Advanced Operating Systems

20MCA172

(Elective)

Contents

- **Overview**: Functions of Operating System– Design Approaches – Types of Advanced Operating Systems.

- **Synchronization Mechanisms**: Concept of Processes and Threads –The Critical Section Problem – Other Synchronization Problems:– Monitor –Serializer – Path Expressions.

- **Distributed Operating Systems**:- Issues in Distributed Operating System – Communication Networks And Primitives Lamport's Logical clocks – Causal Ordering of Messages.

(Mukesh Singhal and Niranjan G. Shivaratri, "***Advanced Concepts in Operating Systems*** – Distributed, Database, and Multiprocessor Operating Systems", Tata McGraw-Hill, 2001.)

What is OS?

- Operating System is a software, which makes a computer to actually work.
- It is the software the enables all the programs we use.
- The OS organizes and controls the hardware.
- OS acts as an interface between the application programs and the machine hardware.
- Examples: Windows, Linux, Unix and Mac OS, etc.,

What is Advanced OS?

- Traditional Operating System : which ran on stand- alone computers with single processors.

- Arise of Multiprocessor Systems and Distributed Systems.

- Due to the high demand and popularity of multiprocessor systems, advanced operating system have gained high importance.

Functions of an Operating System

Two basic functions of operating systems are:

1. **Resource management**
2. **User friendliness**

Functions of an Operating System...

1. Resource management

- A user program accesses several hardware and software resources during its execution.
- Example: CPU, Main memory, I/O devices, and various types of software (compiler, linker-loader, files, etc.).
- Manages the resources and allocates them to users in an efficient and fair manner.
- It encompasses the following functions.
    - Time management (CPU and disk scheduling).
    - Space management (main and secondary storages).
    - Process Synchroziation and deadlock handling.
    - Accouting and status information.

Functions of an Operating System...

2. User friendliness

- It hides the unpleasant, low-level details and singularity of bare H/W machine .
- Friendlier interface to the machine.
- It encompasses the following functions.
  - Execution environment (Process management – creation, control, and termination, file manipulation, interrupt handling, support for I/O operations. Language support).
  - Error detection and handling
  - Protection and security
  - Fault tolerance and failure recovery.

Design Approaches

- A typical operating system that supports a multiprogramming environment can easily be tens of megabytes in length and its design, implementation, and testing amounts to the undertaking of a huge software project.

- Deal with complexities of modern systems.

- How and what should be done in the context of operating systems

- **Separation of Policies and Mechanisms**
  - **Policies** - What should be done
  - **Mechanisms** - How it should be done
  - Ex: In CPU scheduling, *mechanisms* provide the means to implement various scheduling disciplines and *policy* decides which CPU scheduling discipline will be used.

Design Approaches

- A good operating system design must separate policies from mechanisms which provide flexibility.
- A change in policy decisions will not require changes in mechanisms.

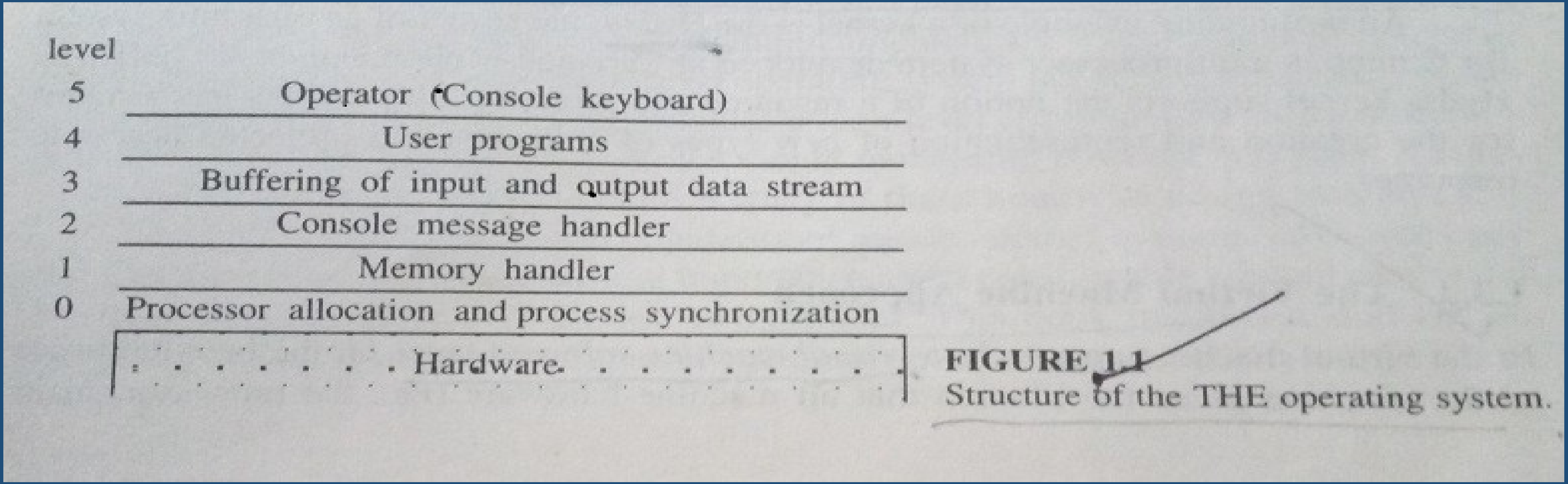
- **Three common approaches:**
  - **Layered Approach**
  - **Kernel Approach**
  - **Virtual Machine Approach**

Layered Approach

- This approach divides the operating system into **several layers**.
- Each layer has **well-defined functionality** and **input-output interfaces** with the two adjacent layers.
- The bottom layer interfaces with **machine hardware** and top layer interfaces with **users or operators**.
- This approach has all the advantages of modular design.
- Each layer can be designed, coded and tested **independently.**
- This approach simplifies the design, specification, and implementation of an operating system.
- **Drawback :** functions must be carefully assigned to various layers because a layer can make use only of the functionality provided by the layers beneath it.
- Example : THE operating system and MULTICS operating system

Layered Approach

• Example is THE Operating System which consists of six layers.



level	
5	Operator (Console keyboard)
4	User programs
3	Buffering of input and output data stream
2	Console message handler
1	Memory handler
0	Processor allocation and process synchronization
	Hardware

FIGURE 1.1
Structure of the THE operating system.

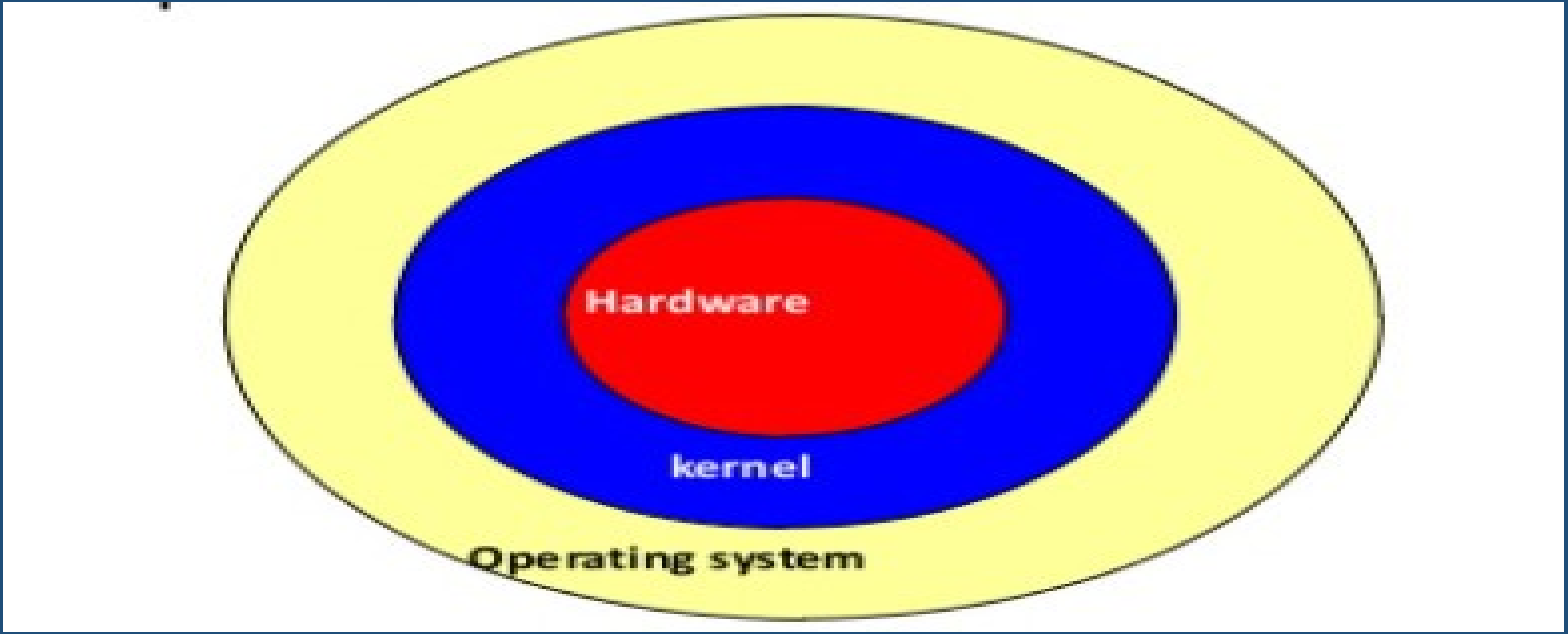
Kernel Approach

- Kernel contains a collection of primitives which are used to build the OS.
- The kernel (nucleus) is a collection of primitive facilities over which the rest of the operating system is built, using the functions provided by the kernel.
- Kernel provides an environment to build operating systems
- Policy and optimization decisions are not made at the kernel level.
- Kernel should support only mechanisms and policy decisions are left to the outer layer.
- OS implements policy .
- Kernel implements mechanisms.

Kernel Approach

- An operating system is an orderly growth of software over the kernel where all decisions regarding process scheduling, resource allocation, execution environment, file system, and resource protection, etc. are made.
- A kernel should contain a minimal set of functionality that is adequate to build an operating system with a given set of objectives.
- Including too much functionality in a kernel results in low flexibility at a higher level, whereas including too little functionality in a kernel results in low functional support at a higher level.
- Example : C.mmp (Kernal is Hydra)

Kernel Approach

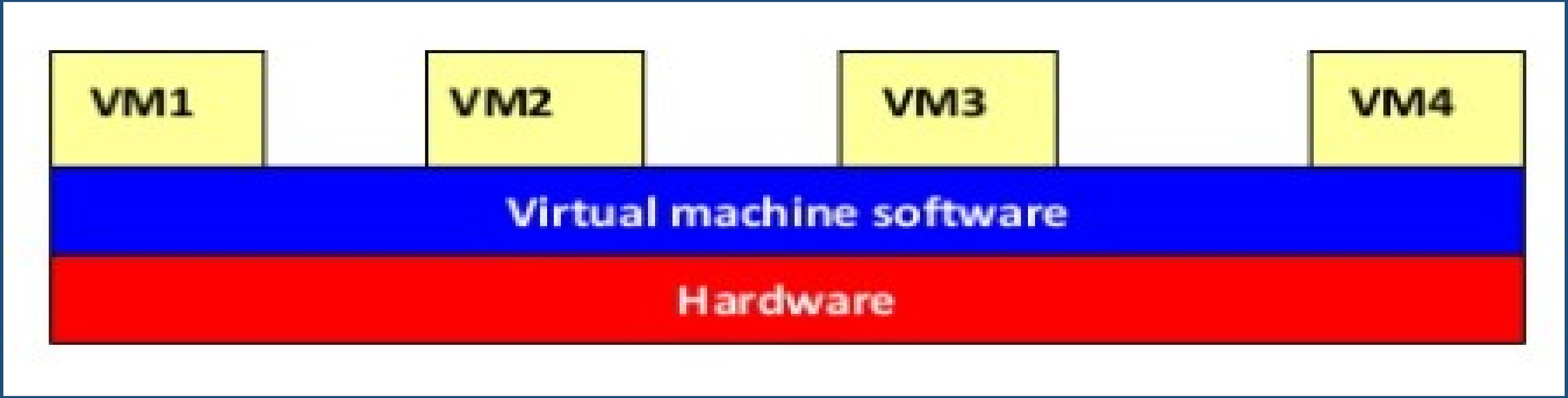


Virtual Machine Approach

- *A virtual machine software layer on the hardware of the machine gives the illusion that all machine hardware (i.e. the processor, main memory, secondary storage..) is the sole disposal of each user.*
- The virtual machine software creates this illusion by appropriately **time-multiplexing the system resources** among all the users of the machine.
- Virtual machine is a **copy** of the bare hardware of the system.
- A user can also run a single-user operating system on this virtual machine.
- The virtual machine concept provides flexibility in that it allows different operating systems to run on different virtual machines.
- This concept does not interfere with other users of the machine.
- Virtual machine software is **huge and complex.**
- Example : IBM 370 system with the virtual machine software VM/370.

Virtual Machine Approach

- Illusion of multiple instances of hardware



Types of Advanced Operating Systems

- The impetus for advanced operating systems has from 2 dimensions.

  1. it has come from advanced operating systems of multicomputer systems and is driven by a wide variety of high-speed architectures

  2. advanced operating systems driven by applications

Types

1. Distributed operating systems
2. Multiprocessor operating systems
3. Database operating systems
4. Real-time operating systems

Types of Advanced Operating Systems

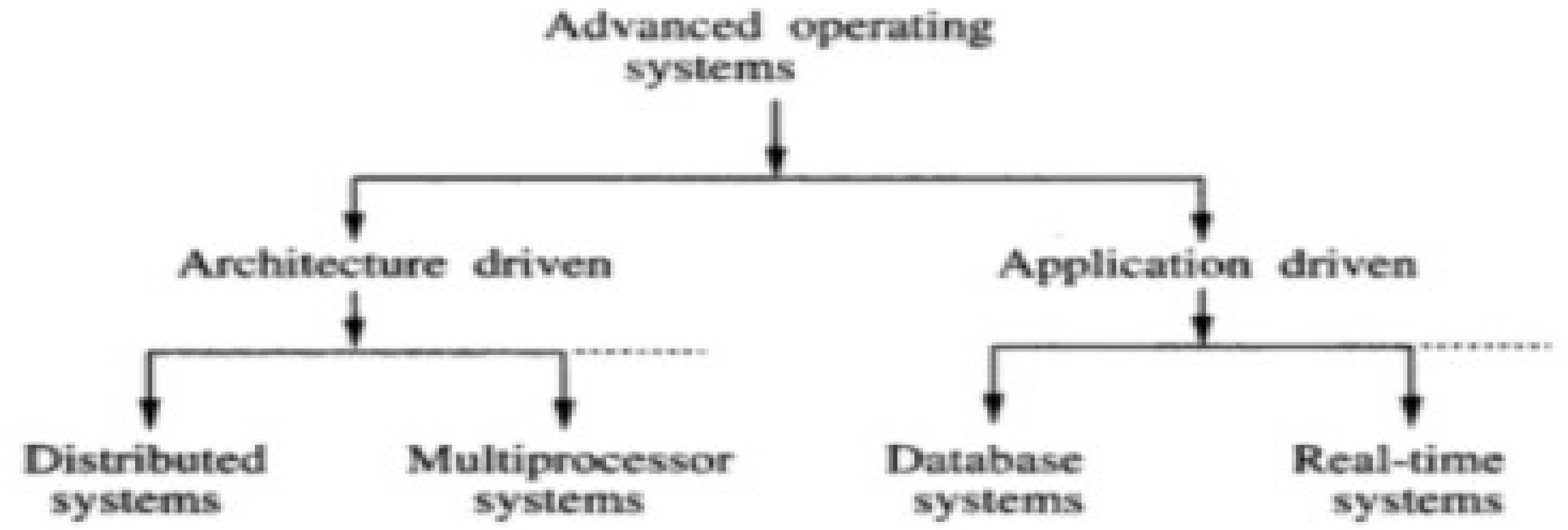


FIGURE 1.3
A classification of advanced operating systems.

