Understanding Operating Systems
Seventh Edition

Chapter 6
Concurrent Processes

Learning Objectives

After completing this chapter, you should be able to describe:

- The critical difference between processes and processors and how they're connected
- The differences among common configurations of multiprocessing systems
- The basic concepts of multicore processor technology
- The significance of a critical region in process synchronization

Learning Objectives (cont'd.)

- The essential ideas behind process synchronization software
- The need for process cooperation when several processors work together
- How processors cooperate when executing a job, process, or thread
- The significance of concurrent programming languages and their applications

What Is Parallel Processing?

- Parallel processing
  - Two or more processors operate in one system at the same time
    - Work may or may not be related
  - Two or more CPUs execute instructions simultaneously
  - Processor Manager
    - Coordinates activity of each processor
    - Synchronizes interaction among CPUs

What Is Parallel Processing? (cont'd.)

- Benefits
  - Increased reliability
    - More than one CPU
    - If one processor fails, others take over: must be designed into the system
  - Faster processing
    - Instructions processed in parallel two or more at a time

What Is Parallel Processing? (cont'd.)

- Faster instruction processing methods
  - CPU allocated to each program or job
  - CPU allocated to each working set or parts of it
  - Individual instructions subdivided
    - Each subdivision processed simultaneously
    - Concurrent programming
- Two major challenges
  - Connecting processors into configurations
  - Orchestrating processor interaction
- Example: six-step information retrieval system
  - Synchronization is key

Originator	Action	Receiver
Processor 1 (the order clerk)	Accepts the query, checks for errors, and passes the request on to the receiver	Processor 2 (the bagger)
Processor 2 (the bagger)	Searches the database for the required information (the hamburger)	
Processor 3 (the cook)	Retrieves the data from the database (the meat to cook for the hamburger) if it's kept off-line in secondary storage	
Processor 3 (the cook)	Once the data is gathered (the hamburger is cooked), it's placed where the receiver can get it (in the hamburger bin)	Processor 2 (the bagger)
Processor 2 (the bagger)	Retrieves the data (the hamburger) and passes it on to the receiver	Processor 4 (the cashier)
Processor 4 (the cashier)	Routes the response (your order) back to the originator of the request	You

(table 6.1)
The six steps of the four-processor fast food lunch stop.

Levels of Multiprocessing

- Multiprocessing occurs at three levels
  - Job level
  - Process level
  - Thread level
- Each level requires different synchronization frequency

Levels of Multiprocessing (cont'd.)

Parallelism Level	Process Assignments	Synchronization Required
Job Level	Each job has its own processor, and all processes and threads are run by that same processor.	No explicit synchronization required after jobs are assigned to a processor.
Process Level	Unrelated processes, regardless of job, can be assigned to available processors.	Moderate amount of synchronization required.
Thread Level	Threads, regardless of job and process, can be assigned to available processors.	High degree of synchronization required to track each process and each thread.

(table 6.2)
Typical levels of parallelism and the required synchronization among processors.

Introduction to Multi-Core Processors

- Multi-core processing
  - Several processors placed on single chip
- Problems
  - Current leakage (tunneling) and heat
- Solution
  - Single chip with two processor cores in same space
    - Allows two sets of simultaneous calculations
  - Two cores each run more slowly than single core chip

