20MCA172
ADVANCED OPERATING SYSTEMS

Module 1-Part 1

Prof. BABY SYLA L.

COLLEGE OF ENGINEERING TRIVANDRUM

Module-1

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

(2 hours)

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

(4 hours)

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

(4 hours)

- Textbook:

Mukesh Singhal and Niranjan G. Shivaratri, "Advanced Concepts in Operating Systems

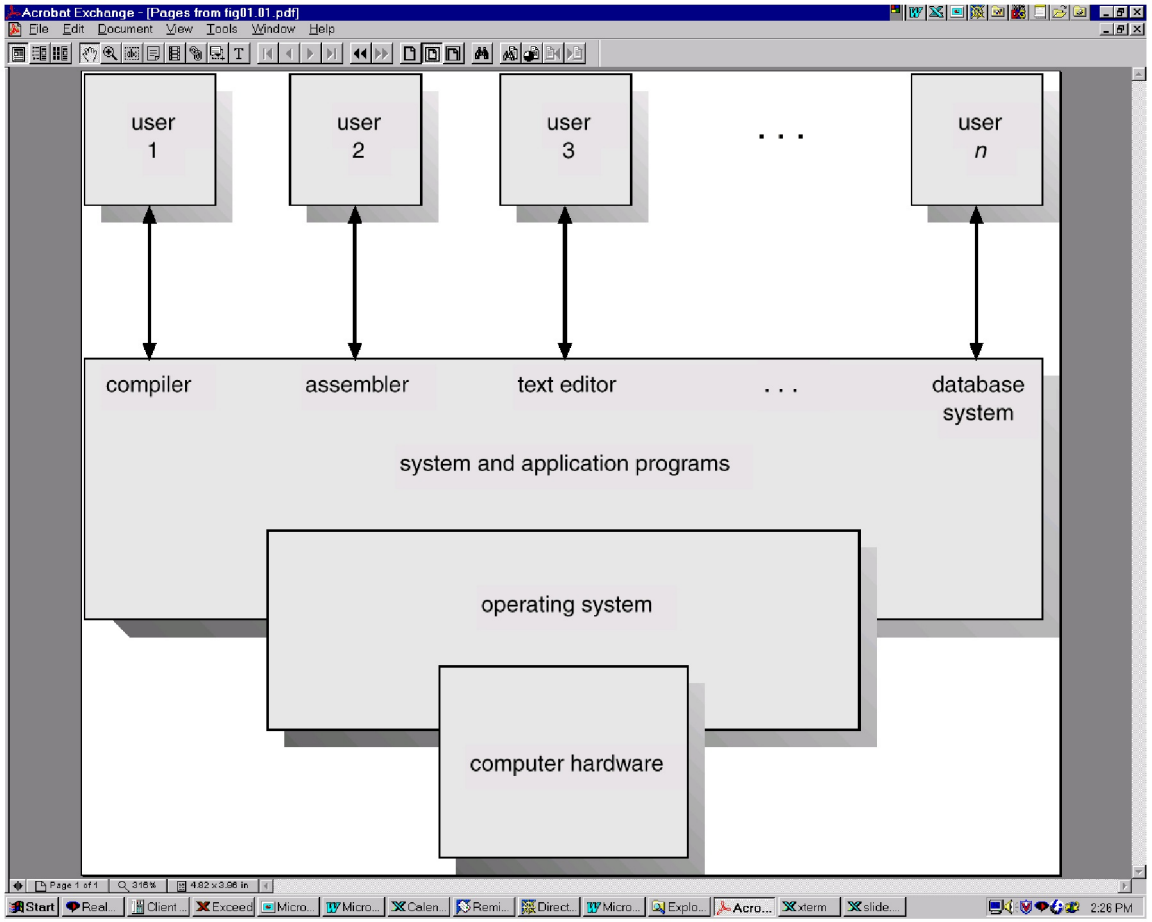
1.1 Operating System Overview

What is an operating system?

- An operating system is a layer of software on a bare hardware machines.

- What are the functions of an Operating system?...

Abstract view of computer system



Functions of an OS

Basic functions of Operating System

1. Resource management

- A user program access several hardware and software resources during its execution.
- It is the operating system that manages the resources and allocates them to users in an efficient and fair manner
- Examples of Resources are CPU, Main memory, input-output devices and various types of software (Compiler ,linker-loader, files etc..)

Resource management Functions:

- Time management(CPU and disk scheduling)
- Space management(Main memory and secondary storage)
- Process synchronization and deadlock handling
- Accounting and status information.

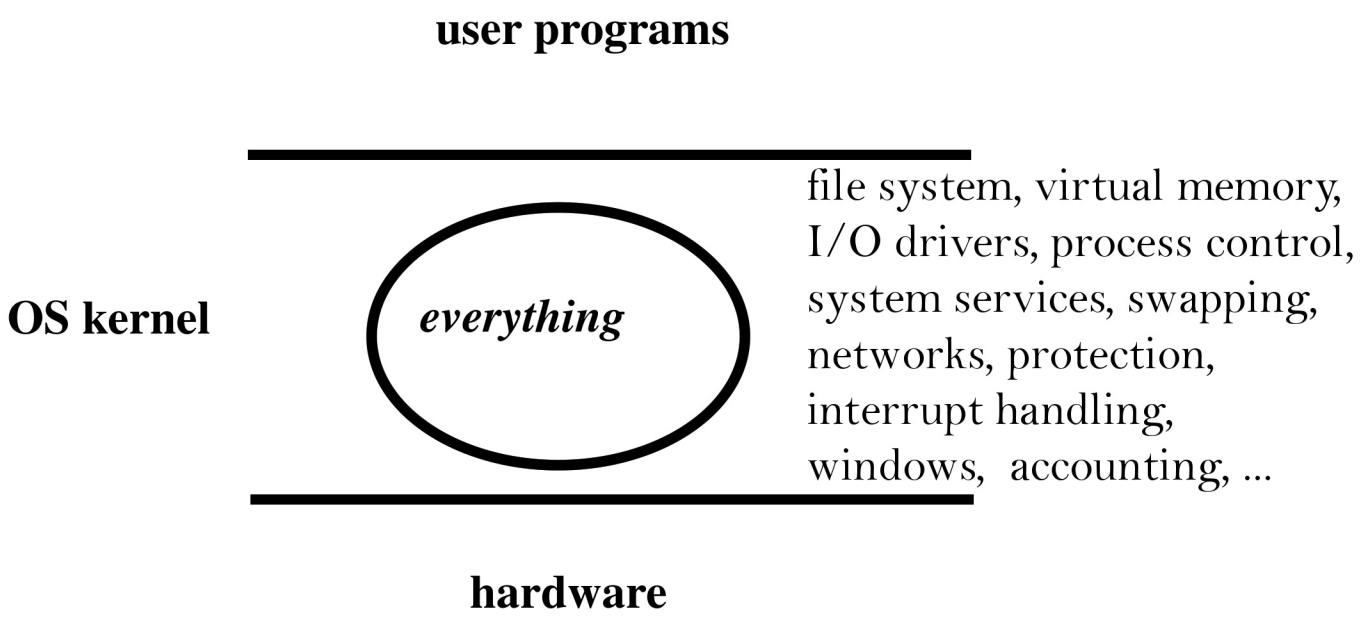
2.User friendliness

- hides the unpleasant, low-level details of a bare hardware machine
- provides users with a much friendlier interface to the machine.
- To load, manipulate, print and execute programs ,high-level commands can be used without the inconvenience of worrying about the low-level details.
- **User friendliness Tasks:**
- Execution environment(process management-creation, control and termination)
- Error detection and handling
- Protection and security
- Fault tolerance and failure recovery

Design of Operating Systems

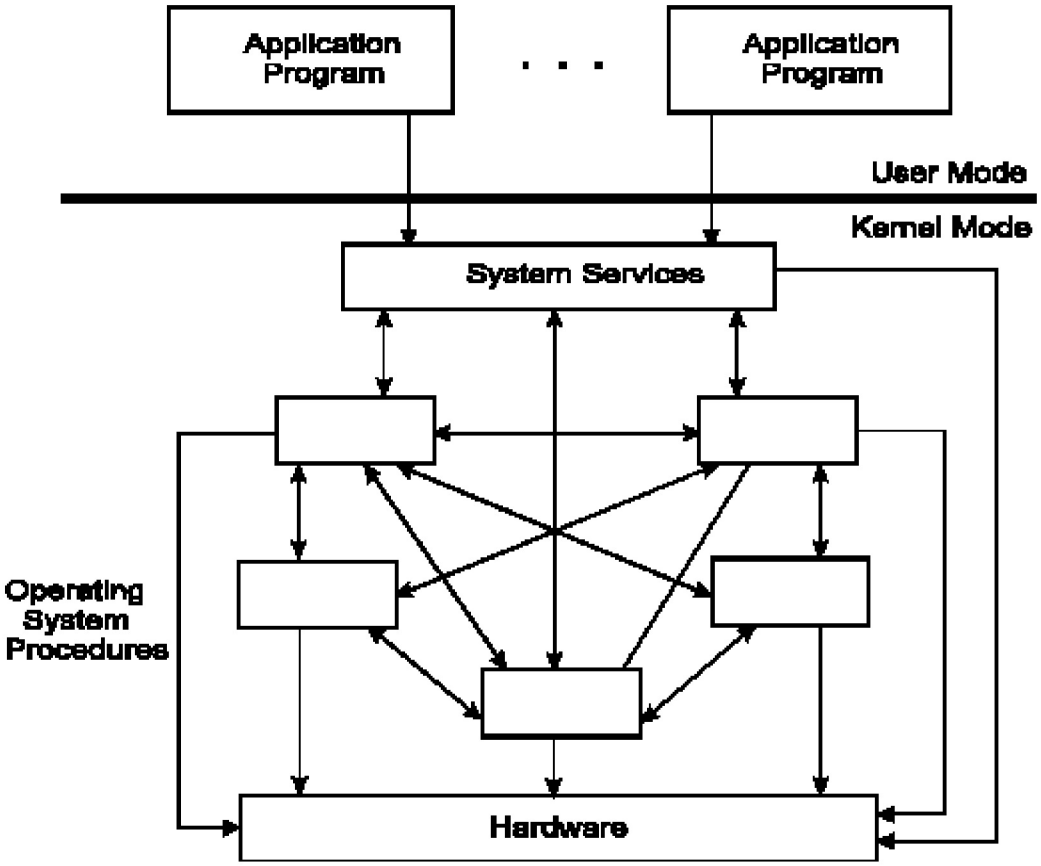
Monolithic kernel Structure

Traditionally, systems such as Unix ,DOS were built as a *monolithic* kernel:



16

Monolithic Operating System



Design Approaches :

1. Layered Approach
2. Kernel based approach
3. Virtual Machine approach

Strategy for good design

- Policies vs Mechanisms
- Policies refer to what should be done .
- Mechanisms refer to how it should be done.

- A good design must separate policies from mechanisms.
  - provides great flexibility.

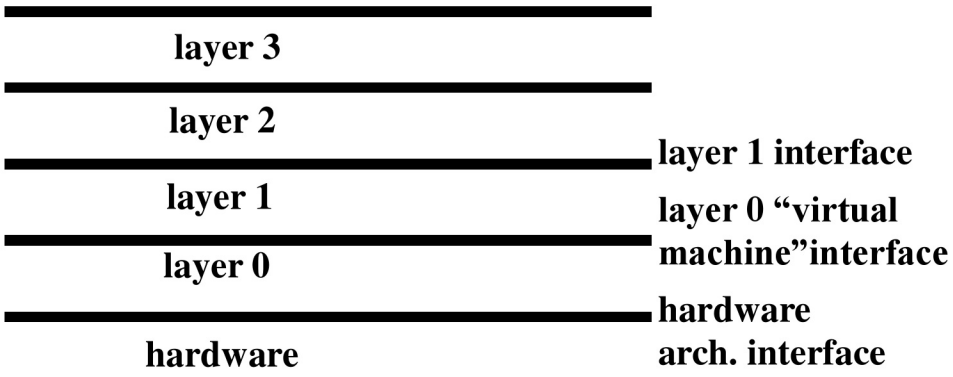
1. Layered Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers.
- Each layer has well define functionality and I/O interfaces with the adjacent layers.
- The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers.

Structuring/Layering

Traditional approach is layering: implement system as a set of layers, where each layer is a *virtual machine* to the layer above.

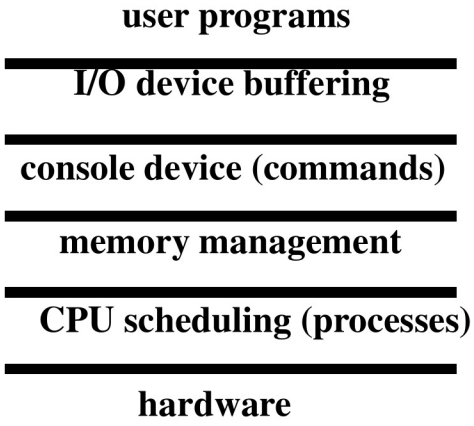
That is, each layer provides a "machine" that has higher level features.



18

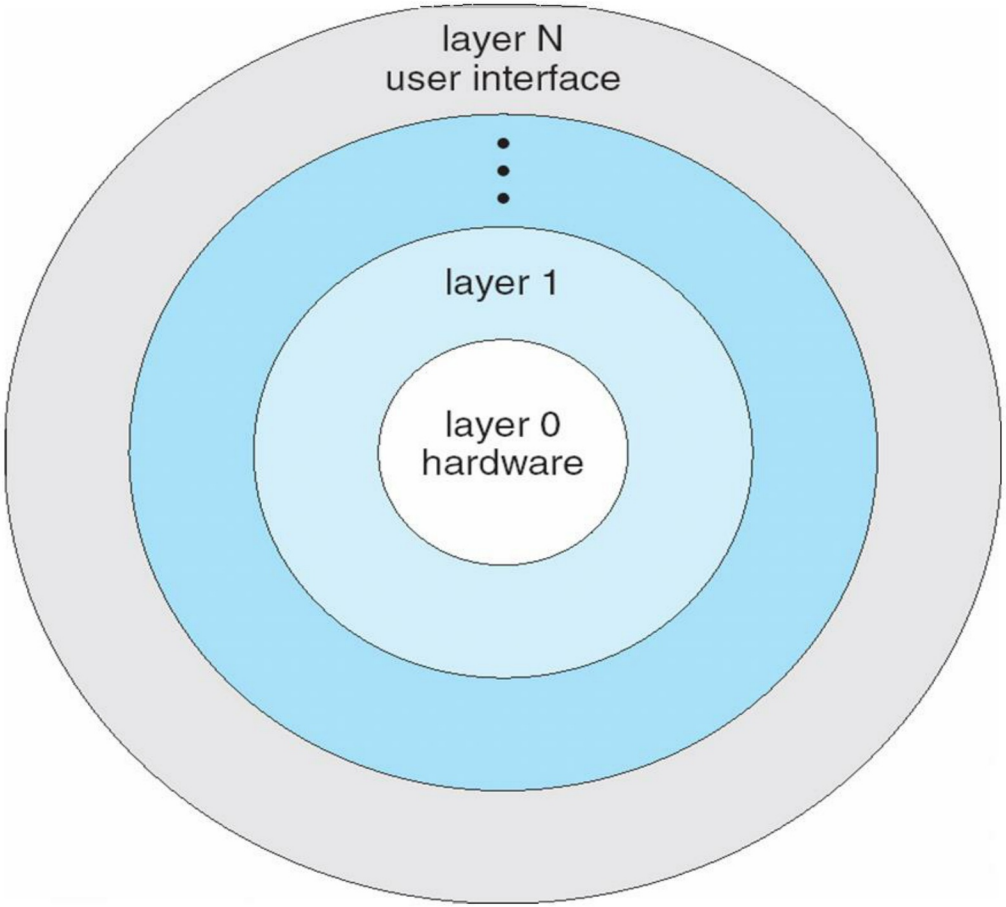
Layering in THE

The first description of this approach was Dijkstra's THE system.



19

Layered Operating System



Older Windows System Layers

Advantage of Layered OS

- Design and implementation is simple
- Easy to debug.