1. Distributed operating systems

- Are operating systems for a network of autonomous computers connected by a communication network.
- Controls and manages the hardware and software resources of a distributed system such that its users view the entire system as a powerful monolithic computer system.
- When a program is executed in a distributed system, the user is not aware of where the program is executed or of the location of the resources accessed.
- Practical issues:

  - lack of shared memory

  - lack of global clock

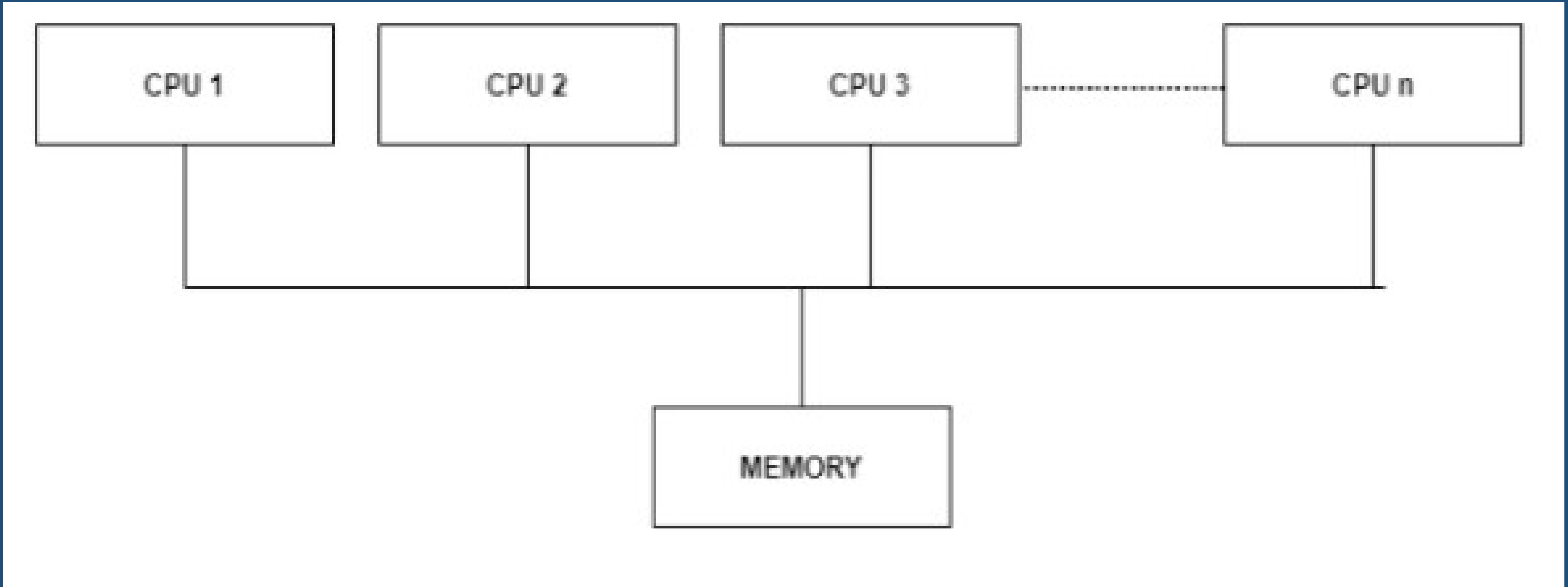
  - unpredictable communication delays.

2. Multiprocessor operating systems

- Consists of a set of processors that shares a set of physical memory blocks over interconnection network.
- Is a tightly coupled system where the processors share an address space.
- Controls and manages the hardware and software resources such that users view the system as a powerful uniprocessor system
- User is not aware of the presence of multiple processors and the interconnection network
- Practical issues:

  - increased complexity of synchronization, scheduling, memory management, protection and security.

2. Multiprocessor operating systems



Multiprocessing Architecture

3. Database operating systems

- Must support
  1. the concept of transaction, operations to store, retrieve, and manipulate a large volume of data efficiently
  2. Primitives for concurrency control and system failure recovery
- Must have buffer management system to store temporary data and data retrieved from secondary storage

4. Real-time operating systems

- Jobs have completion deadlines
- Major issue in the design is the scheduling of jobs in such a way that a maximum number of jobs satisfy their deadlines. Other issue include designing languages and primitives to effectively prepare and execute a job schedule

Synchronization Mechanisms

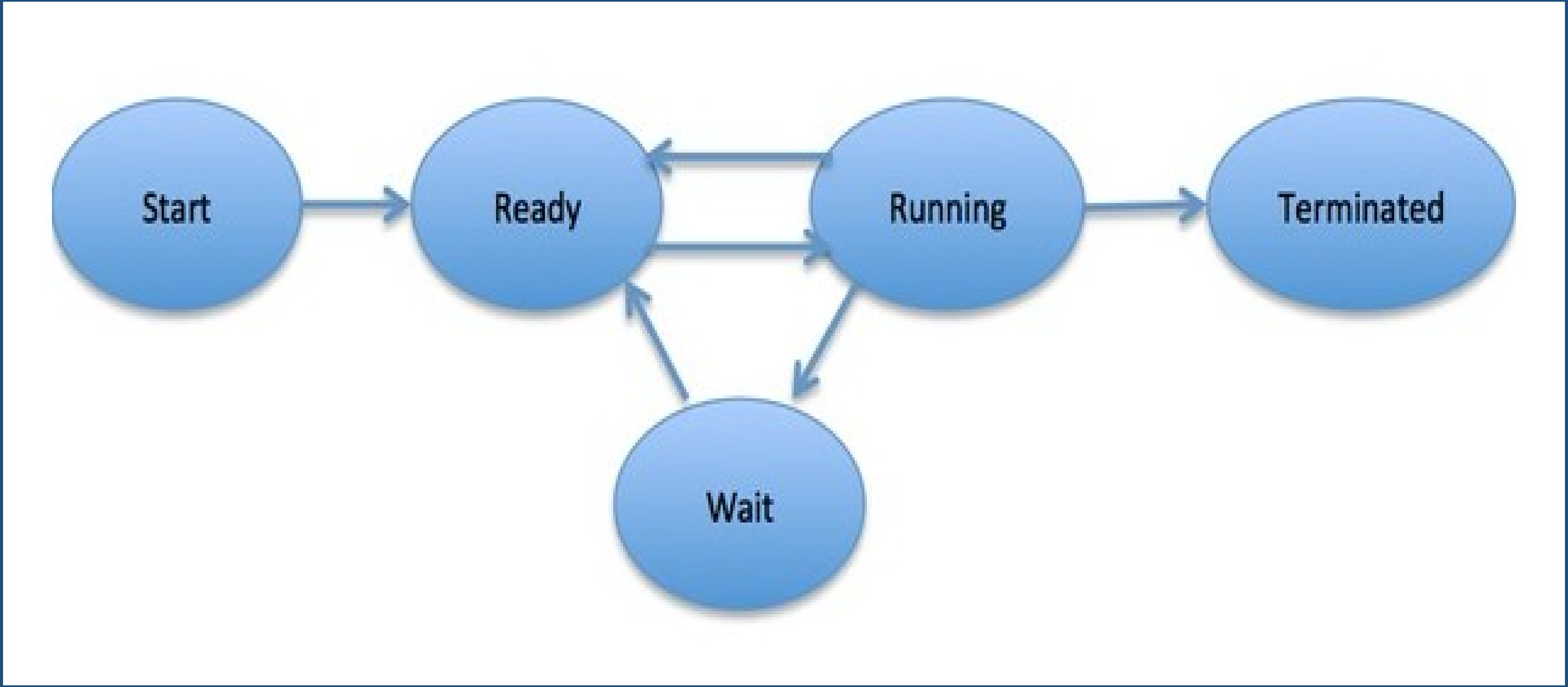
Concept of a Process

- The notion of a process is fundamental to operating systems.
- A process is a program who execution has started but is not yet complete (i.e. a program in execution)

States of a process

1. Running – the processor is executing the instructions of the corresponding process.
2. Ready – the process is ready to be executed, but the processor is not available for the execution of this process.
3. Blocked – the process is waiting for an event to occur (examples of event – an I/O operation waiting to be completed, memory to be made available, a message to be received)
4. A data structure called Process Control Block (PCB) stores complete information about a process, such as id, process state, priority, privileges, virtual memory address translation maps etc….

Process Life Cycle



Serial Processes

- Two processes are serial if the execution of one must be complete before the execution of the other can start.

- **Serial processing performs a single task at a time.**

Concurrent processes

- Two processes **are concurrent if their execution can overlap in time**; that is execution of the second process starts before the first process completes.
- Two processes are said to be concurrent, if they are not serial.

1. **Physical concurrency** – In a single CPU system, physical concurrency is due to the concurrent execution of the CPU and I/O.
2. **Logical concurrency** – If a CPU interleaves the execution of several processes, logical concurrency is obtained.

- Concurrent processes interact through any of the following mechanisms
  - Shared variables – the processes access (read or write) a common variable or common data.
  - Message Passing – the processes exchange information with each other by sending and receiving messages

Concept of thread

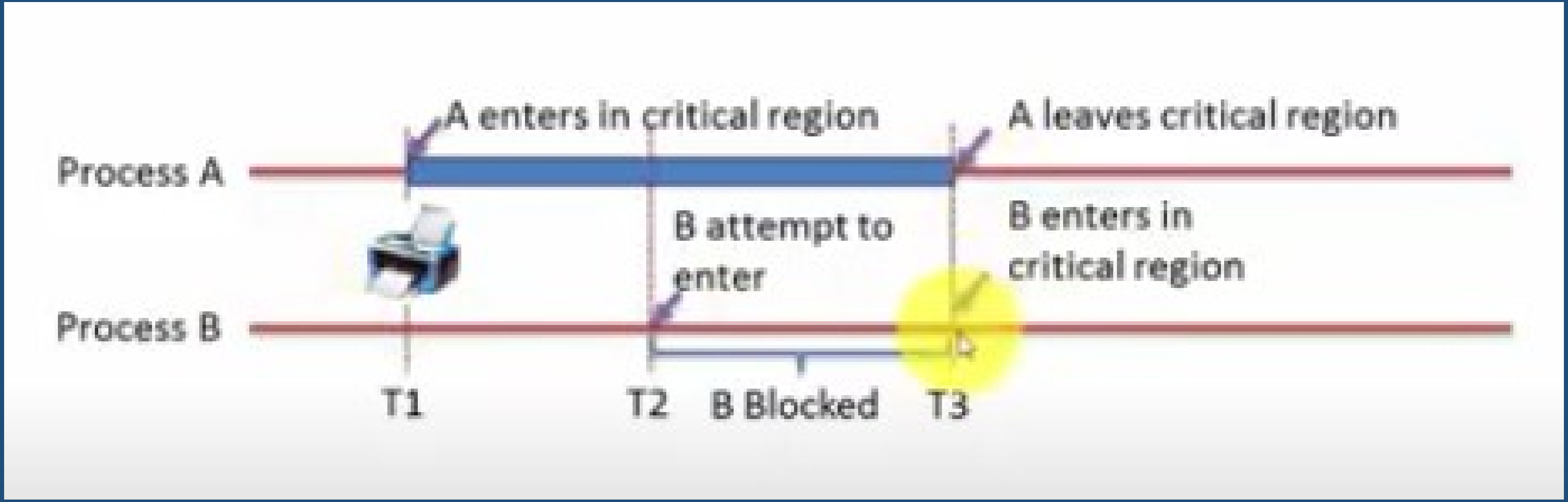
- A process has a single address space and a single thread of control with which to execute a program within that address space.
- To execute a program, a process has to initialize and maintain **state information.**
- The state information is *comprised of* page tables, swap images, file descriptors, outstanding I/O requests, saved register values, etc.
- This information is maintained as per program basis and as per process basis.
- A single thread executes a portion of a program, cooperating with other threads concurrently executing within the same address space.
- Each thread makes use of a separate program counter and a stack of activation records (that describe the state of the execution), and a control block.
- A control block contains the state information necessary for thread management, such as putting thread into a ready list and for synchronizing with other threads.
- Most of the information that is part of a process is common to all the threads

The critical section problem

The critical section problem

- When concurrent processes (or threads) interact through a shared variable, the integrity of the variable may be violated if access to the variable is not coordinated.
- Examples of integrity violations:
  - The variable does not record all the changes
  - A process may read inconsistent values
  - The final value of the variable may be inconsistent
- Solution to this problem requires that processes can be synchronized such that only one process can access the variable at any one time. This problem is called mutual exclusion.
- A critical section is a code segment in a process in which a shared resource is accessed.
- **Mutual Exclusion: A condition in which there is a set of processes, only one of which is able to access a given resource.**
- Mutual exclusion (MUTEX) is a program object that prevents simultaneous access to shared resource.
- This concept is used in concurrent programming with a critical section, a piece of code in which processes or threads access a shared resources.

Critical Section



A solution to the problem of mutual exclusion

- Only one process can execute its critical section at any one time.
- When no process is executing in its critical section, any process that requests entry to its critical section must be permitted to enter without delay.
- When two or more processes compete to enter their respective critical sections, the section cannot be postponed indefinitely.
- No process can prevent any other process from entering its critical section indefinitely; that is, every process should be given a fair chance to access the shared resource.

Early mechanisms for mutual exclusion

- **Busy waiting** – a process that cannot enter its critical section continuously tests the value of a status variable to find if the shared resource is free. The status variable records the status of the shared resource. Problems of this approach are the wastage of CPU cycles and memory access bandwidth.
- **Disabling interrupts** – a process disables interrupts before entering the critical section and enables the interrupts immediately after exiting the critical section. Mutual exclusion is achieved because a process is not interrupted during the execution of its critical section and thus excludes all other processes from entering their critical section.(for uniprocessor systems)
- **Test-and-set instruction-** is a special instruction used to achieve mutual exclusion. This instruction (typically completed in one clock cycle) performs a single indivisible operation on a designated / specific memory location. When this instruction is executed, a specified memory location is checked for a particular value; if they match, the memory location's contents are altered. This instruction can be used as a building block for busy waiting or can be incorporated into schemes that relinquish the CPU when the instruction fails.

