MODERN OPERATING SYSTEMS
Third Edition

ANDREW S. TANENBAUM

Chapter 2
Processes and Threads

Process Concept

- **Process** – a program in execution; process execution must progress in sequential fashion
- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

The Process Model

Multiprogramming: rapid back and forth switching of a processor among multiple processes is called **multiprogramming**

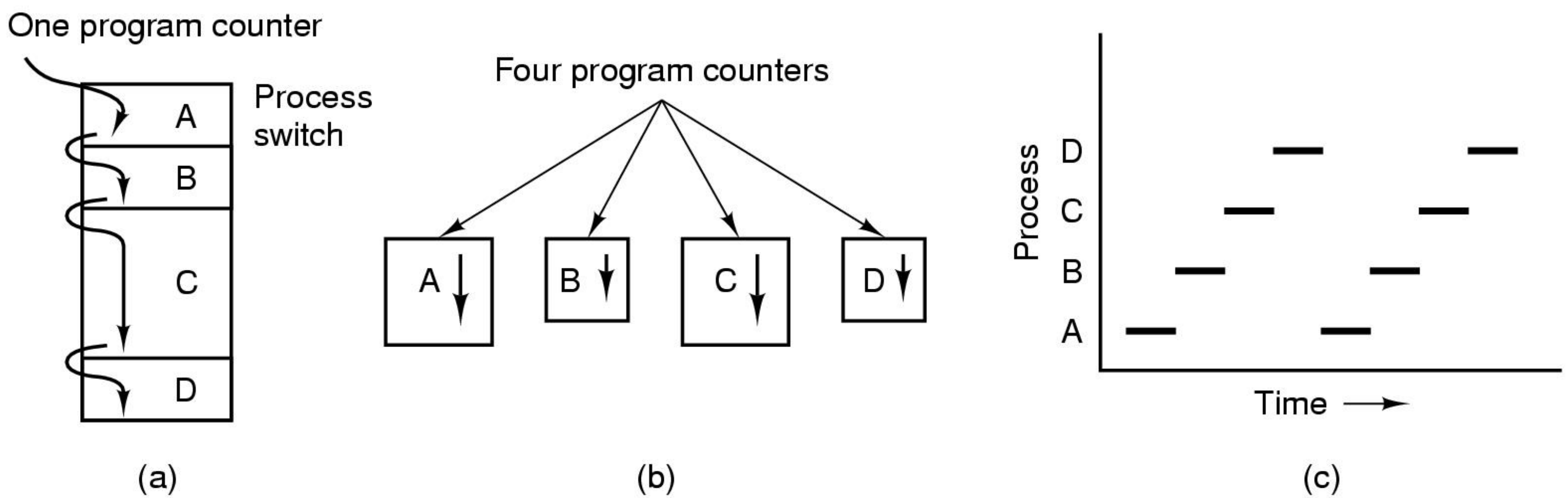


Figure 2-1. (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

Process Creation

Events which cause process creation:

- System initialization.
- Execution of a process-creation system call by a running process (creating a child process)
- A user request to create a new process.

Process Termination

Events which cause process termination:

- **Normal exit (voluntary):** Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Process' resources are deallocated by operating system

- **Fatal error (involuntary):** The second reason for termination is that the process discovers a fatal error. For example, if a user types the command ***cc foo.c*** to compile the program ***foo.c*** *and no such file exists, the compiler simply* announces this fact and exits.

Process Termination contd…

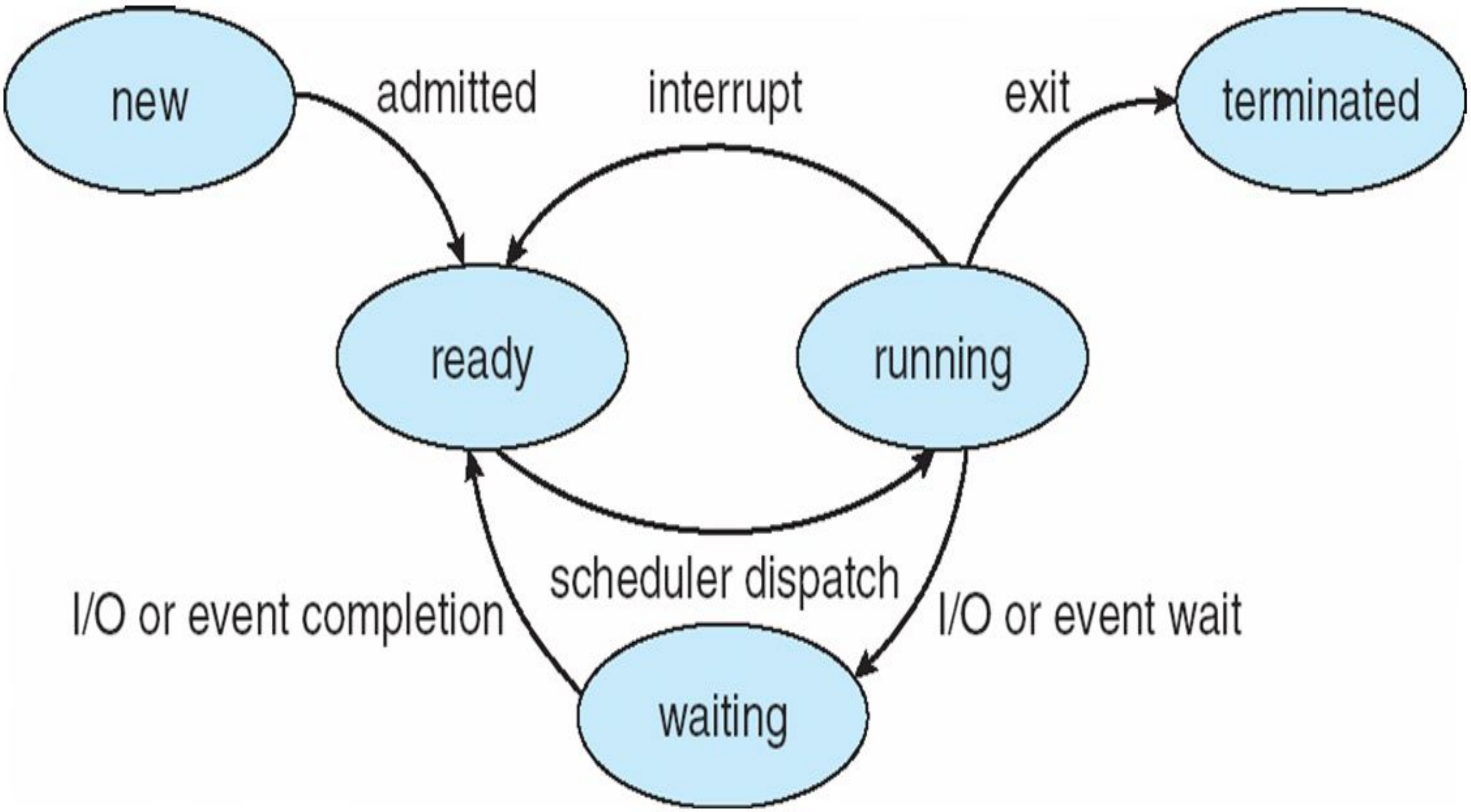
- **Error exit (voluntary):** The third reason for termination is an error caused by the process, often due to a program bug