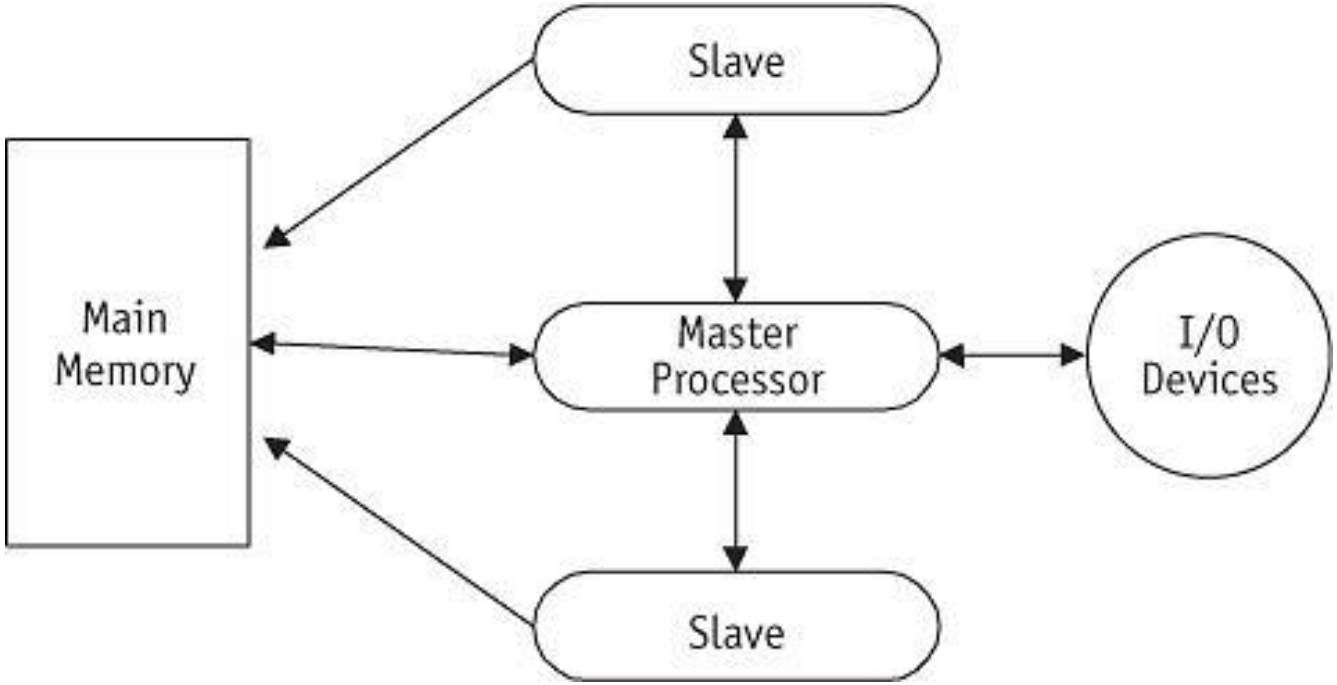
Typical Multiprocessing Configurations

- Multiple processor configuration impacts systems
- Three types
  - Master/slave
  - Loosely coupled
  - Symmetric

Master/Slave Configuration

- Asymmetric multiprocessing system
- Single-processor system
  - Additional slave processors
    - Each managed by primary master processor
- Master processor responsibilities
  - Manages entire system
  - Maintains status of all processes
  - Performs storage management activities
  - Schedules work for other processors
  - Executes all control programs

Master/Slave Configuration (cont'd.)



(figure 6.1)
In a master/slave multiprocessing configuration, slave processors can access main memory directly, but they must send all I/O requests through the master processor.

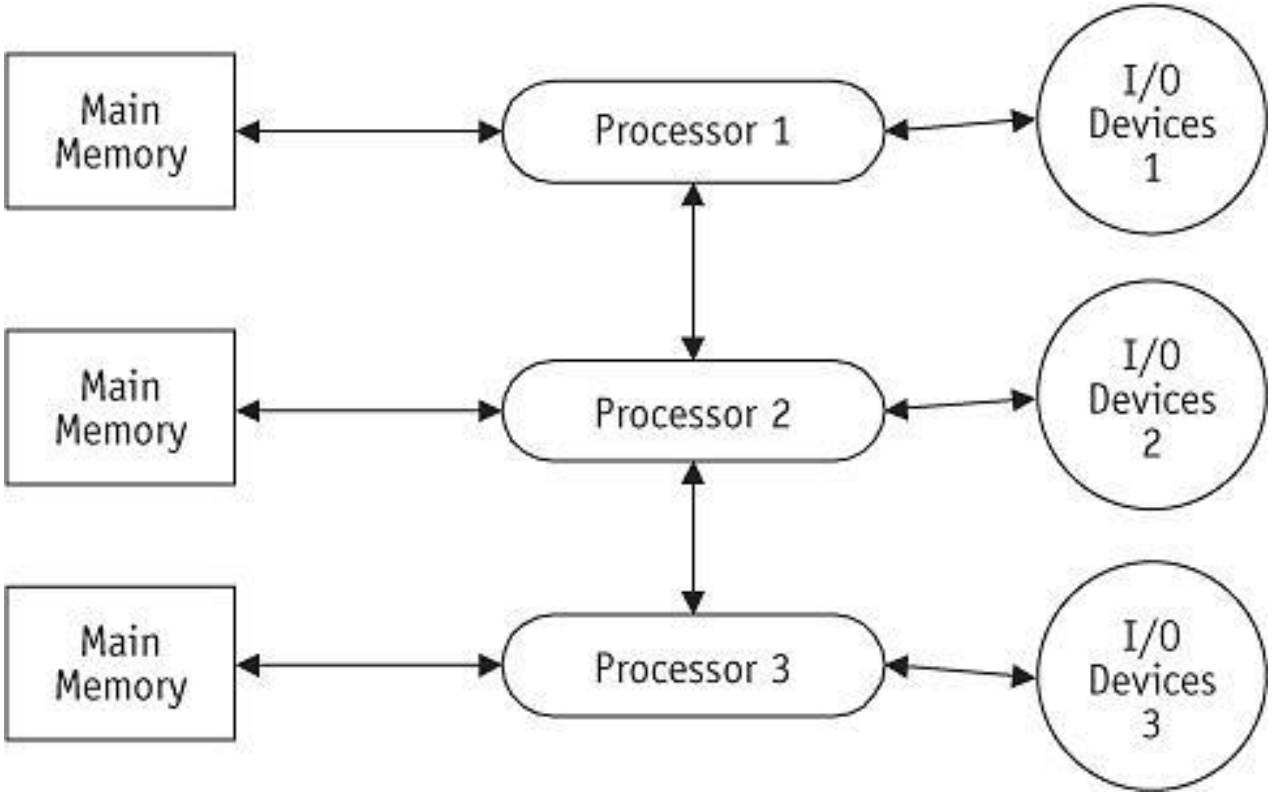
Master/Slave Configuration (cont'd.)

- Advantages
  - Simplicity
- Disadvantages
  - Reliability
    - No higher than single processor system
  - Potentially poor resources usage
  - Increases number of interrupts

Loosely Coupled Configuration

- Several complete computer systems
  - Each with own resources
    - Each maintains commands and I/O management tables
- Independent single-processing difference
  - Each processor
    - Communicates and cooperates with others
    - Has global tables
- Several requirements and policies for job scheduling
- Single processor failure
  - Others continue work independently
  - Difficult to detect

Loosely Coupled Configuration (cont'd.)