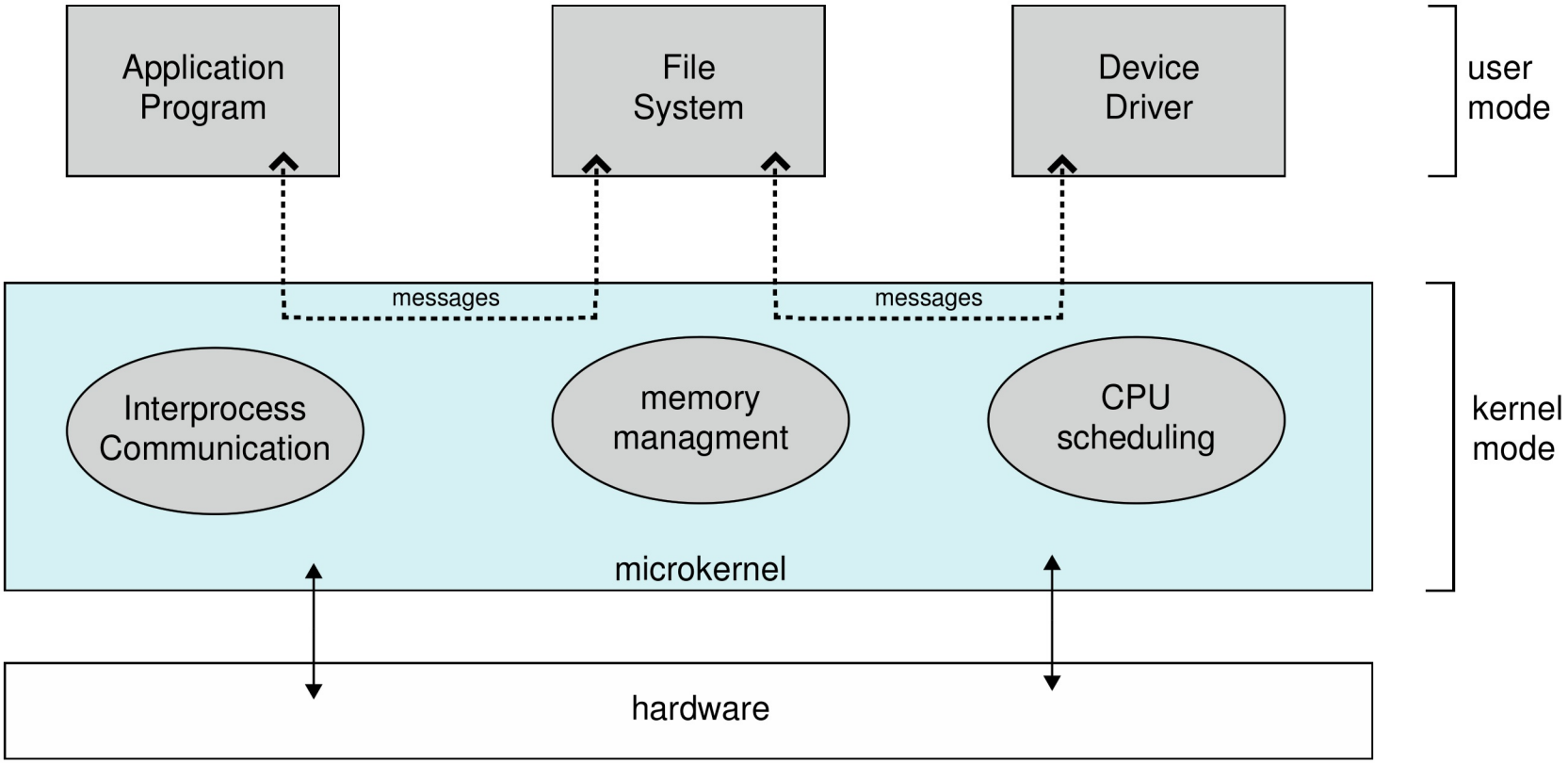
Problems with Layering

- Systems must be hierarchical, but real systems are more complex than that, e.g.,
  - *file system would like to be a process layered on VM*
  - *VM would like to use files for its backing store I/O*
- Approach is not flexible.
- Often has poor performance due to layer crossings.

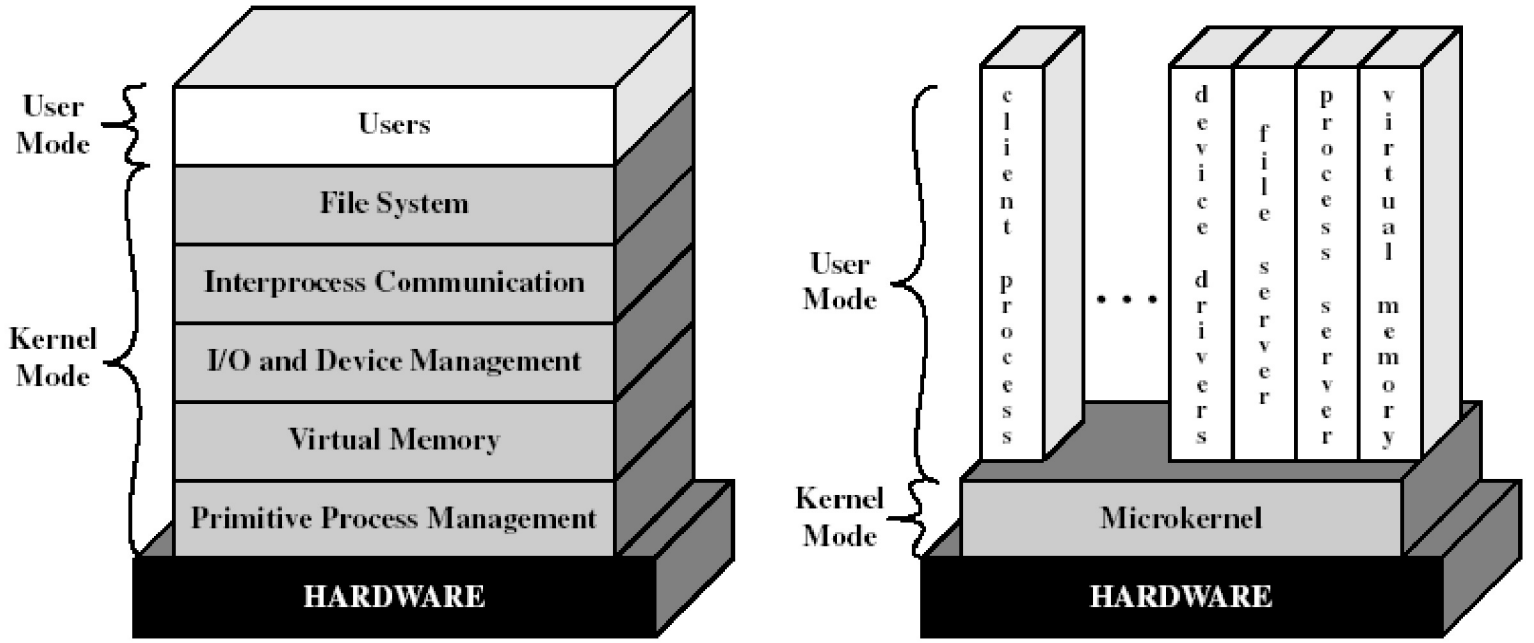
2. Kernel based Approach

- First suggested by Brinch Hansen.
- Only a few essential functions in the kernel:
  - primitive memory management (address space)
  - I/O and interrupt management
  - Inter-Process Communication (IPC)
  - basic scheduling
- Other OS services are provided by processes running in user mode (vertical servers):
  - device drivers, file system, virtual memory…

Microkernel Architecture



Layered vs. Microkernel Architecture



Microkernel System Structure

- Main function of microkernel is to provide a communication facility between client program and various services that are also running in user space.
- Communication takes place between user modules using message passing.
- If a client program wishes to access a file, it must interact with file server.
- Client never interact with fileserver directly, they communicate indirectly by exchanging messages with the microkernel.

Benefits of a Microkernel Organization

- Extensibility/Reliability
  - easier to extend a microkernel
  - easier to port the operating system to new architectures
  - more reliable (less code is running in kernel mode)
  - more secure
  - small microkernel can be rigorously tested.
- Portability
  - changes needed to port the system to a new processor is done in the microkernel, not in the other services.

Benefits of Microkernel Organization

- Distributed system support
  - message are sent without knowing what the target machine is.
- Object-oriented operating system
  - components are objects with clearly defined interfaces that can be interconnected to form software.

Microkernel OS

- First microkernel system was Hydra (CMU, 1970)
- Examples of microkernel systems are the CMU Mach system, Chorus (French Unix-like system), and in some ways Microsoft NT/Windows.

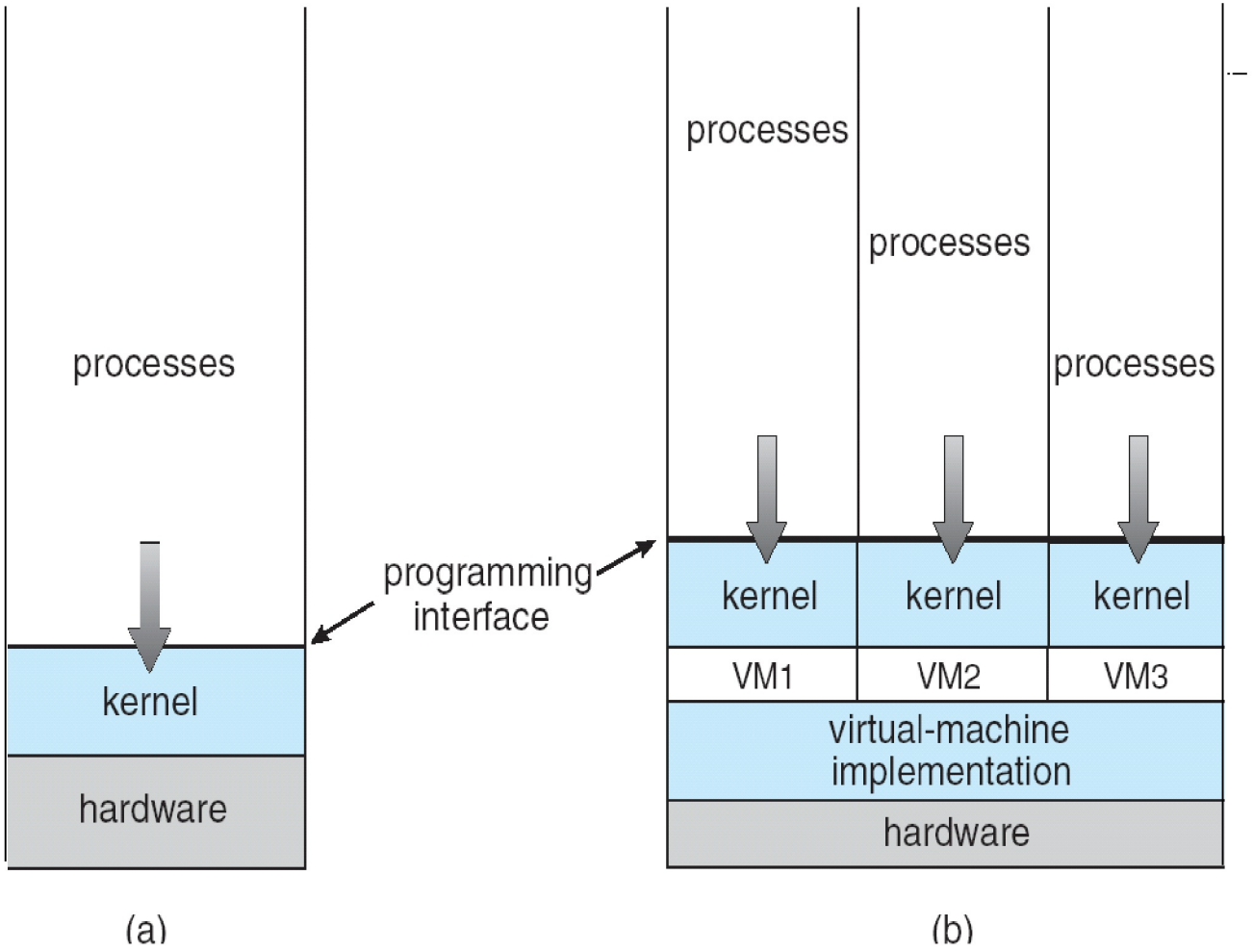
Disadvantage:

- performance overhead caused by replacing service calls with message exchanges between processes.

3. Virtual Machines

- The fundamental idea of a virtual machine is to abstract the hardware of a single machine into several different execution environment .
- It creates an illusion that each seperate execution environment is running its own private computer.
- Virtual machines are created using virtual machine software on bare hardware which creates the illusion by sharing the resources among all users of the machine.
- Classical example of this system is IBM 370 system in which, virtual machine software , VM/370 provides a virtual machine to each user.

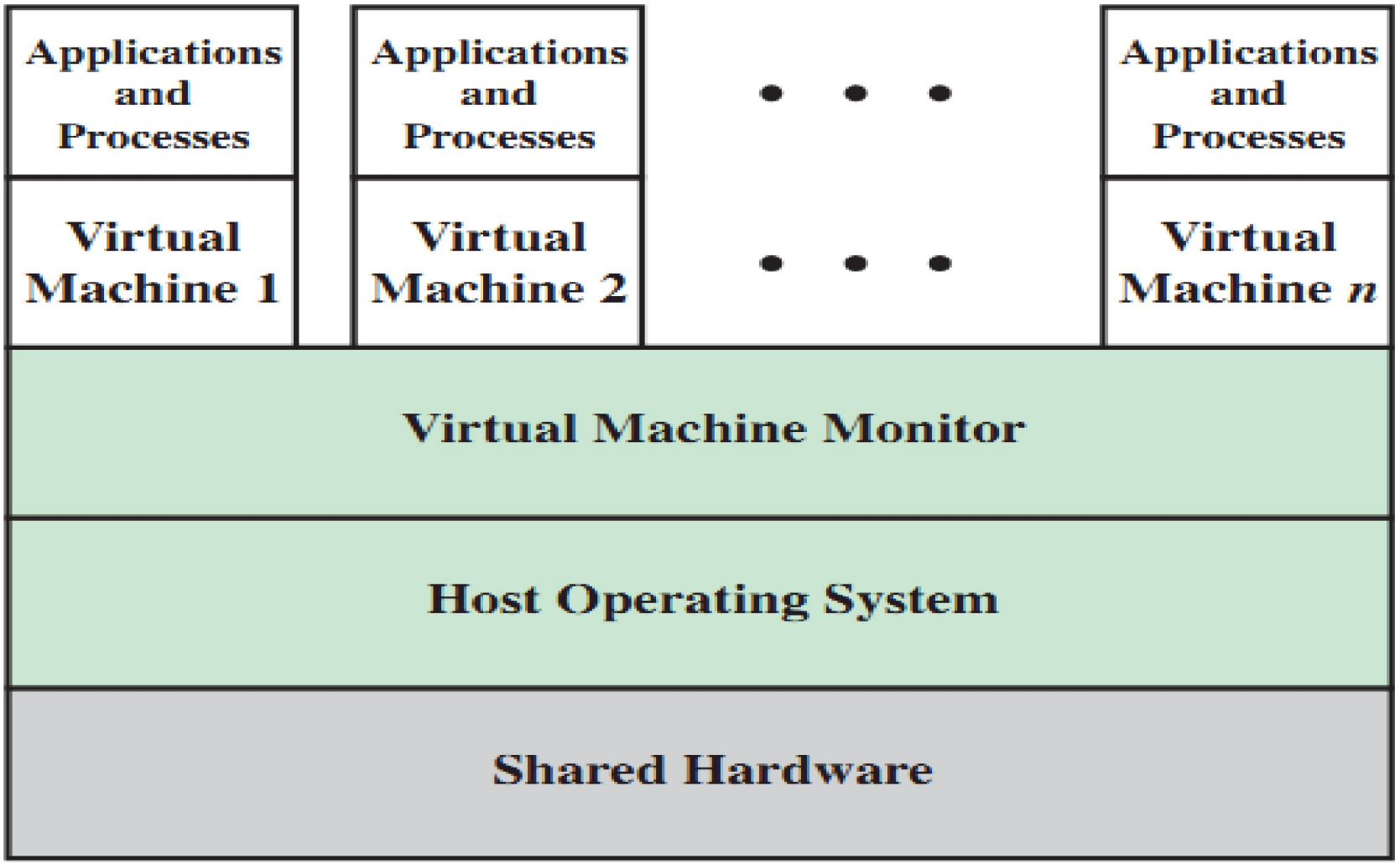
on Bare Machine Implementation VM



Virtual Machines (2)

- User can run a single Operating system on the virtual machine.
- It also allows the flexibility that different OS to run on different VMs.
- Example:VMware
- VMware runs as an application on a host operating systems such as windows or linux and allows to run several guest OS as independent virtual machines.

VM Implementation on Host OS



Advantages/Disadvantages of Virtual Machines

- The virtual-machine concept provides complete protection of system resources since each virtual machine is isolated from all other virtual machines.
- A virtual-machine system is a perfect vehicle for operating-systems research and development. System development is done on the virtual machine, instead of on a physical machine and so does not disrupt normal system operation.
- The virtual machine concept is difficult to implement due to the effort required to provide an *exact* duplicate to the underlying machine.