SEMAPHORES

Semaphores

- **A semaphore is a high-level construct used to synchronize concurrent processes.**

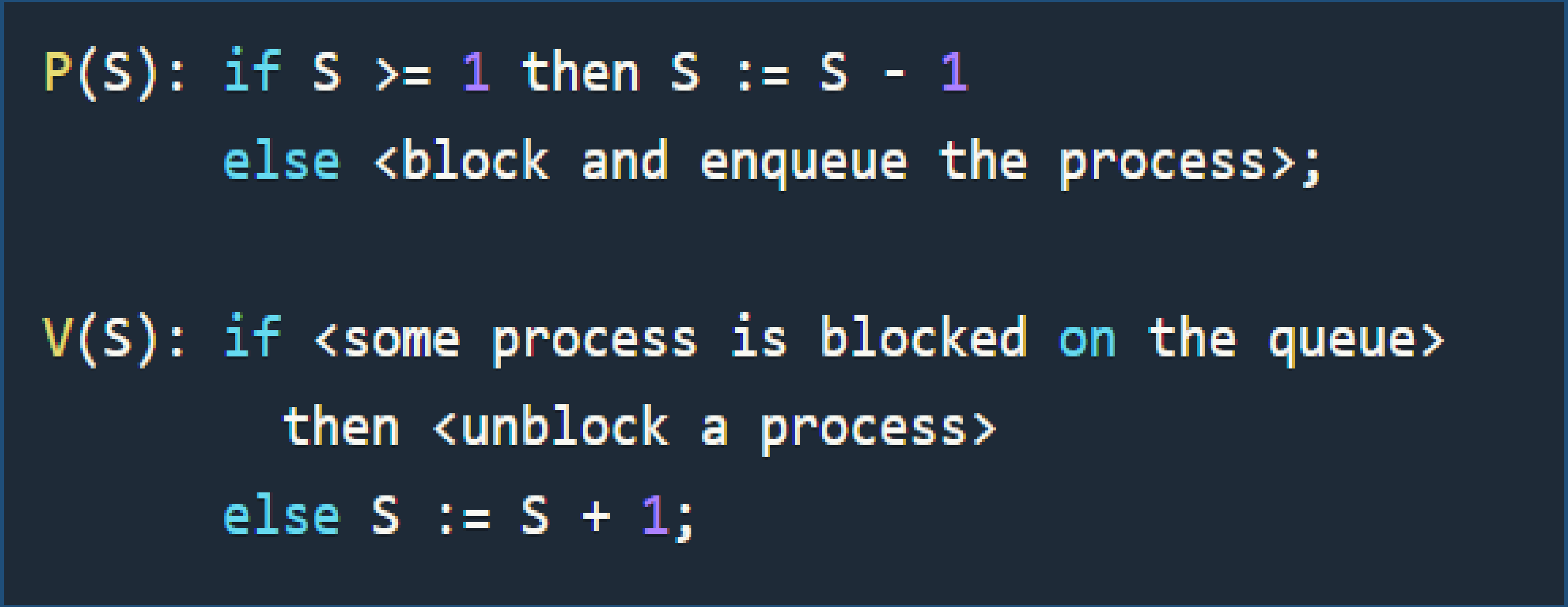
- A new and very significant technique for managing concurrent processes by using the value of a simple integer variable to synchronize the progress of interacting processes.

- This integer variable is called a **semaphore**.

- So it is basically a synchronizing tool and is accessed only through two low standard atomic operations, **wait** and **signal** designated by P(S) and V(S) respectively.

Semaphores

- In very simple words, the **semaphore** is a variable that can hold only a non-negative Integer value, shared between all the threads, with operations **wait** and **signal**, which work as follow:



```
P(S): if S >= 1 then S := S - 1
      else <block and enqueue the process>;

V(S): if <some process is blocked on the queue>
        then <unblock a process>
      else S := S + 1;
```

Semaphores

- **Wait**: This operation decrements the value of its argument S, as soon as it would become non-negative(greater than or equal to 1).
- This Operation mainly helps you to control the entry of a task into the critical section. In the case of the negative or zero value, no operation is executed. wait() operation was originally termed as P; so it is also known as **P(S) operation**.
- The definition of wait operation is as follows:

```
wait(S)
{
while (S<=0);//no operation
S--;
}
```

Semaphores

- **Signal**: Increments the value of its argument S, as there is no more process blocked on the queue.

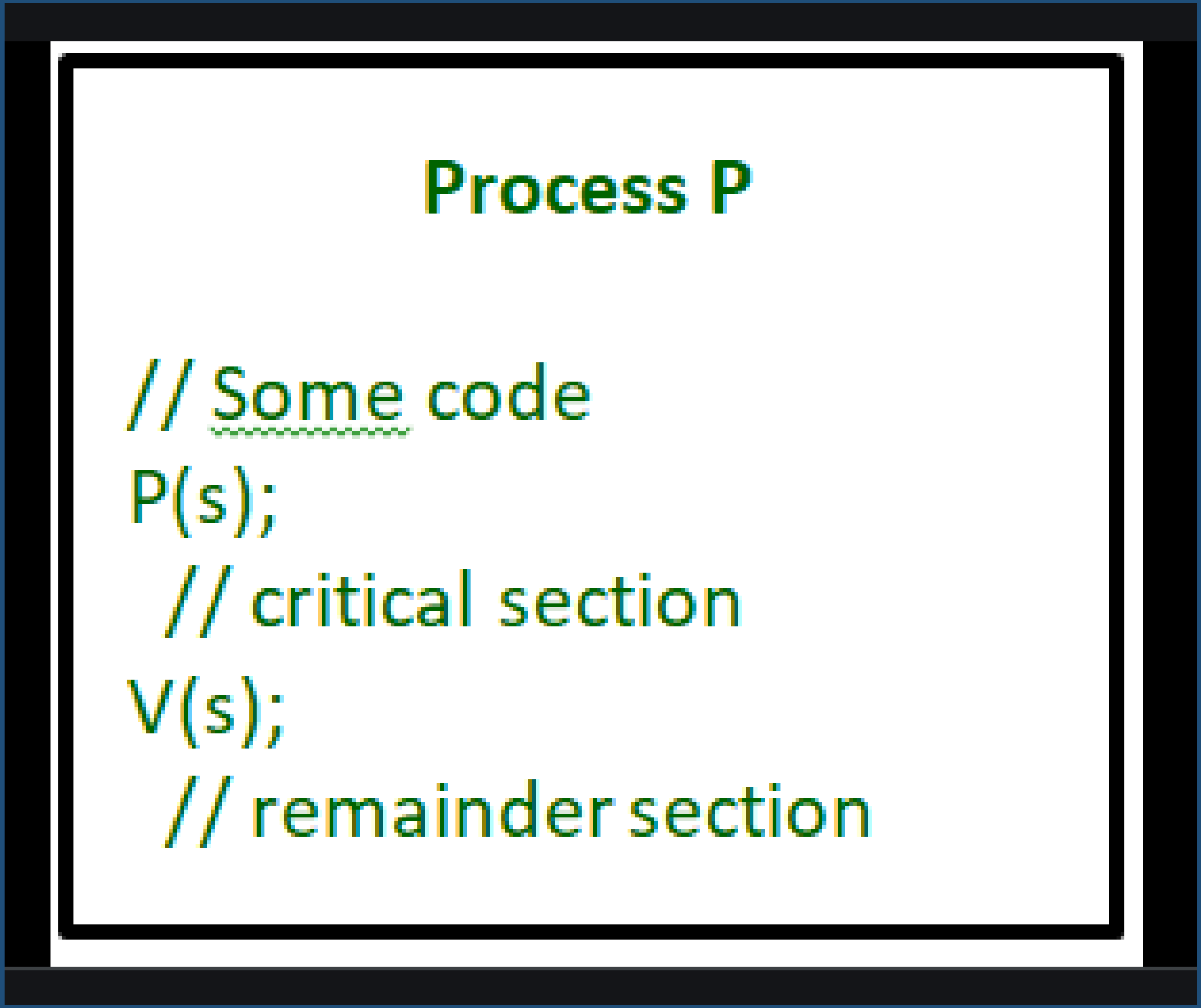
- This Operation is mainly used to control the exit of a task from the critical section.

- Signal() operation was originally termed as V; so it is also known as **V(S) operation**.

- The definition of signal operation is as follows:

```
signal(S)
{
 S++;
}
```

Semaphores

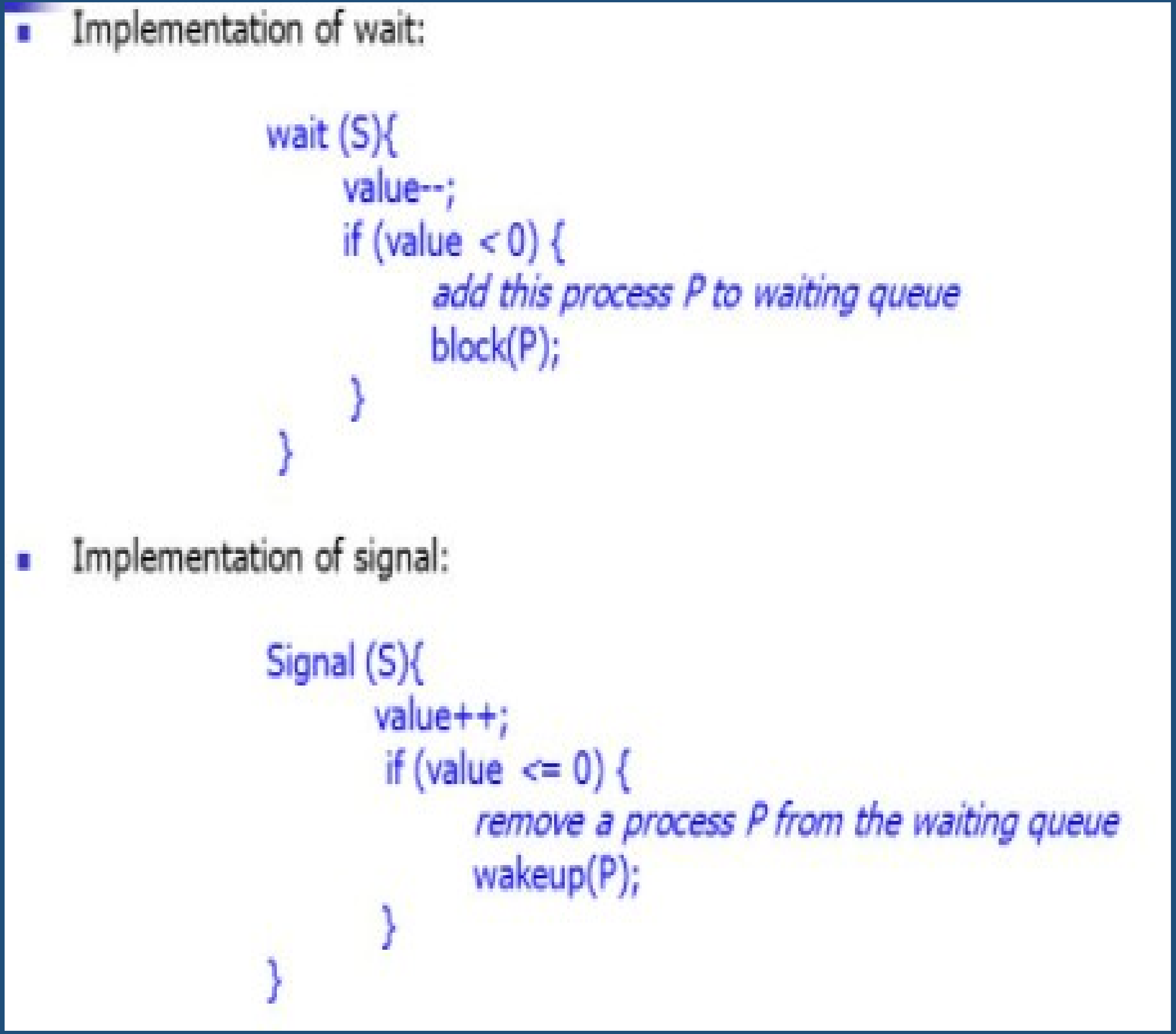


Process P

```
// Some code
P(s);
  // critical section
V(s);
  // remainder section
```

Semaphores

•



- Implementation of wait:

```
wait (S){
      value--;
      if (value  < 0) {
             add this process P to waiting queue
             block(P);
      }
}
```

- Implementation of signal:

```
Signal (S){
        value++;
         if (value  <= 0) {
                remove a process P from the waiting queue
                wakeup(P);
         }
}
```

Properties of Semaphores

- It's simple and always have a non-negative integer value.
- Works with many processes.
- Can have many different critical sections with different semaphores.
- Each critical section has unique access semaphores.
- Can permit multiple processes into the critical section at once, if desirable.

Types of Semaphores

- Semaphores are mainly of two types in Operating system:

1) **Binary Semaphore**

2) **Resource Counting Semaphores**

Binary Semaphore

- It is a special form of semaphore used for implementing mutual exclusion, hence it is often called a **Mutex**.
- A binary semaphore is initialized to 1 and only takes the values 0 and 1 during the execution of a program.
- In Binary Semaphore, the wait operation works only if the value of semaphore = 1, and the signal operation succeeds when the semaphore= 0.
- Binary Semaphores are easier to implement than counting semaphores.

Counting Semaphores

- These are used to implement **bounded concurrency**.
- The Counting semaphores can range over an **unrestricted domain**.
- These can be used to control access to a given resource that consists of a finite number of Instances.
- Here the semaphore count is used to indicate the number of available resources.
- If the resources are added then the semaphore count automatically gets incremented and if the resources are removed, the count is decremented.
- Counting Semaphore has no mutual exclusion.

Example of Use

- Here is a simple step-wise implementation involving declaration and usage of semaphore.

```
Shared var
mutex: semaphore = 1;
Process i
begin
. .
P(mutex);
execute CS;
V(mutex);
. .
End;
```

.

Other Synchronization Problems

Other Synchronization Problems

- In addition to Mutual Exclusion, there are many other situations were process synchronization is necessary.