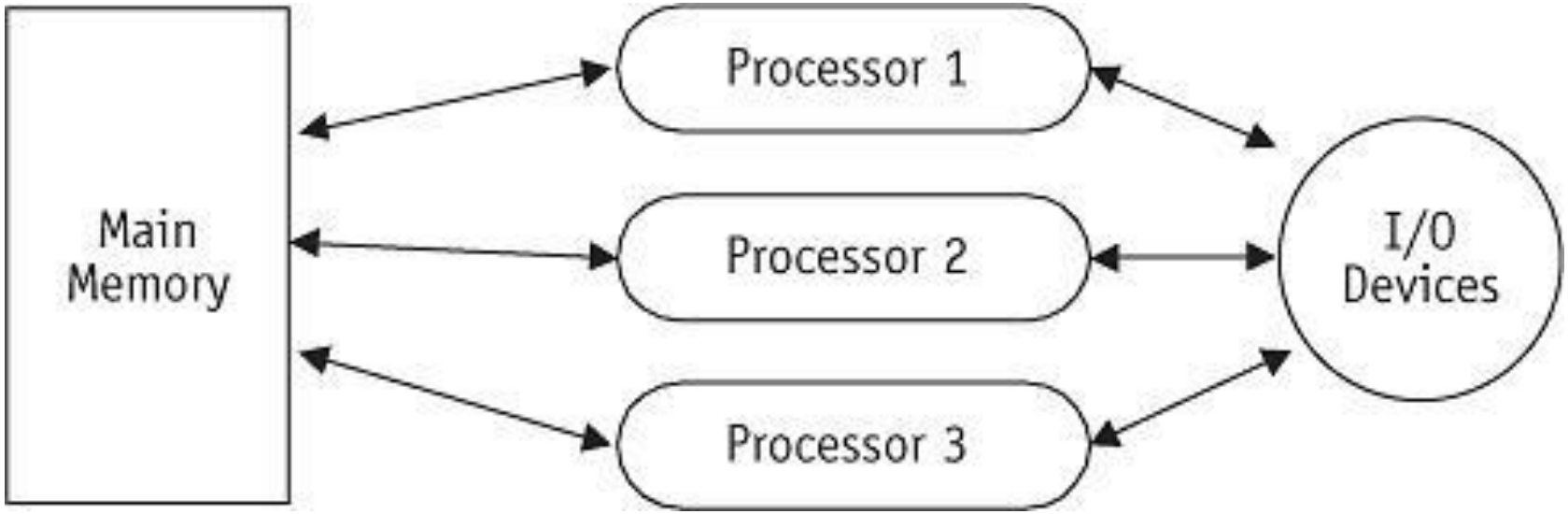


(figure 6.2)
In a loosely coupled multiprocessing configuration, each processor has its own dedicated resources.

Symmetric Configuration

- Decentralized processor scheduling
  - Each processor uses same scheduling algorithm
- Advantages (over loosely coupled configuration)
  - More reliable
  - Uses resources effectively
  - Balances loads well
  - Degrades gracefully in failure situation
- Most difficult to implement
  - Requires well synchronized processes
    - Avoids races and deadlocks

Symmetric Configuration (cont'd.)



(figure 6.3)
A symmetric multiprocessing configuration with homogeneous processors. Processes must be carefully synchronized to avoid deadlocks and starvation.

Symmetric Configuration (cont'd.)

- Interrupt processing
  - Update corresponding process list
  - Run another process
- More conflicts
  - Several processors access same resource at same time
- Algorithms resolving conflicts between processors required

Process Synchronization Software

- Successful process synchronization
  - Lock up used resource
    - Protect from other processes until released
  - Only when resource is released
    - Waiting process is allowed to use resource
- Mistakes in synchronization can result in:
  - Starvation
    - Leave job waiting indefinitely
  - Deadlock
    - If key resource is being used

Process Synchronization Software (cont'd.)

- Critical region
  - Part of a program
  - Critical region must complete execution
    - Other processes must wait before accessing critical region resources
- Processes within critical region
  - Cannot be interleaved
    - Threatens integrity of operation

Process Synchronization Software (cont'd.)

- Synchronization
  - Implemented as lock-and-key arrangement:
  - Process determines key availability
    - Process obtains key
    - Puts key in lock
    - Makes it unavailable to other processes
- Types of locking mechanisms
  - Test-and-set
  - WAIT and SIGNAL
  - Semaphores

Test-and-Set

- Indivisible machine instruction (TS)
- Executed in single machine cycle
  - If key available: set to unavailable
- Actual key
  - Single bit in storage location: zero (free) or one (busy)
- Before process enters critical region
  - Tests condition code using TS instruction
  - No other process in region
    - Process proceeds
    - Condition code changed from zero to one
    - P1 exits: code reset to zero, allowing others to enter

Test-and-Set (cont'd.)

