Chapter 6
Concurrent Processes

Understanding Operating Systems, Fourth Edition

Objectives

You will be able to describe:

- The critical difference between processes and processors, and their connection
- The differences among common configurations of multiprocessing systems
- The significance of a critical region in process synchronization
- The basic concepts of process synchronization software: test-and-set, WAIT and SIGNAL, and semaphores

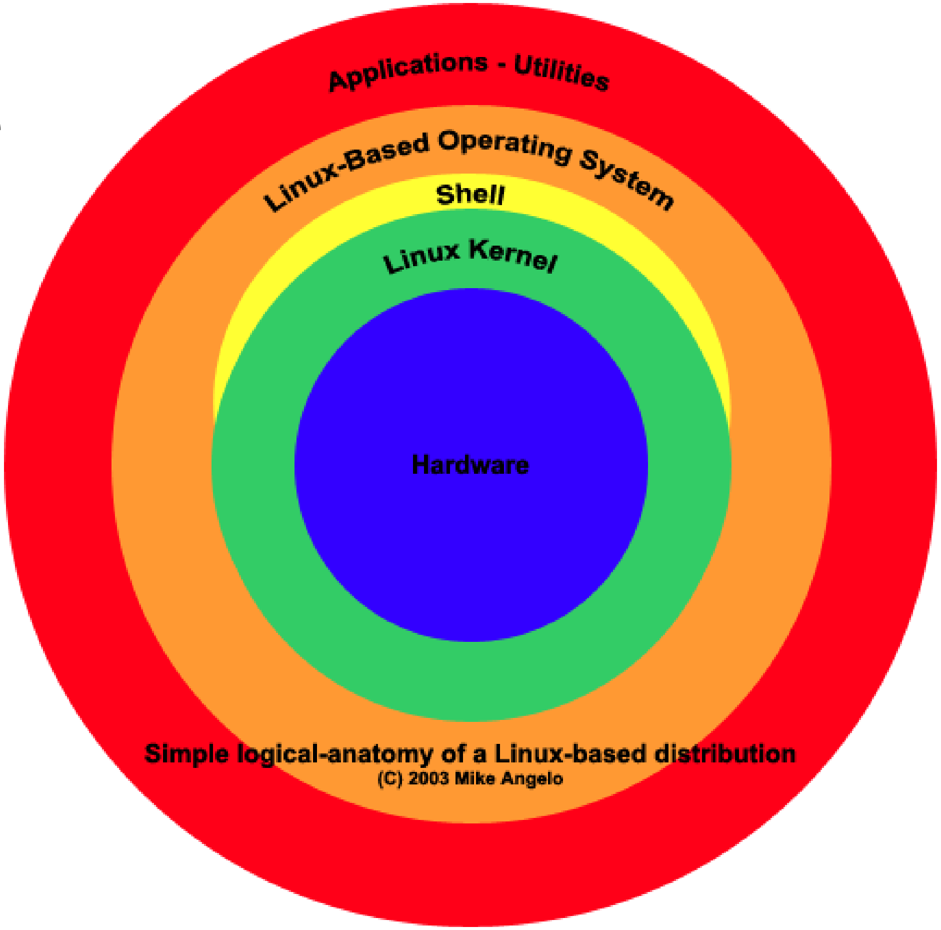
Objectives (continued)

You will be able to describe:

- The need for process cooperation when several processes work together
- How several processors, executing a single job, cooperate
- The similarities and differences between processes and threads
- The significance of concurrent programming languages and their applications

Kernel

- **Kernel** – The internal part of the operating system.
  - Those software components that perform the basic functions required by the computer.
    - File management
    - Memory management (RAM)
    - Security



