OPERATING SYSTEMS

Process Management 9

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

OPERATING SYSTEMS

Course Syllabus - Unit 1

UNIT 1: Introduction and Process Management

Operating-System Structure & Operations, Kernel Data Structures,Computing Environments, Operating-System Services, OperatingSystem Design and Implementation. Process concept: Process in memory, Process State, Process Control Block, Process Creation and Termination, CPU Scheduling and Scheduling Algorithms, IPC - Shared Memory & Message Passing, Pipes - Named and Ordinary. Case Study: Linux/Windows Scheduling Policies.

OPERATING SYSTEMS

Course Outline

Class No.	Chapter Title / Reference Literature	Topics to be covered	% of Portions covered	
			Reference chapter	Cumulative
1	1.1-1.2	What Operating Systems Do, Computer-System Organization?	1	21.4
2	1.3,1.4,1.5	Computer-System Architecture, Operating-System Structure & Operations	1	
3	1.10,1.11	Kernel Data Structures, Computing Environments	1	
4	2.1,2.6	Operating-System Services, Operating System Design and Implementation	2	
5	3.1-3.3	Process concept: Process in memory, Process State, Process Control Block, Process Creation and Termination	3	
6	5.1-5.2	CPU Scheduling: Basic Concepts, Scheduling Criteria	5	
7	5.3	Scheduling Algorithms: First-Come, First-Served Scheduling, Shortest-Job-First Scheduling	5	
8	5.3	Scheduling Algorithms: Shortest-Job-First Scheduling (Pre-emptive), Priority Scheduling	5	
9	5.3	Round-Robin Scheduling, Multi-level Queue, Multi-Level Feedback Queue Scheduling	5	
10	5.5,5.6	Multiple-Processor Scheduling, Real-Time CPU Scheduling	5	
11	5.7	Case Study: Linux/Windows Scheduling Policies	5	
12	3.4,3.6.3	IPC – Shared Memory & Message Passing, Pipes – Named and Ordinary	3,6	

Topics Outline

- Interprocess Communication
  - Shared Memory
  - Message Passing
- Named and Unnamed Pipes

Basic Inter-Process Communication Concepts

- **Inter Process Communication (IPC)** is a mechanism that involves **communication** of one process with another process.
- This usually occurs only in **one** system and also be extended as communication between **two** process on two **different** hosts
- Communication can be of two types
  - Between **related** processes initiating from only one process, such as parent and child processes.
  - Between **unrelated** processes, or two or more different processes.

Basic Inter-Process Communication Concepts

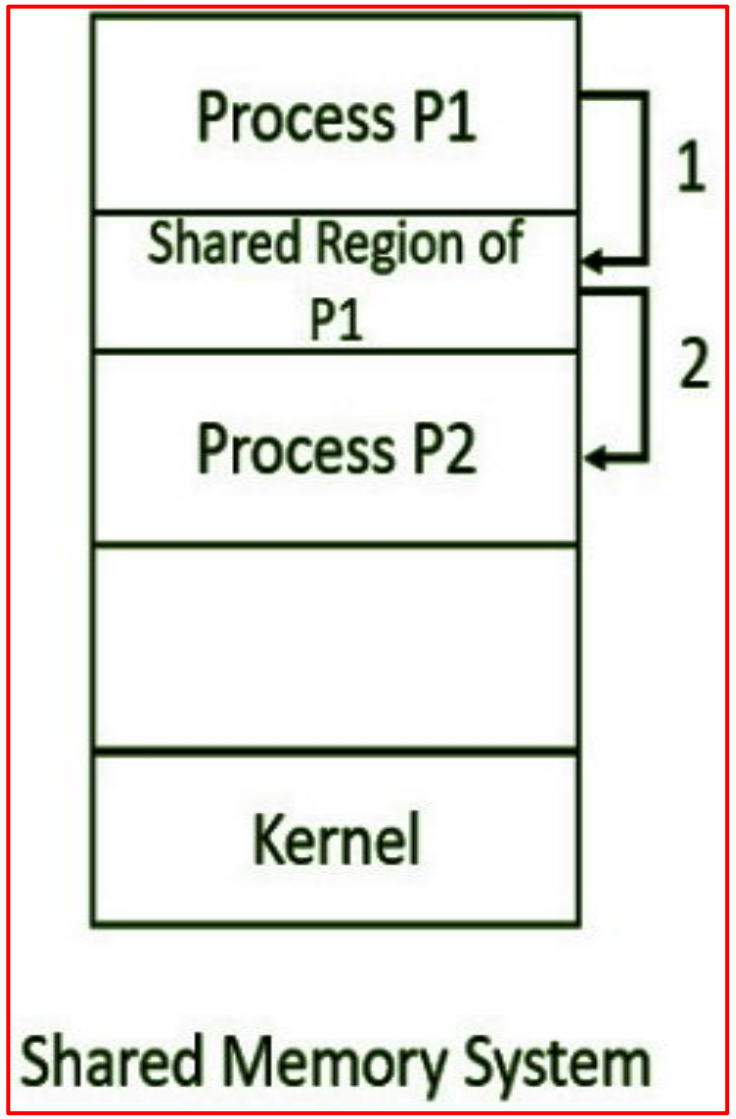
- There are several reasons for providing an environment that allows process cooperation:
  - **Information sharing:** Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
  - **Computation speedup:** If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.
  - **Modularity:** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads
  - **Convenience:** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.

