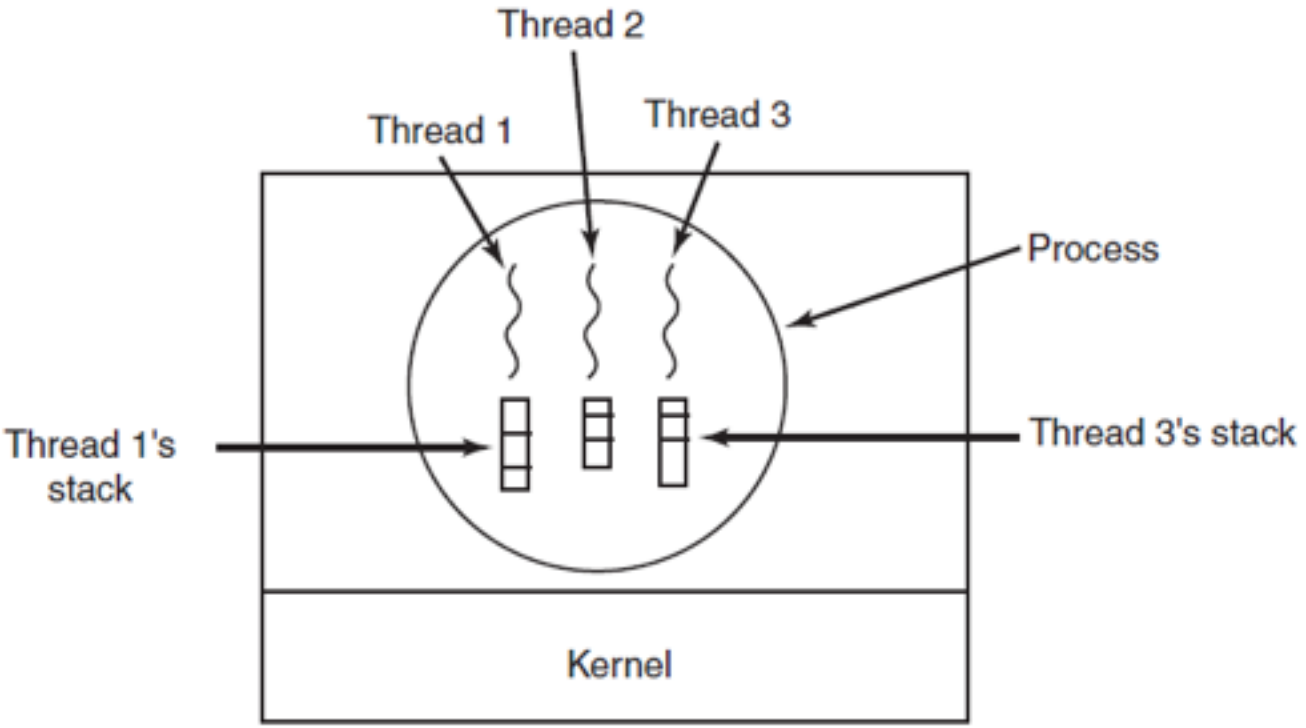

Figure 2-13. Each thread has its own stack.

The Classical Thread Model (11)

- Processes usually start with a single thread that is able to create new threads by calling a library procedure such as ***thread_create***
- A parameter to *thread_create* specifies the name of a procedure for the new thread to run
- New thread automatically runs in the address space of the creating thread
- With or without a hierarchical relationship, the creating thread is usually returned a thread identifier that names the new thread

The Classical Thread Model (12)

- When a thread has finished its work, it can exit by calling a library procedure ***thread_exit***. It then vanishes and is no longer schedulable
- In some thread systems, one thread can wait for a (specific) thread to exit by calling a procedure, for example, ***thread_join***. This procedure blocks the calling thread until a (specific) thread has exited
- In this regard, thread creation and termination is very much like process creation and termination, with approximately the same options as well

The Classical Thread Model (13)

- Another common thread call is ***thread_yield***, which allows a thread to voluntarily give up the CPU to let another thread run
- Such a call is important because there is no clock interrupt to actually enforce multiprogramming as there is with processes
- Thus it is important for threads to voluntarily surrender the CPU from time to time to give other threads a chance to run
- Other calls allow one thread to wait for another thread to finish some work, for a thread to announce that it has finished some work, and so on

The Classical Thread Model (14)

- Using threads brings a number of complications into the programming model:
  - If the parent process has multiple threads, should the child also have them?
  - What happens if a thread in the parent was blocked on a *read* call from the keyboard?
  - When a line is typed, do both threads get a copy of it?
  - What happens if one thread closes a file while another one is still reading from it?

POSIX Threads (1)

- To make it possible to write portable threaded programs, IEEE has defined a standard for threads in (IEEE Std 1003.1c)
- The threads package it defines is called **Pthreads**
- Most UNIX systems support it. The standard defines over 60 function calls. We will study a few of the major ones (Fig. 2-14)
- All Pthreads threads have certain properties:
  - an identifier,
  - a set of registers (including the program counter)
  - a set of attributes, such as the stack size and scheduling parameters which are stored in a structure

POSIX Threads (2)

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 (3)

- Here is an example of how Pthreads work:

```
#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);
}
```