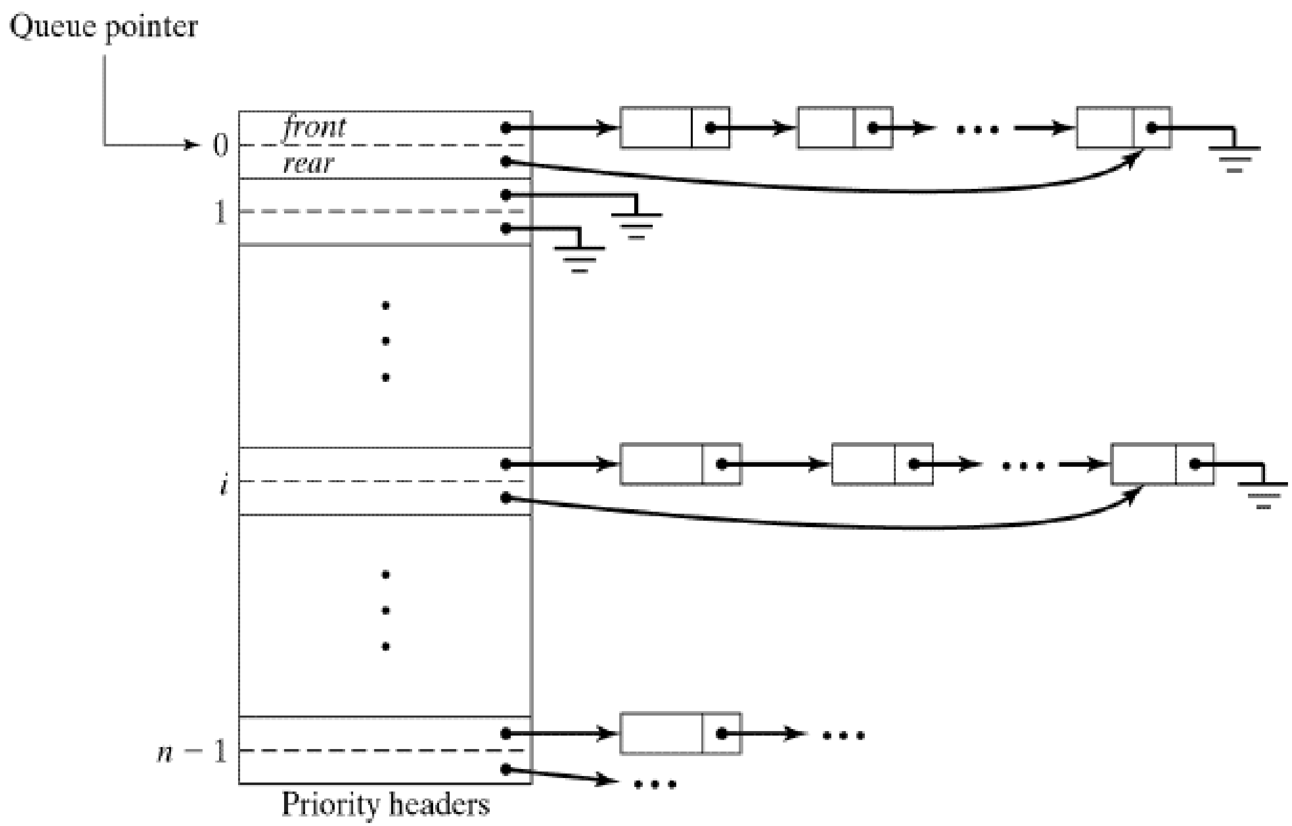
Kernel Definitions and Objects

- Basic set of objects, primitives, data structures, processes
- Rest of OS is built on top of kernel
- Kernel defines/provides ***mechanisms*** to implement various ***policies***
  - Process and thread management
  - Interrupt and trap handling
  - Resource management
  - Input/output

Queues

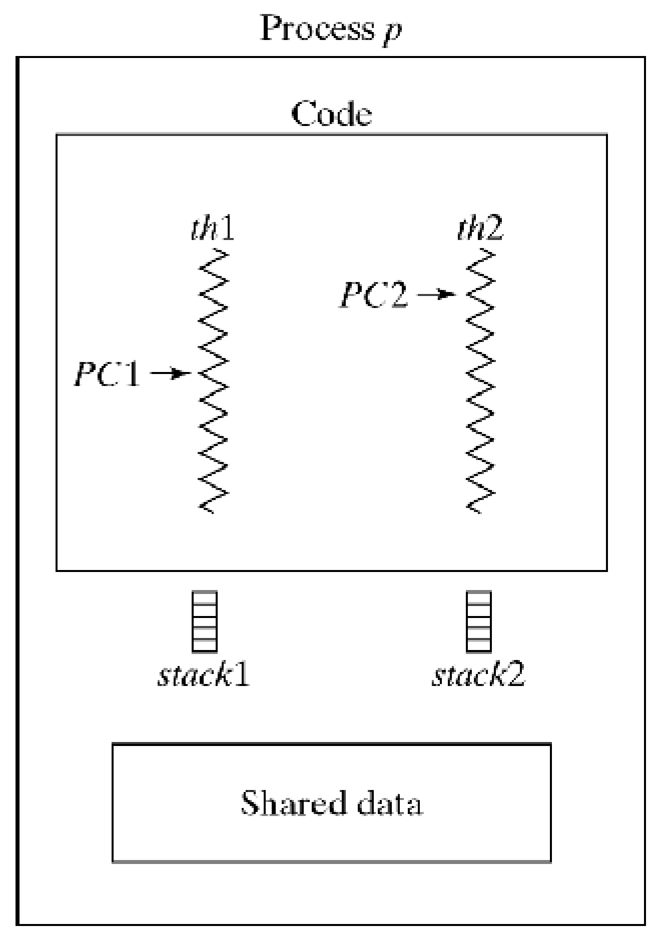
- OS needs many different queues
- Single-level queues
  - Implemented as array
    - Fixed size
    - Efficient for simple FIFO operations
  - Implemented as linked list
    - Unbounded size
    - More overhead, but more flexible operations

Priority Queues



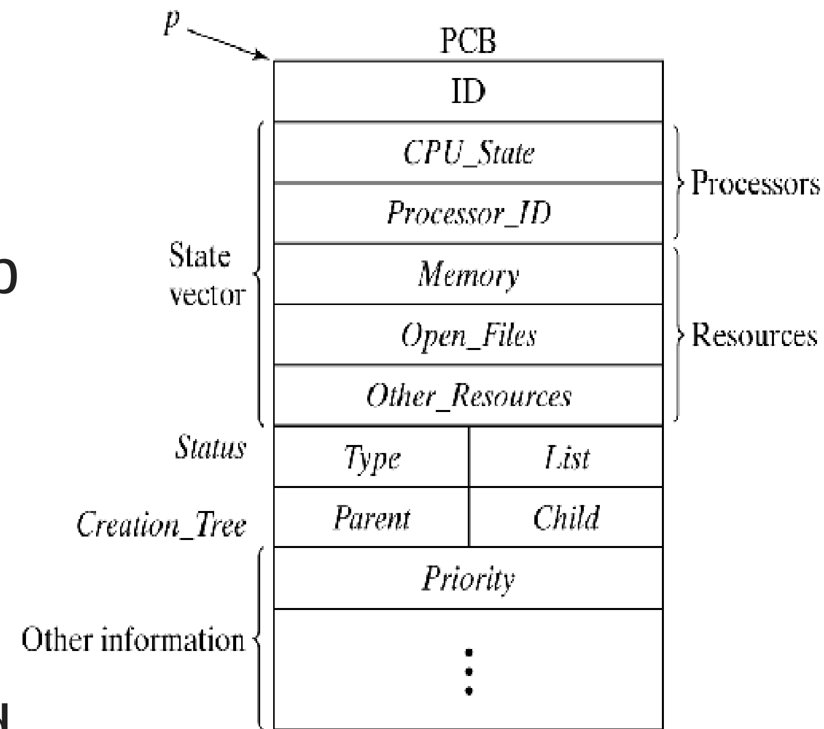
Processes and threads

- Process has one or more threads
- All threads in a process share:
  - Memory space
  - Other resources
- Each thread has its own:
  - CPU state (registers, program counter)
  - Stack
- Implemented in user space or kernel space
- Threads are efficient, but lack protection from each other