Basic Inter-Process Communication Concepts

- There are two fundamental models of interprocess communication: **shared memory** and **message passing.**

- In the **shared-memory model**, a region of **memory** that is **shared** by cooperating processes is established.

    - Processes can then exchange information by **reading** and **writing** data to the shared region.



Shared-Memory Systems

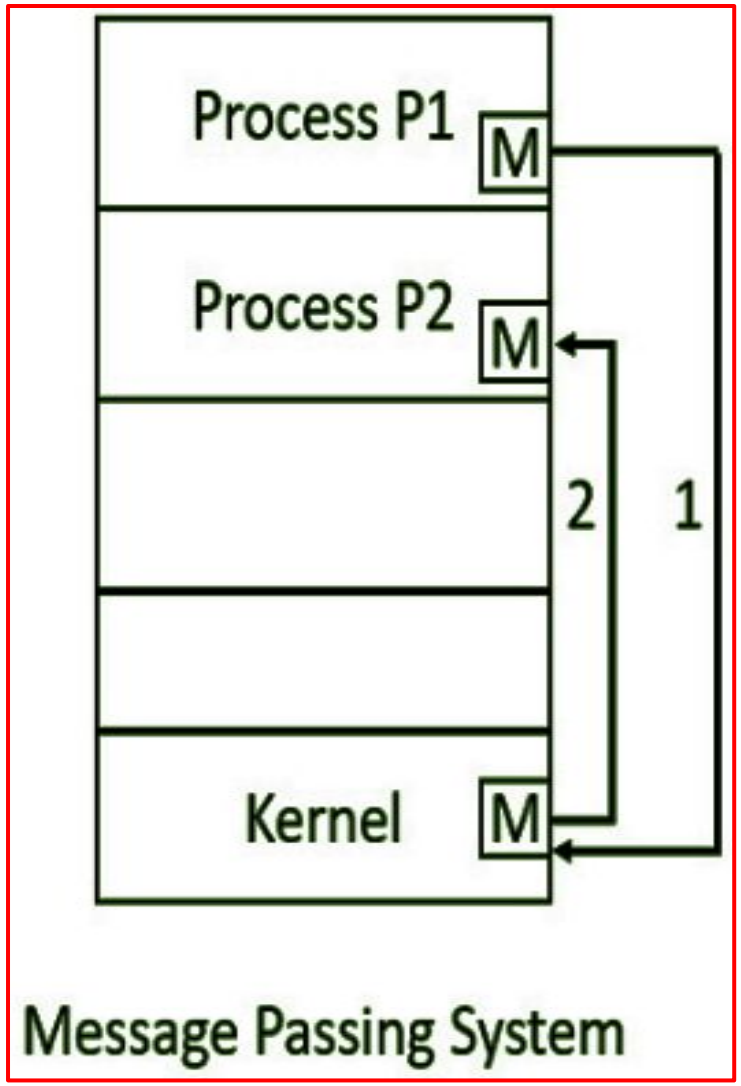
- A **producer** can produce one item while the **consumer** is consuming another item.
- The producer and consumer must be **synchronized**, so that the consumer does not try to consume an item that has not yet been produced.
- Two types of buffers can be used.
- The **unbounded buffer** places no practical limit on the size of the buffer.
    - The consumer may have to wait for new items, but the producer can always produce new items.
- The **bounded** buffer assumes a fixed buffer size.
    - In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.
- **Synchronization** among cooperating processes can be implemented effectively in a shared- memory environment.

```
item next_produced;

while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
       ; /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

The producer process using shared memory.

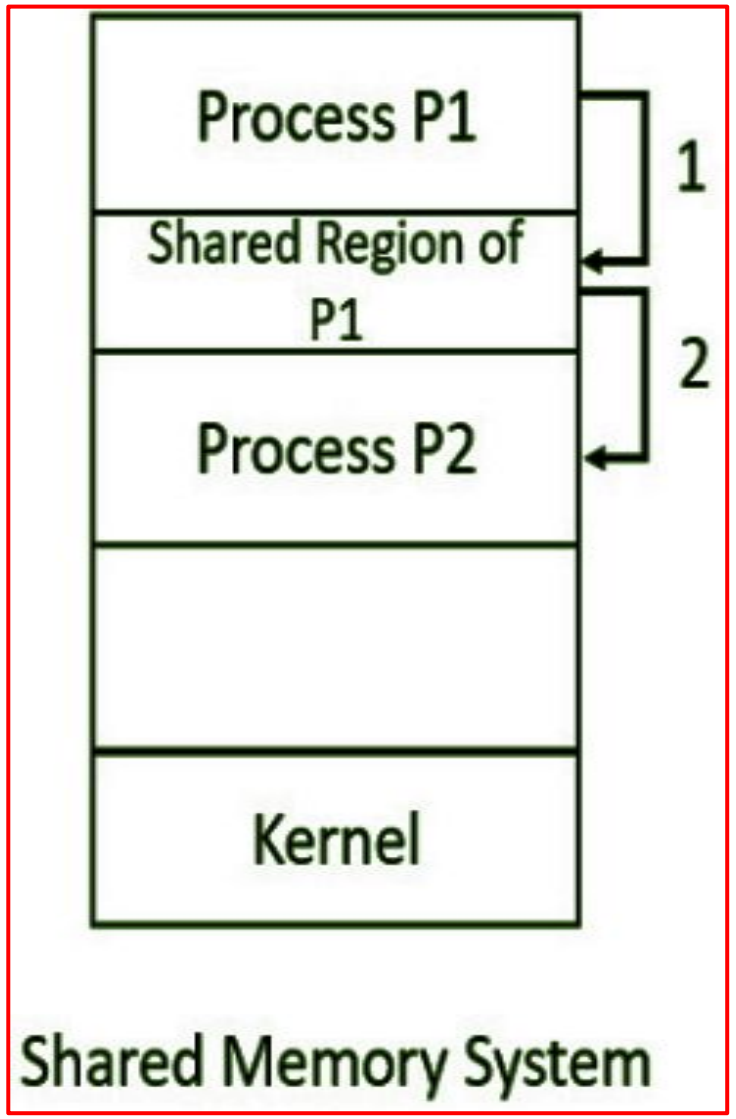