- **Killed by another process (involuntary):** Parent may terminate the execution of children processes using the **abort()** system call. Some reasons for doing so:
    i. Child has exceeded allocated resources
    ii. Task assigned to child is no longer required
    iii. The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Process State

- As a process executes, it changes **state**
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **Waiting/blocked**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution

Diagram of Process State



Implementation of Processes (1)

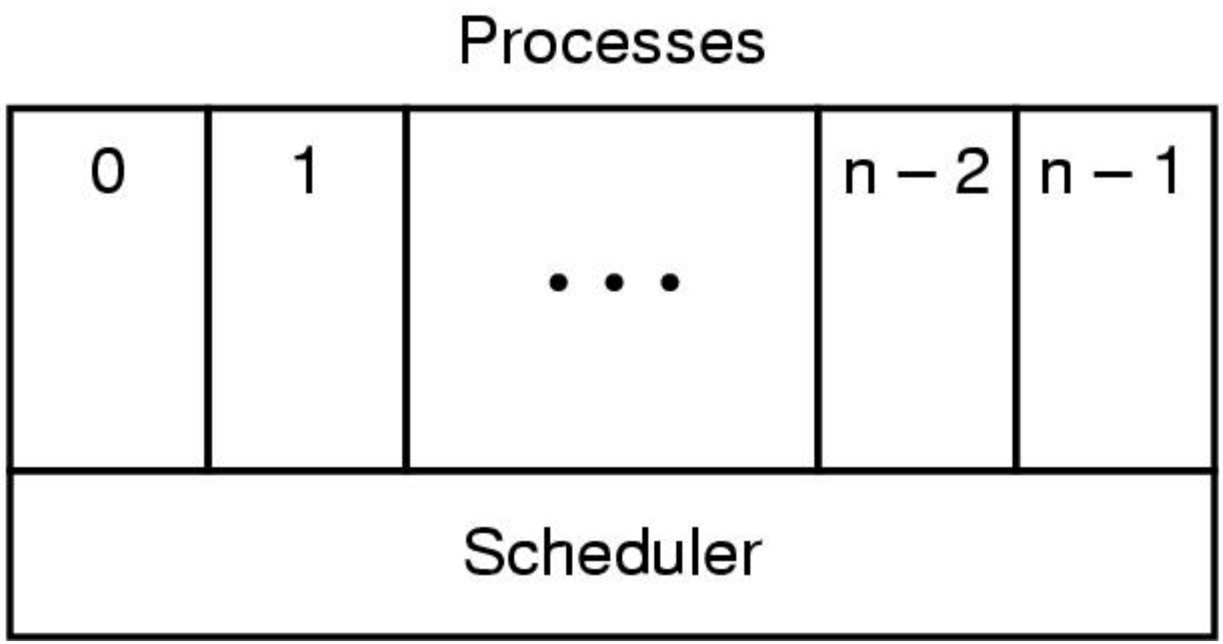


Figure 2-3. The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.

Implementation of Processes (2)

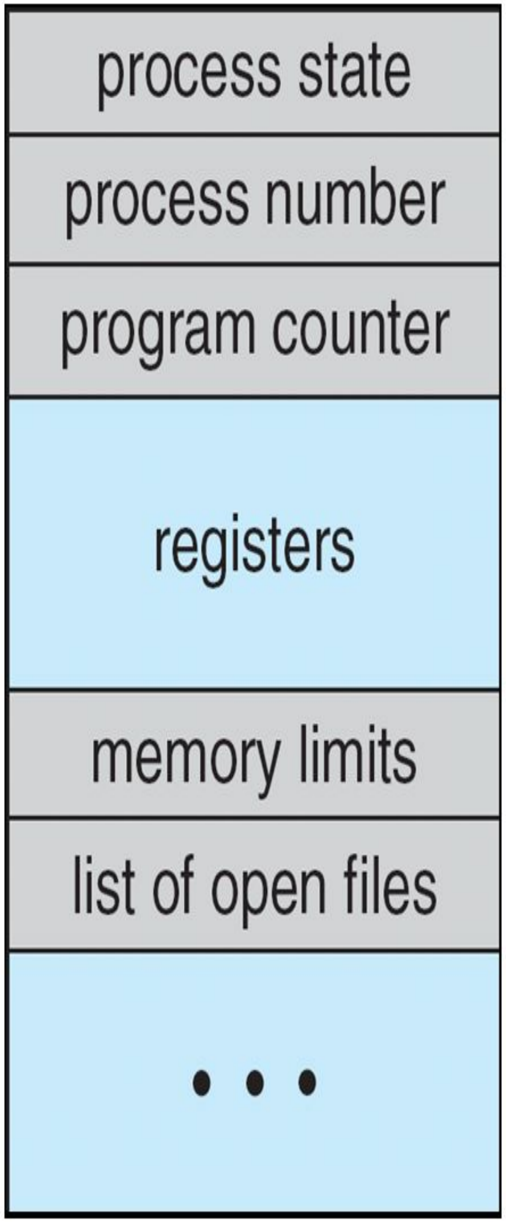
- To implement the process model, the operating system maintains a table (an array of structures), called the **process table**, with one entry per process. (Some authors call these entries process control blocks.)

Implementation of Processes (3)

Information associated with each process

(also called **task control block**)

- Process state – running, waiting, etc.
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



Implementation of Processes (4)

Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children’s CPU time		
Time of next alarm		

Figure 2-4. Some of the fields of a typical process table entry.

Threads

- A kind of a mini-process within a process, these mini-processes, called **threads.**

- Processes are used to group resources together; threads are the entities scheduled for execution on the CPU.

- Because threads have some of the properties of processes, they are sometimes called **lightweight processes.**

Thread Usage (1)

Simplicity: decomposing an application into multiple sequential threads that run in quasi-parallel, the programming model becomes simpler.

Light weighted: A second argument for having threads is that since they are lighter weight than processes, they are easier (i.e., faster) to create and destroy than processes.

Performance: Having substantial computing and also substantial I/O, having threads allows these activities to overlap, thus speeding up the application.

Parallelism: threads are useful on systems with multiple CPUs, where real parallelism is possible.