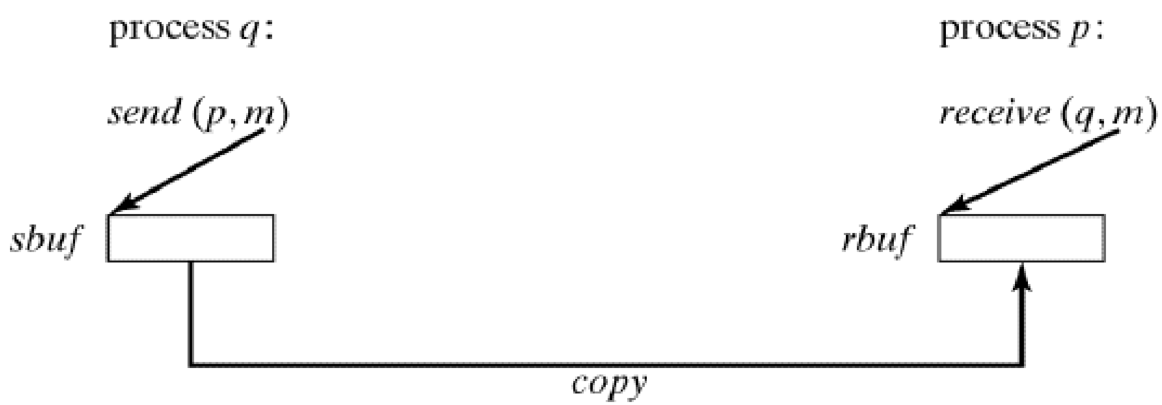
Implementing Processes/Threads

- Process Control Block (PCB)
  - State Vector = Information necessary to run process p
  - Status
    - Basic types: Running, Ready, Blocked
    - Additional types:
      - Ready_active, Ready_suspended
      - Blocked_active, Blocked_suspended



Communication Primitives

send and *receive* each use a buffer to hold message



Operating System Kernel

- “The one program running at all times on the computer” is the **kernel.** Everything else is either a system program (ships with the operating system) or an application program

Computer Startup

- **bootstrap program** is loaded at power-up or reboot
  - Typically stored in ROM or EPROM, generally known as **firmware**
  - Initializes all aspects of system
  - Loads operating system kernel and starts execution

Operating-System Operations

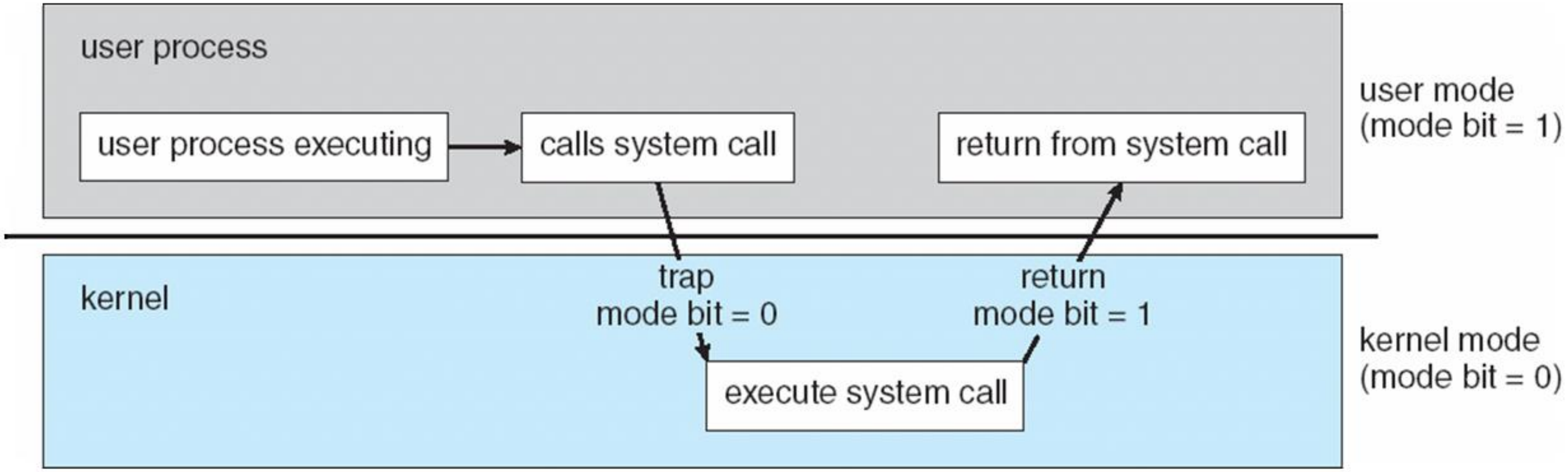
- Interrupt driven by hardware
- Software error or request creates **exception** or **trap**
  - Division by zero, request for operating system service
- Other process problems include infinite loop, processes modifying each other or the operating system

Operating-System Operations

- **Dual-mode** operation allows OS to protect itself and other system components
  - **User mode** and **kernel mode**
  - **Mode bit** provided by hardware
    - Provides ability to distinguish when system is running user code or kernel code
    - Some instructions designated as **privileged**, only executable in kernel mode
    - System call changes mode to kernel, return from call resets it to user

Transition from User to Kernel Mode

- Timer to prevent infinite loop / process hogging resources
  - Set interrupt after specific period
  - Operating system decrements counter
  - When counter zero generate an interrupt
  - Set up before scheduling process to regain control or terminate program that exceeds allotted time



What Is Parallel Processing?