- Advantages
  - Simple procedure to implement
  - Works well for small number of processes
- Drawbacks
  - Starvation
    - Many processes waiting to enter a critical region
    - Processes gain access in arbitrary fashion
  - Busy waiting
    - Waiting processes remain in unproductive, resource-consuming wait loops

WAIT and SIGNAL

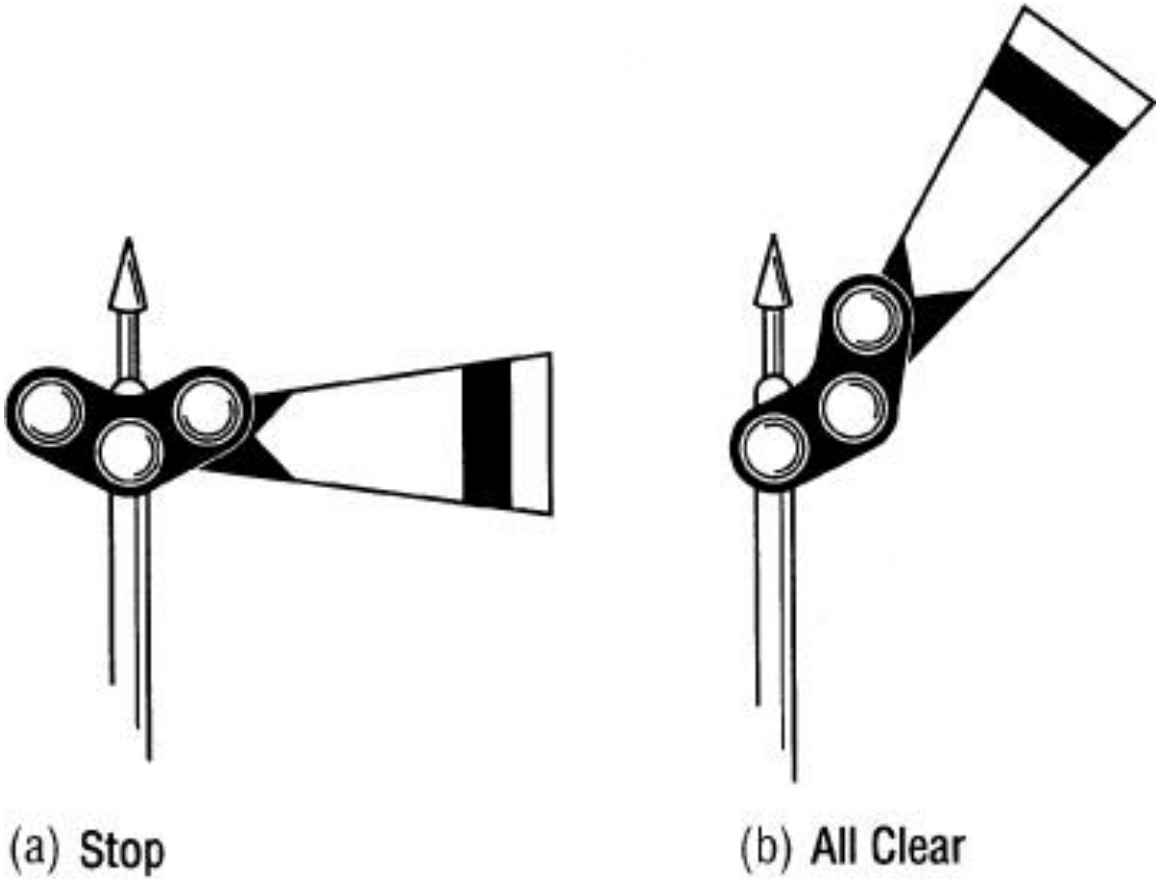
- Modification of test-and-set
  - Designed to remove busy waiting
- Two new mutually exclusive operations
  - WAIT and SIGNAL
  - Part of Process Scheduler's operations
- WAIT
  - Activated when process encounters busy condition code
- SIGNAL
  - Activated when process exits critical region and condition code set to "free"

Semaphores

- Nonnegative integer variable
  - Flag (binary signal)
  - Signals if and when resource is free
    - Resource can be used by a process
- Two operations of semaphore: introduced by Dijkstra (1965)
  - P (proberen means “to test”)
  - V (verhogen means “to increment”)

Semaphores (cont'd.)

(figure 6.4)
The semaphore used by railroads indicates whether your train can proceed. When it's lowered (a), another train is approaching and your train must stop to wait for it to pass. If it is raised (b), your train can continue.



Semaphores (cont'd.)

- Let *s* be a semaphore variable
  - V(*s*): *s*: = *s* + 1
    - Fetch, increment, store sequence
  - P(*s*): If *s* > 0, then *s*: = *s* – 1
    - Test, fetch, decrement, store sequence
- *s* = 0 implies busy critical region
  - Process calling on P operation must wait until *s* > 0
- Waiting job of choice processed next
  - Depends on process scheduler algorithm

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

(table 6.3)

The sequence of states for four processes (P1, P2, P3, P4) calling test and increment (P and V) operations on the binary semaphore s. (Note: The value of the semaphore before the operation is shown on the line preceding the operation. The current value is on the same line.)

Semaphores (cont'd.)