Thread Usage (2)

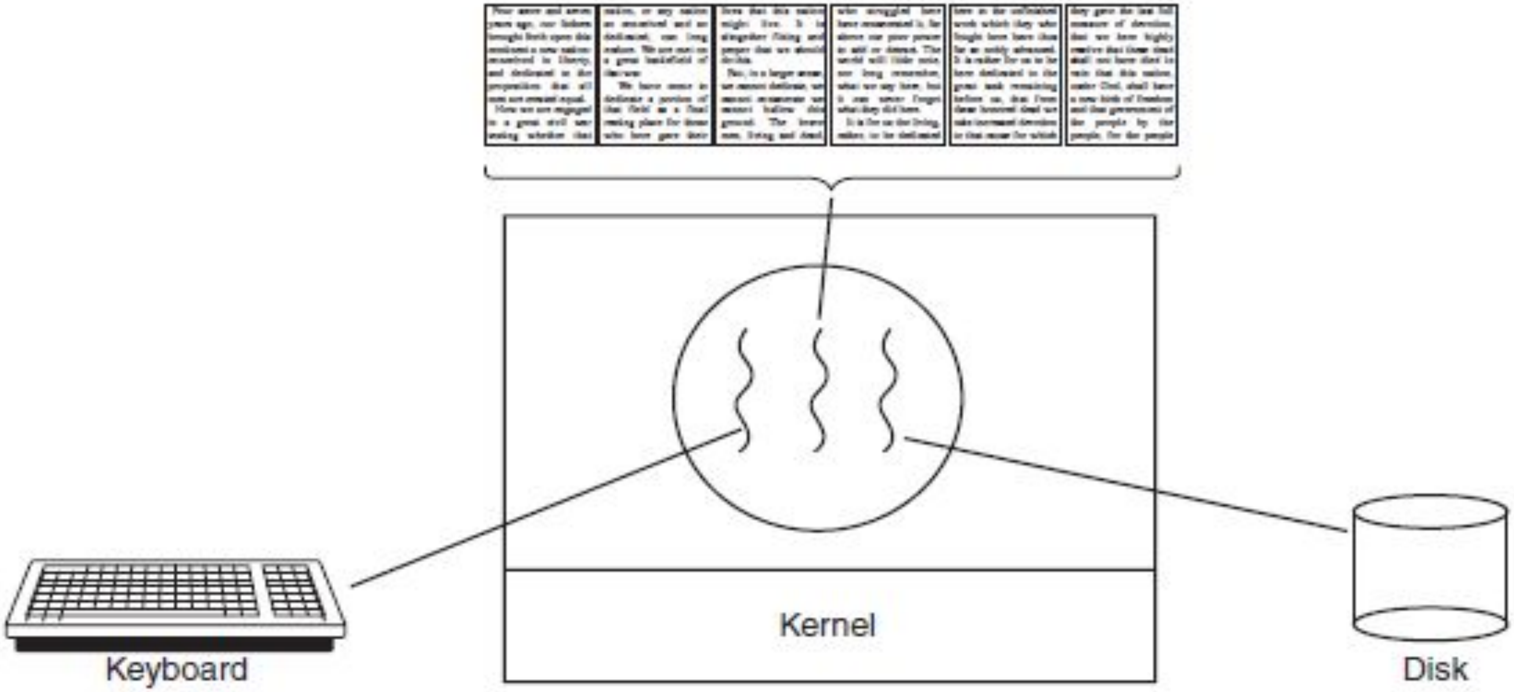


Figure 2-7. A word processor with three threads.

Thread Usage (3)

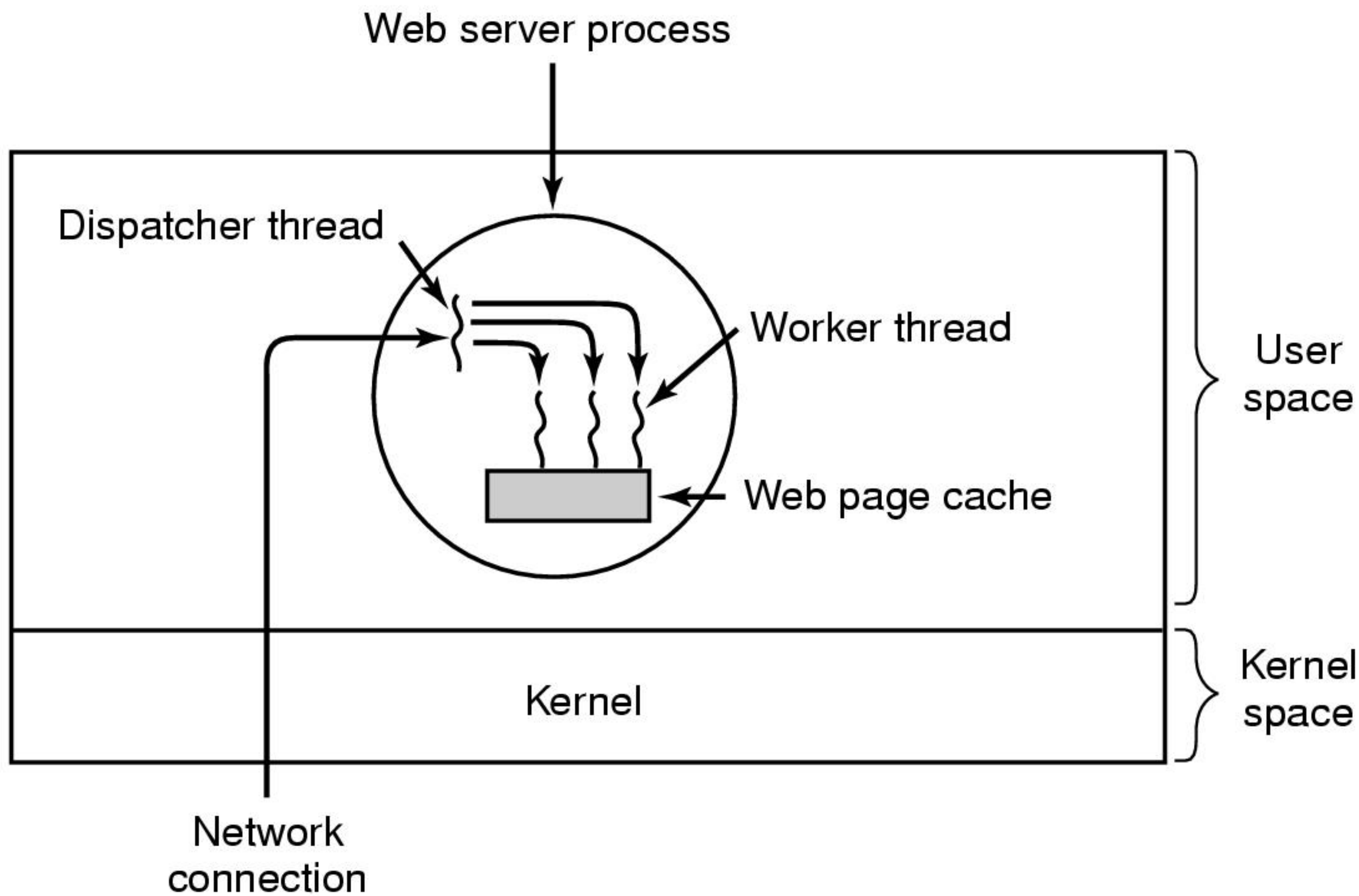


Figure 2-8. A multithreaded Web server.