- **Parallel Processing (multiprocessing):**
  - Two or more processors operate in unison, which means two or more CPUs execute instructions simultaneously
  - Processor Manager needs to coordinate the activity of each processor
  - Processor Manager needs to synchronize the interaction among the CPUs

What Is Parallel Processing? (continued)

- **Reasons for development** of parallel processing:
  - To enhance throughput
  - To increase computing power
- **Benefits** of parallel processing:
  - Increased reliability
    - If one processor fails the other can take over
  - Faster processing
    - Instructions can be processed in parallel

What Is Parallel Processing? (continued)

- **Different methods of parallel processing:**
  - CPU allocated to each program or job
  - CPU allocated to each working set or parts of it
  - Individual instructions are subdivided so each subdivision can be processed simultaneously **(concurrent programming)**
- **Two major challenges:**
  - How to connect the processors into configurations
  - How to orchestrate their interaction

Typical Multiprocessing Configurations

- **Typical Multiprocessing Configurations:**
  - Master/slave
  - Loosely coupled
  - Symmetric

Master/Slave Configuration

- An asymmetric multiprocessing system
- A single-processor system with additional slave processors, each of which is managed by the primary master processor
- Master processor is responsible for
  - Managing the entire system
  - Maintaining status of all processes in the system
  - Performing storage management activities
  - Scheduling the work for the other processors
  - Executing all control programs

Master/Slave Configuration (continued)

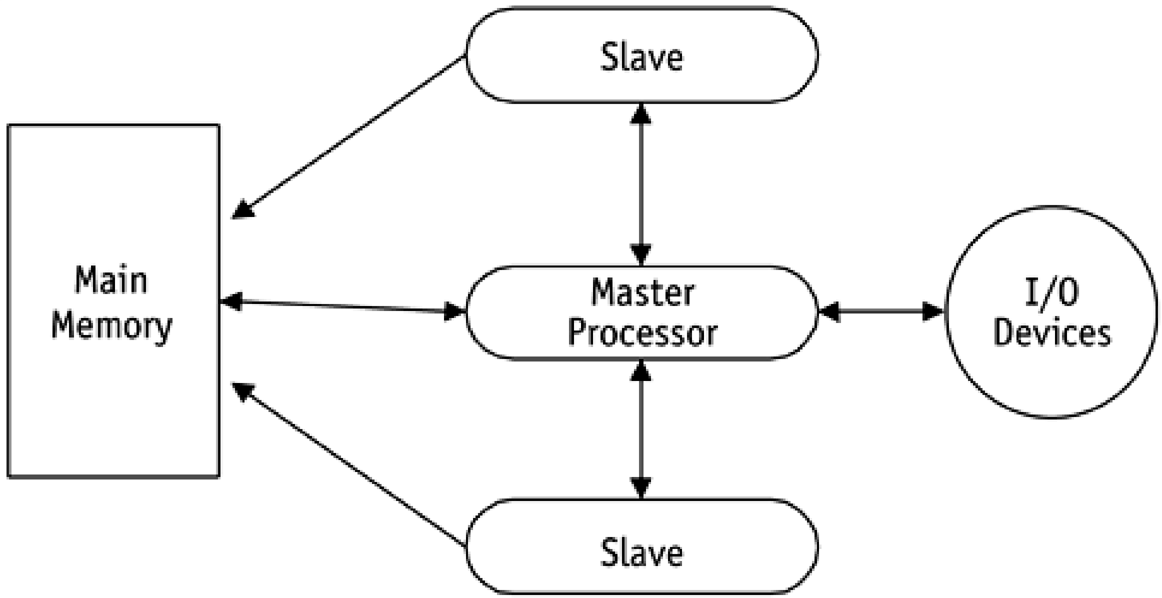


Figure 6.1: Master/slave configuration

Master/Slave Configuration (continued)

- **Advantages:**
  - Simplicity
- **Disadvantages:**
  - Reliability is no higher than for a single processor system
  - Can lead to poor use of resources
  - Increases the number of interrupts

Loosely Coupled Configuration

- Each processor has a copy of the OS and controls its own resources, and each can communicate and cooperate with others
- Once allocated, job remains with the same processor until finished
- Each has global tables that indicate to which processor each job has been allocated
- Job scheduling is based on several requirements and policies
- If a single processor fails, the others can continue to work independently

Loosely Coupled Configuration (continued)