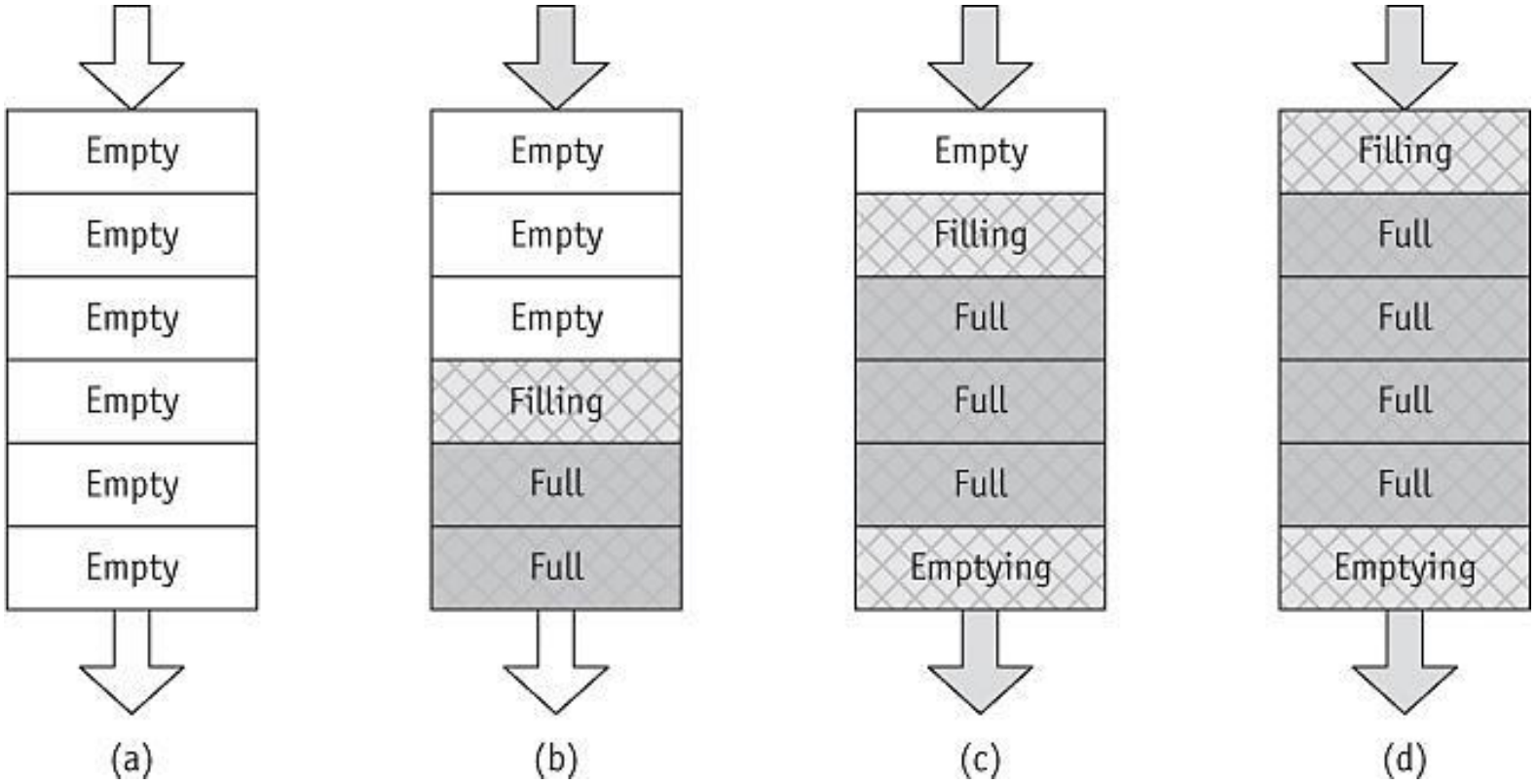
- P and V operations on semaphore *s*
  - Enforce mutual exclusion concept
- Semaphore called mutex (MUTual EXclusion)
  - P(mutex): if mutex > 0 then mutex: = mutex – 1
  - V(mutex): mutex: = mutex + 1
- Critical region
  - Ensures parallel processes modify shared data only while in critical region
- Parallel computations
  - Mutual exclusion explicitly stated and maintained

Process Cooperation

- Several processes work together to complete common task
- Each case requires
  - Mutual exclusion and synchronization
- Examples
  - Producers and consumers problem
  - Readers and writers problem
- Each case implemented using semaphores

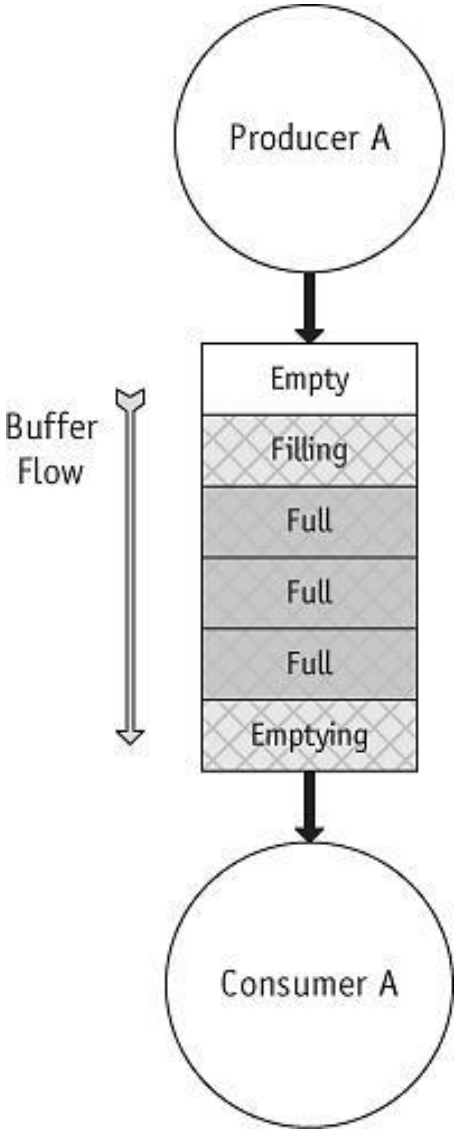
Producers and Consumers

- One process produces data
  - Another process later consumes data
- Example: CPU and printer buffer
  - Delay producer: buffer full
  - Delay consumer: buffer empty
  - Implemented by two semaphores
    - Number of full positions
    - Number of empty positions
  - Mutex
    - Third semaphore: ensures mutual exclusion between processes



(figure 6.6)
Four snapshots of a single buffer in four states from completely empty (a) to almost full (d).