Classification of Advanced OS

Classification of Advanced OS

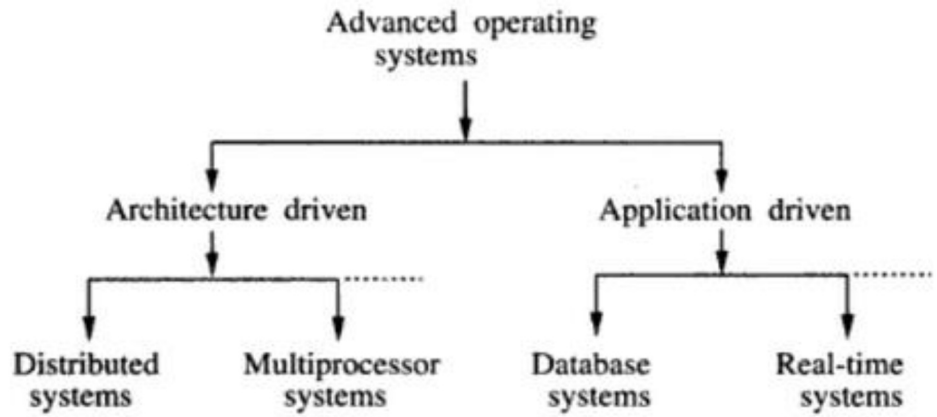
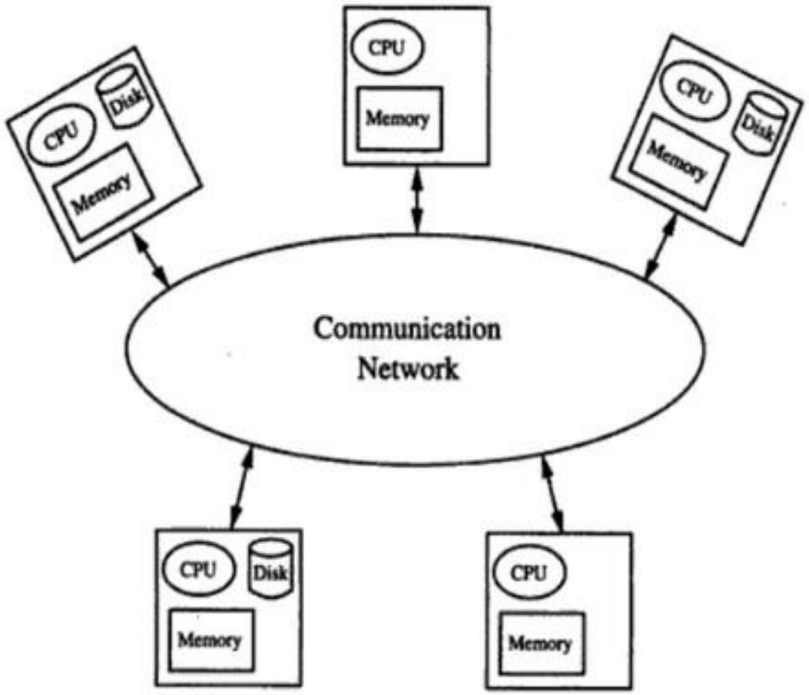


FIGURE 1.3
A classification of advanced operating systems.

1. Distributed operating systems:

- Operating systems for a network of autonomous computers connected by a communication network.
- It controls and manages the hardware and software resources of a distributed systems such that user's 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 resource is accessed.

Architecture of distributed system



Distributed operating systems Issues

- Issues same as in the traditional operating systems,
- Process synchronization
- Deadlocks
- Scheduling
- File systems
- Inter-process communication
- Memory and buffer management
- Failure recovery
- Lack of shared memory and a physical global clock
- Unpredictable communication delays

2. Multiprocessor systems

- It consists of a set of processors that shares a common memory over an interconnection network.
- All the processors operate under the control of a single OS.
- It is called tightly coupled systems where processors share a common address space.

Multiprocessor systems

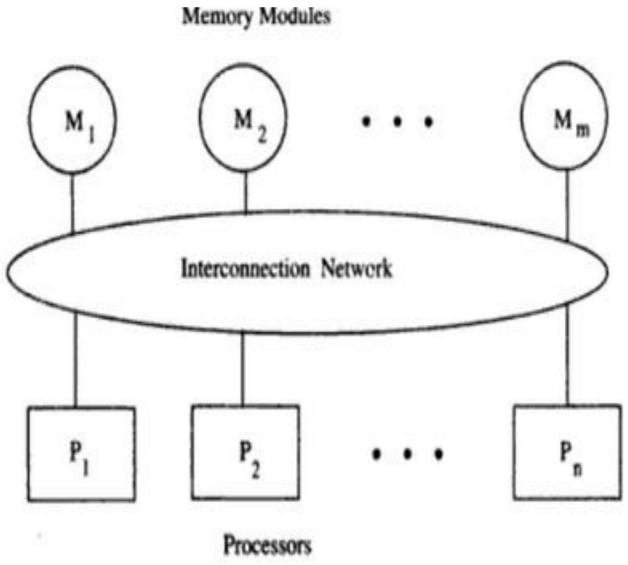


FIGURE 16.1
A tightly coupled multiprocessor system.

A number of processors are connected to shared memory by an interconnection network.
The shared memory is divided in to several modules and multiple modules can be accessed by different processors concurrently.

Multiprocessor operating systems:

- It controls and manages the the hardware and software resources such that the user's view the entire system as a powerful uniprocessor system.
- Issues:
- Process synchronization
- Memory management
- Protection and security

Database operating systems:

Database systems place special requirements on operating systems.

- Database operating systems must support
  - the concept of transaction;
  - operations to store, retrieve and manipulate a large volume of data efficiently;
  - primitives for concurrency control and system failure recovery.
- To store temporary data and data retrieved from secondary storage , the OS must have proper buffer management scheme.

Real time operating systems:

HARD

SOFT

- A distinct feature of Real-time systems is that jobs have completion deadlines.
- A job should be completed before its deadline to be of use.
- **Issues:**
- Schedule maximum number of jobs that should satisfy their deadlines.
- Designing languages and primitives to effectively prepare and execute a job schedule.

Module 1-Part 2

Prof. BABY SYLA L.

COLLEGE OF ENGINEERING TRIVANDRUM

1.2 Synchronisation Mechanisms

Concept of Process

- Process is a program whose execution has started but is not yet complete
- Three basic states of 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 allocated.
    3. Blocked – The process is waiting for an event to occur.

    Examples of event are I/O operation, sending /receiving a message etc.

State transition diagram of a process

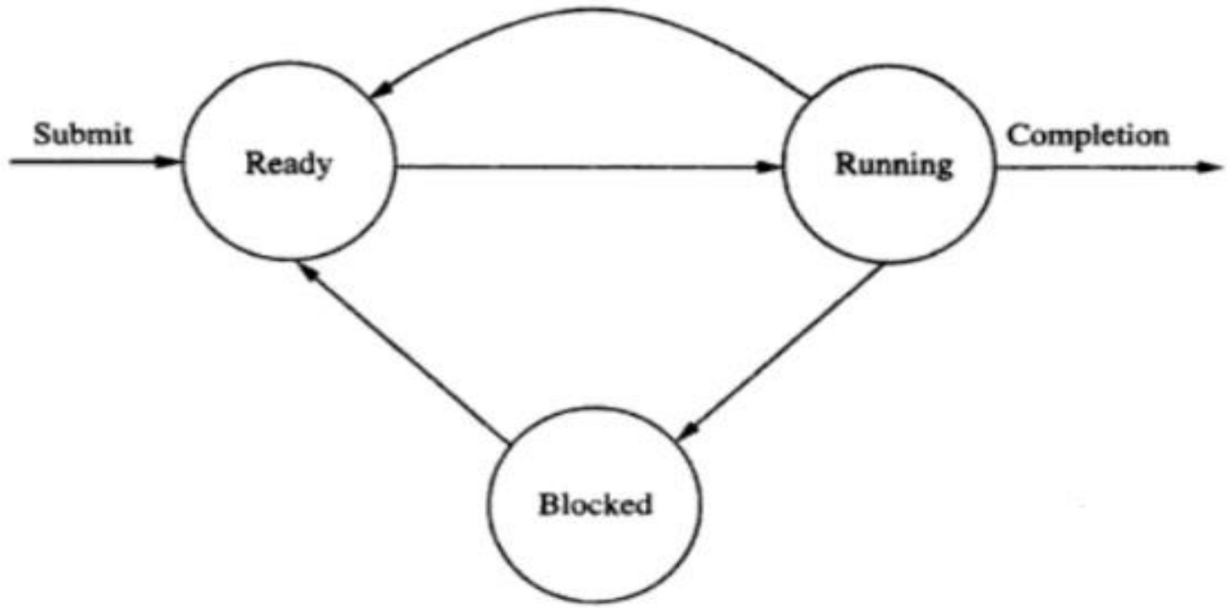


FIGURE 2.1
State transition diagram of a process.

Process Control Block (PCB)

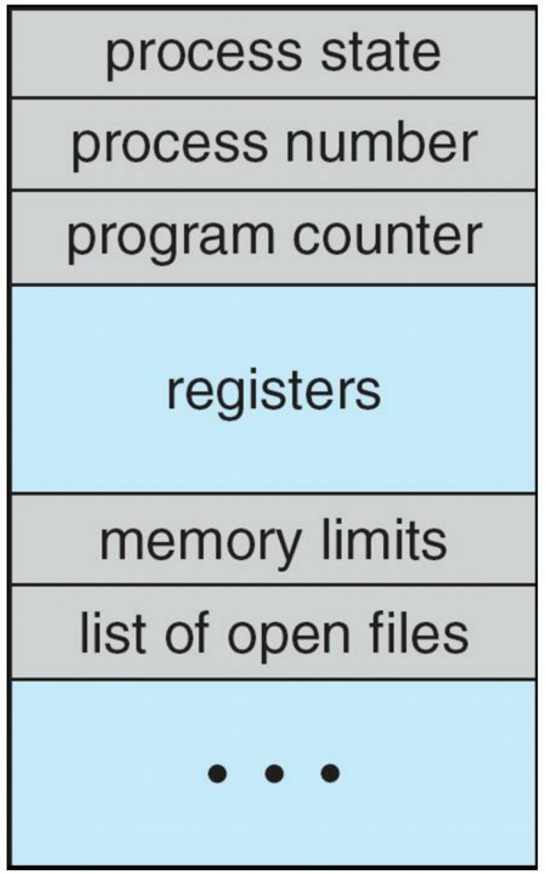
- Process Control Block (PCB) is a data structure used by operating system to store all the information about a process.
- It is also known as Process Descriptor.
- When a process is created, the operating system creates a corresponding PCB.
- Information in a PCB is updated during the transition of process states.
- When a process terminates, its PCB is released.
- Each process has a single PCB.

Process Control Block (PCB)

Information associated with each process

(also called **task control block**)

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



Serial Processes Vs Concurrent processes

- Concurrent process – are not serial, execution can overlap in time.
- Concurrency in multiprocessor systems are easy to visualise.
- In single processor systems , physical concurrency occurs by interleaving the CPU time and I/O time.
- Concurrent processes generally interact for information sharing , computation speed up etc (Cooperating processes)

Inter process communication (IPC) mechanisms

1. Shared variables

   The processes access (read/write) a common variable or data

2. Message Passing

   The processes exchange information by passing messages (Send/Receive) with each other

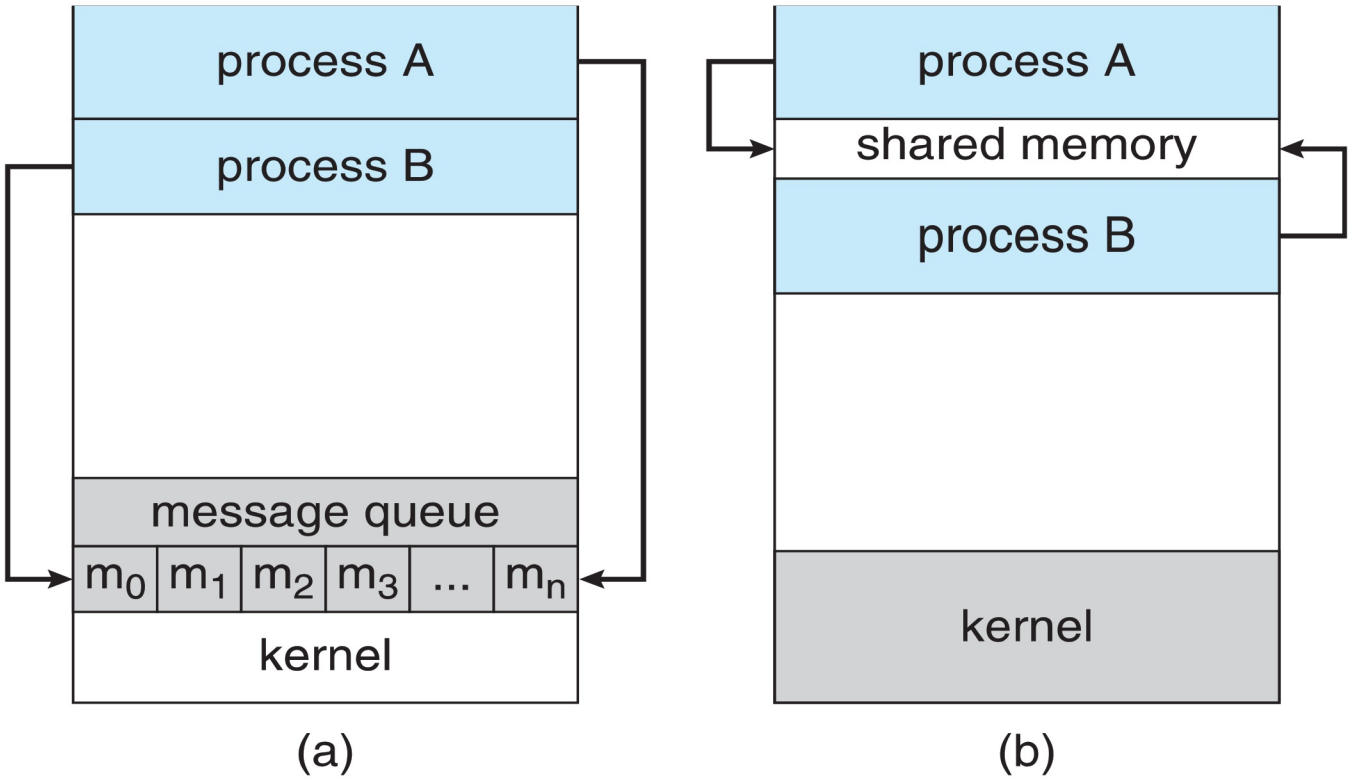
Note: If the processes do not communicate , then it is same as serial execution.

IPC mechanisms

a) Message passing

b) Shared memory

SEND(MSG,B)
RECEIVE (&BUF, A)



Process and Thread

Process

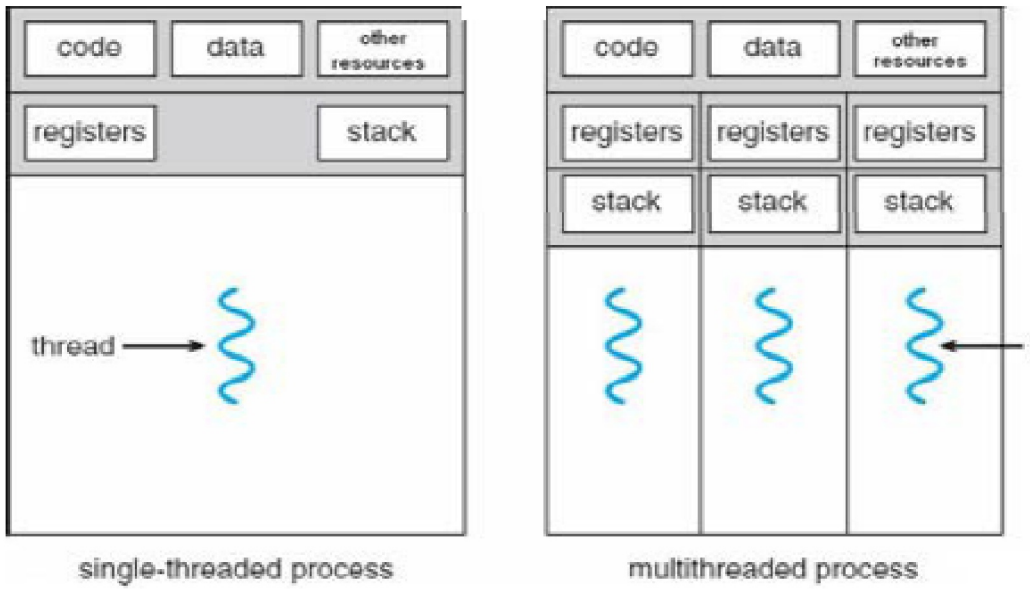
- A process has a single address space and a single thread of control.
- Process information is stored in PCB
- Context switching is expensive .
- called heavy weight process

Thread

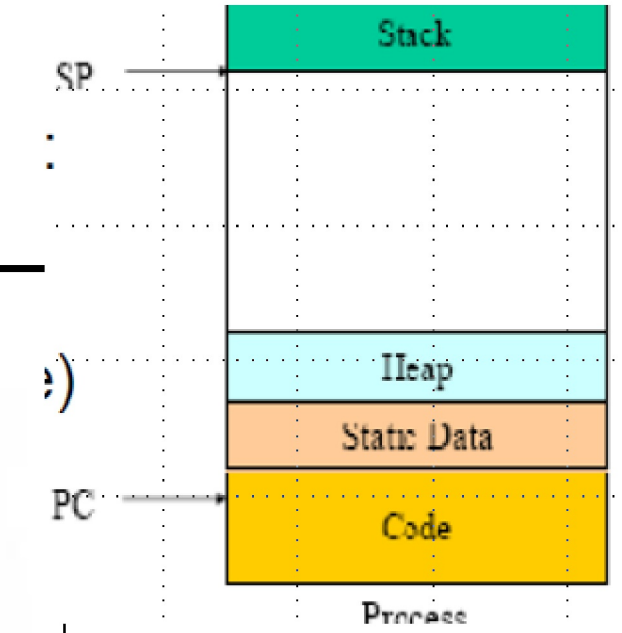
- A process can have multiple executions units called threads. All the threads of a single process share the same address space and have multiple threads of control .
- Each thread have separate thread control block (TCB) to manage the threads .
- Context switching is easy .
- Also called light weight processes.