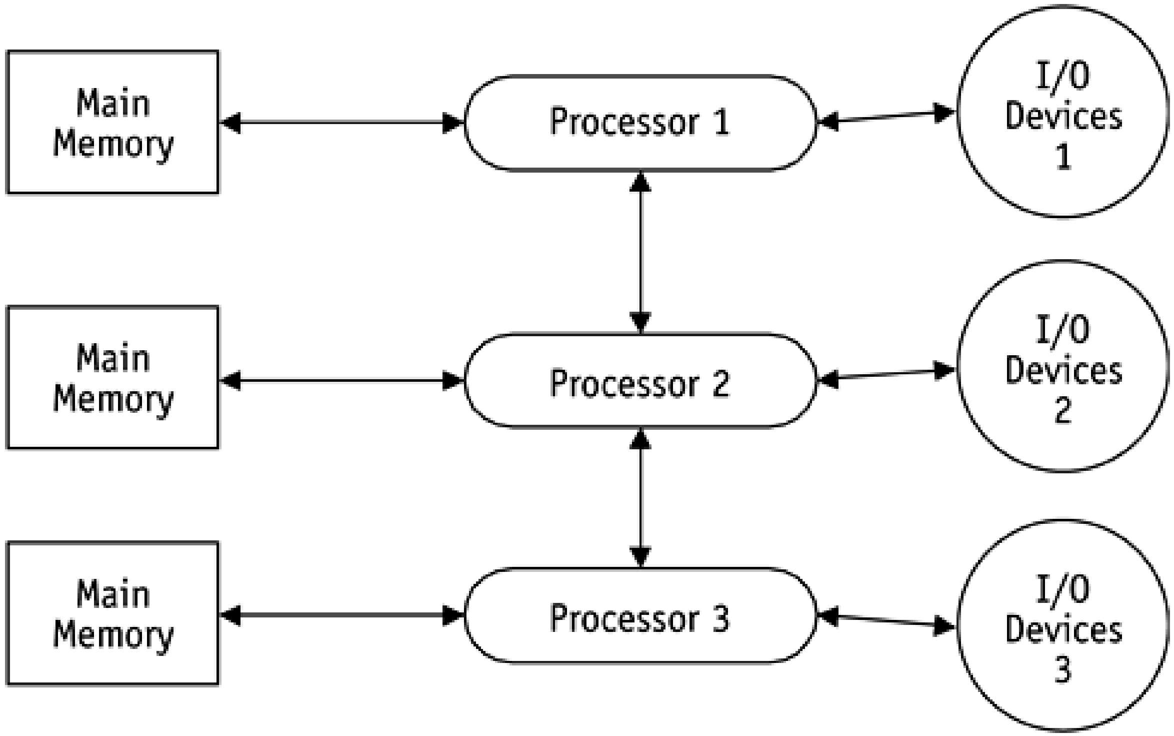


Figure 6.2: Loosely coupled configuration

Symmetric Configuration

- Processor scheduling is decentralized and each processor is of the same type
- **Advantages** over loosely coupled configuration:
  - More reliable
  - Uses resources effectively
  - Can balance loads well
  - Can degrade gracefully in the event of a failure

Symmetric Configuration (continued)

- All processes must be well synchronized to avoid races and deadlocks
- Any given job or task may be executed by several different processors during its run time
- More conflicts as several processors try to access the same resource at the same time
- **Process synchronization:** algorithms to resolve conflicts between processors

Symmetric Configuration (continued)

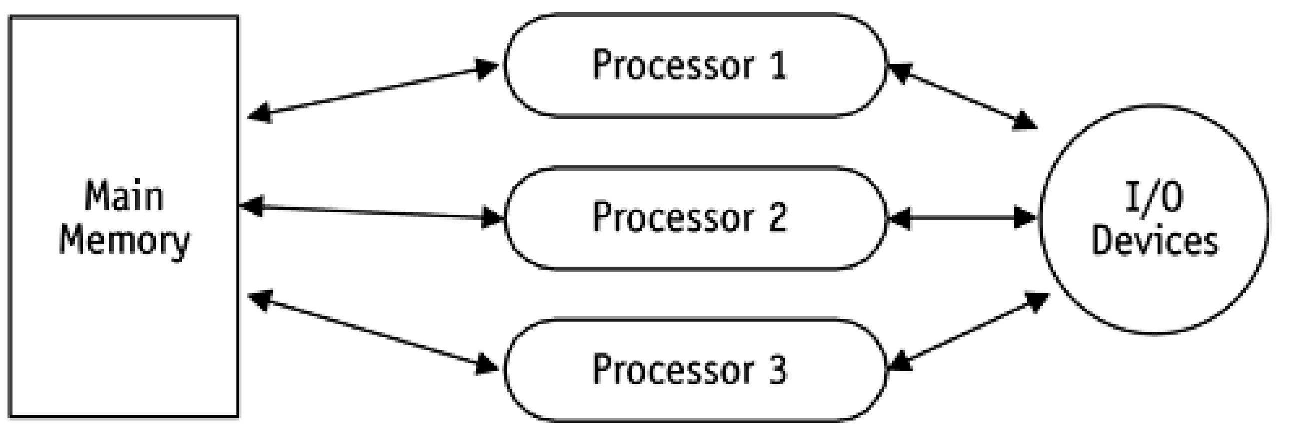


Figure 6.3: Symmetric configuration

Process Synchronization Software

- For a successful process synchronization:
  - Used resource must be locked from other processes until released
  - A waiting process is allowed to use the resource only when it is released
- A mistake could leave a job waiting indefinitely or if it is a key resource, cause a deadlock

Process Synchronization Software (continued)

- **Critical region:** A part of a program that must complete execution before other processes can have access to the resources being used
- Processes within a critical region can’t be interleaved without threatening integrity of the operation

Process Synchronization Software (continued)

- Synchronization is sometimes implemented as a lock-and-key arrangement:
  - Process must first see if the key is available
  - If available, process must pick it up and put it in the lock to make it unavailable to all other processes
- **Types of locking mechanisms:**
  - Test-and-set
  - WAIT and SIGNAL
  - Semaphores

Test-and-Set

- **Test-and-set:**
  - An indivisible machine instruction executed in a single machine cycle to see if the key is available and, if it is, sets it to unavailable
  - The actual key is a single bit in a storage location that can contain a 0 (free) or a 1 (busy)
  - A process P1 tests the condition code using TS instruction before entering a critical region
    - If no other process in this region, then P1 is allowed to proceed and condition code is changed from 0 to 1
    - When P1 exits, code is reset to 0, allows other to enter

Test-and-Set (continued)

- **Advantages:**
  - Simple procedure to implement
  - Works well for a small number of processes
- **Drawbacks:**
  - Starvation could occur when many processes are waiting to enter a critical region
    - Processes gain access in an arbitrary fashion
  - Waiting processes remain in unproductive, resource-consuming wait loops (busy waiting)

WAIT and SIGNAL

- Modification of test-and-set designed to remove busy waiting
- Two new mutually exclusive operations, **WAIT** and **SIGNAL** (part of Process Scheduler’s operations)
- WAIT is activated when process encounters a busy condition code
- SIGNAL is activated when a process exits critical region and the condition code is set to “free”

Semaphores

- A nonnegative integer variable that's used as a flag and signals if and when a resource is free and can be used by a process
- Two operations to operate the semaphore
  - P (*proberen* means to test)
  - V (*verhogen* means to increment)

Semaphores (continued)