(figure 6.7)
Typical system with one producer, one consumer, and a single buffer.



Readers and Writers

- Formulated by Courtois, Heymans, and Parnas (1971)
- Two process types need to access shared resource, e.g., file or database

Readers and Writers (cont'd.)

- Example: airline reservation system
  - Implemented using two semaphores
    - Ensures mutual exclusion between readers and writers
  - Resource given to all readers
    - Provided no writers are processing (W2 = 0)
  - Resource given to a writer
    - Provided no readers are reading (R2 = 0) and no writers writing (W2 = 0)

Concurrent Programming

- Another type of multiprocessing
- Concurrent processing system
  - One job uses several processors
    - Executes sets of instructions in parallel
  - Requires programming language and computer system support
- Two broad categories of parallel systems
  - Data level parallelism (DLP)
  - Instruction (or task) level parallelism (ILP)

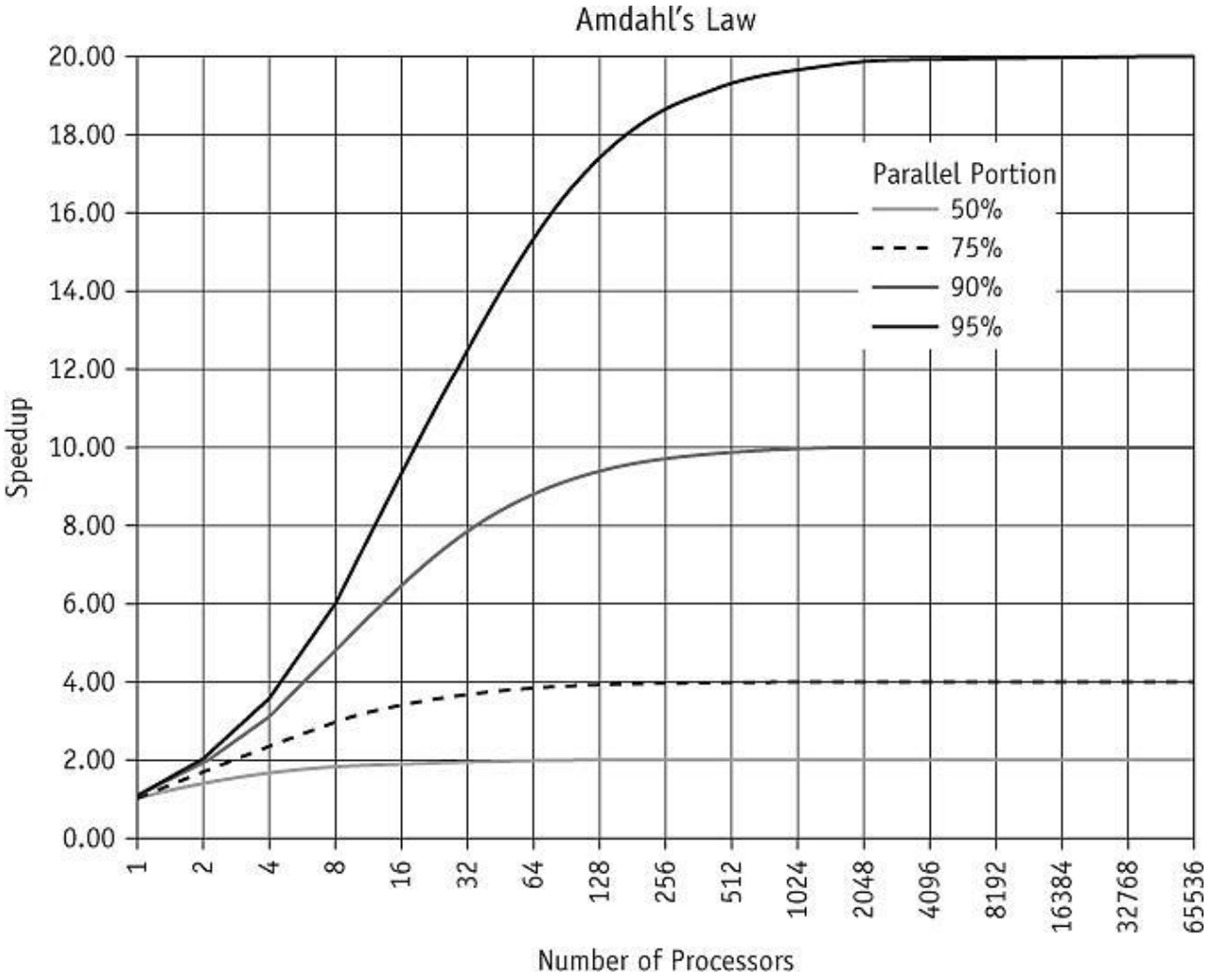
Concurrent Programming (cont'd.)