The Classical Thread Model (1)

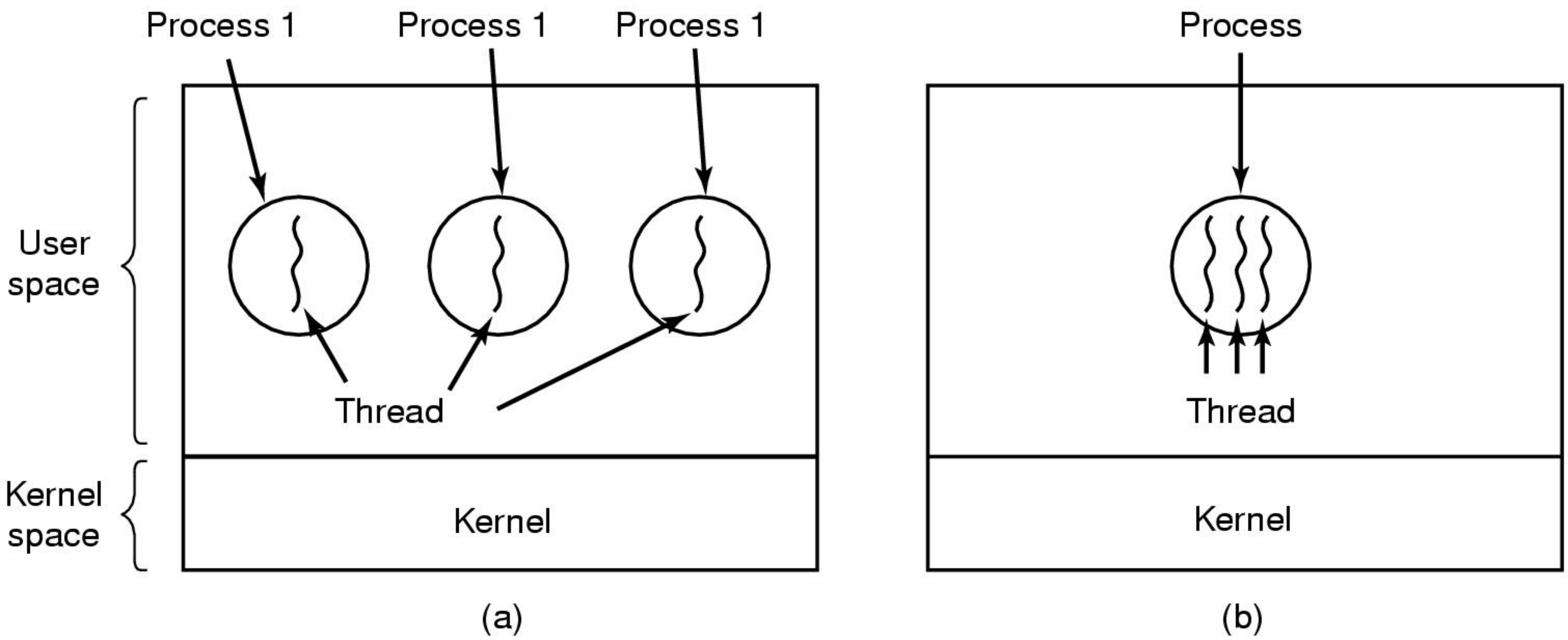


Figure 2-11. (a) Three processes each with one thread. (b) One process with three threads.

The Classical Thread Model (2)

- The thread has a program counter that keeps track of which instruction to execute next.
- It has registers, which hold its current working variables.
- It has a stack, which contains the execution history.

The Classical Thread Model (3)

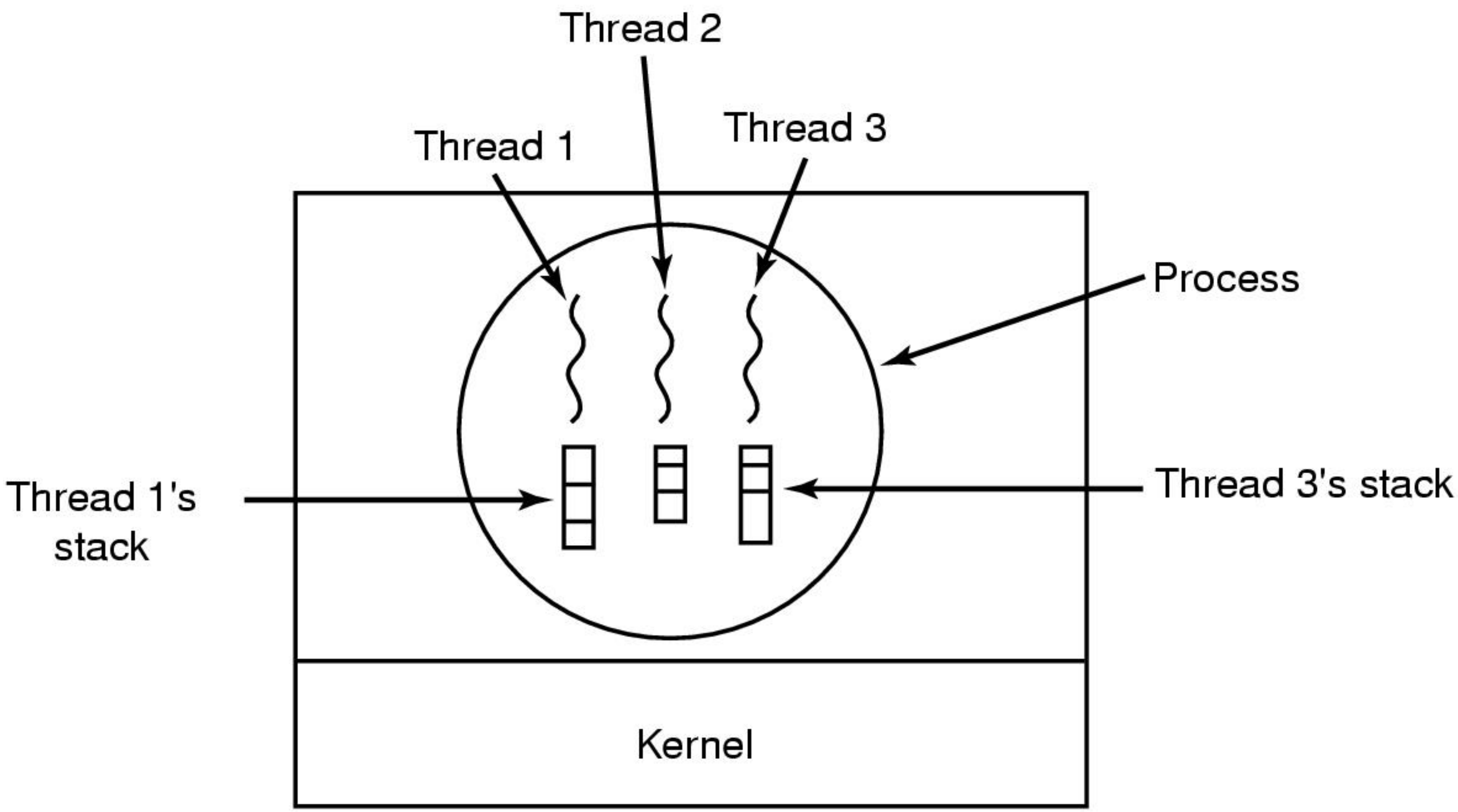


Figure 2-13. Each thread has its own stack.

The Classical Thread Model (4)

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.

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.

Implementing Threads in User Space

- This method is to put the threads package entirely in user space.

- The kernel knows nothing about them. As far as the kernel is concerned, it is managing ordinary, single-threaded processes.

- The first, and most obvious, advantage is that a user-level threads package can be implemented on an operating system that does not support threads.

- With this approach, threads are implemented by a library.

Implementing Threads in the Kernel

- Consider having the kernel know about and manage the threads.
- No run-time system is needed in each process.
- There is no thread table in each process.
- The kernel has a thread table that keeps track of all the threads in the system.
- When a thread wants to create a new thread or destroy an existing thread, it makes a kernel call, which then does the creation or destruction by updating the kernel thread table.