1. The Dining Philosophers Problem.
2. The Producer- Consumer Problem.
3. The Readers – Writers Problem.

1. The Dining Philosophers Problem.

- The dining philosophers problem states that there are 5 philosophers sharing a circular table and they eat and think alternatively. There is a bowl of rice for each of the philosophers and 5 chopsticks. A philosopher needs both their right and left chopstick to eat.
- A philosopher alternates between two phase: ***thinking and eating***
- A hungry philosopher may only eat if there are both chopsticks available (left and right).
- Otherwise a philosopher puts down their chopstick and begin thinking again.
- Once a philosopher starts eating, the forks are not released until the eating phase is over.
- When the eating phase concludes, both forks are put back in their original position and the philosopher re-enters the thinking phase.
- ***The dining philosopher is a classic synchronization problem as it demonstrates a large class of concurrency control problems.***

Dining Philosophers Problem

- No two philosophers can use the same fork at the same time (mutual exclusion)

- No philosopher must starve to death (avoid deadlock and starvation)

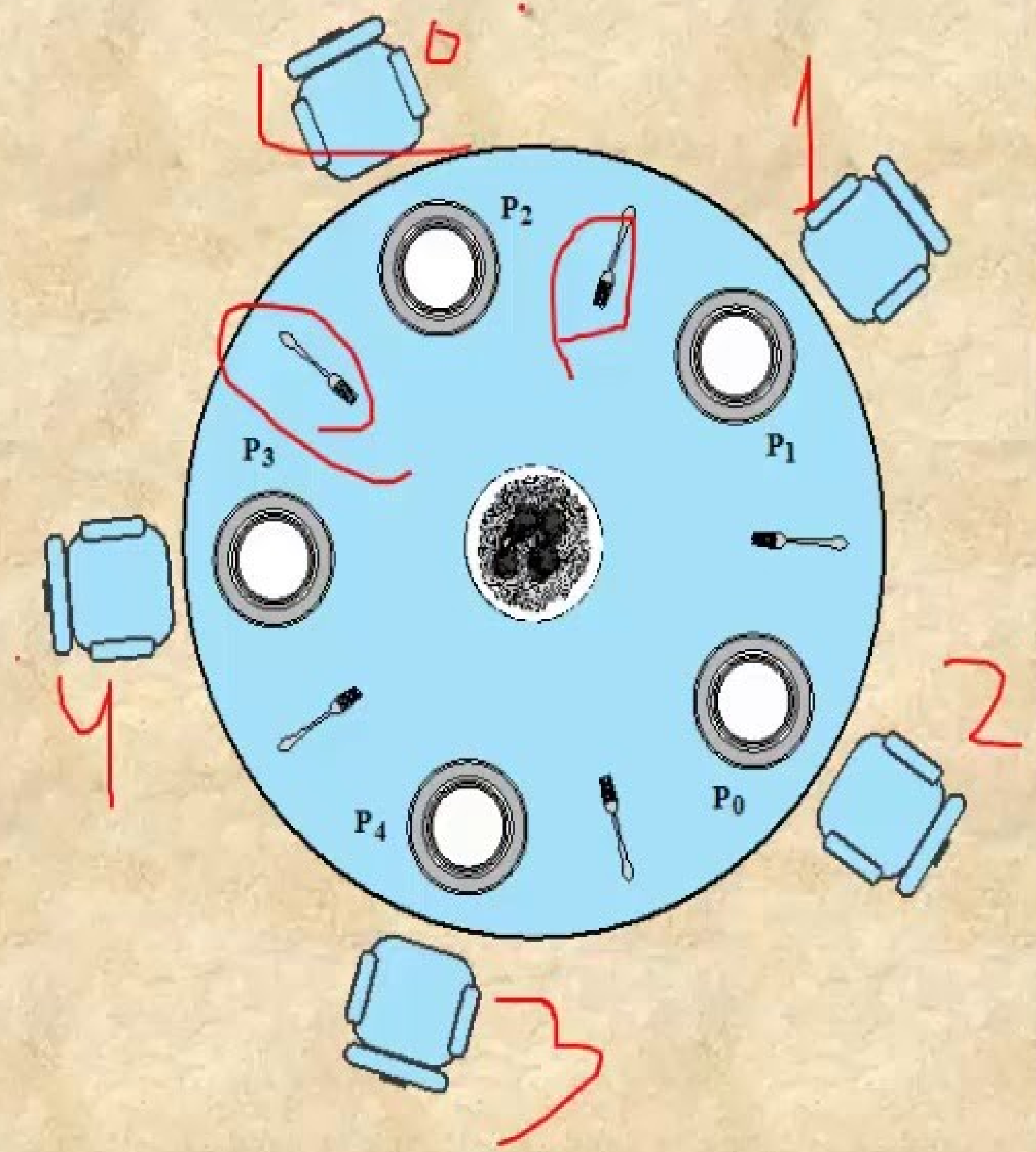


Figure 6.11 Dining Arrangement for Philosophers

The Dining Philosophers Problem.

- No two neighbouring philosophers can eat simultaneously.
- The act of picking up a fork by a philosopher must be a critical section.

2. The Producer - Consumer Problem

- In computing, the producer-consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem.
- The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue.
- The producer's job is to generate data, put it into the buffer, and start again.
- At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

The Producer- Consumer Problem

- **Problem**

i. To make sure that the producer won’t try to add data into the buffer if it’s full.
ii. The consumer won’t try to remove data from an empty buffer.

- **Solution**
  The producer is to either go to sleep or discard data if the buffer is full.
- The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again.
- In the same way, the consumer can go to sleep if it finds the buffer to be empty.
- The next time the producer puts data into the buffer, it wakes up the sleeping consumer.
- An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

The Readers – Writers Problem

- The readers-writers problem relates to an object such as a file that is shared between multiple processes.

- Some of these processes are readers i.e. they only want to read the data from the object and some of the processes are writers i.e. they want to write into the object.

- Reader Process: Simply Read the Information from file.

- Writer Process: may change the information in the file.

- The basic synchronization constraint is that any number of readers should be able to concurrently access the file, but only one writer can access the file at a given time.

The Readers – Writers Problem

- **Versions:**
  - Based on whether readers or writers are given priority.

  - **Reader's Priority:** Arriving readers receive priority over waiting writers.

    - A waiting or arriving writer gain access to file only when there are no readers in the system.

    - Writers may starve.

  - **Writer's Priority**: Arriving writer receives priority over waiting readers.

    - A waiting or an arriving reader gain access to file only when there are no writers in the system.

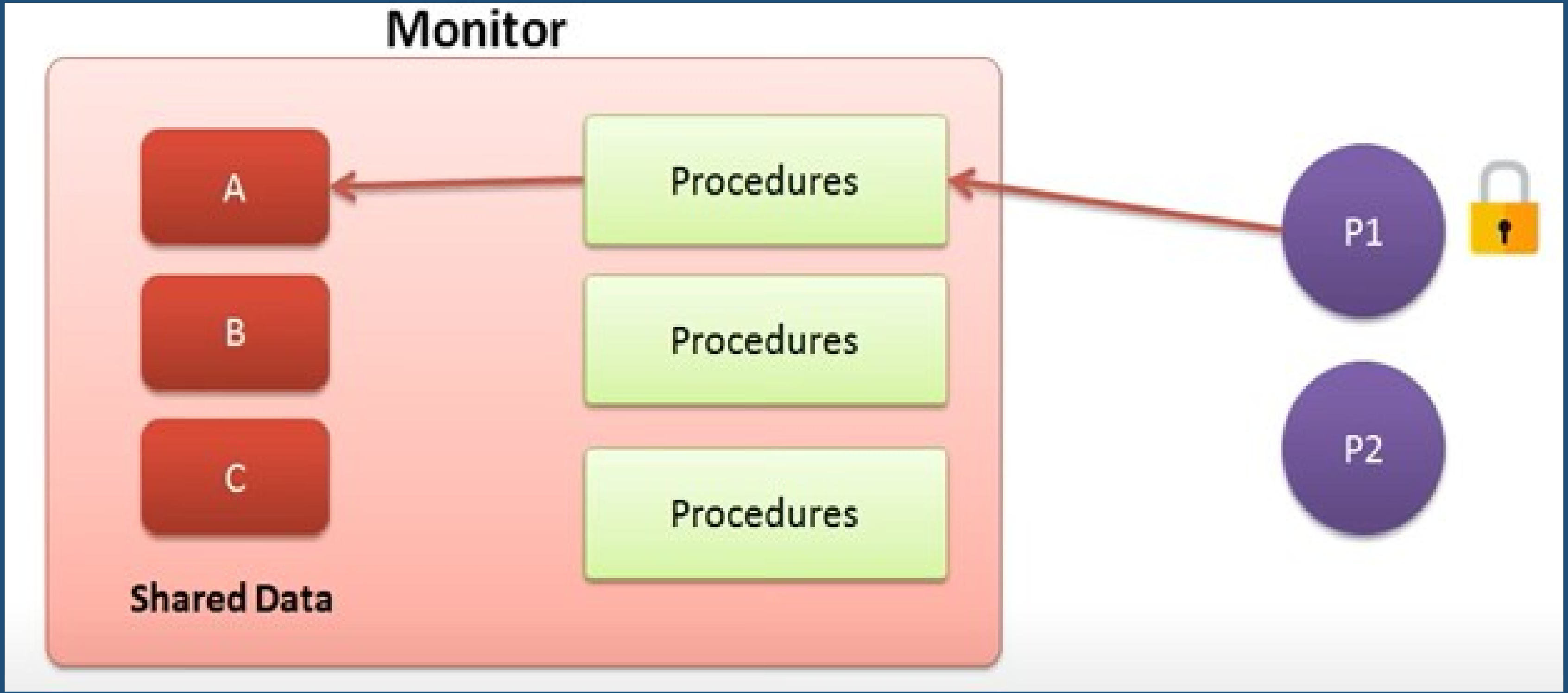
    - Readers may starve.

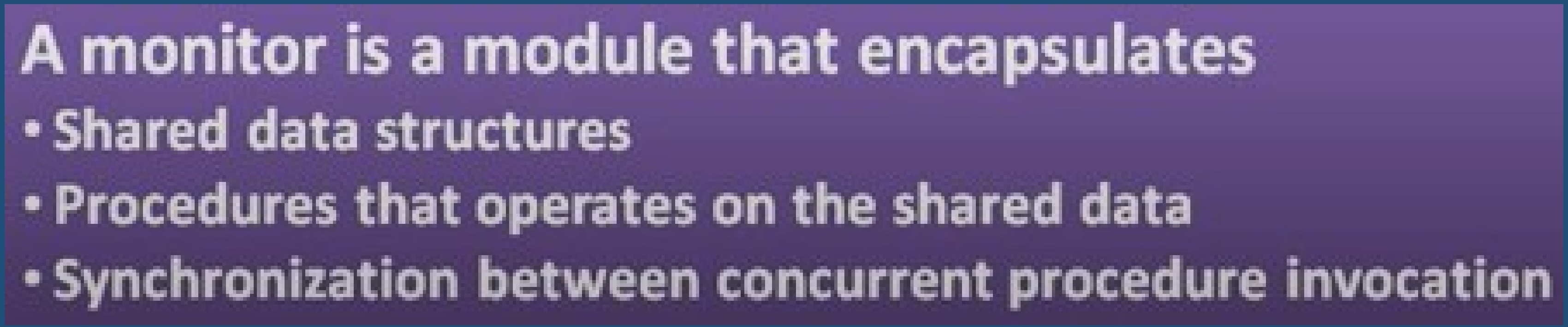
Monitors

Monitors in Operating System

- Monitors are used for process synchronization.
- With the help of programming languages, we can use a monitor to achieve mutual exclusion among the processes.
- In other words, monitors are defined as the construct of programming language, which helps in controlling shared data access.
- The Monitor is a module or package which encapsulates shared data structure, procedures, and the synchronization between the concurrent procedure invocations.
- A monitor consists of procedures, the shared object (resource), and administrative data.
- Procedures are the gateway to the shared resource and are called by the processes needing to access the resource.

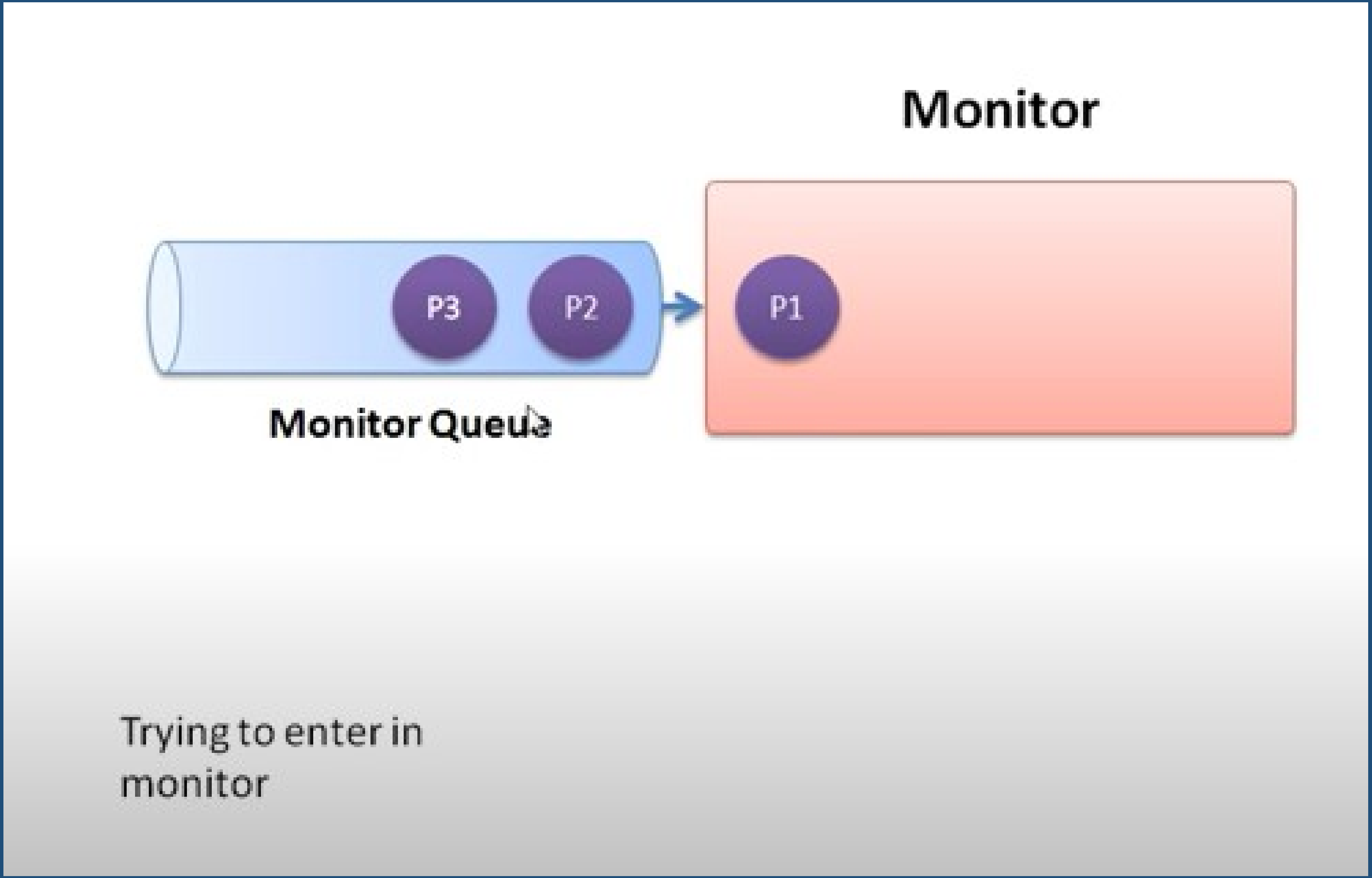
Monitor




A monitor is a module that encapsulates
- Shared data structures
- Procedures that operates on the shared data
- Synchronization between concurrent procedure invocation

Monitor

•



Trying to enter in monitor

Characteristics of Monitors

1. Inside the monitors, we can only execute one process at a time.
2. Monitors are the group of procedures, and condition variables that are merged together in a special type of module.

3. If the process is running outside the monitor, then it cannot access the monitor's internal variable. But a process can call the procedures of the monitor.