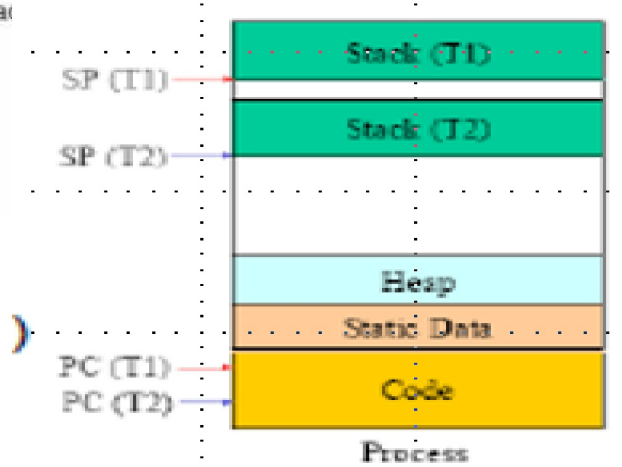
Process Vs Thread

Difference between Single vs. Multithread Process



- A process by itself can be viewed a single thread and is traditionally known as a heavy weight process





Process with 2 threads

More about threads …

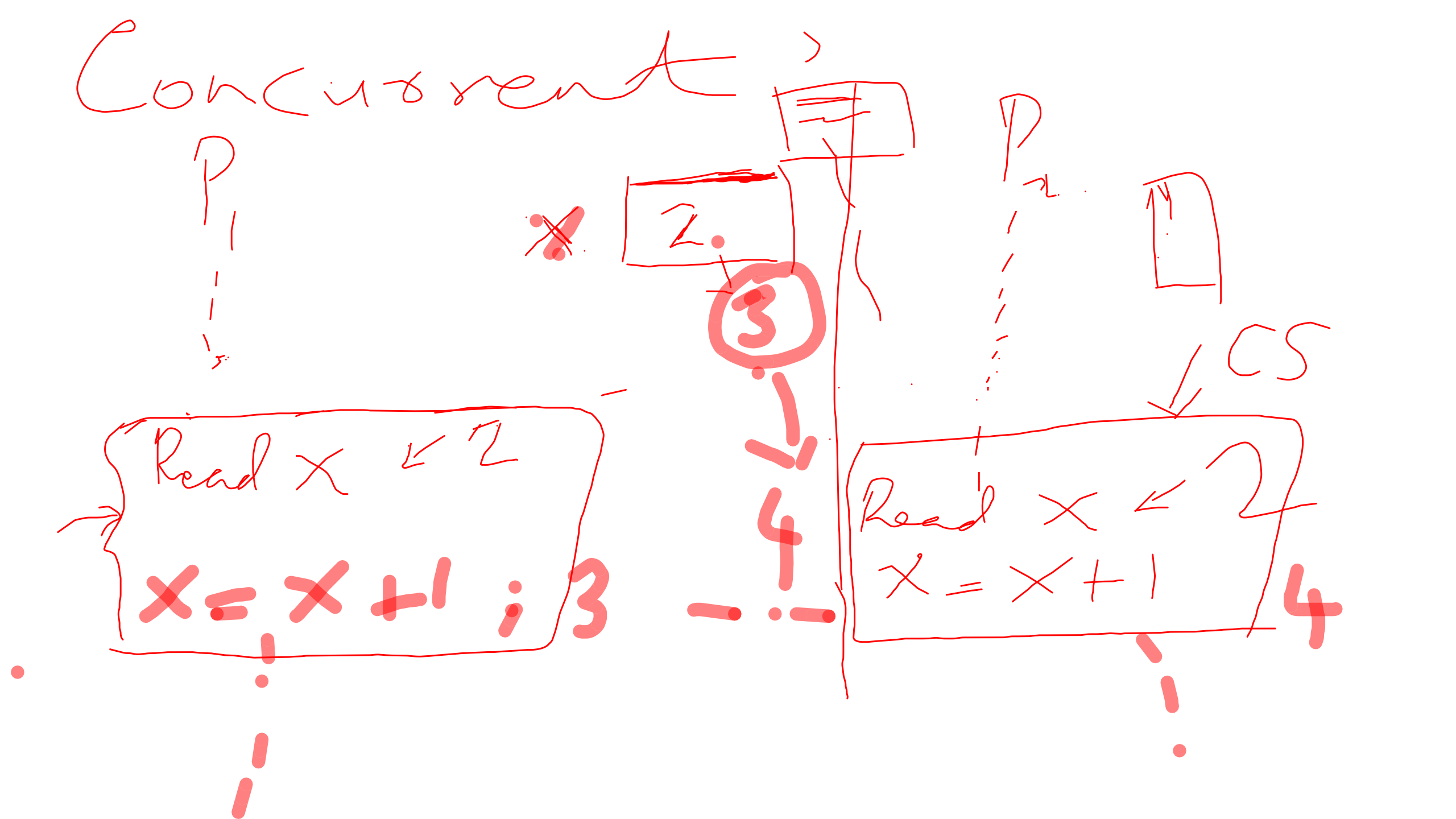
- A single thread executes a portion of a program ,cooperating with each other threads concurrently
- Each thread make use of a separate program counter and a stack of activation records (that describes the state of execution) and a thread control block.
- The control block contains the state information necessary for thread management.
- Most of the information that is part of a process is common to all threads executing within a single address space .

Process Synchronisation

Prof. BABY SYLA L

COLLEGE OF EMGINEERING TRIVANDRUM

1. Need for process Synchronization
2. Critical Section Problem
3. Solutions of CS problem


Concurrent
P1
P2
Z
3
4
CS
Read X ← Z
X = X + 1 ; 3
Read X ← Z
X = X + 1
4

Critical Section Problem

- Critical section problem - problem which occurs when two or more coordinating processes /threads trying to access a shared variable/ resource .
- It may violate the integrity of a shared variable
- Examples are
  - Variable does not record all the changes.
  - A process may read inconsistent values.
  - Final value of the variable do not be consistent.

- Critical section is a code segment in a process in which shared resource is accessed.

Solution to Critical Section Problem

- Process Synchronisation is necessary to solution to CS Problem.
  - Solution requires that processes be synchronised such that only one process access the shared resource at a time.
- This is why CS problem is also called mutual exclusion problem.
- Different solutions were proposed .
- Requirements for solutions to mutual exclusion problems:

1. Only one process can execute CS at a time.
2. When no process is executing CS ,then requests for other process should be permitted without any delay.
3. When two processes compete for entering to CS ,the selection can not be postponed indefinitely
4. No process can prevent any other other process from entering its CS ,means each process should get a fair chance to share the resource.

Critical Section

- General structure of process P_i

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

Request

Release

Early mechanisms of mutual exclusion

1. **Hardware support:** Some hardware instructions are provided to support the programmer
   - (e.g., busy waiting , disabling interrupts and 'test&set' instruction)
2. **Operating System Support:** Operating system supports for the declaration of data structures and also operations on those data structures
   - (e.g., semaphores)
3. **High Level Language Support:** Language provides support for data structures and operations on them to help with synchronization.
   - (e.g., critical regions, monitors, serializers, etc)

1. Hardware support:

- Busy waiting:
  - Process continuously tests the status variable to find if the shared resource is free .
  - Disadvantage: Wastage of CPU cycles and memory access bandwidth

Disaling interrupts:

- Process disables the interrupt before entering its CS and enables immediately after exiting the CS.
- Applicable is only for uniprocessor systems.

Test and Set Instruction:

- Used in multiprocessor systems .
- This instruction performs a single indivisible operation in one clock cycle.
- A specific memory location is checked for a particular value; if they match ,its contents are altered.

Busy waiting

- General structure of process P_i (other process P_j)

do {

check the status variable

critical section

reset the status variable

reminder section

} **while (1)**;

Disabling interrupts

- General structure of process P_i (other process P_j)

do {

Disable interrupts

critical section

Enable interrupts

reminder section

} **while (1)**;

Mutual Exclusion with Test-and-Set

- Shared data:

  boolean lock = false;

- Process P_i

```
do {
    while (TestAndSet(&lock)) ;
        critical section
    lock = false;
        remainder section
} while(1);
```

2. Operating system support:

- Semaphores: is a high level construct used to synchronise concurrent processes
- It is an integer variable
  - Can only be accessed via two indivisible (atomic) operations
    - P(S) and V(S)

  - Definition of the P(S) operation

wait (S)

```
{ while (S <= 0)
        ; // busy wait (block the process on semaphore queue)
  S--;
  }
```

  - Definition of V(S) operation

signal (S)

```
{ S++; }//
```

if some processes are blocked on the semaphore S
unblock a process

Semaphore Usage

- Proposed in 1969 by Dijkstra for process synchronization "Cooperating Sequential Processes"

Depending on the value of a semaphore allowed to take:

- **Binary semaphore** – integer value can range only between 0 and 1. (Intial value=1)
- **Counting semaphore** – integer value can range over an unrestricted domain (Initial value >1).
- A semaphore is initialized by the system.

Note: for any semaphore , the number of P operations – number of V operations <= initial value

Binary semaphore

```
Shared var
mutex: semaphore (= 1);

Process i (i = 1, n);

    begin
    .
    .
    .
    P(mutex);
    execute CS;
    V(mutex);
    .
    .
    .
    end.
```

FIGURE 2.2
Solution to mutual exclusion using a semaphore.

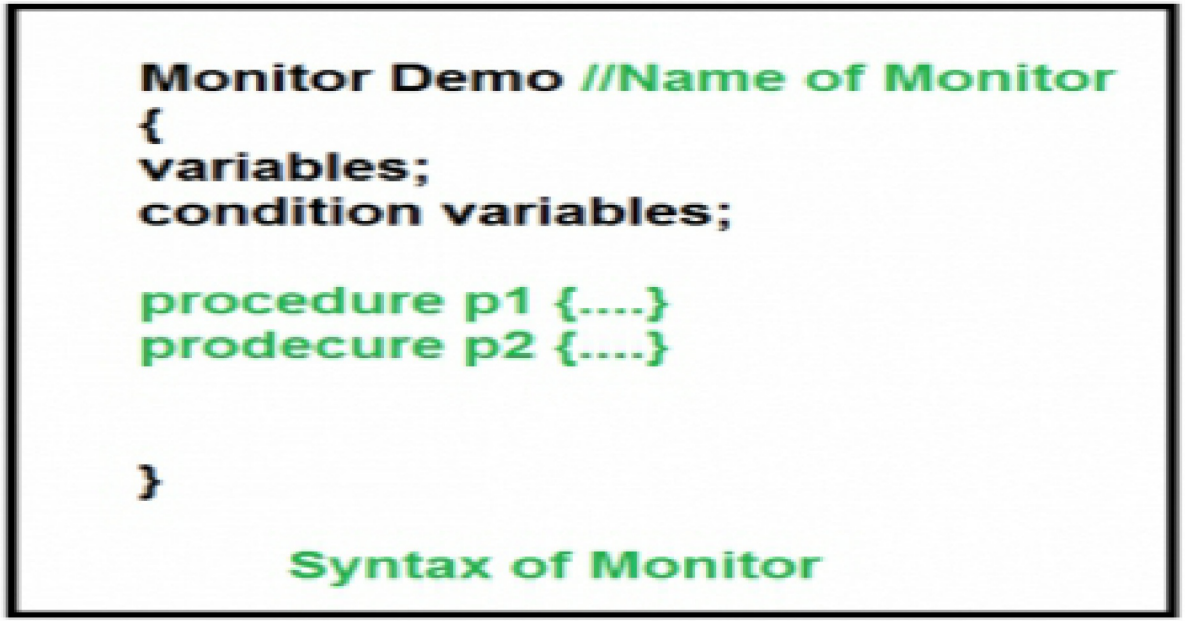
Semaphore Implementation

- Must guarantee that no two processes can execute the `P(S)` and `V(S)` on the same semaphore at the same time.
- Thus, the implementation becomes the critical section problem where **`P(S)`** and **`V(S)`** code are placed in the critical section.
- Semaphore operations must be carefully installed in a process.
- The omission of P or V may results in inconsistencies.
- Programs using semaphores can be extremely hard to verify for correctness.

  - Note that applications may spend lots of time in critical sections and therefore this is not a good solution

3. Language support:

- Monitors: are abstract datatypes for defining shared resources.
- It consists of condition variables and procedures combined together in a special kind of module or a package.
- Procedures are called by the process to access the resource.



Monitors

- The execution of a monitor obeys the following constraints:
- Only one process can be active within the monitor at a time.When a process is active wuthin the monitor, processes trying to enter the monitor are placed in the monitor’ entry queue.
- Procedures of a monitor can only access data local to the monitor ,they cannot access an outside variable.
- The variables or data local to monitor cannot be directly accessed from outside monitor.

Monitors

Two different operations are performed on the condition variables of the monitor.

1. wait() - suspends the caller process ,caller relinquishes control of monitor and placed on an urgent queue.

2. signal() – It causes exactly one waiting process to immediately regain the control of monitor.

- Processes in the urgent queue have a higher priority than processes trying to enter the monitor ,when a process relinquishes control of the monitor.

Process Synchronisation

Part-2

3. Language support:

- Monitors: are abstract datatypes for defining shared resources.
- It consists of condition variables and procedures combined together in a special kind of module or a package.
- Procedures are called by the process to access the resource.