Implementing Threads in User/Kernal Space

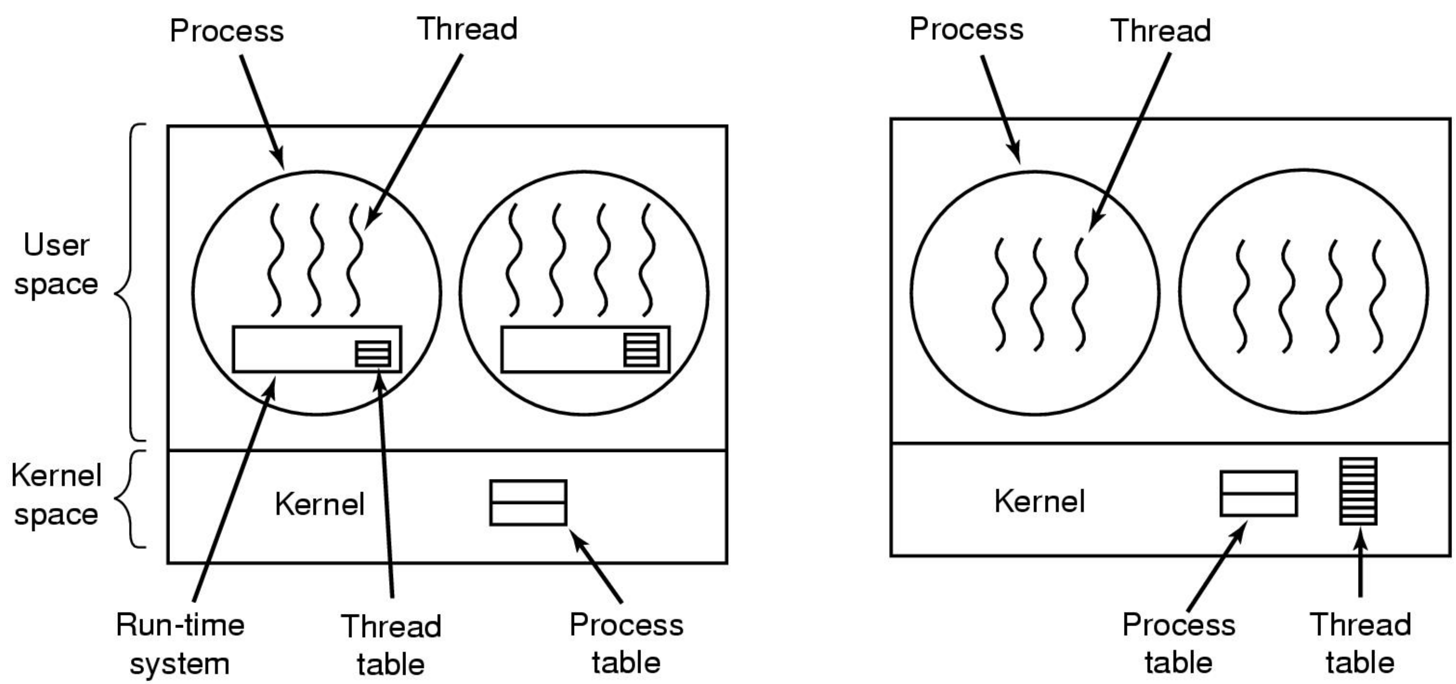


Figure 2-16. (a) A user-level threads package. (b) A threads package managed by the kernel.

Hybrid Implementations

- One way is use kernel-level threads and then multiplex user-level threads onto some or all of them.

- The programmer can determine how many kernel threads to use and how many user-level threads to multiplex on each one.

- The kernel is aware of *only the kernel-level threads and* schedules those.

- Some of those kernel threads may have multiple user-level threads multiplexed on top of them.

Hybrid Implementations

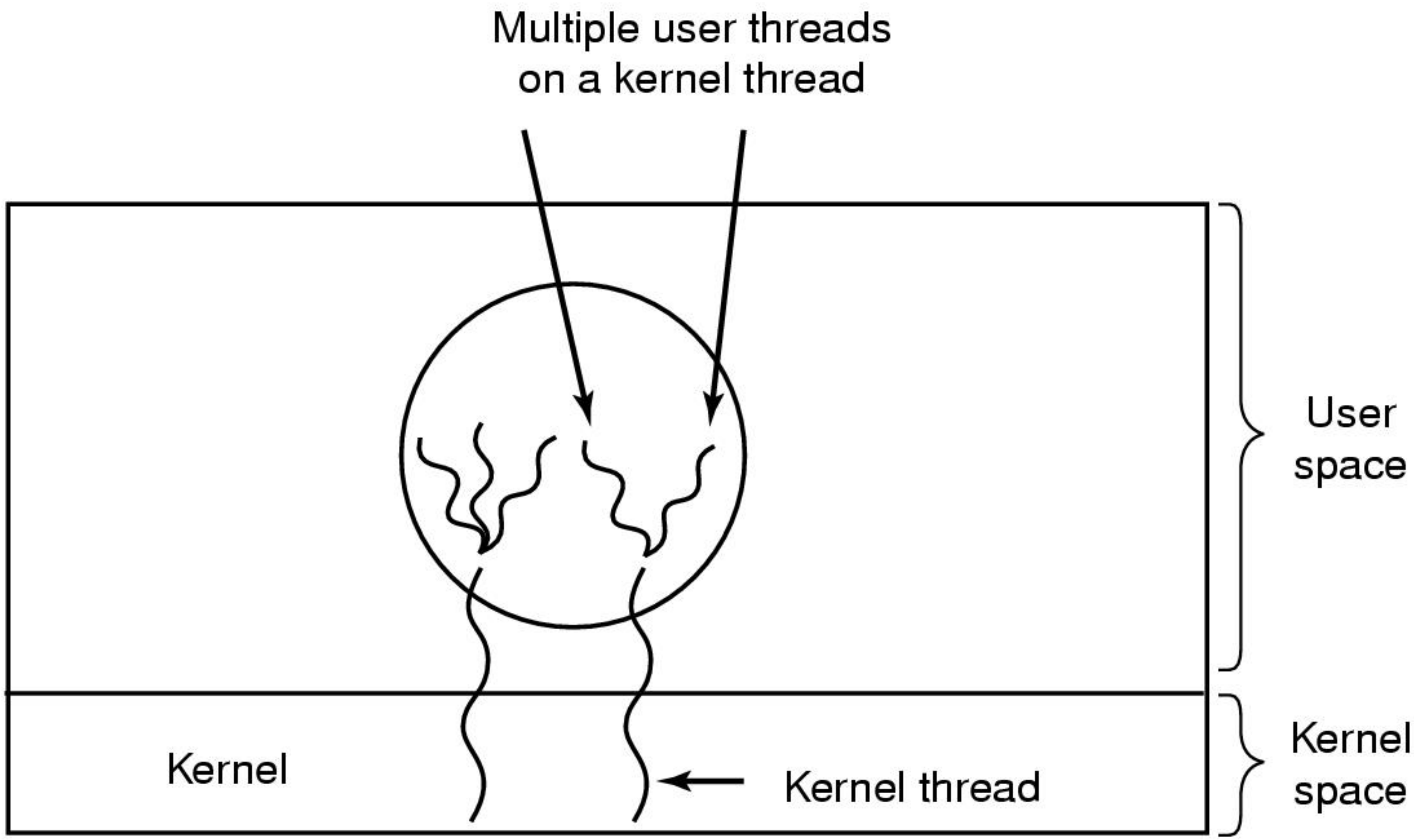


Figure 2-17. Multiplexing user-level threads onto kernel-level threads.