4. Monitors offer high-level of synchronization.

5. Monitors were derived to simplify the complexity of synchronization problems.

6. There is only one process that can be active at a time inside the monitor.

Monitors

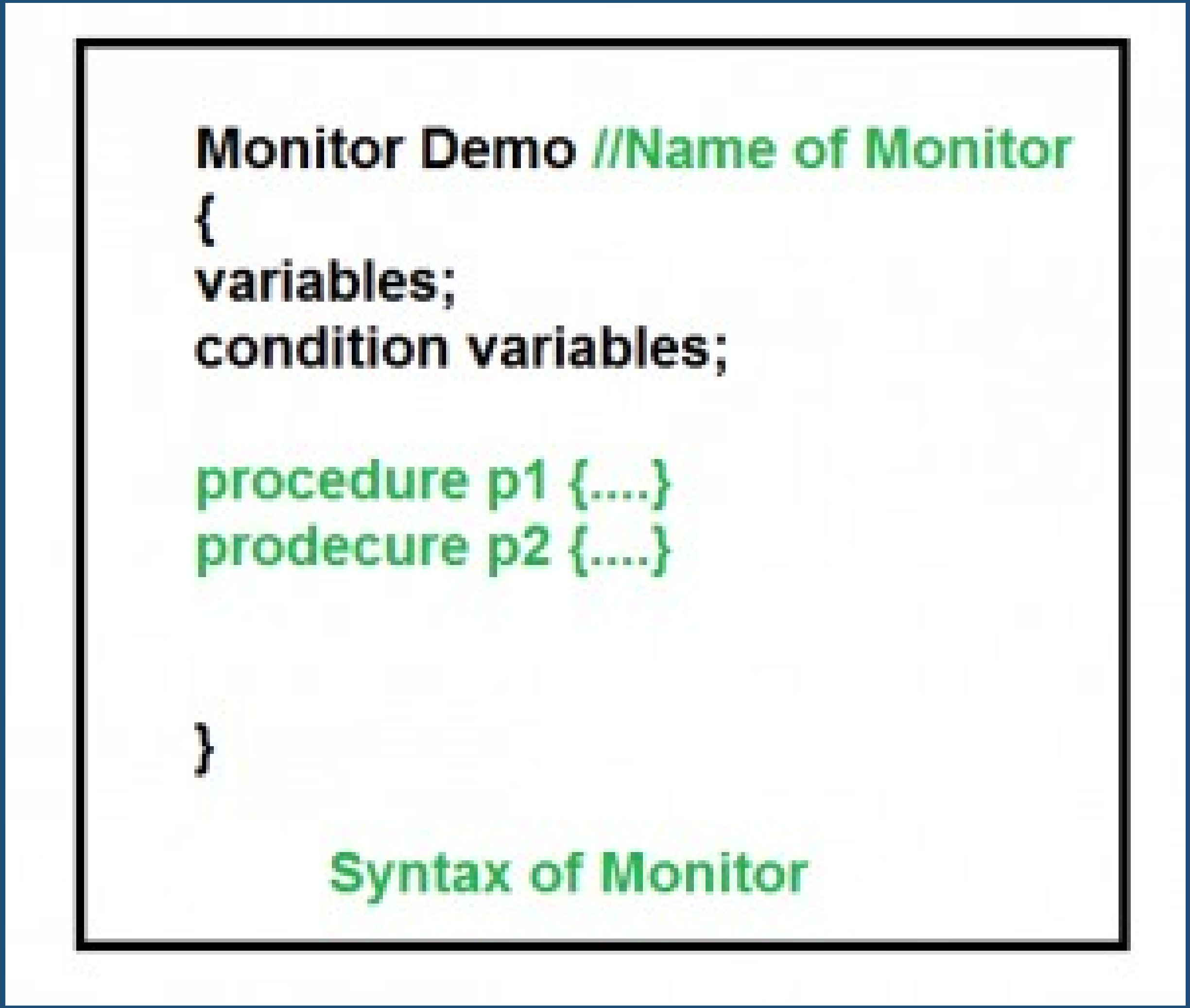
- Since the main function of a monitor is to control access to a shared resource, it should be able to delay and resume the execution of the processes calling monitor's procedures.
- Synchronization of processes is accomplished with two operations:
    - Wait
    - Signal
- Both operations are executed within the monitor's procedures.
- Executing wait operation suspends the caller process.
- Executing a signal operation causes exactly one waiting process to immediately regain control of the monitor.

Monitors

- The signalling process is suspended on an urgent queue.
- The processes in an urgent queue have a higher priority for regaining control of the monitor than the process waiting in the monitor’s entry queue.
- When a waiting process is signalled, it starts execution from the very next statement following the wait statement.
- If there are no waiting processes, the signal has no effect.

Monitors

•



```
Monitor Demo //Name of Monitor
{
variables;
condition variables;

procedure p1 {....}
prodecure p2 {....}


}
```

Syntax of Monitor

Monitors

- ***Condition Variables:***
  - A condition variable associated with wait and signal operations helps to distinguish the processes to be blocked or unblocked.
  - The conditional variable is associated with a queue of processes that are currently waiting on that condition.
  - The operation <condition variable>.queue returns true if the queue associated with the condition variable is not empty.
  - The syntax of wait and signal operations associated with a condition is:

```
< condition variable > .wait;
< condition variable > .signal;
```

Monitors

- ***Condition Variables:***
  - The two different operations are performed on the condition variables of the monitor.
  - let say we have 2 condition variables
    condition x, y;
- **Wait operation**
  - x.wait() : Process performing wait operation on any condition variable are suspended. The suspended processes are placed in block queue of that condition variable.
  - Each condition variable has its unique block queue.
- **Signal operation**
  - x.signal(): When a process performs signal operation on condition variable, one of the blocked processes is given chance.

Monitors

- **Advantages of Monitor:**
  - Monitors have the flexibility in scheduling the processes waiting in queues. First – in-first-out is generally used with queues.

  - less error prone than using techniques such as semaphore.

- **Disadvantages of Monitor:**
  - Absence of concurrency if a monitor encapsulates the resource, since only one process can be active within a monitor at a time.
- Monitors have to be implemented as part of the programming language . The compiler must generate code for them.

  - Some languages that do support monitors are Java, C#,V isual Basic, Ada ets.

Serializers

Serializers

- **A synchronization mechanism to overcome some of the deficiencies of monitors.**
- **Serializers allow concurrency inside and thus the shared resource can be encapsulated in a serializer.**
- **Serializers replace explicit signalling required by monitors with automatic signalling.**
- Serializers are abstract data types defined by a set of procedures (or operations) and can encapsulate the shared resources to form a protected resource object.
- Serializers are abstract data types very similar to monitor.
- When a process access a serializer it gains control of the serializer and is said to be in possession of the serializer.
- But in the procedures of a serializer there are certain regions in which multiple process can be active.
- These regions are known as **Hollow regions**.

Serializers

- As soon as a process enters a hollow region, it releases the serializer so that some other process can access it.
- Thus concurrency is achieved in the hollow regions of a serializer.
- Remember that in a hollow region the process just releases the serializer and does not exit it. So that the process can regain control when it gets out of the hollow region.

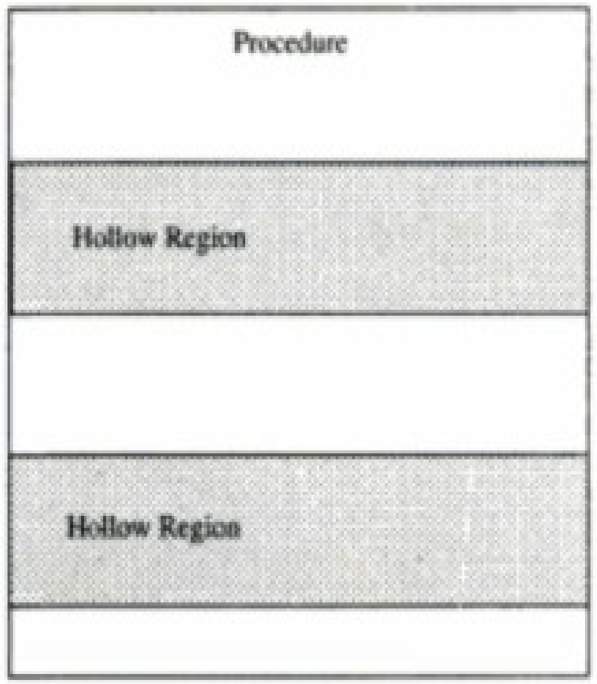


FIGURE 2.7
The structure of a procedure of a serializer.

Serializers

- **W**henever a process requests to gain or regain access of a serializer certain conditions are checked.
- The process is held in a waiting queue until the condition is true. This is accomplished using an *enque* operation.
- The syntax of the enque command is

    enque (<priority>, <queue-name>) **until** (<condition>)

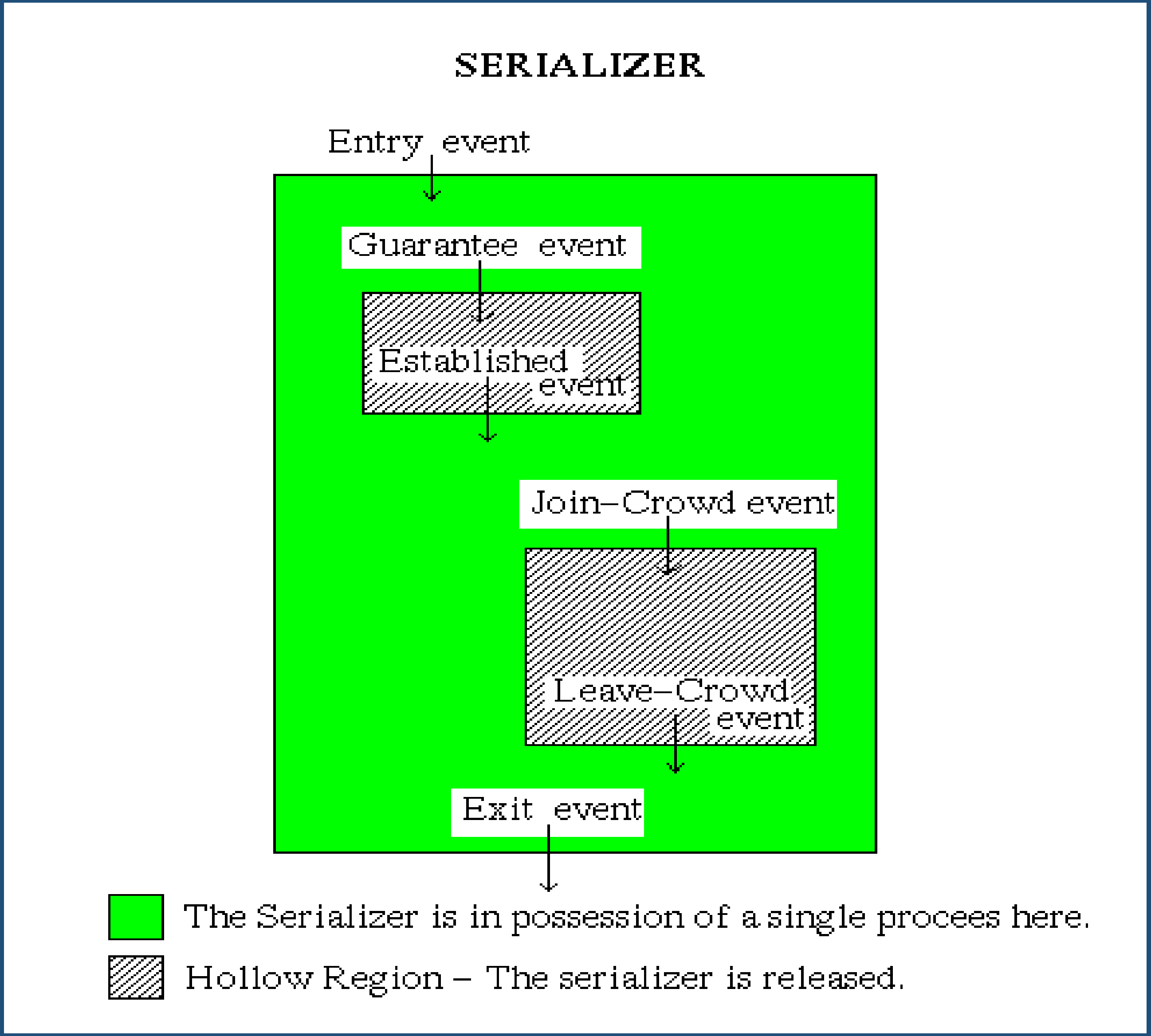
- The queue name specifies the name of the queue in which the process has to be held and the priority options specifies the priority of the process to be delayed.

Serializers

- A hollow region in a procedure is specified by a *join-crowd* operation.
- The syntax of the join-crowd command is

  join-crowd (<crowd>) **then** <body> **end**
- On invocation of a join-crowd operation, possession of the serializer is released, the identity of the process invoking the join-crowd is recorded in the *crowd*, and the list of statements in the *body* is executed.
- When the process completes execution of the body, a *leave-crowd* operation is executed.
- As a result of the leave-crowd operation the process regains control of the serializer.
- Please note that, if the serializer is currently in possession of some other process then the process executing the leave crowd operation will result in a wait queue.

The operation of a serializer can be explained by the following figure:


SERIALIZER
Entry event
Guarantee event
Established event
Join–Crowd event
Leave–Crowd event
Exit event
The Serializer is in possession of a single procees here.
Hollow Region – The serializer is released.

- As shown in the above figure, every operation in a serializer can be identified as an event.
- An Entry event can be a request for serializer, in which a condition will be checked.

  (For eg., Is the serializer free for the process to enter ?)
- If the condition is true the process gains control of the serializer.
- Then before the process accesses the resource, a guarantee event is executed.
- The guarantee event results in an established event if the condition is true, else the process releases the control of the serializer and waits in its queue.
- When a resource is available the process enters (Join-Crowd event) the Crowd and accesses the process.
- After completing the job with the resource, the process leaves (Leave-Crowd event) the crowd and regains control of the serializer (if the serializer is available, else it has to wait).
- serializers also allow a *timeout event* which can avoid processes waiting for a condition longer than the specified period.

Serializers

- **Drawbacks**

  - More complex and hence less efficient.

  - Automatic signalling process increases overhead, since whenever a resource or serializer is released all the conditions are to be checked for the waiting processes.

Monitors vs Serializers

- Serializers have several advantages over the monitors.

1. Only one process can execute inside a monitor.
   Only one process can have possession of the serializer at a time. But in the hollow regions the process releases the control and thereby facilitates several process to execute concurrently inside the hollow region of the serializers.
2. Nesting of monitors can cause deadlock. If inner process is waiting the outer one will be tied up.
   Nesting of serializers are allowed in the hollow regions.
3. In monitors the conditions are not clearly stated to exit from a wait state.
   The conditions are clearly stated for an event to occur in a serializer.
4. In monitors - explicit signalling.
   In serializers implicit signalling.

Path Expressions

Path Expressions

- A quite different approach to process synchronization.
- A path expression restricts the set of admissible execution histories of the operations on the shared resource so that no incorrect state is ever reached and it indicates the order in which operations on a shared resource can be interleaved.
- A path expression has the following form

  Path S **end;**
- Where S denotes possible execution histories.
- It is an expression whose variables are the operations on the resource and whose operators are:
  - Sequencing (;)
  - Selection (+)
  - Concurrency ({ })

Path Expressions

- Sequencing (;)
    - Defines a sequencing order among operations.
    - Eg: **path** open; read; close; **end** means that an open must be performed first, followed by a read and a close in that order.
    - There is no concurrency in the execution of these operations.

- Selection (+)
    - It signifies that only one of the operations connected by a + operator can be executed at a given time.
    - Eg: **path** read+write **end** means that only read or only write can be executed at a given time, but the order of execution of these operations does not matter.