(table 6.4)
The four classifications of Flynn's Taxonomy for machine structures

	Number of Instructions	
	Single Instruction	**Multiple Instructions**
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

- Opportunities for parallelism
  - SISD: few if any
  - MISD: might allow instruction level parallelism; little or no data level parallelism without additional software
  - SIMD: multiple data streams
  - MIMD: both instruction level and data level parallelism

Amdahl's Law



(figure 6.8) Amdahl's Law. Notice that all four graphs level off and there is no speed difference even though the number of processors increased from 2,048 to 65,536 (Amdahl, 1967).

Order of Operations

- Precedence of operations or rules of precedence
- Solving an equation–all arithmetic calculations performed from the left and in the following order:
  1. Perform all calculations in parentheses
  2. Calculate all exponents
  3. Perform all multiplications and divisions: resolved from the left
  4. Perform all additions and subtractions: resolved from the left

Order of Operations (cont'd.)

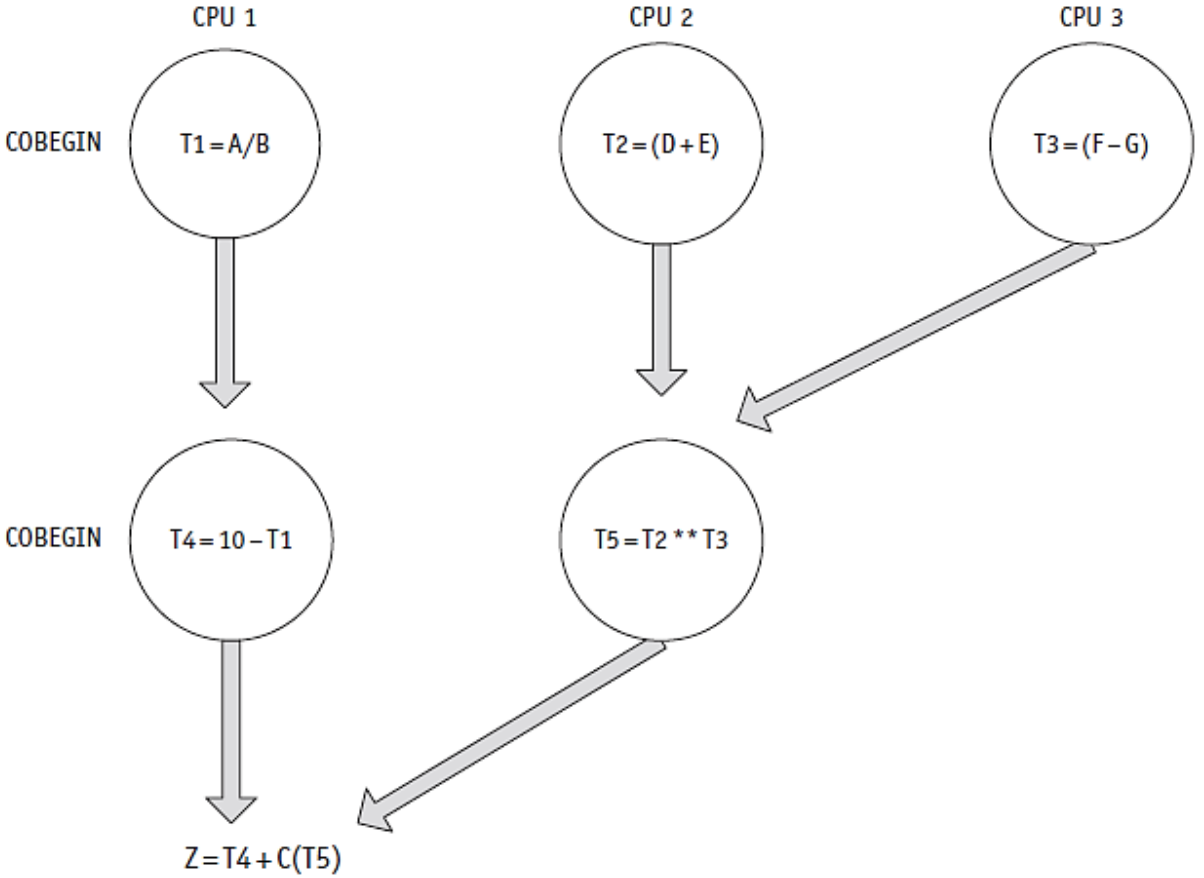
$$Z = 10 - A / B + C (D + E) ** (F - G)$$

(table 6.5)
The sequential computation of the expression requires several steps. (In this example, there are six steps, but each step, such as the last one, may involve more than one machine operation.)

Step No.	Operation	Result
1	(D + E) = M	Store sum in M
2	(F – G) = N	Store difference in N
		Now the equation is Z = 10 – A/B + C(M^N)
3	(M) ** (N) = P	Store power in P
		Now the equation is Z = 10 – A/B + C(P)
4	A / B = Q	Store quotient in Q
		Now the equation is Z = 10 – Q + C(P)
5	C * P = R	Store product in R
		Now the equation is Z = 10 – Q + R
6	10 – Q + R = Z	Store result in Z

Applications of Concurrent Programming

$$Z = 10 - A / B + C (D + E) ** (F - G)$$



(figure 6.9) Three CPUs can perform the six-step equation in three steps.