Pop-Up Threads

- Threads are frequently useful in distributed systems. An important example is how incoming messages, for example requests for service, are handled.

- The arrival of a message causes the system to create a new thread, called the Pop-Up Thread, to handle the message.

- Advantage: since they are brand new, they do not have any history—registers, stack etc.

- Each one starts out fresh and each one is identical to all the others & makes it possible to create such a thread quickly.

- The new thread is given the incoming message to process & the latency between message arrival and the start of processing can be made very short.

Pop-Up Threads

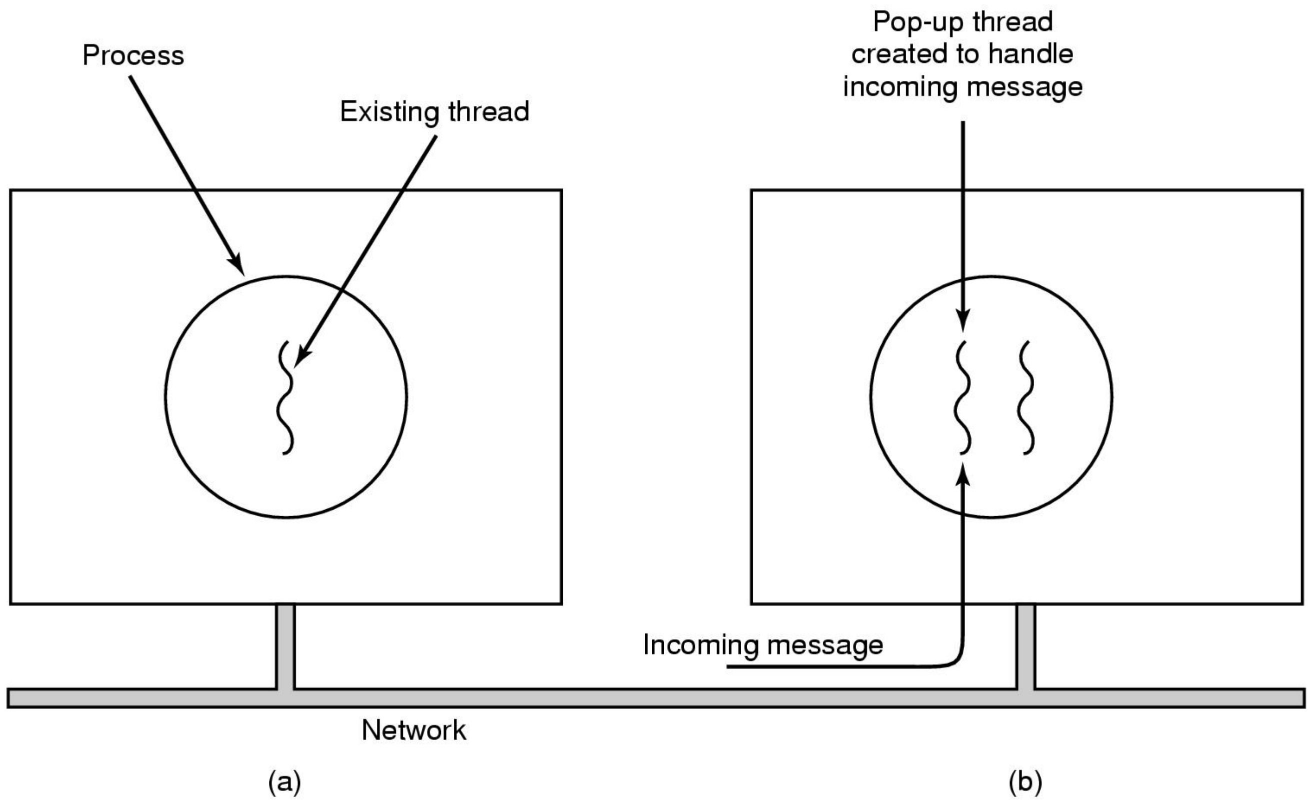


Figure 2-18. Creation of a new thread when a message arrives.
(a) Before the message arrives.
(b) After the message arrives.

Race Conditions

- A race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run.

- It is also defined as; an execution ordering of concurrent flows that results in undesired behavior is called a race condition-a software defect and frequent source of vulnerabilities.

Race Conditions

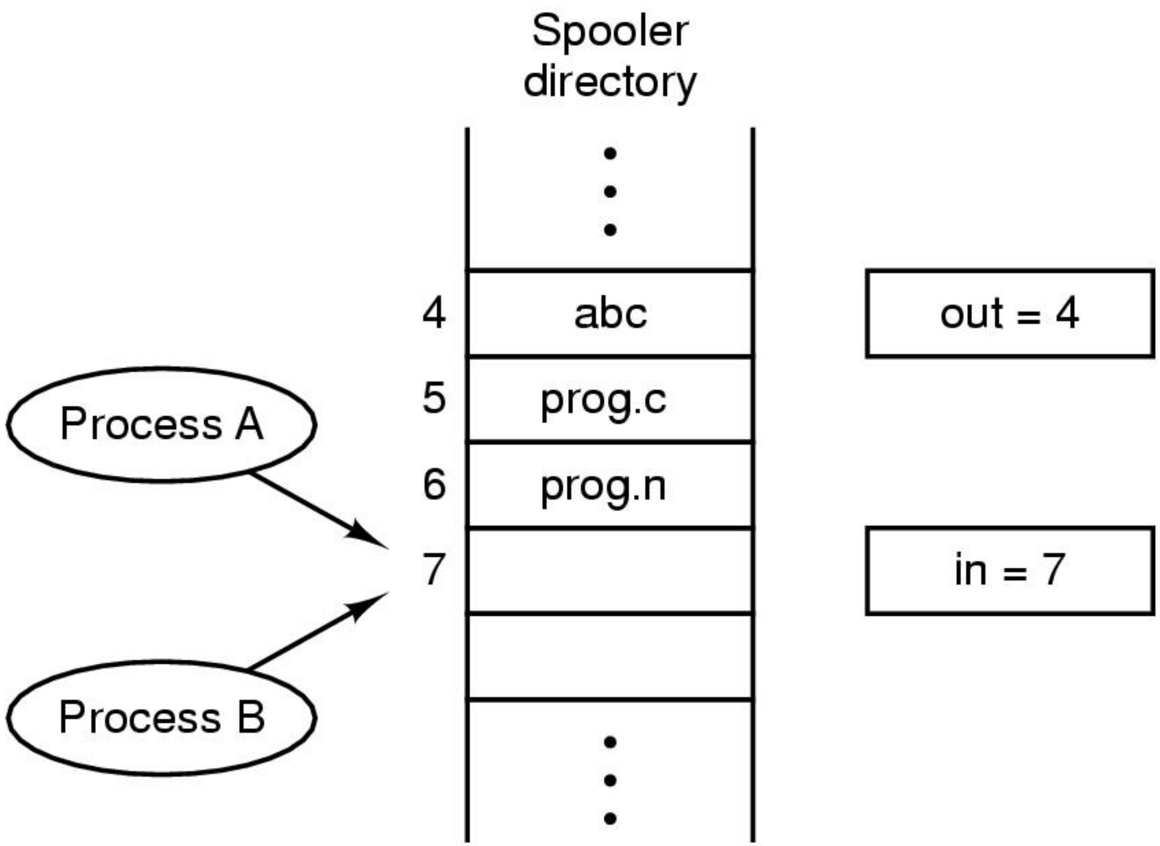


Figure 2-21. Two processes want to access shared memory at the same time.