Path Expressions

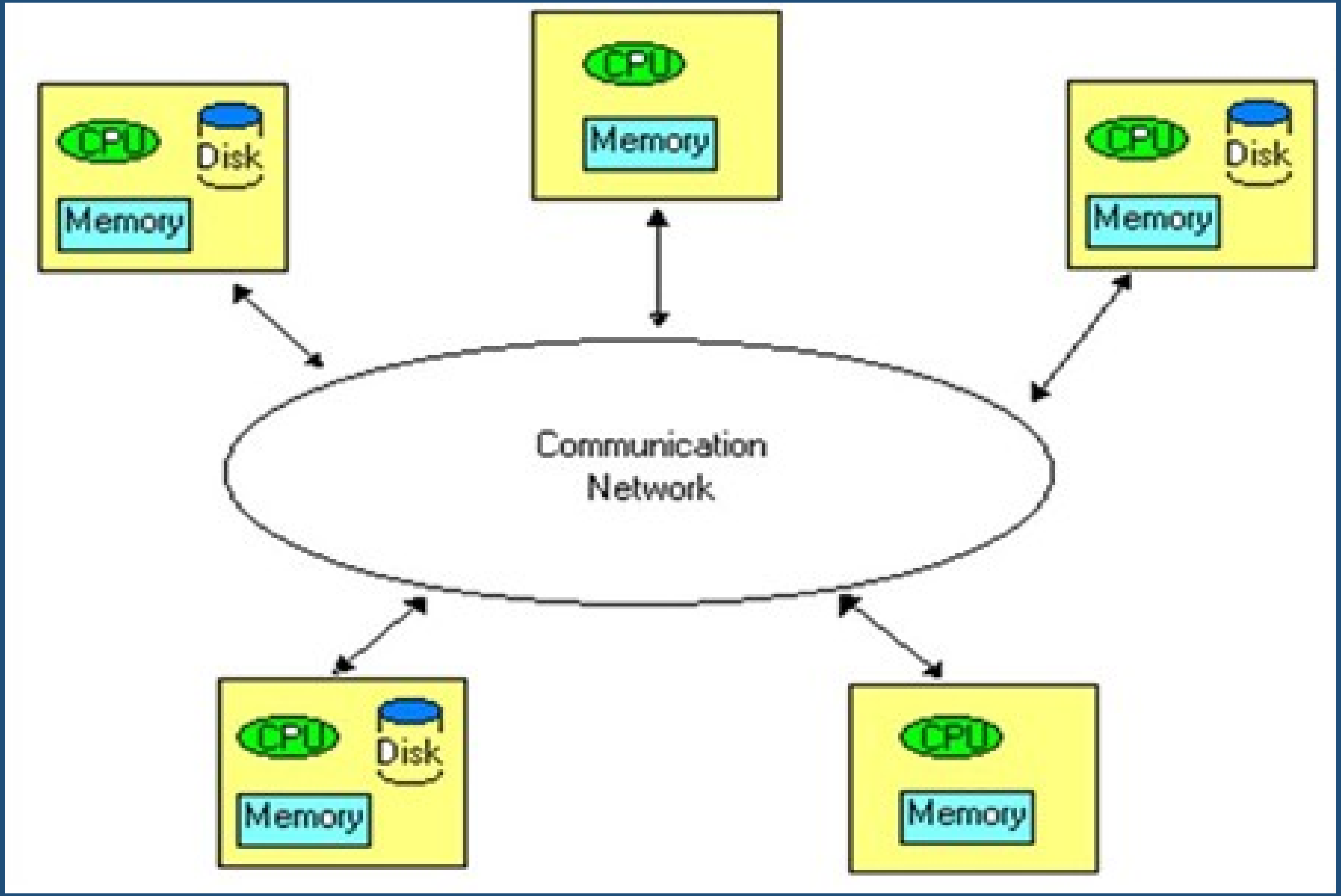
- Concurrency ({ })
  - It signifies that any number of instances of the operation delimited y { and } can be in execution at a time.
  - Eg: **path** {read} **end** means that any number of read operations can be executed concurrently.
  - **path** write; {read} **end** allows either several read operations or a single write operation to be executed at any time.

Distributed Operating Systems

Distributed Operating Systems

- The term distributed system is used to describe a system with the following characteristics:
  - It consists of several computers that do not share a memory or a clock.
  - The computers communicate with each other by exchanging messages over a communication network.
  - Each computers has its own memory and runs its own operating system.
  - The resource owned and controlled by a computer are said to be local to it.

Architecture of a Distributed Systems



Advantages of Distributed Systems

- Resource Sharing
  - A computer can request a service from another computer by sending a request to it over the communication network.
  - Hardware and software resources can be shared among computers.
  - Eg: a printer, a compiler, a text processor, a database at a computer can be shared with remote computers.

- Enhanced Performance
  - Capable for providing rapid response time and higher system throughput.
  - Many task can be concurrently executed at different computers.
  - Can employ a load distributing technique to improve response time.

Advantages of Distributed Systems

- Improved Reliability and Availability
  - Few components of the system can fail without affecting the availability of the rest of the system.
  - Through the replication of data (files and directories) and services, distributed systems can be made fault tolerant.

- Modular Expandability
  - New software and hardware resources can be easily added without replacing the existing resources.

Issues in Distributed Operating Systems

Issues in DOS

- Some Important issues are:
  1. Unavailability of up-to-date global knowledge
  2. Naming
  3. Scalability
  4. Compatibility
  5. Process synchronization
  6. Resource management
  7. Security
  8. Structuring of the operating system

Issues in DOS

1. Unavailability of up-to-date global knowledge

- In case of shared memory system, up-to-date state of all the processes and resources are known.
- Due to the unavailability of a global memory and a global clock, it is practically impossible for a computer to collect up-to-date information about the global state of the distributed computing system.
- A computer does not know the current and complete status of the global state.
- Absence of a global clock arises the question of how to order all the events that occur at different times at different computers present in the system.
- Temporal ordering of events is a fundamental concept in the design and development of distributed systems.

Issues in DOS

2. Naming

- Naming are used to refer to objects.
- Objects that can be named in computer systems include computers, printers, services, files and users.
- Eg: Name Service. [Store names and their physical addresses are used for mapping names to their addresses]
- In distributed system, the directories may replicated and stored at different locations.
- Algorithm depend on structure of names.
- Method of naming objects.

3. Scalability

- Systems grow with time.
- The techniques used in designing a system should not result in system unavailability or degraded performance when growth occurs.

Issues in DOS

4. Compatibility

- Refers to the notion of interoperability among the resources in a system.
- Three levels of compatibility that exist in distributed systems are:
  - *Binary level* – all processors execute the same binary instruction, even though the processors may differ in performance and in input-output. Distributed system cannot include computers with different architectures.
  - *Execution level* –if the same source code can be compiled and executed properly on any computer in the system
  - *Protocol level* – if all system components support a common set of protocols. Individual computers can run different Operating Systems.

Issues in DOS

5. **Process synchronization**
    - The is difficult because of the unavailability of shared memory.

6. **Resource management** –
    - Concerned with making both local and remote resources available to users in an effective manner.
    - Users should be able to access remote resources as easily as they can access local resources.
    - In other words, the specific location of resources should be hidden from the users.

Issues in DOS

7. **Security –**
   - The security of a system is the responsibility of the operating system.
   - Two issues are authentication and authorization
   - Authentication is the process of guaranteeing that an entity is what it claims to be.
   - Authorization is the process of deciding what privileges an entity has and making only these privileges.

8. **Structuring –**
   - Defines how various parts of the operating system are organized.

Communication networks

Communication networks

- All the computers in a distributed system are interconnected through a communication network.

- A computer can exchange messages with other computers and access data stored at another through this network.

- Communication networks are classified into
  1. Wide Area Networks
  2. Local Area Networks

1. Wide Area Networks (WANs)

- Used to **interconnect various devices (such as computers and terminals) spread over a wide geographic area that may cover different cities, states and countries.**
- Also referred to as **Long Haul Networks.**
- The communication facility in a WAN consists of switches that are interconnected by communication links, which are established through telephone lines, satellites, microwave links, or any combination of the three.
- WANs employ a technique known as **point-to-point or store-and-forward** where data is transferred between computers through a series of switches.
- **Switches** are special purpose computers responsible for routing data from one point to another through an appropriate path while avoiding network congestion.
- A path or portion of a path become congested due to heavy data communication through that path or due to limited bandwidth.

1. Wide Area Networks (WANs)

- The data being communicated in a WAN can be lost for any of the following reasons:
  - Switch crashes
  - Communication link failure
  - Limited buffer capacity at switches
  - Transmission error

  etc...

1. Wide Area Networks (WANs)

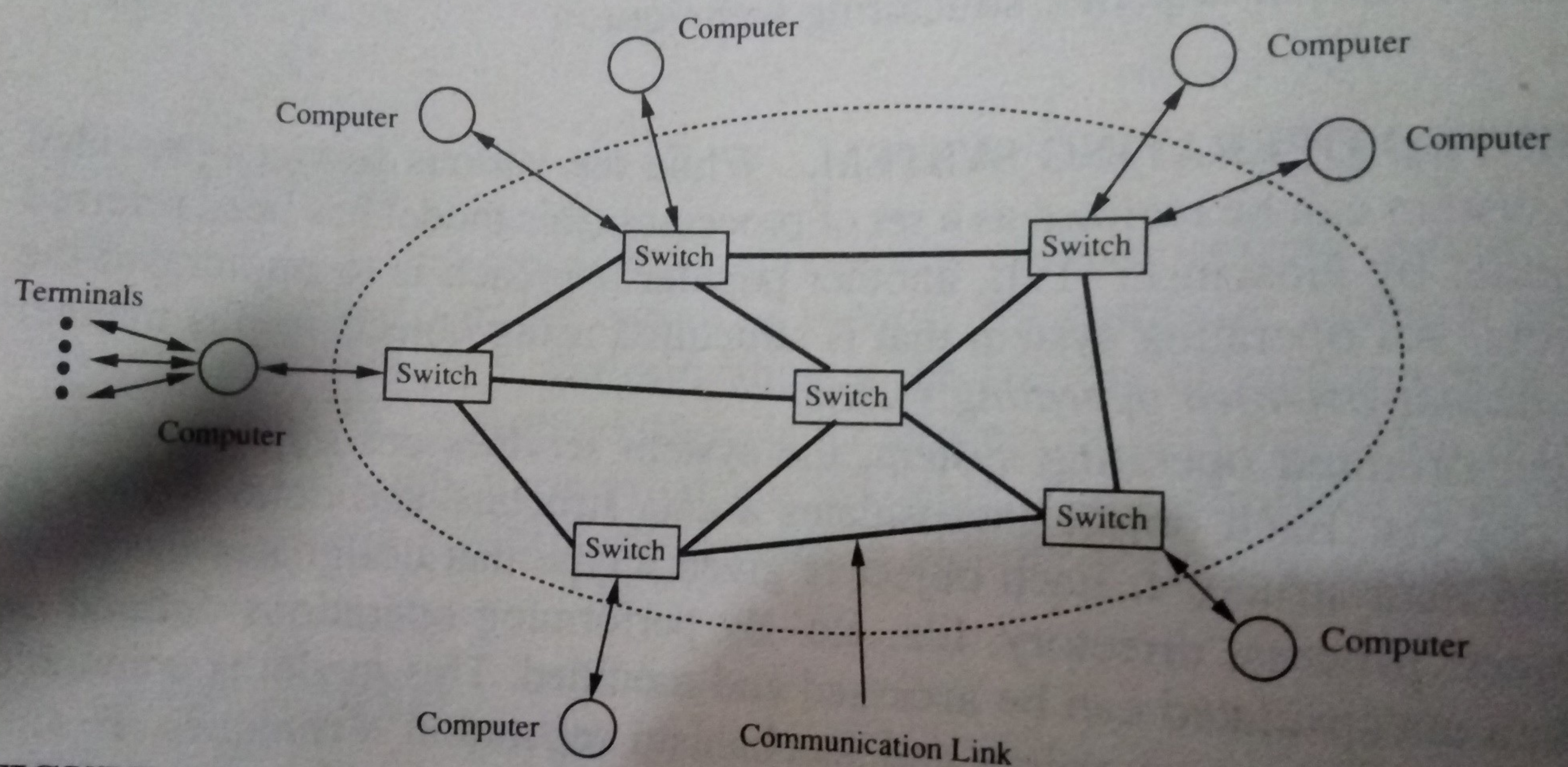


FIGURE 4.2
A point-to-point network.

Utilizing communication networks - modes

- The communication network can be utilized in one of the following two modes:
  1. Circuit Switching
  2. Packet Switching

1. Circuit switching

- A dedicated path is established between two devices wishing to communicate, and the path remains intact for the entire duration in which the two devices communicate.
- The path is broken when one side terminates the conversation.
- Example – telephone system

Utilizing communication networks - modes

2. Packet switching

- A connection established between the source device (terminal or computer) and its nearest switch.
- The data or message to be communicated is broken down into smaller units called **packets**, with each packet containing the address of the destination.
- The packets are then sent to the nearest switch.
- These packets are then sent to the nearest switch.
- These packets are routed from one switch to another switch in the communication network until they arrive at the switch connected to the destination device.
- A communication path(switches or links) is not reserved by any two devices wishing to communicate, but rather is dynamically shared among many devices on a demand basis.
- The network can be shared between many devices.

The ISO OSI Reference Model

- WANs must interconnect heterogeneous types of equipment (Computers, Printers etc).
- These equipment may differ from each other in their speed, word length, information representation or in many other criteria.
- **To communicate in heterogeneous environment, the ISO OSI reference model provides a framework for communication protocols.**
- It organizes the protocols as seven layers and specifies the functions of each layer.
- In this model, user programs run in the application layer.