POSIX Threads (4)

```
int main(int argc, char *argv[]) {
  /* The main program creates 10 threads and exits. */
  pthread tthreads[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);

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

Making Single-Threaded Code Multithreaded (1)

- Many existing programs were written for single-threaded processes. Converting these to multithreading is tricky. Some of the pitfalls are:
  - having to deal with the variables that a global to the process (Fig. 2-11)
  - many library procedures are not reentrant
  - dealing with non-thread-specific signals (or even with thread-specific signals in user space)
  - stack management

Making Single-Threaded Code Multithreaded (2)

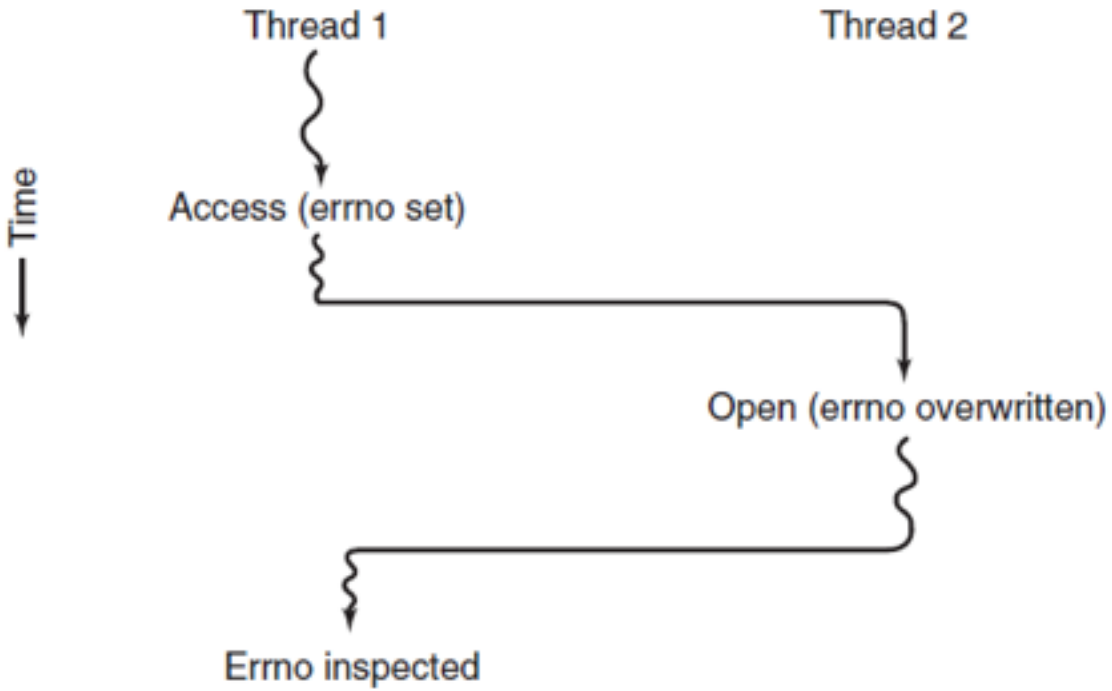

Figure 2-19. Conflicts between threads over the use of a global variable.

Making Single-Threaded Code Multithreaded (3)

- One of the solutions to the first problem is to prohibit global variables
- Another one is to introduce private global variables specific for each thread (Fig. 2-20)
- The third one is to create special library procedures for creating, setting and reading threadwide global variables, such as:
  - create_global(“bufptr");
  - set_global(“bufptr", &buf);
  - bufptr = read_global(“bufptr");

Making Single-Threaded Code Multithreaded (4)

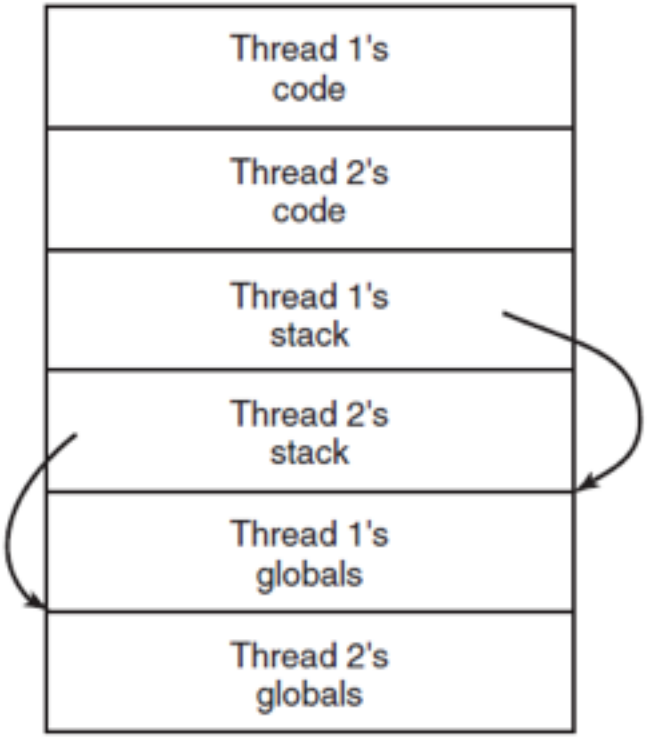

Figure 2-20. Threads can have private global variables.

Making Single-Threaded Code Multithreaded (5)

- The other problems are harder to deal with
- For example, to solve the second problem, one would need to rewrite the whole library or use a jacket (wrapper) that sets a bit to mark the library as in use and will not allow to call a procedure while this bit is set. Such an approach greatly eliminates potential parallelism
- All of these problems are more or less manageable, but they do show that just introducing threads into an existing system without a fairly substantial system redesign will not work at all

End

Week 04 – Lecture 2

References

- Tanenbaum & Bo, Modern Operating Systems: 4th edition, 2013 Prentice-Hall, Inc.