Critical Regions (1)

Conditions required to avoid race condition:

- Mutual Exclusion: No two processes may be simultaneously inside their critical regions.
- No assumptions may be made about speeds or the number of CPUs.
- No process running outside its critical region may block other processes.
- No process should have to wait forever to enter its critical region.

Critical Regions (2)

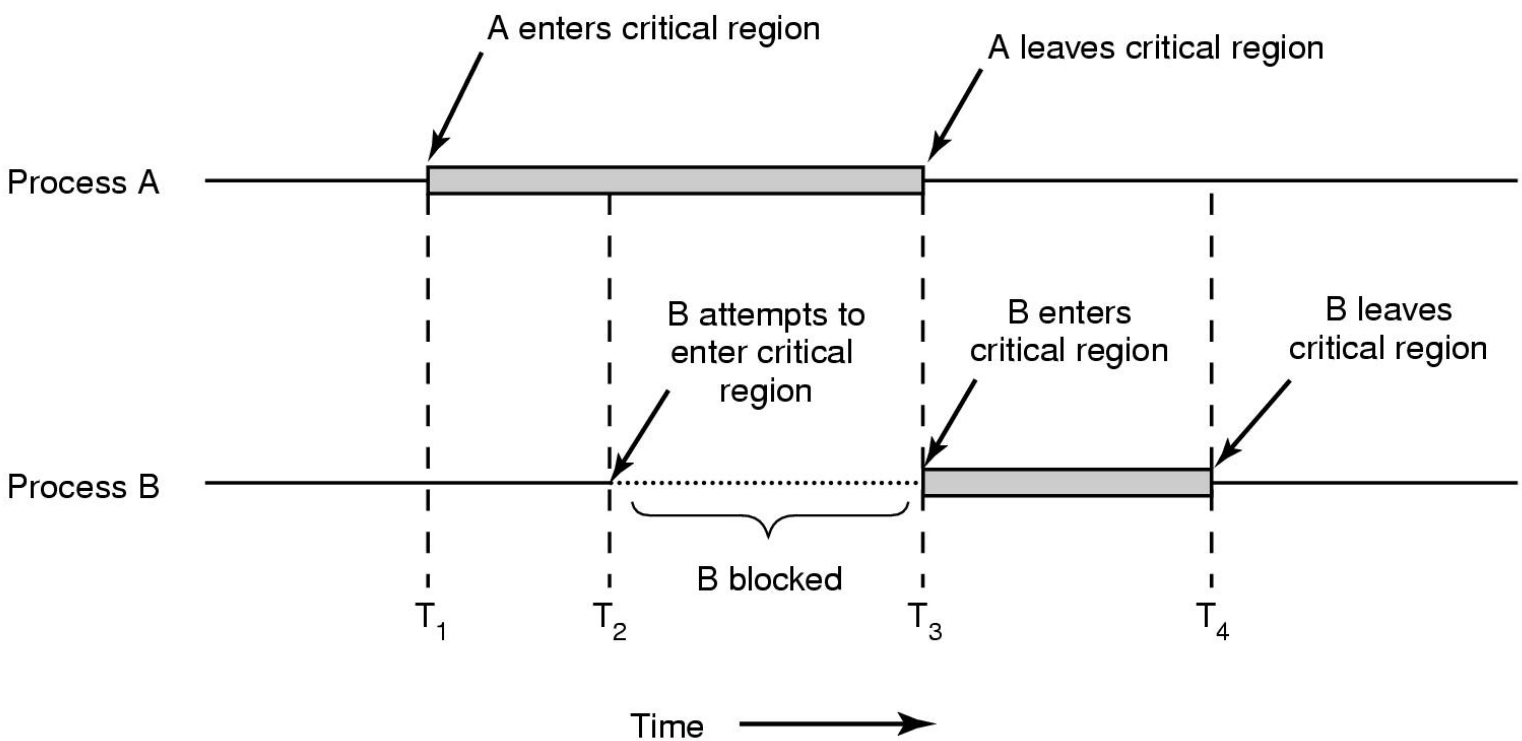


Figure 2-22. Mutual exclusion using critical regions.

Mutual Exclusion with Busy Waiting

Proposals for achieving mutual exclusion:

Disabling Interrupts: The simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it. Hence CPU will not be switched to another process.

Lock Variable: A software solution with variable 0 and 1. When a process wants to enter its critical region, it first tests the lock. If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0.

Mutual Exclusion with Busy Waiting

Strict Alteration: The integer variable *turn*, initially 0, keeps track of whose turn it is to enter the critical region and examine or update the shared memory. Initially, process 0 inspects *turn*, finds it to be 0, and enters its critical region. Process 1 also finds it to be 0 and therefore sits in a tight loop continually testing turn to see when it becomes 1. Continuously testing a variable until some value appears is called busy waiting that should be avoid as it waste the CPE time.

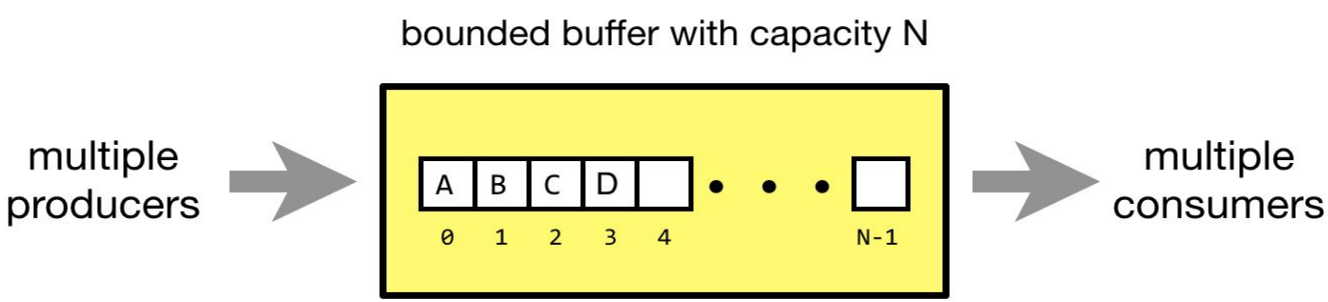
<u>READING ASSIGNMENT:</u>

Peterson's Solution

The TSL Instruction

Sleep and Wakeup: The Producer-Consumer Problem

- The bounded-buffer problems (aka the producer-consumer problem) is a classic example of concurrent access to a shared resource. A bounded buffer lets multiple producers and multiple consumers share a single buffer. Producers write data to the buffer and consumers read data from the buffer.
- Producers must block if the buffer is full.
- Consumers must block if the buffer is empty.