ISO OSI reference model

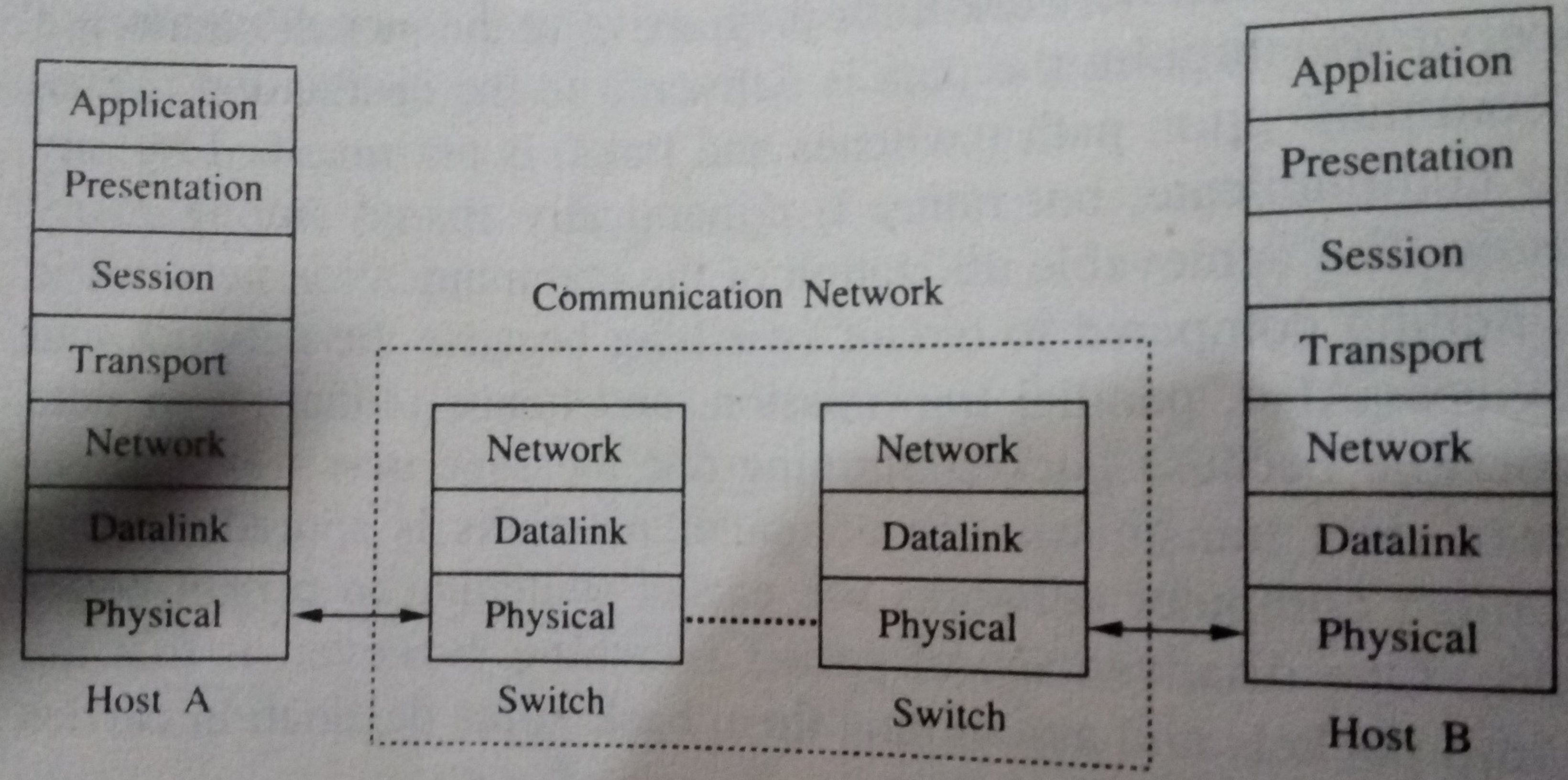


FIGURE 4.3
The ISO OSI reference model.

The ISO OSI Reference Model

- When an application program at computer A wants to send a message to an application program at computer B, the chain of events are:
  - The application layer of A passes the message down to presentation layer.
  - The Presentation layer transforms the data, add a header containing some control information to the message and passes to session layer.
  - The session layer adds its own header to the message and passes to next layer.
  - This continue until the message reaches the physical layer of Computer A.
  - The physical layer on A transmits the raw data bits into physical layer running at computer B. (Message is routed to B through various intermediate switches).
  - At computer A, after receiving message, performs the necessary processing identified by the header and passes on to the next layer.
  - This continues until the message reaches the destination.

The ISO OSI Reference Model

- The ISO OSI model does not specify how the layers should be implemented.
- Every layer is aware only the protocols and header formats of its own.
- It does not understand the header or the protocols used by other layers.
- Each layer is independent, so any layer can change its protocol without affecting other layers.

2. Local Area Networks (LANs)

- A LAN is a communication network that interconnects a variety of data communication devices within a small geographic area.

- **Characteristics of LANs**
  - High data transmission rates.
  - The geographic scope is small, generally confined to a single building or several buildings (eg: college campus)
  - Low transmission error rate.

Network topologies of LAN

- Widely used network topologies of LAN are:
    1. Bus
    2. Tree
    3. Ring

- The communication media can be coaxial cable, twisted pair wire or optical fibre.

Bus / Tree topology

- In bus topology, the communication devices transmit data in the form of packets where each packet contains the address of the destination and a message.
- A packet propagates throughout the medium, the bus, and is available to all other devices, but is received by the addressed device.
- In the bus topology, all the devices connected to the LAN are allowed to transmit at any time.
- Since all the devices share a common data path (bus), a protocol to control the access to the bus is necessary.
- A tree topology LAN is obtained by interconnecting many bus topology LANs to a common bus.
- Protocols to control access to the bus are,
    1. **CSMA/CD** (Carrier Sense Multiple Access with Collision Detection) **protocol**
    2. **Token bus protocol**

Bus / Tree topology

1. **CSMA/CD** (Carrier Sense Multiple Access with Collision Detection) **protocol**
   - A device wishing to transmit listens to the medium to determine whether another transmission is in progress.
   - If so, the device waits for a random amount of time before trying again.
   - If no other transmission is in progress, the device starts transmitting data and continues to listen to the medium while it is transmitting.
   - If another device starts transmitting simultaneously, the two transmission collide.
   - If a collision is detected, a short jamming signal is transmitted over the bus to inform all the devices that there has been a collision.
   - The device will then wait for a random amount of time before attempting to transmit again.
   - Advantage : Simplicity
   - Disadvantage: Heavy load, performance degraded due to collision.

Bus / Tree topology

2. Token bus protocol

- Devices physically organized in a bus/tree topology form a logical ring, and each device knows the identity of the devices preceding and following it.
- Access to the bus is controlled through a token (Control Packet).
- The device holding the token is allowed to transmit.
- The device not holding the token can receive message.
- A device is allowed to keep the token for a specific amount of duration, after which it has to send the token to the device following it on the logical ring.

Ring topology

- Data is transmitted point-to-point.
- At each point, the address on the packet is copied and checked to see if the packet is meant for the device connected at that point.
- If the address of the device and the address in the packet match, the rest of the packet is copied, otherwise, the entire packet is retransmitted to the next device on the ring.
- Protocols to control access to the ring are,
    1. Token ring protocol
    2. Slotted ring protocol

Ring topology

- Protocols to control access to the ring are,

1. **Token ring protocol**
   - Widely used protocol.
   - The token is labelled free when no device is transmitting.
   - When a device wishes to transmit, it waits for the token to arrive, labels the token as busy and retransmit the token. Immediately the device starts transmitting the data.
   - The transmitting device will mark the token free when the busy token returns to the device and the device has completed its transmission.
   - Advantage: Not sensitive to the load on the network.
   - Disadvantage: Complex. If the token is lost and care must be taken to generate only one token.

Ring topology

- Another protocols to control access to the ring are,

2. Slotted ring protocol

- A number of fixed length slots continuously circulate around the ring.
- A device wishing to transmit data waits for a slot marked *empty* to arrive, marks it *full*, and inserts the destination’s address and the data into the slot as it goes by.

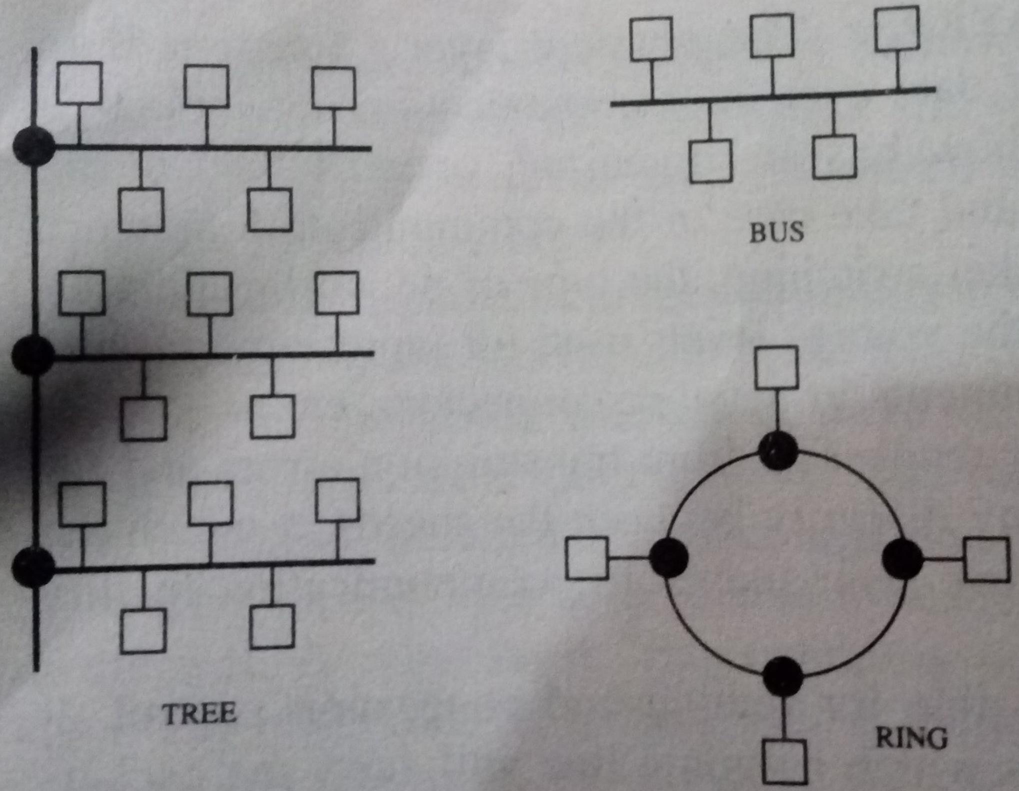


FIGURE 4.4
Network topologies.

Communication primitives

- The communication network provides a means to send and receive raw bit streams of data in distributed systems.
- The communication primitives are high-level constructs with which programs use the underlying communication network.
- They influence:
    1. A programmer’s choice of algorithms
    2. Ultimate performance of the programs.
    3. Ease of use of the system
    4. Efficiency of applications that are developed for the system
- Communication models that provide communication primitives are,
    1. The message passing model
    2. Remote procedure call

The message passing model

- This model provides two basic communication primitives ,
    - SEND and RECEIVE.
- The SEND primitive has two parameters :
    - a message and its destination.
- The RECEIVE primitive has two parameters :
    - the source of the message and a buffer for storing the message.
- Example: client server model
    - The client process needing some service sends a message to the server and waits for a reply message. After performing the task, the server process sends the result in the form of a reply message to the client process.
- The SEND and RECEIVE primitives provide the basic communication ability to programs.
- The *semantics* of these primitives also play a significant role in ease of developing programs.

The message passing model

- In the standard message passing model, messages are copied three times:
  - From user buffer to the kernel buffer
  - From kernel buffer on the sending computer to the kernel buffer on the receiving computer.
  - From the kernel buffer on the receiving computer to a user buffer.

- This is known as the *buffered* option.

- The two design issues that decide the semantics of these two primitives are:

1. BLOCKING VS. NONBLOCKING PRIMITIVES

2. SYNCHRONOUS VS. ASYNCHRONOUS PRIMITIVES

1.BLOCKING VS. NONBLOCKING PRIMITIVES

- With NONBLOCKING PRIMITIVES, the SEND primitive returns control to the user process as soon as the message is copied from the user buffer onto the kernel buffer.
- The corresponding RECEIVE primitive signals its intention to receive a message and provides a buffer to copy the message. The receiving process may either periodically check for the arrival of a message or be signalled by the kernel upon arrival of a message. Here programming is difficult, but flexible.
- With BLOCKING PRIMITIVES, the SEND primitive does not return control to the user program until the message has been sent or until an acknowledgement has been received. The user buffer can be reused as soon as the control is returned to the user program. The corresponding RECEIVE primitive does not return control until a message is copied to the user buffer. A reliable RECEIVE primitive automatically sends an acknowledgement, while an unreliable RECEIVE primitive does not send an acknowledgement. Here the behavior of the programs is predictable, and programming is easy., but lack of flexibility in programming and absence of concurrency.

SYNCHRONOUS VS. ASYNCHRONOUS PRIMITIVES

- With synchronous primitives, a SEND primitive is blocked until a corresponding RECEIVE primitive is executed at the receiving computer. Also called rendezvous. A blocking synchronous primitive can be extended to an unblocking-synchronous primitive by first copying a message to a buffer at the sending side, and then allowing the process to perform other computational except another SEND.

- With asynchronous primitives, the messages are buffered. A SEND primitive does not block even if there is no corresponding execution of RECEIVE primitive.
- The corresponding RECEIVE primitive can either be a blocking or a nonblocking primitive. Disadvantage – buffering of messages is more complex and what to do with the messages that are meant for processes that have already died.

Remote procedure call

- RPC is an interaction between a client and a server, where the client needing a service invokes a procedure at the server.
- On invoking a remote procedure, the calling process (the client) is suspended and parameters, if any are passed to the remote machine(the server) where the procedure will execute.
- On completion of the procedure execution, the results are passed back from the server to the client and the client resumes execution as if it had called local procedure.
- While the RPC mechanism looks simple, the issues that arise in designing and implementing are,
    1. Structure
    2. Binding
    3. Parameter and result passing
    4. Error handling, semantics and correctness

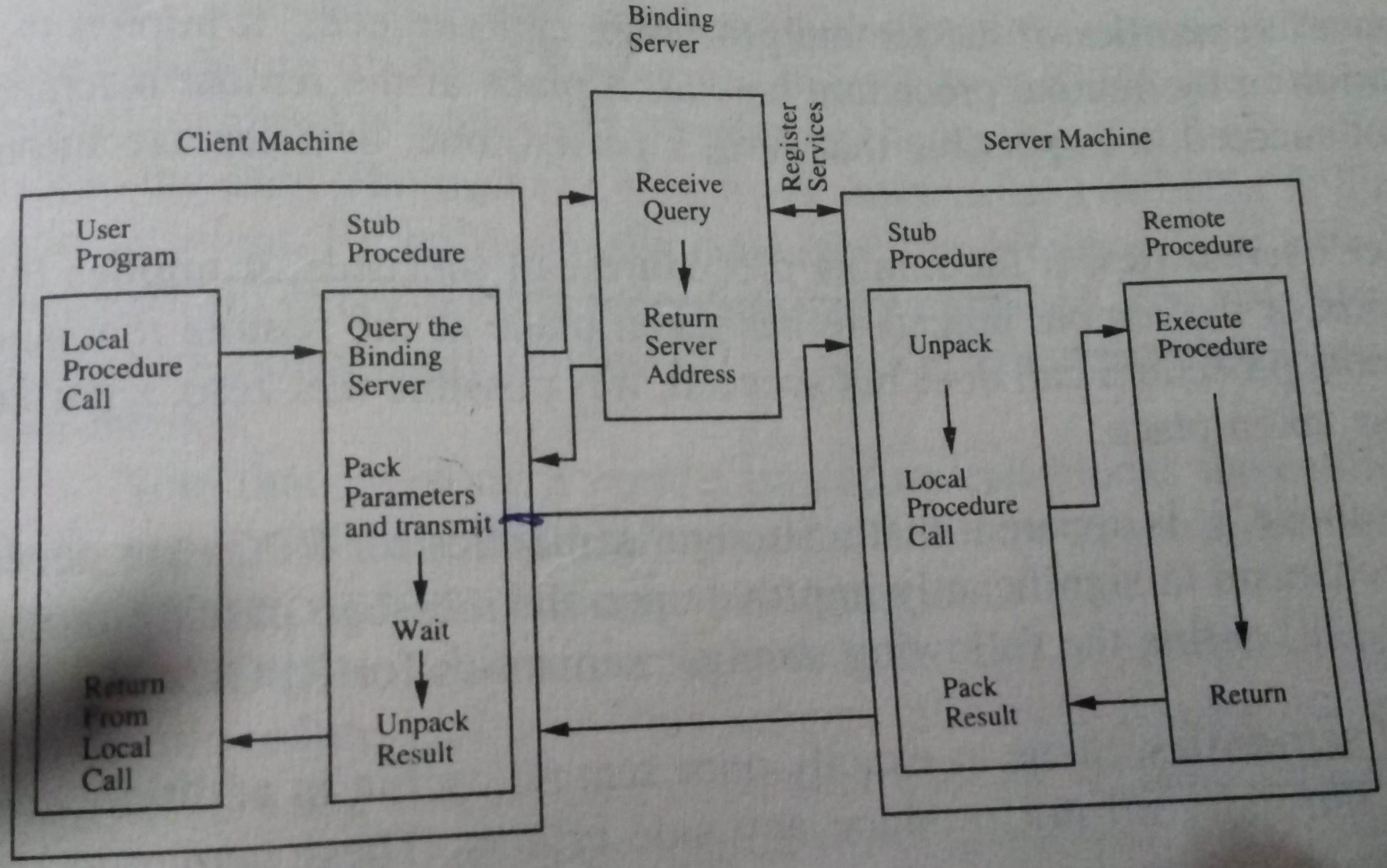


FIGURE 4.5
Remote procedure call.