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

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


}
```

Syntax of Monitor

Monitors

- The execution of a monitor obeys the following constraints:
- Only one process can be active within the monitor at a time.
- When a process is active within the monitor, processes trying to enter the monitor are placed in the monitor' entry queue.
- Procedures of a monitor can only access data local to the monitor ,they cannot access an outside variable.
- The variables or data local to monitor cannot be directly accessed from outside monitor.

Monitors

- To synchronize tasks within the monitor, a condition variable is used to delay processes executing in a monitor.
- Two different operations are performed on the condition variables each with its own queue.

1. wait() - suspends the caller process ,caller relinquishes control of monitor and placed on an urgent queue.

2. signal() – It causes exactly one waiting process to immediately regain the control of monitor.

Processes in the urgent queue have a higher priority than processes trying to enter the monitor , when a process relinquishes control of the monitor.

Monitors :

Advantages

- Flexibility in scheduling processes

- Disadvantages
- Only one active process inside a monitor: no concurrency.
- Responsibility of programmers to ensure proper synchronization.
- Nested monitor calls can lead to deadlocks:
- Responsibility of valid programs shifts to programmers, difficult to validate correctness.

Serializers

- **Serializers** was a mechanism proposed in 1979 to overcome some of the monitors' shortcomings
- ie, it allows concurrency inside.
- Proposed by Hewitt and Atkinson
- Basic structure is similar to monitors

Definition:
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.

- more automatic, high-level mechanism
- Automatic signaling: Condition for resuming the execution of a waiting process to be explicitly stated when a process waits.

Concurrency ?

- As in a Monitor only one process can have accesses or control over a serializer at a given time.
- 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**.
- 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.

Serializers –Hollow region

- 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.

Structure of a procedure

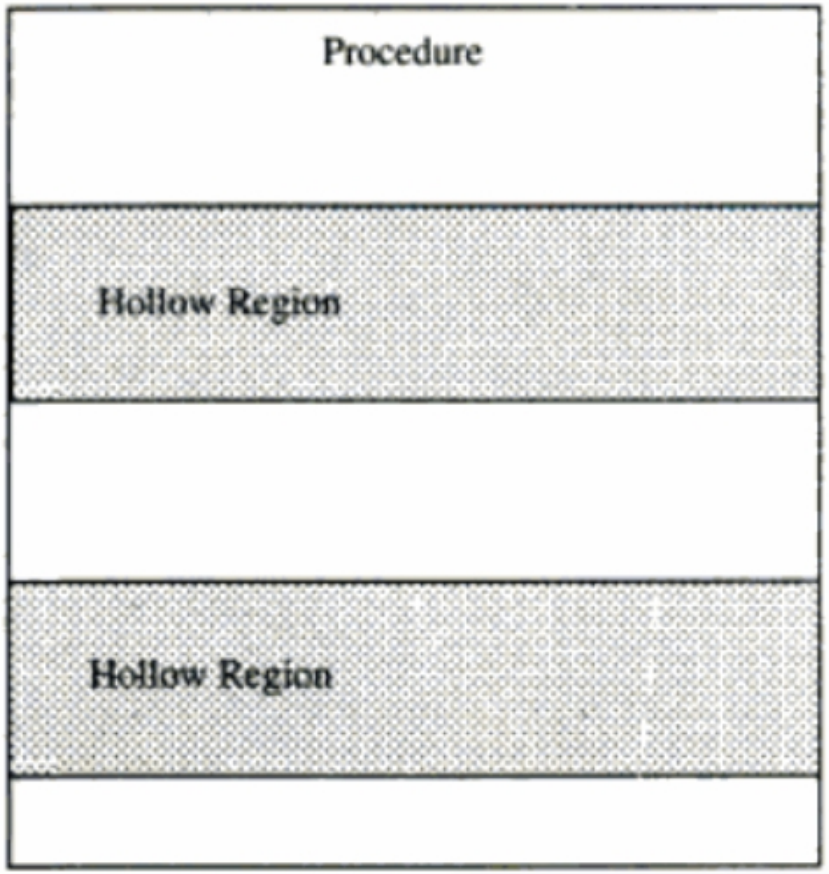


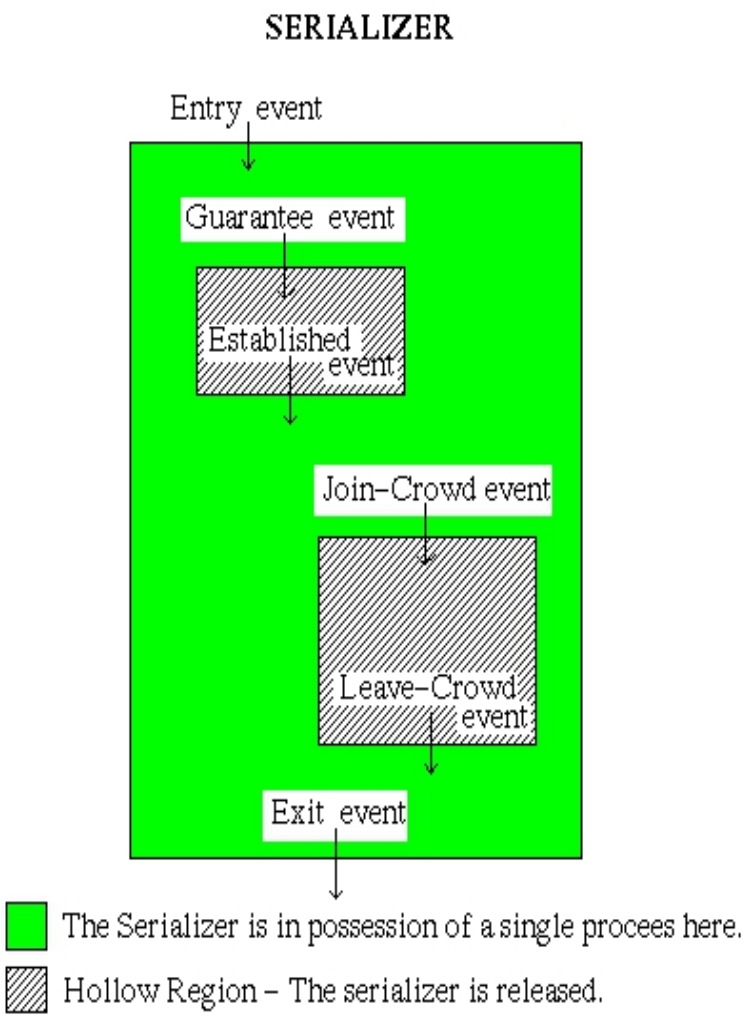
FIGURE 2.7
The structure of a procedure of a serializer.

Serializers- Queue variables

- whenever 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.

OPERATION OF SERIALIZER



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 resource. 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.

Monitors vs Serializers

- Serializers have several advantages over the monitors.
- 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.
- 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.
- 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.
- In monitors - explicit signalling.
  In serializers implicit signalling.

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.

PATH EXPRESSION

- **What is a Path Expression?**
- It is a declarative specification of synchronization between procedures.
- Automatically generated code uses semaphores for the automatic enforcement of the synchronization.

Path Expresiion

The Translator

Path expression $\rightarrow$ translator $\rightarrow$ code

- The path expression takes the form:

  path ⟨ expr ⟩ , ⟨ expr ⟩ ,...., ⟨ expr ⟩ **end**

- The translator is a simple set of rules that does the automatic code generation.
- The code is a set of procedures with a prologue and an epilogue.
  The prologue checks to see if the procedure can begin.
  The epilogue tells when the procedure is finished.

PATH EXPRESSION

The concept of *path expression* was proposed by Campbell and Habermann [2]. Conceptually, a "path expression" is a quite different approach to process synchronization. A path expression restricts the set of admissible exeeution 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:

Operators

Sequencing (;). It defines a sequencing order among operations. For example, **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 time. For example, **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.

Concurrency ({}). It signifies that any number of instances of the operation delimited by { and } can be in execution at a time. For example, **path** {read} **end** means that any number of read operations can be executed concurrently. The path expression **path** write; {read} **end** allows either several read operations or a single write operation to be executed at any time (read and write operations exclude each other). However, whenever the system is empty after all readers have finished, the writer must execute

Operators contd..

first. Between every two write operations, at least one read operation must be executed. The path expression **path** {write; read} **end** means that at any time there can be any number of instantiations of the path write; read. At any instant, the number of read operations executed is less than or equal to the number of write operations executed. The path expression **path** {write + read} **end** is meaningless and does not impose any restriction on the execution of read and write operations.

Operators

Operators

The expressions are formed from the following operators (where x and y are procedures or path expressions):

- sequencer: x;y
  x must execute before y.
  put; get
- restrictor: n:(x)
  x has a maximum of n concurrent executions.
  1:(write)
- derestrictor: [x]
  x can have an unlimited number of concurrent executions.
  [read]
- grouping: (...)
  expresses precedence or nesting.
- compound example:
  1:(write),[read]

The Comma

The comma can be a

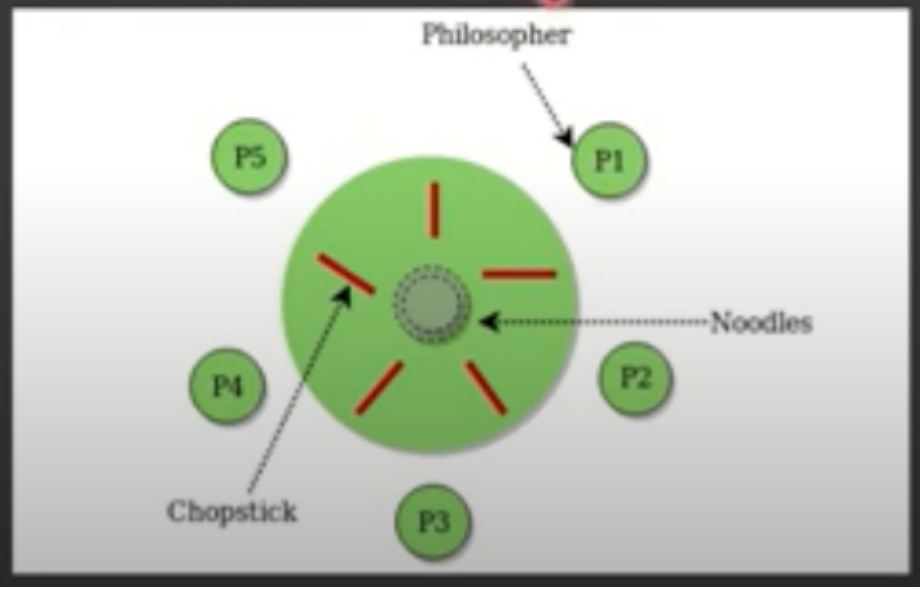
- OR - if procedure names on either side of the comma are different.
- composition (seperator) - if procedure names on either side of the comma are the same.

OTHER SYNCHRONISATION PROBLEMS

- One of the problem studied- Mutual exclusion
- Other problems :
- 1.Producer Consumer Problem
- 2.Reader’s writer Problem
- Dining Philosopher Problem

Dining Philosophers Problem (DPP)

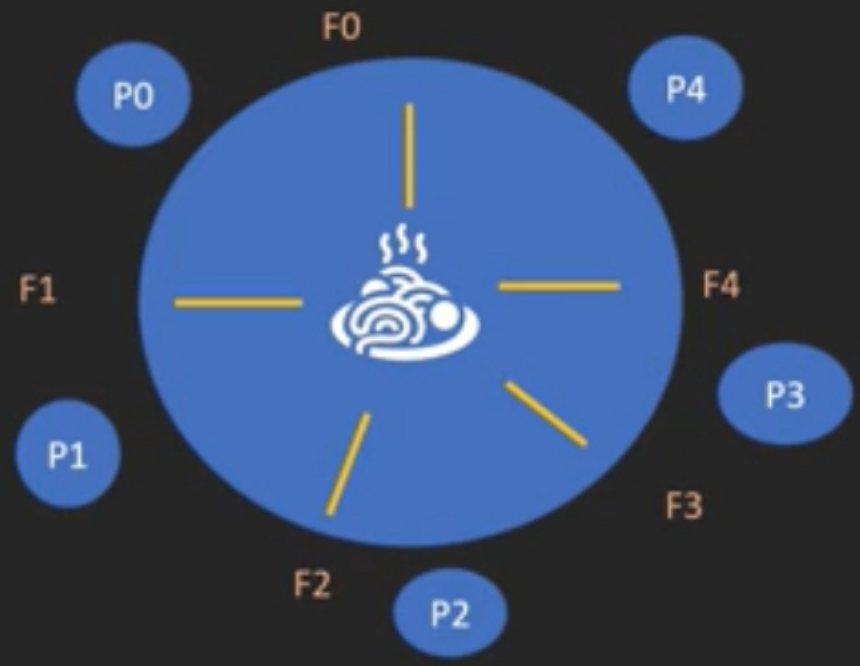
- Dining Philosopher Problem states that 5 philosophers seated around a circular table with one fork between each pair of philosophers
- There is one fork between each philosopher. A philosopher may eat if he can pickup the two forks adjacent to him.
- Every philosopher has 2 states, thinking and Eating
- The dining philosopher is a classic synchronization problem as it demonstrates a large class of concurrency control problems.



The dining philosophers Problem

```
Void philosopher (void)
{
        while(true)
        {
                Thinking()
                take_fork(i); // Left fork
                take_fork((i+1)% N); //  Right Fork
                EAT();
                put_fork(i);
                put_fork((i+1)% N);


        }
}
```



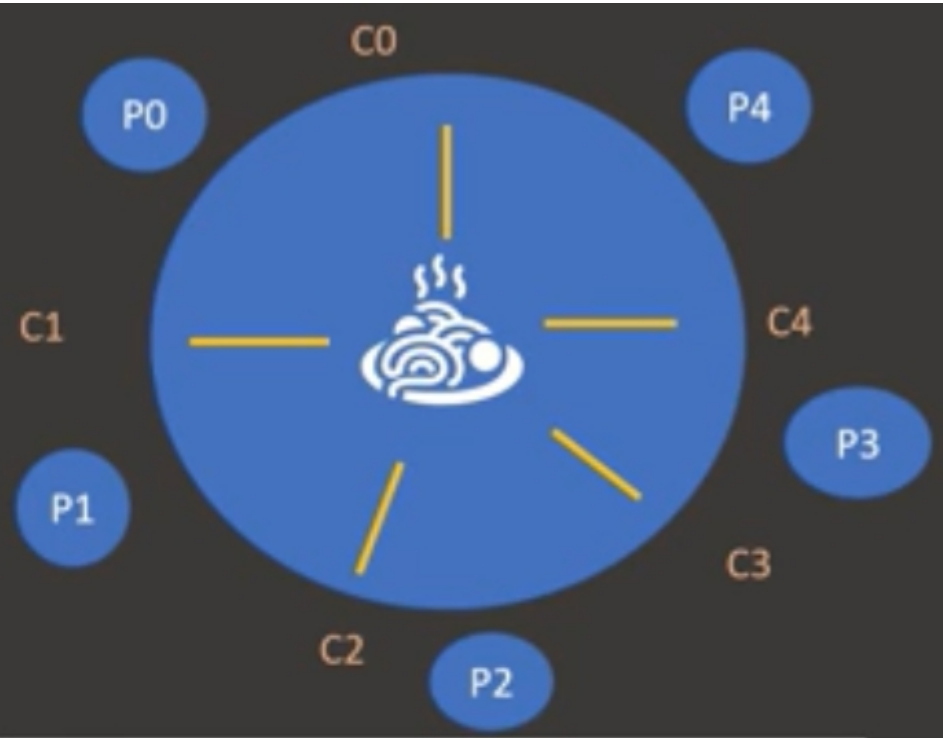
Solution of Dining Philosophers Problem

The structure of a random philosopher

```
Void philosopher (void)
{
        while(true)
        {
                Thinking();
                wait(take_fork(si)); // Left fork
                wait(take_fork (s(i+1) % N); // Right fork
                EAT();
                signal(put_fork(i));
                signal(put_fork(i+1)% N);