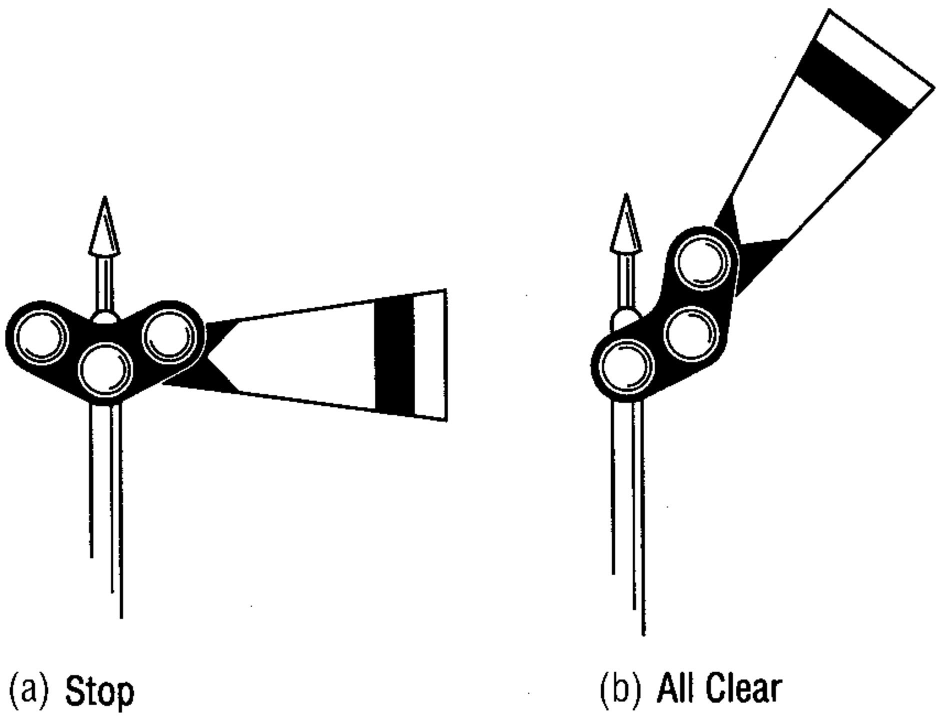


Figure 6.4: Semaphore used by railroads indicates whether the train can proceed

Semaphores (continued)

- If “s” is a semaphore variable, then:
  - **V(s): s: = s + 1**
    - (fetch, increment, and store sequence)
  - **P(s): If s > 0 then s: = s – 1**
    - (test, fetch, decrement, and store sequence)
- s = 0 implies busy critical region and the process calling on the P operation must wait until s > 0
- Choice of which of the waiting jobs will be processed next depends on the algorithm used by this portion of the Process Scheduler

Semaphores (continued)

Actions			Results		
State Number	Calling Process	Operation	Running in Critical Region	Blocked on *s*	Value of *s*
0					1
1	P1	P(*s*)	P1		0
2	P1	V(*s*)			1
3	P2	P(*s*)	P2		0
4	P3	P(*s*)	P2	P3	0
5	P4	P(*s*)	P2	P3, P4	0
6	P2	V(*s*)	P3	P4	0
7			P3	P4	0
8	P3	V(*s*)	P4		0
9	P4	V(*s*)			1

Table 6.1: P and V operations on the binary semaphore *s*

Semaphores (continued)

- P and V operations on semaphore *s* enforce the concept of mutual exclusion
- Semaphore is called mutex (MUTual EXclusion)

  P(mutex): if mutex > 0 then mutex: = mutex – 1

  V(mutex): mutex: = mutex + 1
- **Critical region** ensures that parallel processes will modify shared data only while in the critical region
- In parallel computations, mutual exclusion must be explicitly stated and maintained

Process Cooperation

- **Process cooperation:** When several processes work together to complete a common task
- Each case requires both mutual exclusion and synchronization
- Absence of mutual exclusion and synchronization results in problems
  - **Examples:**
    - Problems of producers and consumers
    - Problems of readers and writers
- Each case is implemented using semaphores

Producers and Consumers

- Arises when one process produces some data that another process consumes later
- **Example:** Use of buffer to synchronize the process between CPU and line printer:
  - Buffer must delay producer if it's full, and must delay consumer if it's empty
  - Implemented by two semaphores – one for number of full positions and other for number of empty positions
  - Third semaphore, mutex, ensures mutual exclusion

Producers and Consumers (continued)

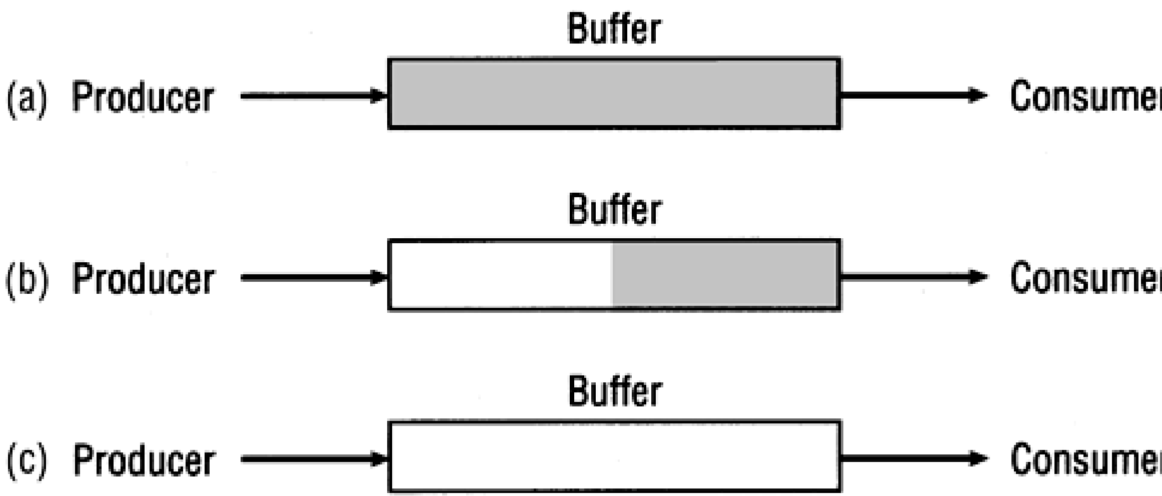


Figure 6.5: The buffer can be in any one of these three states: (a) full buffer, (b) partially empty buffer, or (c) empty buffer

Producers and Consumers (continued)

- **Definitions** of producer and consumer processes:

Producer	**Consumer**
produce data	P (full)
P (empty)	P (mutex)
P (mutex)	read data from buffer
write data into buffer	V (mutex)
V (mutex)	V (empty)
V (full)	consume data