Synchronization Problem

- A bounded buffer with capacity N has can store N data items. The places used to store the data items inside the bounded buffer are called slots. Without proper synchronization the following errors may occur.
- The producers doesn’t block when the buffer is full.
- A Consumer consumes an empty slot in the buffer.
- A consumer attempts to consume a slot that is only half-filled by a producer.
- Two producers writes into the same slot.
- Two consumers reads the same slot.
- And possibly more …

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**

Solving the Producer-Consumer Problem Using Semaphores

- Producers must block if the buffer is full. Consumers must block if the buffer is empty. Two **counting semaphores** can be used for this.
- Use one **semaphore** named **empty** to count the empty slots in the buffer.
- **Initialise** this semaphore to **N**.

- A **producer** must **wait** on this semaphore before writing to the buffer.
- A **consumer** will **signal** this semaphore after reading from the buffer.
- Use one **semaphore** named **data** to count the number of data items in the buffer.
- **Initialise** this semaphore to **0**.
- A **consumer** must **wait** on this semaphore before reading from the buffer.
- A **producer** will **signal** this semaphore after writing to the buffer.

Mutex Locks

- As the synchronization hardware solution is not easy to implement for everyone, a strict software approach called Mutex Locks was introduced.
- In this approach, in the entry section of code, a LOCK is acquired over the critical resources modified and used inside critical section, and in the exit section that LOCK is released.
- As the resource is locked while a process executes its critical section hence no other process can access it.

Mutexes in Pthreads (1)

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Figure 2-30. Some of the Pthreads calls relating to mutexes.

Mutexes in Pthreads (2)

Thread call	**Description**
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them

Figure 2-31. Some of the Pthreads calls relating to condition variables.

Message Passing (1)

This method of inter-process communication uses two primitives, send and receive. As such, they can easily be put into library procedures, such as;

send(destination, &message);
and
receive(source, &message);

Producer-Consumer Problem with Message Passing

PROBLEM:

If the producer works faster than the consumer, all the messages will end up full, waiting for the consumer; the producer will be blocked, waiting for an empty to come back. If the consumer works faster, then the reverse happens: all the messages will be empties waiting for the producer to fill them up; the consumer will be blocked, waiting for a full message.

Producer-Consumer Problem with Message Passing

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox

Barriers

- This synchronization mechanism is intended for groups of processes.

- Some applications are divided into phases and have the rule that no process may proceed into the next phase until all processes are ready to proceed to the next phase.

- This behavior may be achieved by placing a barrier at the end of each phase.

- When a process reaches the barrier, it is blocked until all processes have reached the barrier.

- This allows groups of processes to synchronize.

Barriers

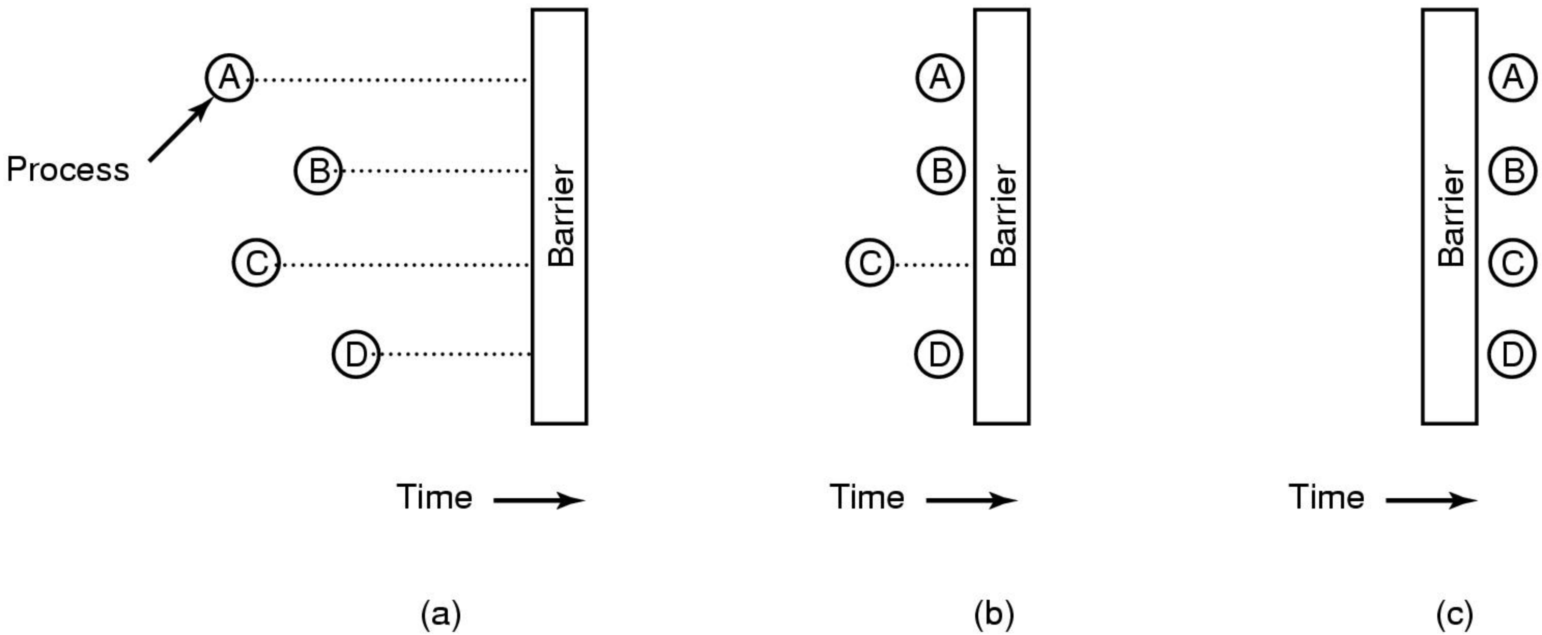


Figure 2-37. Use of a barrier. (a) Processes approaching a barrier. (b) All processes but one blocked at the barrier. (c) When the last process arrives at the barrier, all of them are let through.

Scheduling Algorithm Goals

All systems

Fairness - giving each process a fair share of the CPU
Policy enforcement - seeing that stated policy is carried out
Balance - keeping all parts of the system busy

Batch systems

Throughput - maximize jobs per hour
Turnaround time - minimize time between submission and termination
CPU utilization - keep the CPU busy all the time

Interactive systems

Response time - respond to requests quickly
Proportionality - meet users' expectations

Real-time systems

Meeting deadlines - avoid losing data
Predictability - avoid quality degradation in multimedia systems

Figure 2-39. Some goals of the scheduling algorithm under different circumstances.

Scheduling Algorithms

- First-come first-served
- Shortest job first
- Shortest remaining Time next
- Round-robin scheduling
- Priority scheduling
- Multiple queues
- Shortest process next
- Guaranteed scheduling
- Lottery scheduling
- Fair-share scheduling

ASSIGNMENT

Classical IPC Problems

- The Dining Philosophers Problem
- The Readers and Writers Problem