        }
}
```

S0	S1	S2	S3	S4
1	1	1	1	1



P0	S0	S1
P1	S1	S2
P2	S2	S3
P3	S3	S4
P4	S4	S0

Producer – Consumer Problem

- The producer consumer problem is a synchronization problem
- There is a fixed size buffer and the producer produces items and enters them into the buffer.
- The consumer removes the items from the buffer and consumes them.

Problem Parameters

- A producer should not produce items into the buffer when the consumer is consuming an item from the buffer and vice versa.
- The buffer should only be accessed by the producer or consumer at a time.

Readers - writers problem : Synchronization problem

- There is a file that is shared between multiple processes. Some of these processes are readers(read data from file), others
- are writers (write data into file).
- Multiple readers can access the file at the same time.
- If any user editing the file, no other person can read or write data at the same time

Problem parameters

To solve this situation, a writer should get exclusive access to an object i.e. when a writer is accessing the object, no reader or writer may access it. However, multiple readers can access the object at the same time.

Producer - Consumer Problem (Text book)

In the producer-consumer problem, a set of *producer* processes supplies messages to a set of *consumer* processes. These processes share a common buffer pool where messages are deposited by producers and removed by consumers. All the processes are asynchronous in the sense that producers and consumers may attempt to deposit and remove messages, respectively, at any instant. Since producer processes may outpace consumer processes (or vice versa), two constraints need to be satisfied; no consumer process can remove a message when the buffer pool is empty and no producer process can deposit a message when the buffer pool is full.

Integrity problems may arise if multiple consumers (or multiple producers) try to remove messages (or try to put messages) in the buffer pool simultaneously. For examples, associated data structures (e.g., pointers to buffers) may not be updated consistently, or two producers may try to put messages in the same buffer. Therefore, access to the buffer pool and the associated data structures must constitute a critical section in these processes.

Readers- writers Problem(Text book)

In the readers-writers problem, the shared resource is a file that is accessed by both the reader and writer processes. Reader processes simply read the information in the file without changing its contents. Writer processes 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. Moreover, readers and writers must always exclude each other.

There are several versions of this problem depending upon whether readers or writers are given priority.

Different options

Reader's Priority. In the reader's priority case, arriving readers receive priority over waiting writers. A waiting or an arriving writer gains access to the file only when there are no readers in the system. When a writer is done with the file, all the waiting readers have priority over the waiting writers.

Writer's Priority. In the writer's priority case, an arriving writer receives priority over waiting readers. A waiting or an arriving reader gains access to the file only when there are no writers in the system. When a reader is done with the file, waiting writers have priority over waiting readers to access the file.

In the reader's priority case, writers may *starve* (i.e., writers may wait indefinitely) and vice-versa. To overcome this problem, a *weak reader's priority* case or a *weak writer's priority* case can be used. In a weak reader's priority case, an arriving reader still has priority over waiting writers. However, when a writer departs, both waiting readers and waiting writers have equal priority (that is, a waiting reader or a waiting writer is chosen randomly).

The dining philosophers Problem (Text book)

The dining philosophers problem is a classic synchronization problem that has formed the basis for a large class of synchronization problems. In one version of this problem, five philosophers are sitting in a circle, attempting to eat spaghetti with the help of forks. Each philosopher has a bowl of spaghetti but there are only five forks (with one fork placed to the left, and one to the right of each philosopher) to share among them. This creates a dilemma, as both forks (to the left and right) are needed by each philosopher to consume the spaghetti.

A philosopher alternates between two phases: thinking and eating. In the *thinking* mode, a philosopher does not hold a fork. However, when hungry (after staying in the thinking mode for a finite time), a philosopher attempts to pick up both forks on the left and right sides. (At any given moment, only one philosopher can hold a given fork, and a philosopher cannot pick up two forks simultaneously). A philosopher can start eating only after obtaining both forks. Once a philosopher starts eating, the forks are not relinquished until the eating phase is over. When the eating phase concludes (which lasts for finite time), both forks are put back in their original position and the philosopher reenters the thinking phase.

Note that no two neighboring philosophers can eat simultaneously. In any solution to this problem, the act of picking up a fork by a philosopher must be a critical section. Devising a deadlock-free solution to this problem, in which no philosopher starves, is nontrivial.

Design issues of distributed operating system

Module-1

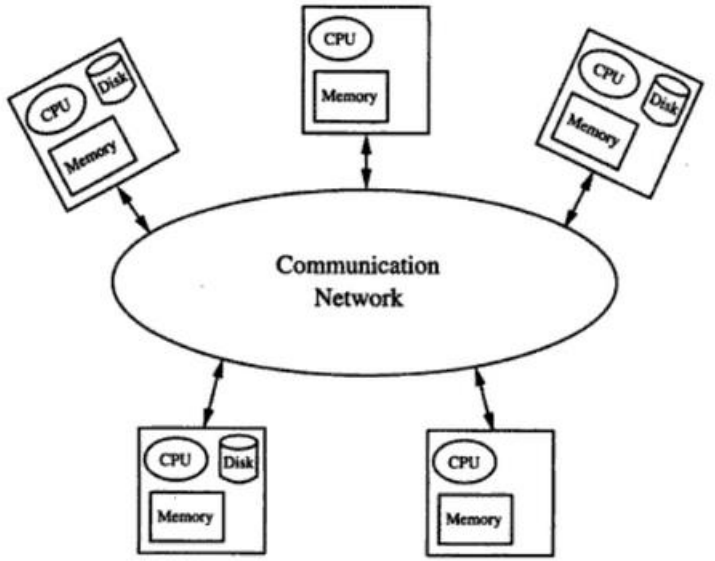
Distributed System

It is a collection of autonomous computers connected by a communication network

Properties:

1.Each computers has its own memory and clock.

2. Computers communicate with each other by exchanging messages.

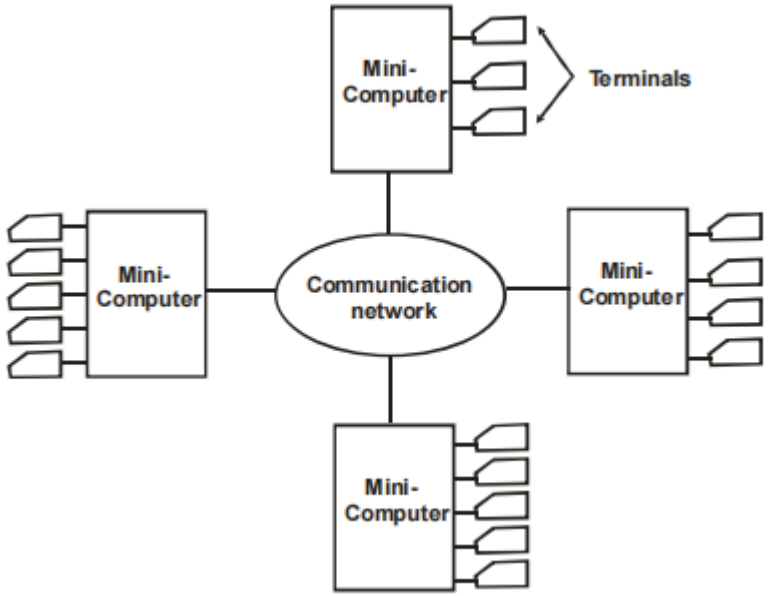


System Architectures

- Minicomputer model
- Workstation model
- Processor pool model

Minicomputer model

- It consists of a few minicomputers interconnected by a communication network.
- Each minicomputer usually has several interactive terminals attached to it.
- Each user is logged on to one specific minicomputer, with remote access to other minicomputers.
- Advantage: Resource sharing

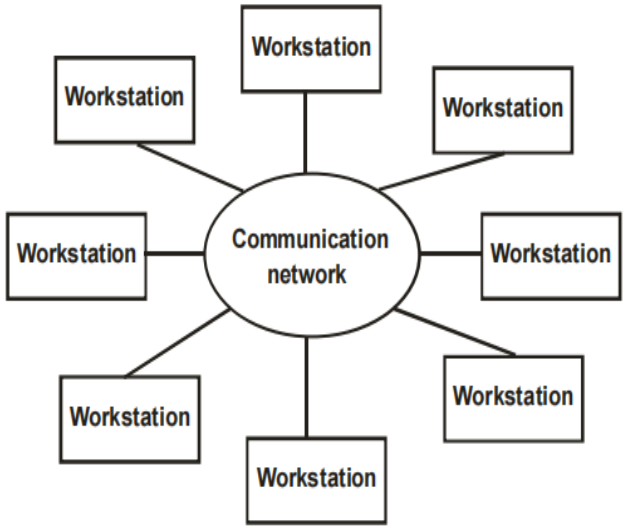


Example:
The early ARPAnet is an example of a distributed computing system based on the minicomputer model.

Minicomputer model

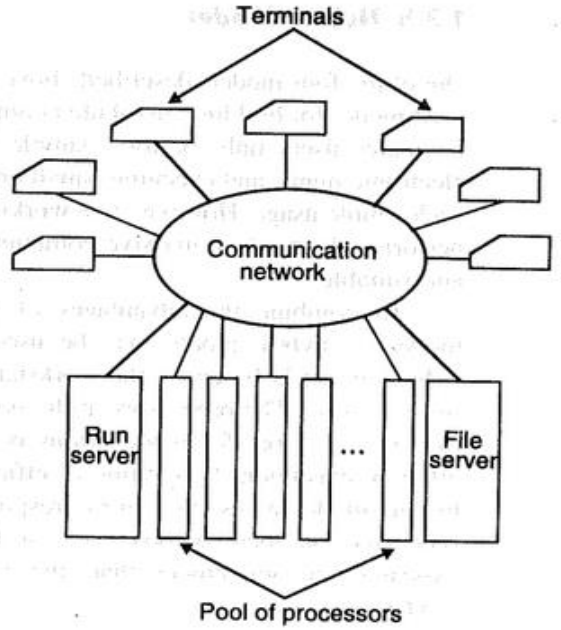
- It consists of several workstations interconnected by a communication network.
- In this model, a user logs onto one of the workstations called his or her “home” workstation and submits jobs for execution.
- Normal computation activities required by the user’s processes are preformed at the user’s home workstation.
- When the system finds that the user’s workstation does not have sufficient processing power for executing the processes of the submitted jobs efficiently, it transfers one or more of the process from the user’s workstation to some other workstation that is currently idle and gets the process executed there, and finally the result of execution is returned to the user’s workstation.

Examples:
Athena and Andrew



Processor pool model

- Processors are pooled together to be shared by the users as needed.
- In this model, there is no concept of home machine.
- Users login to the system as a whole through the terminals and submit tasks.
- The run server allocates appropriate number of processors to the task .
- When the computation is completed, the processors are returned to the pool for use by other users.



Examples:
Amoeba and Cambridge distributed computing systems

Advantages of distributed system

- Main advantage :Price/Performance ratio is high.

- Other advantages:
- 1.Resource sharing: Computers can send a request for other resources (s/w or h/w) available in the system.
- 2. Enhanced performance: shorter response time and higher system throughput.
- 3.Improved reliability and availability: System is fault tolerant . Failure of single system does not affect the whole

- 4.Modular Expandability: New resources can be added without replacing the existing resources.

Operating systems for Distributed systems