Producers and Consumers (continued)

- **Definitions** of variables and functions:
  - **Given:** Full, Empty, Mutex defined as semaphores
  - **n:** maximum number of positions in the buffer
  - **V (x):** x: = x + 1 (x is any variable defined as a semaphore)
  - **P (x):** if x > 0 then x: = x – 1
  - **mutex = 1** means the process is allowed to enter the critical region

Producers and Consumers (continued)

- **Producers and Consumers Algorithm:**

  empty: = n

  full: = 0

  mutex: = 1

  COBEGIN

  repeat until no more data PRODUCER

  repeat until buffer is empty CONSUMER

  COEND

Producers

```
int itemCount;
procedure producer()
{ while (true)
        { item = produceItem();
                if (itemCount == BUFFER_SIZE)
                        { sleep(); }
                putItemIntoBuffer(item);
                itemCount = itemCount + 1;
                if (itemCount == 1)
                { wakeup(consumer); }
        }
}
```

Consumers

```
procedure consumer()
{ while (true)
        { if (itemCount == 0)
                { sleep(); }
        item = removeItemFromBuffer();
        itemCount = itemCount - 1;
        if (itemCount == BUFFER_SIZE - 1)
                { wakeup(producer); }
        consumeItem(item);
        }
}
```

Readers and Writers

- Arises when two types of processes need to access shared resource such as a file or database
- **Example:** An airline reservation system
  - Implemented using two semaphores to ensure mutual exclusion between readers and writers
  - A resource can be given to all readers, provided that no writers are processing (W2 = 0)
  - A resource can be given to a writer, provided that no readers are reading (R2 = 0) and no writers are writing (W2 = 0)

Concurrent Programming

- **Concurrent processing system:** Multiprocessing where one job uses several processors to execute sets of instructions in parallel
- **Sequential programming:** Instructions are executed one at a time
- **Concurrent programming:** Allows many instructions to be processed in parallel

Applications of Concurrent Programming (continued)

A= 3 * B * C + 4 / (D + E) ** (F – G)

Step No.	Operation	Result
1	(F – G)	Store difference in T1
2	(D + E)	Store sum in T2
3	(T2) ** (T1)	Store power in T1
4	4 / (T1)	Store quotient in T2
5	3 * B	Store product in T1
6	(T1) * C	Store product in T1
7	(T1) + (T2)	Store sum in A

Table 6.2: Sequential computation of the expression

Applications of Concurrent Programming (continued)

A= 3 * B * C + 4 / (D + E) ** (F – G)

Step No.	Processor	Operation	Result
1	1	3 * B	Store difference in T1
	2	(D + E)	Store sum in T2
	3	(F – G)	Store difference in T3
2	1	(T1) * C	Store product in T4
	2	(T2) ** (T3)	Store power in T5
3	1	4 / (T5)	Store quotient in T1
4	1	(T4) + (T1)	Store sum in A

Table 6.3: Concurrent programming reduces 7-step process to 4-step process

Applications of Concurrent Programming (continued)

- **Explicit parallelism:** Requires that the programmer explicitly state which instructions can be executed in parallel
- **Disadvantages:**
  - Coding is time-consuming
  - Leads to missed opportunities for parallel processing
  - Leads to errors where parallel processing is mistakenly indicated
  - Programs are difficult to modify

Applications of Concurrent Programming (continued)

- **Implicit parallelism:** Compiler automatically detects which instructions can be performed in parallel
- **Advantages:**
  - Solves the problems of explicit parallelism
  - Dramatically reduces the complexity of
    - Working with array operations within loops
    - Performing matrix multiplication
    - Conducting parallel searches in databases
    - Sorting or merging file

Threads and Concurrent Programming

- **Threads:** A smaller unit within a process, which can be scheduled and executed
- Minimizes the overhead from swapping a process between main memory and secondary storage
- Each active thread in a process has its own processor registers, program counter, stack and status
- Shares data area and the resources allocated to its process