Design Issues in RPC

- Structure
- Binding
- Parameter and result passing
- Error handling, semantics, and correctness

Structure

- RPC is based on stub procedures
- Client stub and server stub are used
- Client stub send message to server stub
- Server stub procedure passes result back to client stub procedure

Binding

Approach 1

- Binding is a process that determines the remote procedure and the machine on which it will be executed
- Also check the compatibility of parameters passed and the procedure type
- A binding server is used, which store the server machine addresses with the services they provide.
- Servers register with binding server.
- Client stub procedure obtains server address from binding server.

Approach 2

- Client specifies the machine and service required, binding server returns the port number for communicating with the service required

Parameter and result passing

- Stub procedure has to convert the parameters and results into an appropriate representation and pack them to a buffer in a form suitable for transmission.
- Conversion is expensive

Ways to avoid conversion

1. Send parameters along with a code so that receiver can do conversion, this requires the machine to know how to convert all formats. If new representation is there, existing software needs to be updated.
2. Each data type may have a standard format, sender will convert data to standard format, receiver will convert to local representation.

- Dealing with passing parameters by value and by reference. Passing by value is easy, but by reference is complicated because along with file pointers previleges associated with calling procedure have to be passed.

Error handling, semantics and correctness

- RPC may fail due to computer failures and communication failures

1. Messages lost occasionally, result in more than one execution of procedure call.
2. Client machine crashes, then no machine to receive result. If client machine recovers quickly and reissues RPC, results in more than one execution of procedure call.

Semantics of RPC

- “At least once” semantics
- “Exactly once” semantics

Stronger semantics

- “At most once” semantics (Zero-or-one” semantics)

Lamport’s logical clocks

Lamport’s logical clocks

- Two types of Clock in Distributed System,
  - Logical Clock
  - Vector Clock

- For event ordering, we have Lamport’s Logical Clock.

- Logical Clock is not based on timing, but on the ordering of events.

- Due to the absence of perfectly synchronized clocks and global time in distributed systems, the order in which two events occur at two different computers cannot be determined based on the local time at which they occur.

- However, under certain conditions, it is possible by analysing the behaviour exhibited by the underlying computation.

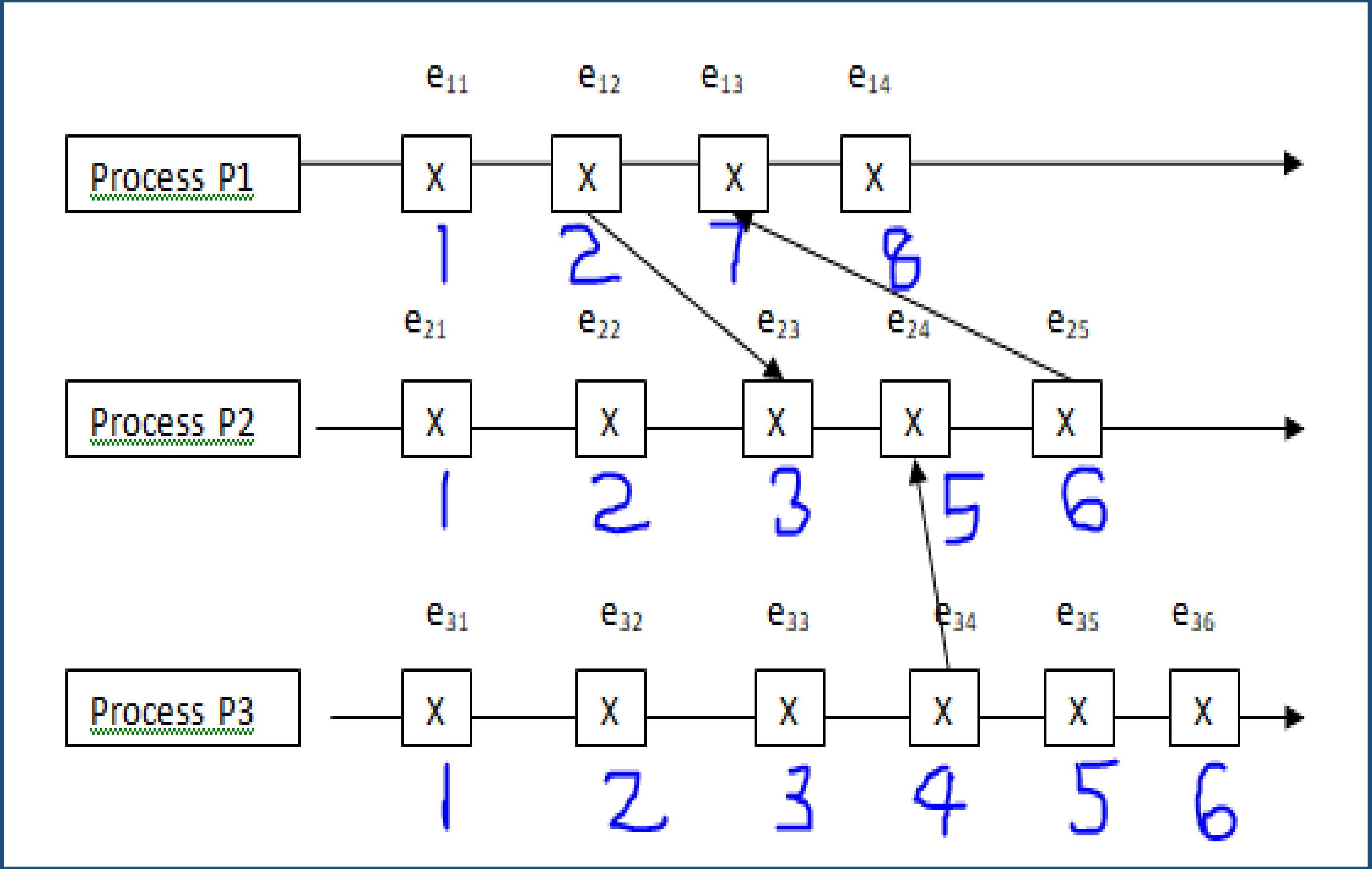
Lamport’s logical clocks

- Is a scheme to order events in a distributed system using logical clocks.
- The execution of events is characterized by a sequence of events.
- Depending on the application, the execution of a procedure could be an event or the execution of an instruction could be one event.
- When processes exchange messages, the execution of a message constitutes one event and receiving a message constitutes another event.

Lamport’s logical clocks

- **Lamport clocks** allow processes to assign sequence numbers (“timestamps”) to messages and other events so that all cooperating processes can agree on the order of related events.
- There is no assumption of a central time source and no concept of *when* events took place.
- Each process maintains a single Lamport timestamp counter.
- Each event in the process is tagged with a value from this counter.
- The counter is incremented before the event timestamp is assigned.
- If a process has four events, *a, b, c, d*, the events would get Lamport timestamps of *1, 2, 3, 4*, respectively.

Example



Let’s look at an example. The figure shows a bunch of events on three processes. Some of these events represent the sending of a message, others represent the receipt of a message, while others are just local events (e.g., writing some data to a file).

Implementation of logical clocks n Implementation Rules

- Implementation Rules(guarantee that the logical clocks satisfy the correctness conditions):

[IR1]

Clock Ci must be incremented between any two successive events in process Pi :

$C_i := C_i + 1$

[IR2]

If event a is the event of sending a message m in process Pi , then message m is assigned a timestamp tm =Ci (a) When that same message m is received by a different process Pk , Ck is set to a value greater than curent value of the counter and the timestamp carried by the message

$C_k := \max(C_k , tm)+1$

Lamport’s logical clocks -Definitions

1. HAPPENED BEFORE RELATION(->)
2. CASUALLY RELATED EVENTS
3. CONCURRENT EVENTS

1. HAPPENED BEFORE RELATION (->)

- The happened before relation captures the casual dependencies between events, i.e., whether two events are casually related or not. The relation -> is defined as follows:
  - a->b, if and b are events in the same process and a occurred before b.
  - a->b, if a is the event of sending a message m in a process and b is the event of receipt of the same message m by another process.
  - If a->b and b->c, then a->c, i.e., "->" relation is transitive.
- In distributed systems, processes interact with each other and affect the outcome of events of processes.
- Past events influence future events and this influence among causally related events is referred to as ***causal affects***.

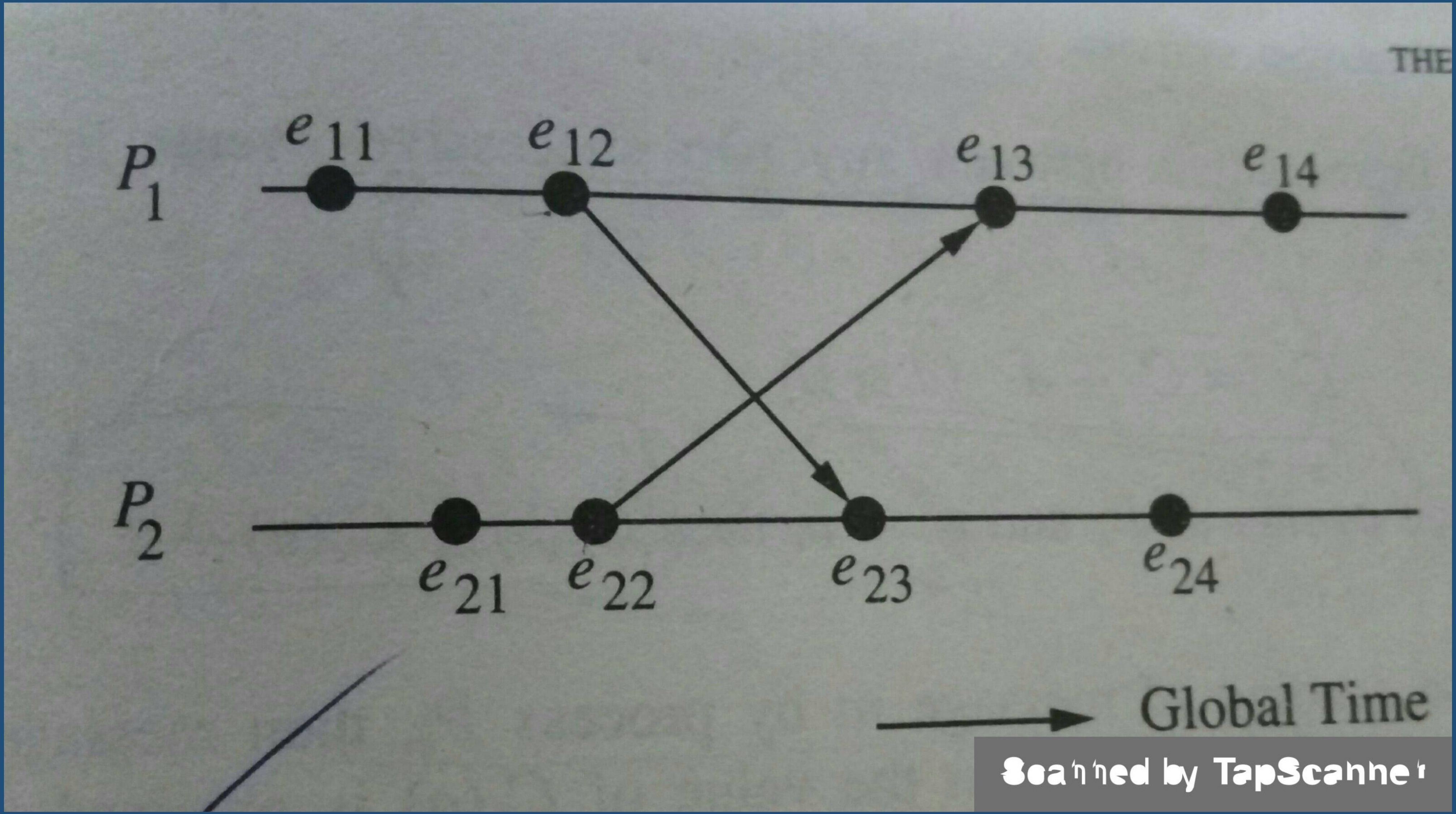
2. CAUSALLY RELATED EVENTS

- Event a causally affects event b if a->b.

3. CONCURRENT EVENTS

- Two distinct events a and b are said to be concurrent (denoted by a||b) if $a \not\rightarrow b$ and $b \not\rightarrow a$.
- Concurrent events do not affect each other.
- For two events and b in a system, either a->b, b->a, or a||b.

Space time diagram



Causal ordering of messages

- Deals with the notion of maintaining the same causal relationship that holds among “message send” events with the corresponding “message receive” events
- If Send(M1)->Send(M2), then every recipient of both messages M1 and M2 must receive M1 before M2.
- There are two protocols to make use of vector clocks for the causal ordering.
    - 1. Briman – Schiper –Stephenson Protocol
    - 2. Schiper – Egglii – Sandoz Protocol

Birman Schiper Stephenson Protocol

- This protocol is used to maintain the causal ordering of the messages.
- i.e. the message which is sent first should be received first.
- If send (M1)–>send(M2) then for all processes which receive the messages M1 and M2 should receive M1 before M2.

Features :

- Broadcast based messaging.
- Size of the messages are small.
- More no. of messages are sent.
- Limited state information.

Key Points :

- Each process increases its vector clock by 1 upon sending of messages.
- Message is delivered to a process if the process has received all the messages preceding to it.
- Else buffer the message.
- Update the vector clock for the process.

Schiper – Egglii – Sandoz Protocol

- There are three basic principles to this algorithm:
- All messages are time stamped by the sending process.

[Note: This time is separate from the global time talked about in the previous sections. Instead each element of the vector corresponds to the number of messages sent (including this one) to other processes.]

- A message can not be delivered until:
  - All the messages before this one have been delivered locally.
  - All the other messages that have been sent out from the original processs has been accounted as delivered at the receiving process.
- When a message is delivered, the clock is updated.
- This protocol requires that the processes communicate through broadcast messages since this would ensure that only one message could be received at any one time (thus concurrently timestamped messages can be ordered).