- Network Operating systems
- Distributed operating systems

- 3 features are used to differentiate the two operating systems above.
- 1.system image
- 2. autonomy
- 3. fault tolerance capability

Difference

Network OS

- Users are aware about existence of multiple computers.
- Each computer functions independently and runs its own OS and resources are managed locally.
- No fault tolerant capability

Distributed OS

- It hides the existence of multiple computers and provides a single system image to its users.(Transparency)
- There is a single system wide OS and each computer runs a part/identical copies of the operating system .They work in close cooperation with each other . Resources are managed globally.
- Fault tolerant capability and users can complete the task with a small loss in performance

Issues in Distributed Operating system .

1. Global Knowledge
2. Naming
3. Scalability
4. Compatibility
5. Process synchronisation
6. Resource management
7. Security
8. Structure of OS
9. Client server computing model.

1.Global knowledge

- Issue:
  - It is practically impossible to collect up-to-date information about global state of distributed system.
- Reason:
  - Lack of global clock and lack of global memory.
- Design should incorporate:
  - Temporal ordering of events, scheduling of jobs based on arrival etc
- Techniques are needed to solve this problem .

2.Naming

A good naming system for a distributed system should have the features described below.

- *1. Location transparency*
  - means that the name of an object should not reveal any hint as to the physical location of the object.

2. Location independency

  - means that the name of an object need not be changed when the object's location changes.
  - Furthermore, a user should be able to access an object by its same name irrespective of the node from where he or she accesses it.

Naming contd…

- A *name server* is a process that maintains information about named objects and It acts to bind an object's name to object's location.
- In distributed systems, name servers/look up tables may be replicated and stored in different locations for reliability.
- A location-independent naming system must support a dynamic mapping scheme so that it can map the same object name to different locations at two different instances of time.
- 2 drawbacks of replication are
    1. requires more storage
    2. synchronisation requirements need to met

    - when one entry is updated/deleted ,changes should be made at all its copies.

3. Scalability

- *Scalability* refers to the capability of a system to adapt to increased service load.
- It is inevitable that a distributed system will grow with time since it is very common to add new machines or an entire subnetwork to the system.
- A distributed operating system should be designed to easily cope with the growth of nodes and users in the system.
- That is, such growth should not cause serious disruption of service or significant loss of performance to users.
- Design suggestions:- Avoid centralised entities, centralised algorithmsand performs computation in client worksataion itself.

4. Compatibility

- Compatibility refers to notion of interoperability among the resources in a system.
- 3 different levels exist

1. Binary Level
    - All processors execute the same binary instruction even though the processors may differ in performance .
    - Advantage : System development is easy but not recommended for building distributed systems because it do not support heterogenous systems.

Compatibility

2. Execution Level

- Same source code can be compiled and executed on any computer in the system

3. Protocol level

All the system components to support a common set of protocols.

Advantage: Individual computers can run different operating systems while not sacrificing interoperability.

5. Process Synchronisation

- Synchronisation of processes is difficult because there is no shared memory.
- But it is essential when different systems trying to access a shared resource.(eg:- fileserver)
- For ensuring correctness, it is necessary that the shared resource be accessed by a single process at a time.
- This problem is known as mutual exclusion problem.

6. Resource Management

- Resource management is concerned with making both local and remote resources available to users.
- Users should be able to access remote resources as easily as they can access local resources.
- In other words ,Specific location of resources should be hidden from users.

It can done using different ways.

Resource Management cntd..

1.Data migration

- Data is brought to the location where it is needed
- Data may be a file or contents of a physical memory.
- If any changes made ,the orginal location have to be updated.

2. Computation migration

- Computation migrates to another location.
- Mechanism used is RPC (Remote Procedure call)

3. Distributed Scheduling

- Processes are transferred from one location to another by the Distributed OS.
- It is desirable when a computer where process originated is overloaded or it does not have necessary resources required for a process .

7. Security

- Two issues must be considered in the design of security

1. Authentication
   - Process of guaranteeing that an entity is what it claims to be.
2. Authorisation
   - Process of deciding what privileges an entity has and making only those privileges available.

8. Structuring

- Structuring defines how various parts of OS are organised.
- Different structures:
- 1.Collective Kernel structure (Microkernel)
  - OS services such as distributed memory management, scheduling, name services, RPC, time management etc are implemented as independent processes.
  - Nucleus ,also called microkernel supports the interaction between processes.
  - Other functions of kernel are task management, processor management ,etc.
  - The microkernel runs on all computers in a distributed system.
  - The other processes may or may not run at a computer.
  - Examples:
    - Mach, Chorus

Structuring contd..

2. Object oriented OS
   - In Collective Kernel structure, services are implemented as process whereas in object oriented OS ,system services are implemented as objects.
   - Each object encapsulates a data structure and defines a set of operations on that data structure.
   - Examples: Amoeba, Cloud

9. Client server model

- Processes are categorised as client and servers.
- Process needs service (client) sends request to servers.
- Severs respond with the request and result may be send back to the client.
- In systems with multiple servers, location and conversations among the servers are transparent to the clients.

Communication Networks and Primitives

- Reading Assignment:

  Communication Networks

Communication models and the primitives used for communication

There are two communication models that provide communication primitives.

1. Message passing model
2. Remote Procedure call

1.Message Passing model

- A form of communication between two processes
- A physical copy of message is sent from one process to the other
- 2 primitives

1.Send(msg, destination)

2.Receive(source, buf)

These two can be :

1.Blocking VS Non blocking

2.Synchronous VS Asynchronous

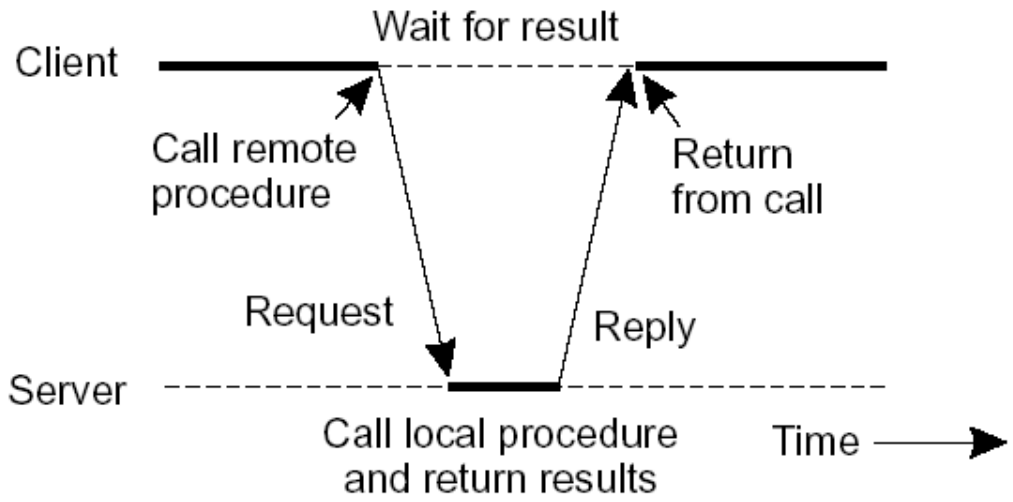
Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**

2.RPC

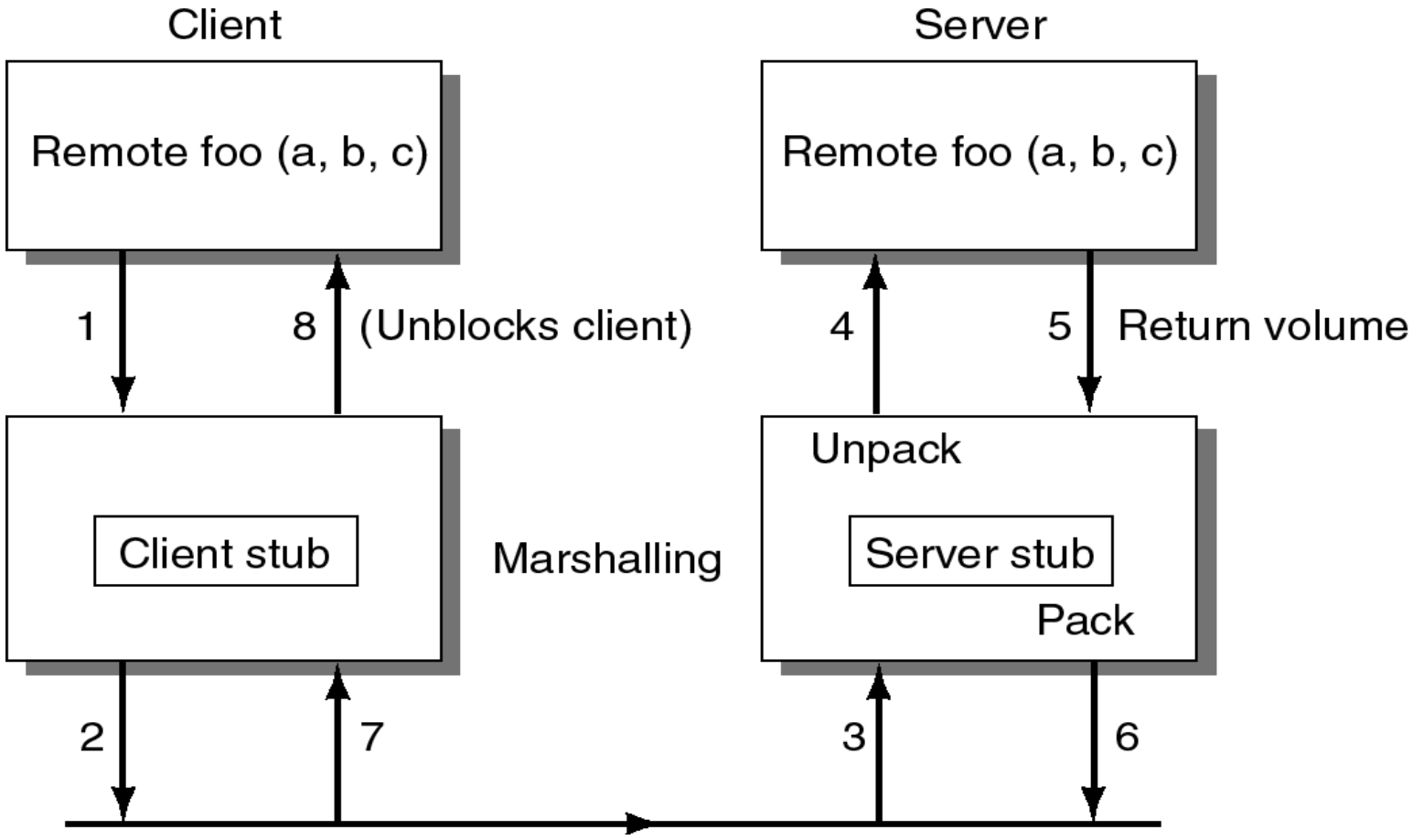
- RPC –Remote Procedure Call

Basic RPC Operation



Note: Communication between caller & callee can be hidden by using procedure-call mechanism.

RPC Implementation (2/2)



Steps involved in doing a remote “foo” operation

Remote Procedure Calls (1)

A remote procedure call occurs in the following

steps:

1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message and calls the local operating system.
3. The client's OS sends the message to the remote OS.
4. The remote OS gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.

Continued …

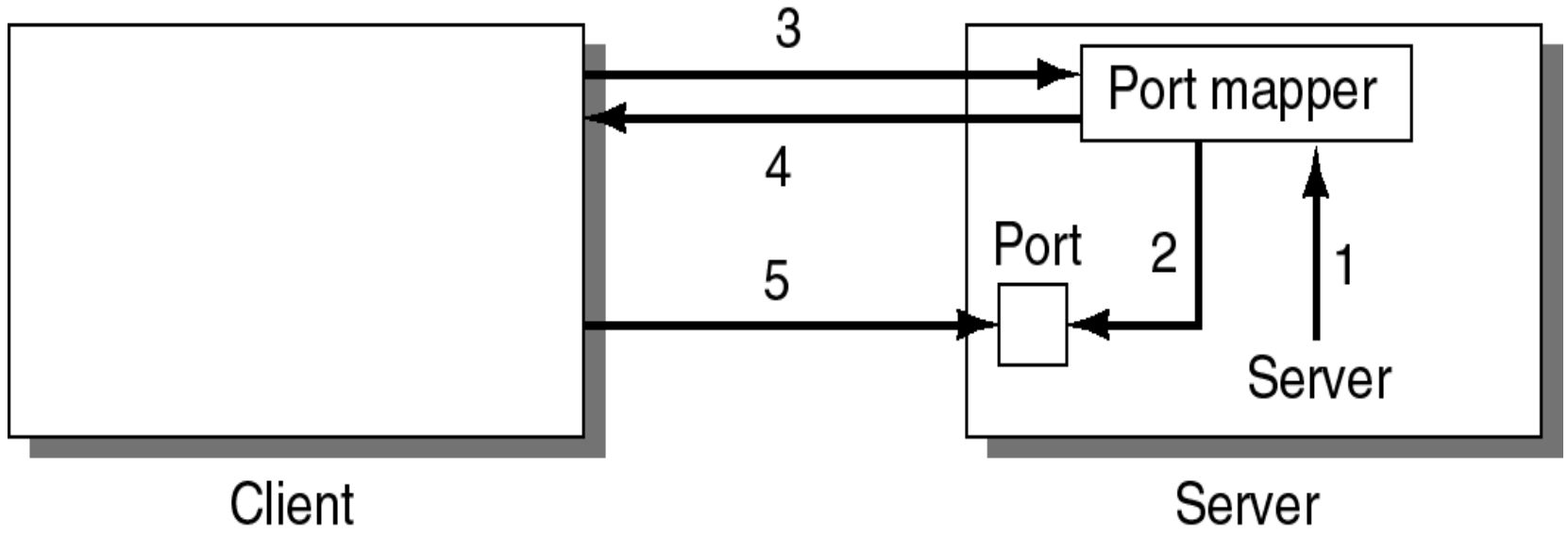
Remote Procedure Calls (2)

A remote procedure call occurs in the following

steps (continued):

6. The server does the work and returns the result to the stub.
7. The server stub packs it in a message and calls its local OS.
8. The server's OS sends the message to the client's OS.