Thread States

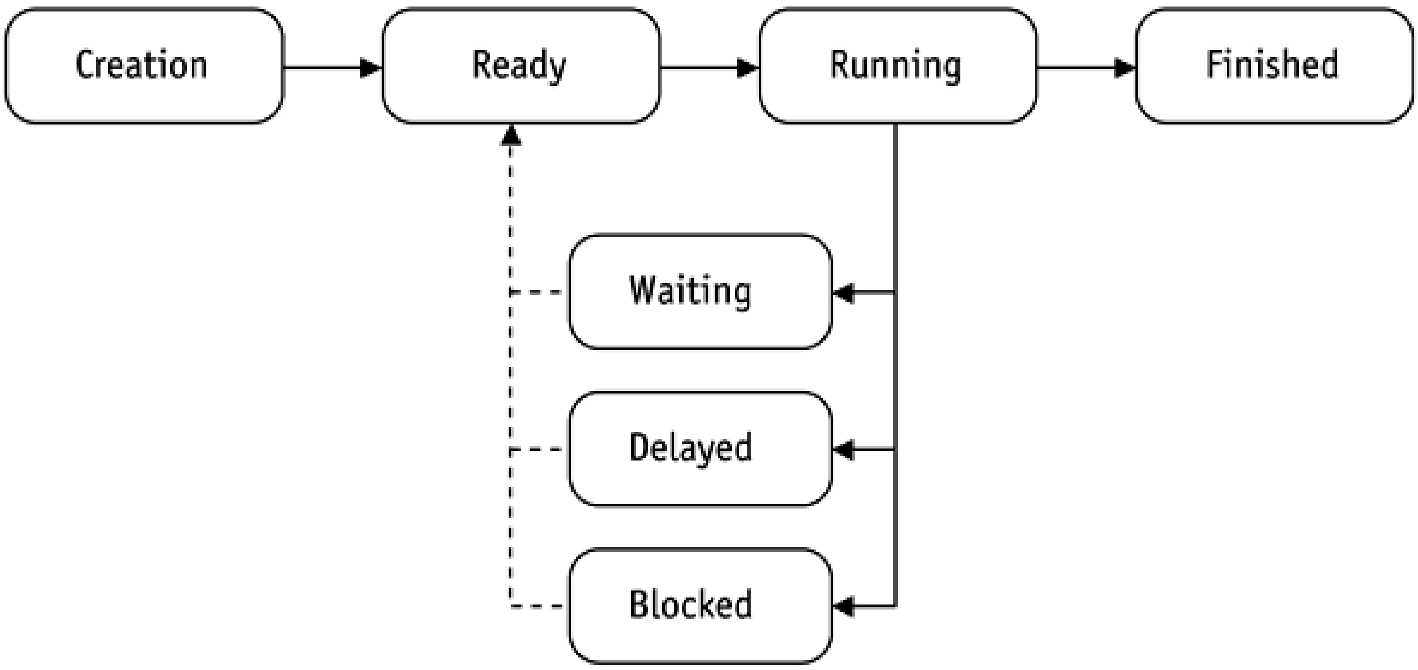


Figure 6.6: A typical thread changes states as it moves through the system.

Thread States (continued)

- **Operating system must be able to support**
  - Creating new threads
  - Setting up a thread so it is ready to execute
  - Delaying, or putting to sleep, threads for a specified amount of time
  - Blocking, or suspending, threads that are waiting for I/O to complete
  - Setting threads on a WAIT state until a specific event has occurred

Thread States (continued)

- (continued)
  - Scheduling threads for execution
  - Synchronizing thread execution using semaphores, events, or conditional variables
  - Terminating a thread and releasing its resources

Thread Control Block

Contains information about the current status and characteristics of a thread

Thread ID
Thread state
CPU information: Program counter Register contents
Thread priority
Pointer to process that created this thread
Pointer(s) to other thread(s) that were created by this thread

Figure 6.7: Typical Thread Control Block (TCB)

Concurrent Programming Languages

- **Ada:**
  - High-level concurrent programming language developed by the U.S Department of Defense
  - Initially intended for real-time and embedded systems
  - Made available to the public in 1980, named after Augusta Ada Byron
  - Standardized by ANSI in 1983 and nicknamed Ada83
  - Latest standard is ANSI/ISO/IEC-8652:1995 Ada 95

Java

- First software platform that promised to allow programmers to code an application once, that would run on any computer
- Developed at Sun Microsystems, Inc. (1995)
- Uses both a compiler and an interpreter
- **Solves several issues:**
  - High cost of developing software applications for different incompatible computer architectures
  - Needs of distributed client-server environments
  - Growth of the Internet and the World Wide Web

The Java Platform

Java platform is a software-only platform that runs on top of other hardware-based platforms

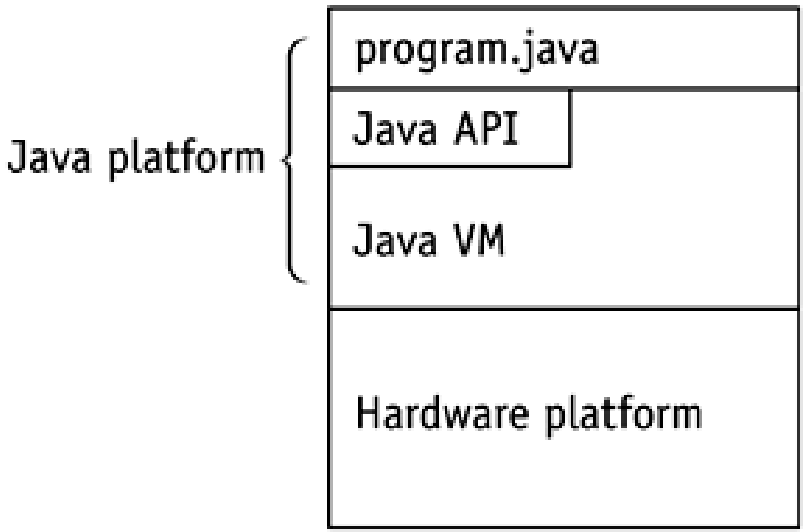


Figure 6.8: A process used by the Java platform to shield a Java program from a computer's hardware

The Java Language Environment

- Looks and feels like C++
- Object oriented and fits well into distributed client-server applications
- Memory allocation done at run time
- Compile-time and run-time checking
- Sophisticated synchronization capabilities
  - Supports multithreading at the language level

Case Study: Process Management in Linux

- Linux scheduler scans the list of processes in the READY state and, using predefined criteria, chooses which process to execute
- Three scheduling policies:
  - Two for real-time processes and one for normal processes
- Each process has three attributes:
  - Associated process type
  - Fixed priority
  - Variable priority

Case Study: Process Management in Linux (continued)

- Combination of type and priority determines which scheduling policy to use on the processes in the ready queue
- For example, each process is one of three types
  - **SCHED_FIFO** for nonpreemptible “real time” processes
  - **SCHED_RR** for preemptible “real time” processes
  - **SCHED_OTHER** for “normal” processes

Summary

- Multiprocessing occurs in single-processor systems between interacting processes that obtain control of the one CPU at different times
- Multiprocessing also occurs in systems with two or more CPUs; synchronized by Processor Manager
- Each processor must communicate and cooperate with the others
- Systems can be configured as master/slave, loosely coupled, and symmetric

Summary (continued)

- Success of multiprocessing system depends on the ability to synchronize the processors or processes and the system’s other resources
- Mutual exclusion helps keep the processes with the allocated resources from becoming deadlocked
- Mutual exclusion is maintained with a series of techniques including test-and-set, WAIT and SIGNAL, and semaphores (P, V, and mutex)

Summary (continued)

- Hardware and software mechanisms are used to synchronize many processes
- Care must be taken to avoid the typical problems of synchronization: missed waiting customers, the synchronization of producers and consumers, and the mutual exclusion of readers and writers
- Java offers the capability of writing a program once and having it run on various platforms without having to make any changes