Applications of Concurrent Programming (cont'd.)

- Reducing complexity via concurrent programming
  - Case 1: array operations
  - Case 2: matrix multiplication
  - Case 3: searching databases
  - Case 4: sorting and merging files

Threads and Concurrent Programming

- Threads: lightweight processes
  - Smaller unit within process
    - Scheduled and executed
- Minimizes overhead
  - Swapping process between main memory and secondary storage
- Each active process thread
  - Processor registers, program counter, stack, and status
- Shares data area and resources allocated to its process

Threads and Concurrent Programming (cont'd.)

- Web server: improved performance and interactivity with threads
  - Requests for images or pages: each served with a different thread
  - After thread's task completed: thread returned to pool for assignment to another task

Two Concurrent Programming Languages

- Ada
  - First language providing specific concurrency commands
  - Developed in late 1970's
  - ADA 2012: International Standards Organization (ISO) standard replacing ADA 2005
- Java
  - Designed as universal Internet application software platform
  - Allows programmers to code applications that can run on any computer

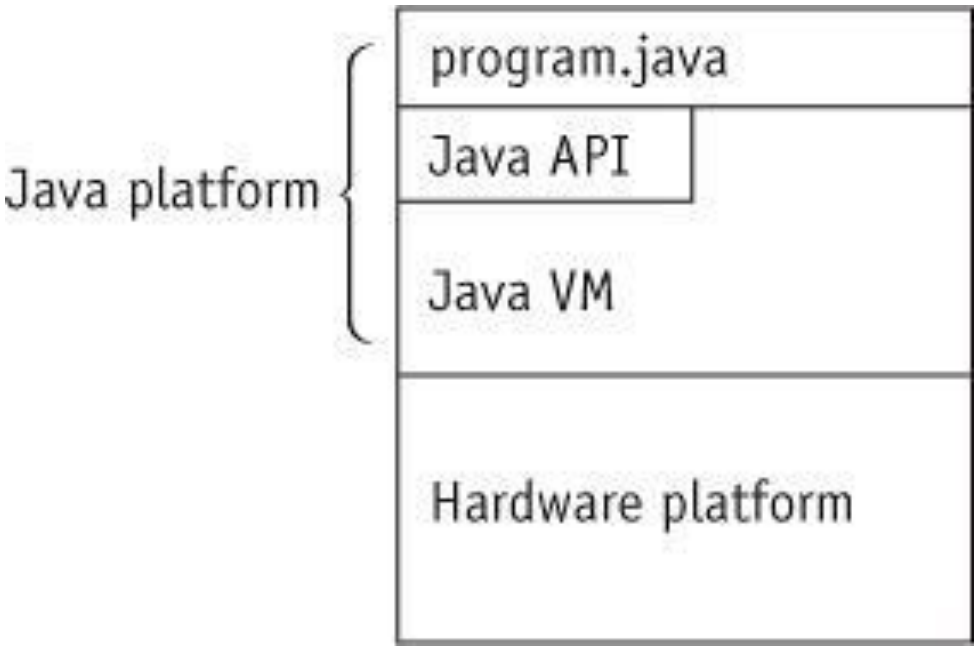
Java

- Developed at Sun Microsystems, Inc. (1995)
- Solves several issues
  - High software development costs for different incompatible computer architectures
  - Distributed client-server environment needs
  - Internet and World Wide Web growth
- Uses compiler and interpreter
  - Easy to distribute Java applications

The Java Platform

- Software only platform
  - Runs on top of other hardware-based platforms
- Two components
  - Java Virtual Machine (Java VM)
    - Foundation for Java platform
    - Contains the interpreter
    - Runs compiled bytecodes
  - Java application programming interface (Java API)
    - Collection of software modules
    - Grouped into libraries by classes and interfaces

The Java Platform (cont'd.)



(figure 6.12)
A process used by the Java platform to shield a Java program from a computer's hardware.

The Java Language Environment

- Designed for experienced programmers (similar to C++)
- Object oriented
  - Exploits modern software development methods
    - Fits into distributed client-server applications
- Memory allocation features
  - Done at run time
  - References memory via symbolic “handles”
  - Translated to real memory addresses at run time
  - Not visible to programmers

The Java Language Environment (cont'd.)

- Security
  - Built-in feature
    - Language and run-time system
  - Checking
    - Compile-time and run-time
- Sophisticated synchronization capabilities
  - Multithreading at language level
- Popular features
  - Single program runs on various platforms; robust feature set; Internet and Web integration

Conclusion

- Multiprocessing
  - Single-processor systems
    - Interacting processes obtain control of CPU at different times
  - Systems with two or more CPUs
    - Control synchronized by processor manager
    - Processor communication and cooperation
  - System configuration
    - Master/slave, loosely coupled, and symmetric

Conclusion (cont'd.)

- Multiprocessing: several configurations
  - Single-processor with interacting processes
  - Multiple processors: synchronized by Process Manager
- Mutual exclusion
  - Prevents deadlock
  - Maintained with test-and-set, WAIT and SIGNAL, and semaphores (P, V, and mutex)
- Synchronize processes using hardware and software mechanisms

Conclusion (cont'd.)

- Avoid typical problems of synchronization
  - Missed waiting customers
  - Synchronization of producers and consumers
  - Mutual exclusion of readers and writers
- Concurrent processing innovations
  - Threads and multi-core processors