9. The client's OS gives the message to the client stub.
10. The stub unpacks the result and returns to the client.

Establishing Communication for RPC.



1. I need a port
2. Here is your port
3. I need a handle
5. Communicate using handle

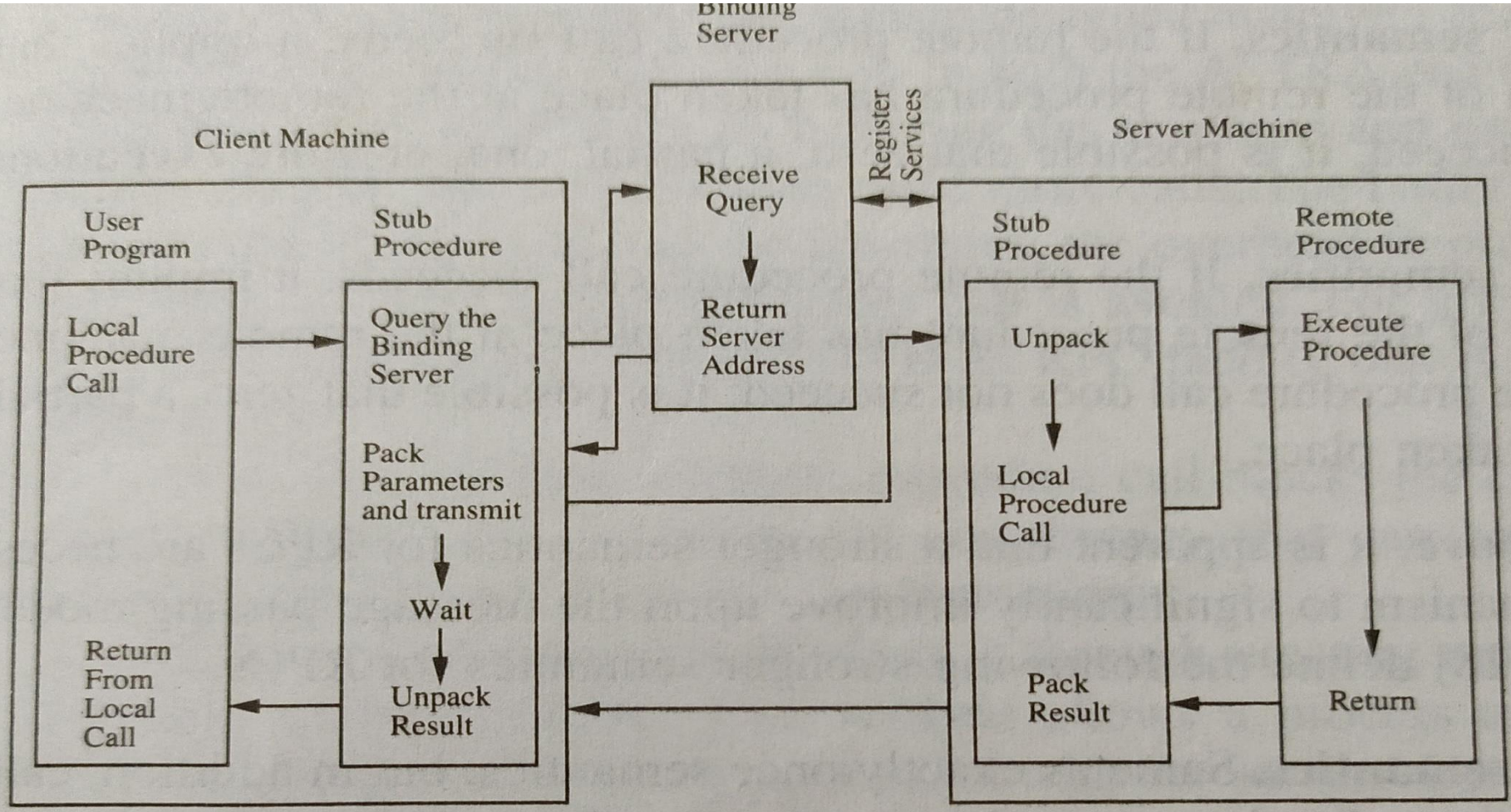


FIGURE 4.5
Remote procedure call.

Clock Synchronization

In distributes systems ,there is no global cloavk.

- How do we synchronize clocks with real-world time?
- How do we synchronize clocks with each other?
- How the events in a distributed system are ordered globally ?

Lamport's Logical Clock

- Lamport' propsed a scheme to order events in distributed system using a logical clock concept.
- For partial ordering of events, Lamport defined a new relation called *happened before* and introduced the concept of logical clocks for ordering of events based on the happened-before relation

The Happened-Before Relationship

The **happened-before** relation on the set of events in a distributed system is the smallest relation satisfying:

- If a and b are two events in the same process, and a comes before b, then $\boldsymbol{a \rightarrow b}$. (***a*** happened before ***b***)
- If a is the sending of a message, and b is the receipt of that message, then $\boldsymbol{a \rightarrow b}$.
- If $a \rightarrow b$ and $b \rightarrow c$, then $\boldsymbol{a \rightarrow c}$. (transitive relation)

Note: if two events, x and y, happen in different processes that do not exchange messages, then they are said to be **concurrent.**

Note: this introduces a **partial ordering** of events in a system with concurrently operating processes.

Logical Clocks Concept

Problem:

To determine that an event *a* happened before an event *b,* either a common clock or a set of perfectly synchronized clocks is needed.

Lamport [1978] provided a solution for this problem by introducing the concept of logical clocks.

The logical clocks concept is a way to associate a timestamp (which may be simply a number independent of any clock time) with each system event so that events that are related to each other by the happened-before relation (directly or indirectly) can be properly ordered in that sequence.

Solution: attach a timestamp $C(e)$ to each event e, satisfying the following properties:

P1: If a and b are two events in the same process, and $a \rightarrow b$, then we demand that $C(a) < C(b)$

P2: If a corresponds to sending a message m, and b to the receipt of that message, then also $C(a) < C(b)$

Implementation of Logical Clocks using counters

Each process Pi maintains a **local** counter Ci and adjusts this counter according to the following rules:

(1) For any two successive events that take place within Pi, Ci is incremented by 1.

(2) Each time a message m is sent by process Pi, the message receives a timestamp $Tm = Ci$.

(3) Whenever a message m is received by a process Pj, Pj adjusts its local counter Cj:

$$C_j \leftarrow \max\{C_j + 1, T_m + 1\}.$$

This is called the **Lamport's Algorithm**

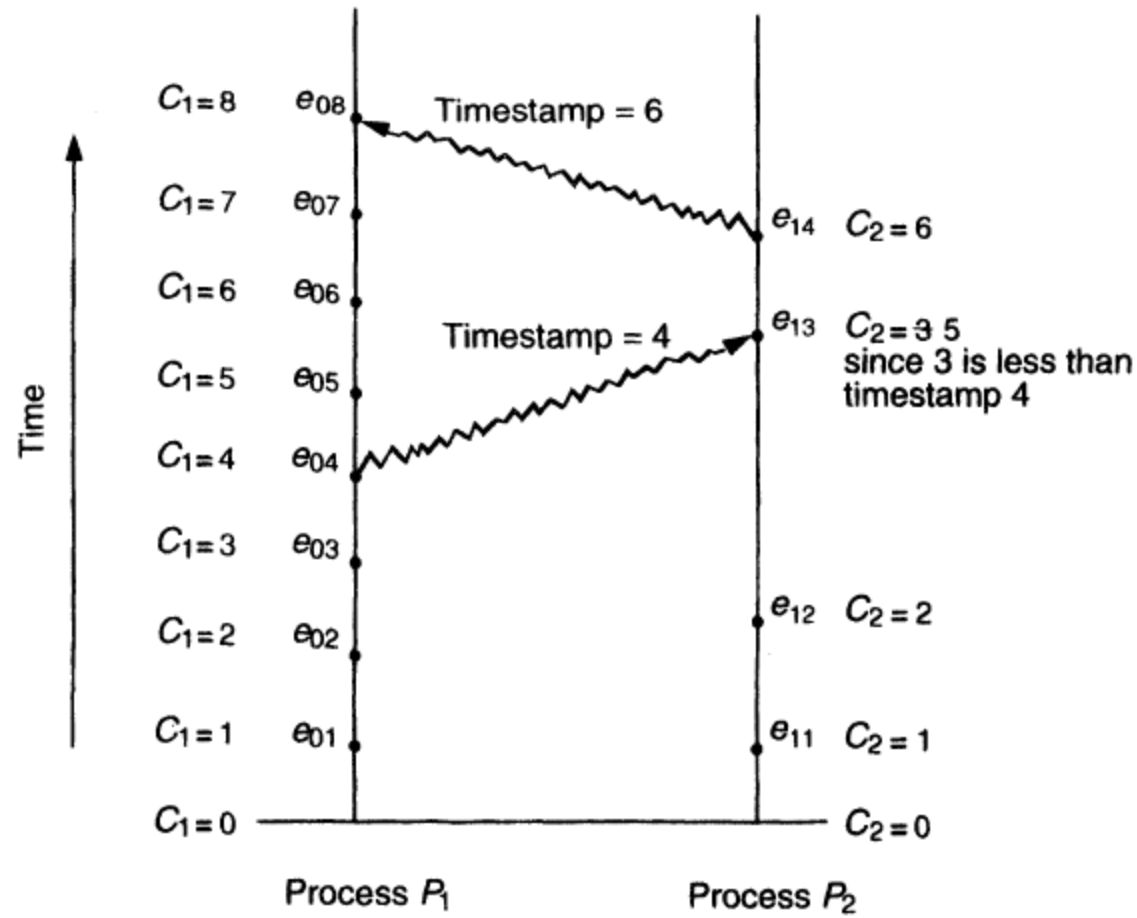


Fig. 6.4 Example illustrating the implementation of logical clocks by using counters.

Logical Clocks – Implementation using physical clocks

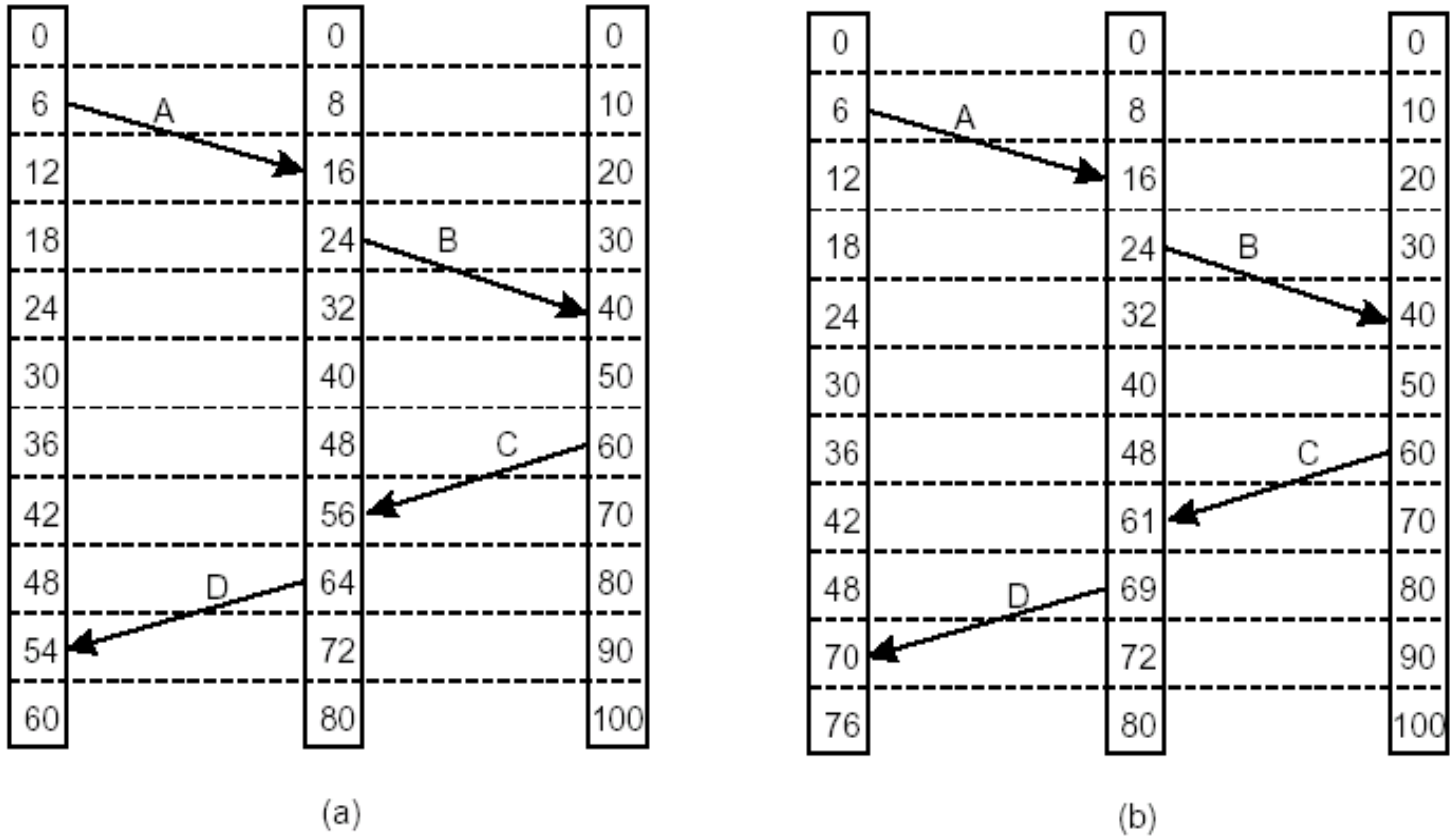
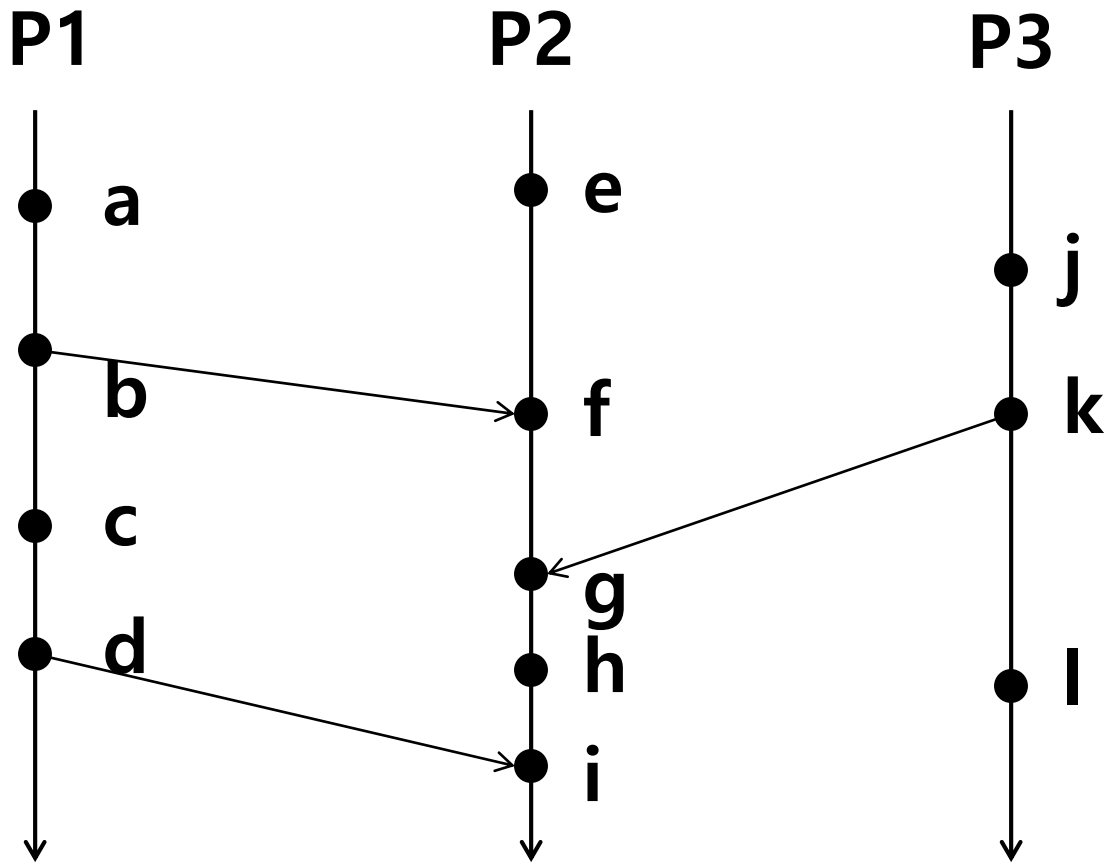
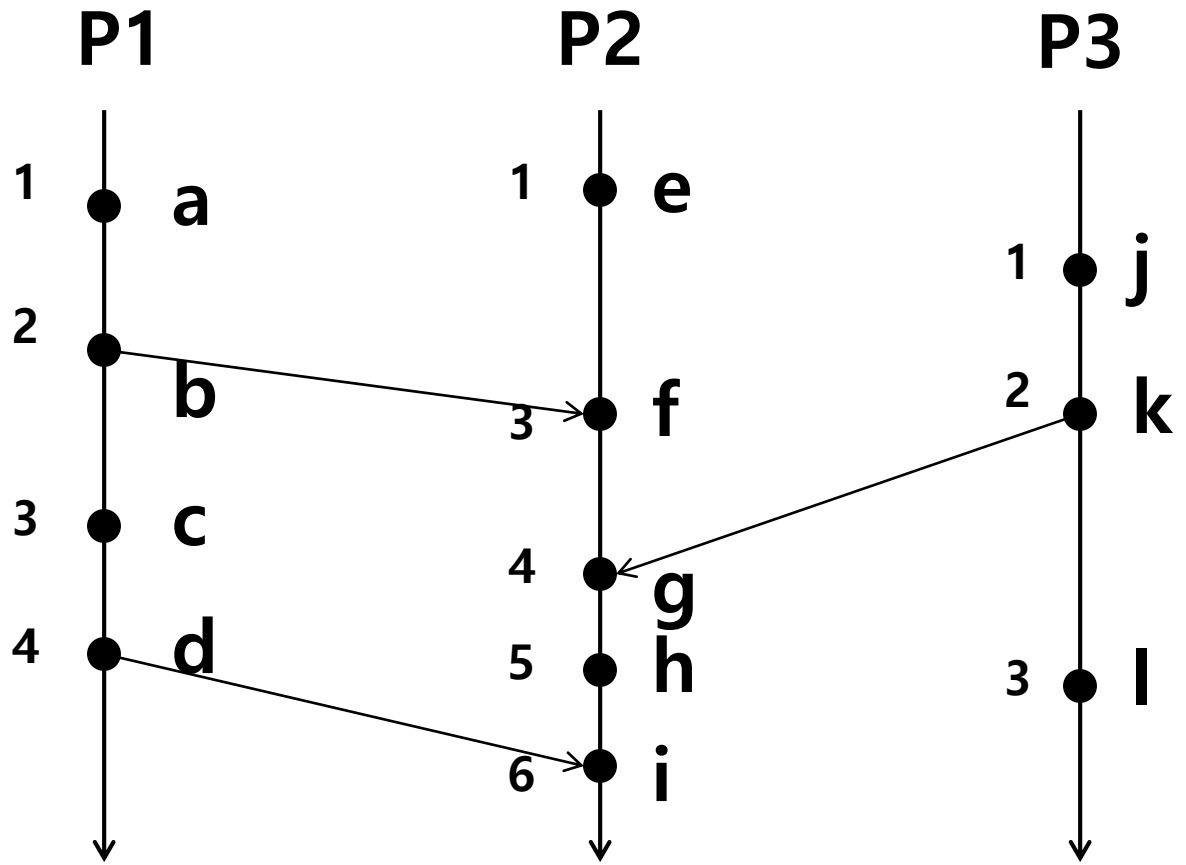


Fig 5-7. (a) Three processes, each with its own clock. The clocks run at different rates. (b) Lamport's algorithm corrects the clocks



- Assign the Lamport's logical clock values for all the events in the above timing diagram. Assume that each process's local clock is set to 0 initially.



From the above timing diagram, what can you say about the following events?

- **between a and b:** $a \rightarrow b$
- **between b and f :** $b \rightarrow f$
- **between e and k:** concurrent
- **between c and h:** concurrent
- **between k and h:** $k \rightarrow h$

Total Ordering with Logical Clocks

Problem: it can still occur that two events happen at the same time. Avoid this by attaching a process number to an event:

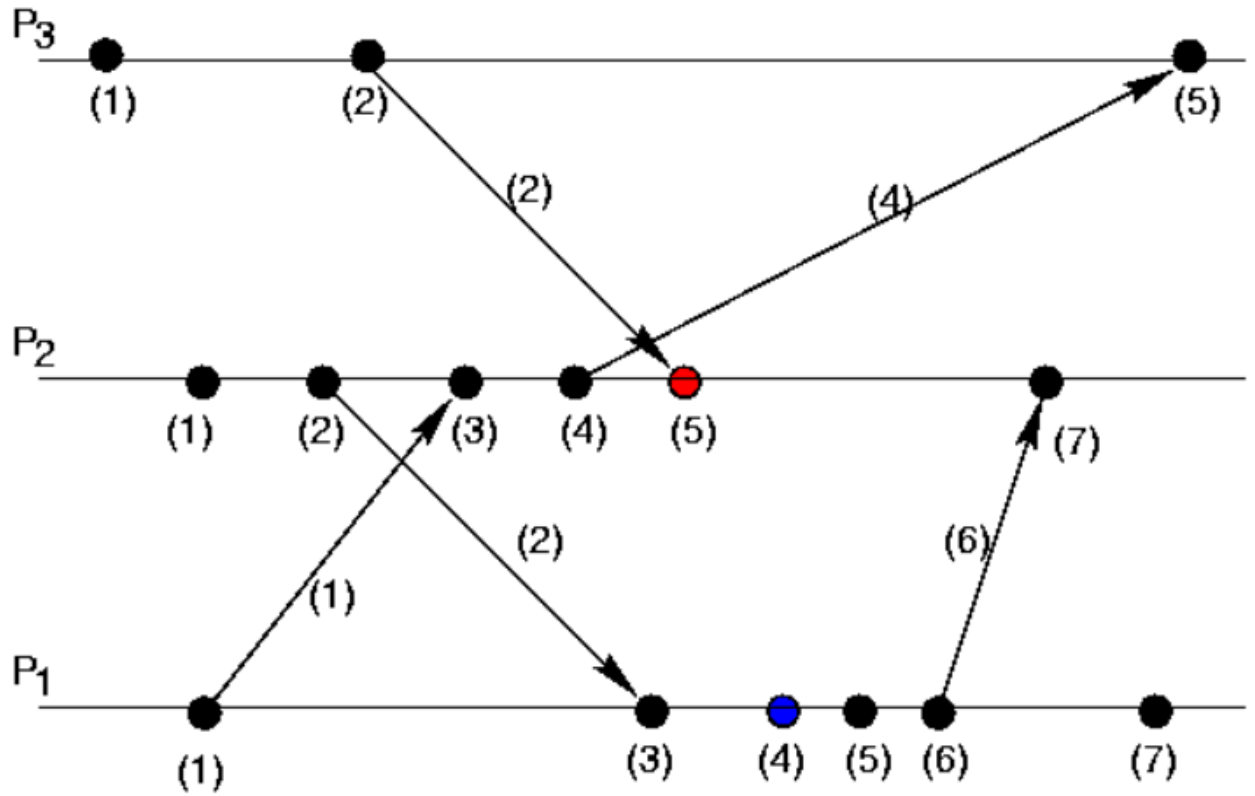
Pi timestamps event e with $Ci\ (e)\ i$

Then: $Ci\ (a)\ i$ happened before $Cj\ (b)\ j$ if and only if:

1: $Ci\ (a) < Cj\ (a)$; or

2: $Ci\ (a) = Cj\ (b)$ and $i < j$

Limitations of Lamport's logical clock



$C(a) < C(b)$ does not imply a ◆ b

Causal Ordering of Messages

- Message delivery is said to be causal if the order in which messages are received is consistent with the order in which they are sent. That is, if *Send*(M_1) ->*Send* ($M2$) then for every recipient of both messages, M_1 is received before M_2.

- Basic idea: Buffer each message until the message that immediately precedes it is delivered.
- It make use of vector clocks

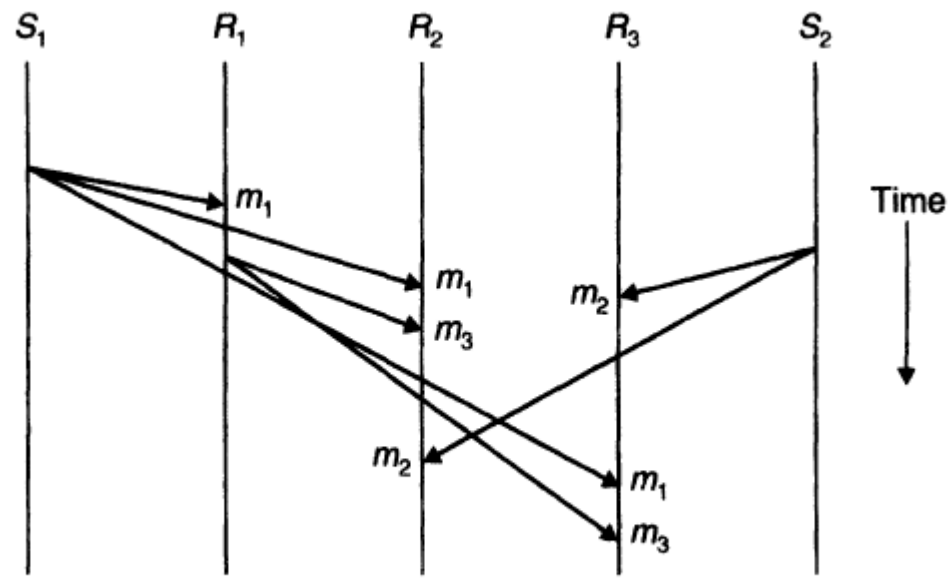


Fig. 3.17 Causal ordering of messages.

- In this example, sender SI sends message m1 to receivers *R1, R2 ,* and *R3* and sender *S2* sends message *m2* to receivers *R2 ,* and *R3 .*
- On receiving *m1 ,* receiver *R1* inspects it, creates a new message

m3, and sends *m3;* to *R2* and *R3 .*

- Note that the event of sending *m3* is causally related to the event of sending *m1,* because the contents of *m3* might have been derived in part from ml; hence the two messages must be delivered to both *R2* and *R3* in the proper order, *m1* before *m3.*
- Also note that since *m2* is not causally related to either *m1* or *m3 , m2* can be delivered at any time to *R2* and *R3* irrespective of *m,* or *m3.*

One method for implementing causal-ordering semantics is the *CBCAST protocol* of the ISIS system [Birman et al. 1991]. It works as follows:

1. Each member process of a group maintains a vector of n components, where n is the total number of members in the group. Each member is assigned a sequence number from 0 to n, and the ith component of the vectors corresponds to the member with sequence number i. In particular, the value of the ith component of a member's vector is equal to the number of the last message received in sequence by this member from member i.

2. To send a message, a process increments the value of its own component in its own vector and sends the vector as part of the message.

3. When the message arrives at a receiver process's site, it is buffered by the runtime system. The runtime system tests the two conditions given below to decide whether the message can be delivered to the user process or its delivery must be delayed to ensure causal-ordering semantics. Let S be the vector of the sender process that is attached to the message and R be the vector of the receiver process. Also let i be the sequence number of the sender process. Then the two conditions to be tested are

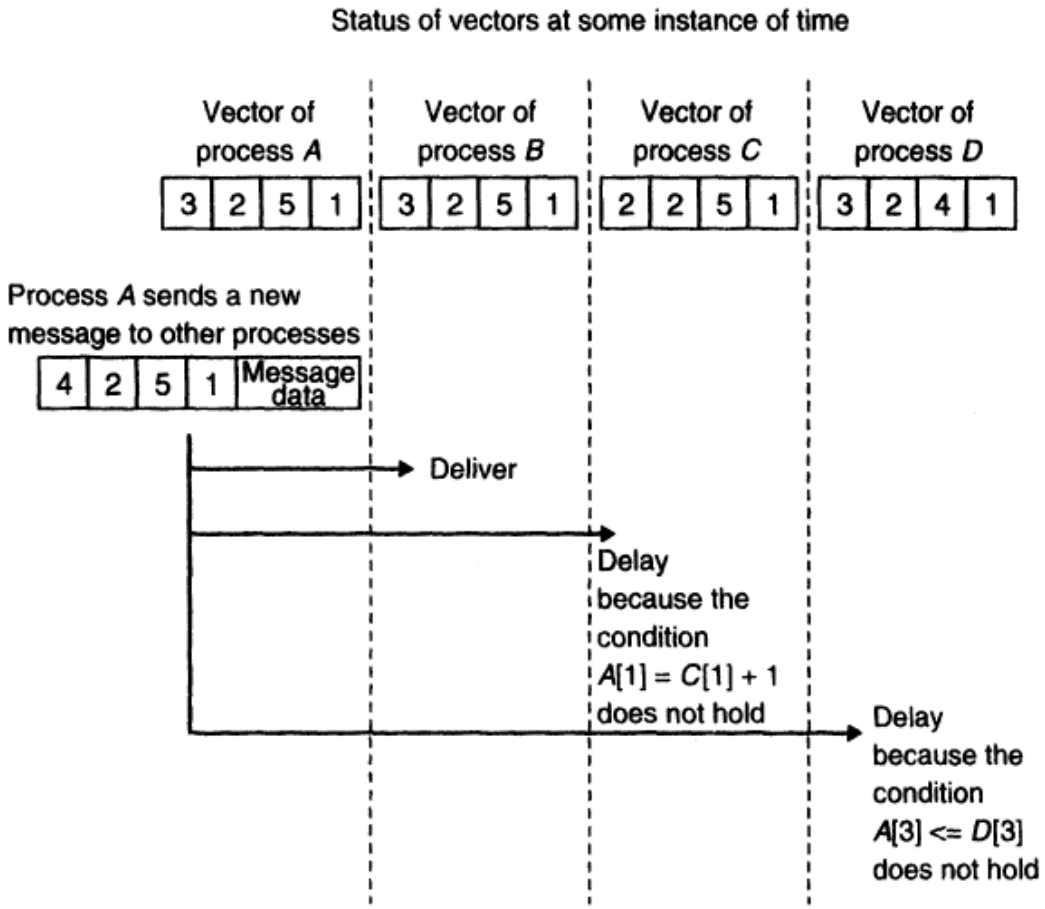
$$S[i] = R[i] + 1 \quad \text{and} \quad S[j] \leq R[j] \quad \text{for all } j \neq i$$

The first condition ensures that the receiver has not missed any message from the sender. This test is needed because two messages from the same sender are always causally related. The second condition ensures that the sender has not received any message that the receiver has not yet received. This test is needed to make sure that the sender's message is not causally related to a message missed by the receiver.

If the message passes these two tests, the runtime system delivers it to the user process. Otherwise, the message is left in the buffer and the test is carried out again for it when a new message arrives.

Example

(3,2,5,1) means that, until now, *A* has sent three messages, *B* has sent two messages, C
has sent five messages, and *D* has sent one message to other processes



Birman-Shiper-Stephenson Causal Message Ordering

- Before P_i broadcasts m, it increments $VT_{Pi}[i]$ and timestamps m. Thus $VT_{Pi}[i]$-1 is the number of messages from P_i preceding m.
- When P_j (j ≠ i) receives message m with timestamp VT_m from P_i, delivery is delayed locally until both of the following are satisfied:
  - $VT_{Pj}[i] = VT_m[i] - 1$
  - $VT_{Pj}[k] \leq VT_m[k]$ for all k ≠ i
    Delayed messages are queued at each process, sorted by their vector timestamps, with concurrent messages ordered by time of receipt.
- When m is delivered to P_j, VT_{Pj} is updated as usual for vector clocks.