Basic Inter-Process Communication Concepts

- In the **message-passing** model, **communication** takes place by means of **messages exchanged** between the **cooperating processes**.
- **Message passing** is useful for exchanging smaller amounts of data, because no conflicts need be avoided.
- **Message passing** is also easier to implement in a distributed system than shared memory.



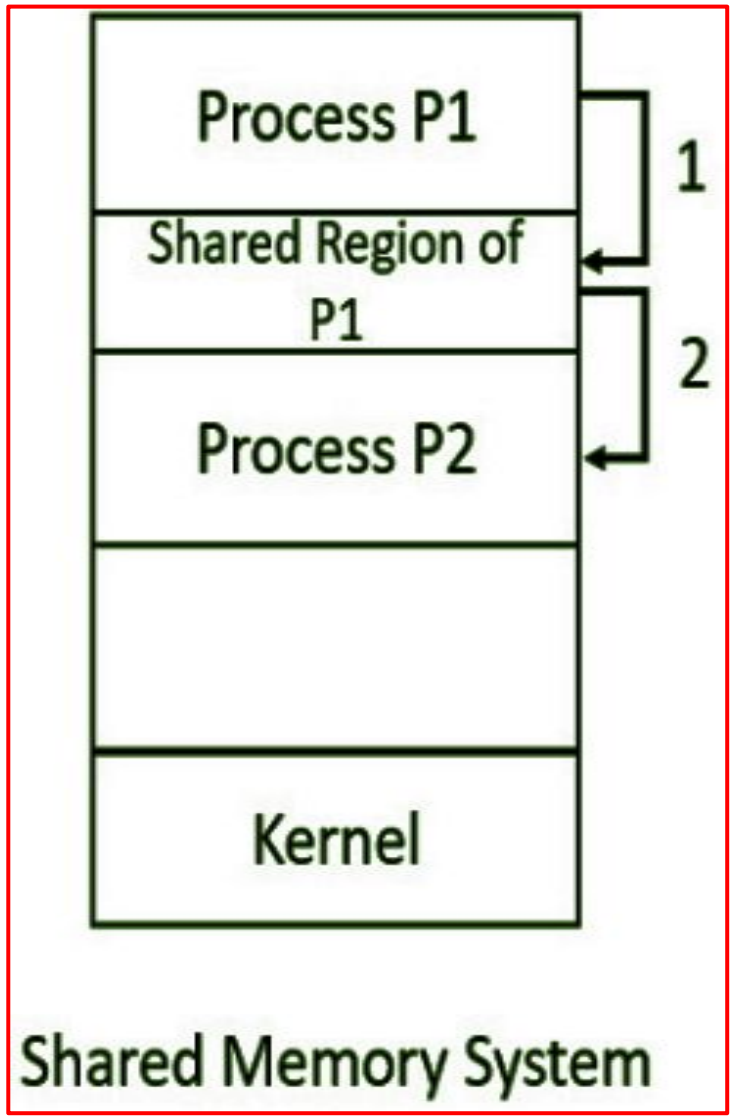
Basic Inter-Process Communication Concepts

- Shared memory can be **faster** than message passing, since message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention.
- In shared-memory systems, system calls are required only to establish shared-memory regions.
- Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.



Shared-Memory Systems

- Typically, a **shared-memory region** resides in the **address space** of the **process creating** the shared-memory segment.
- **Other** processes that wish to communicate using this shared-memory segment **must attach** it to their address space.
- Typically the operating system tries to prevent one process from accessing another process's memory.
- Shared memory requires that two or more processes **agree** to remove this restriction.
- They can then exchange information by reading and writing data in the shared areas.



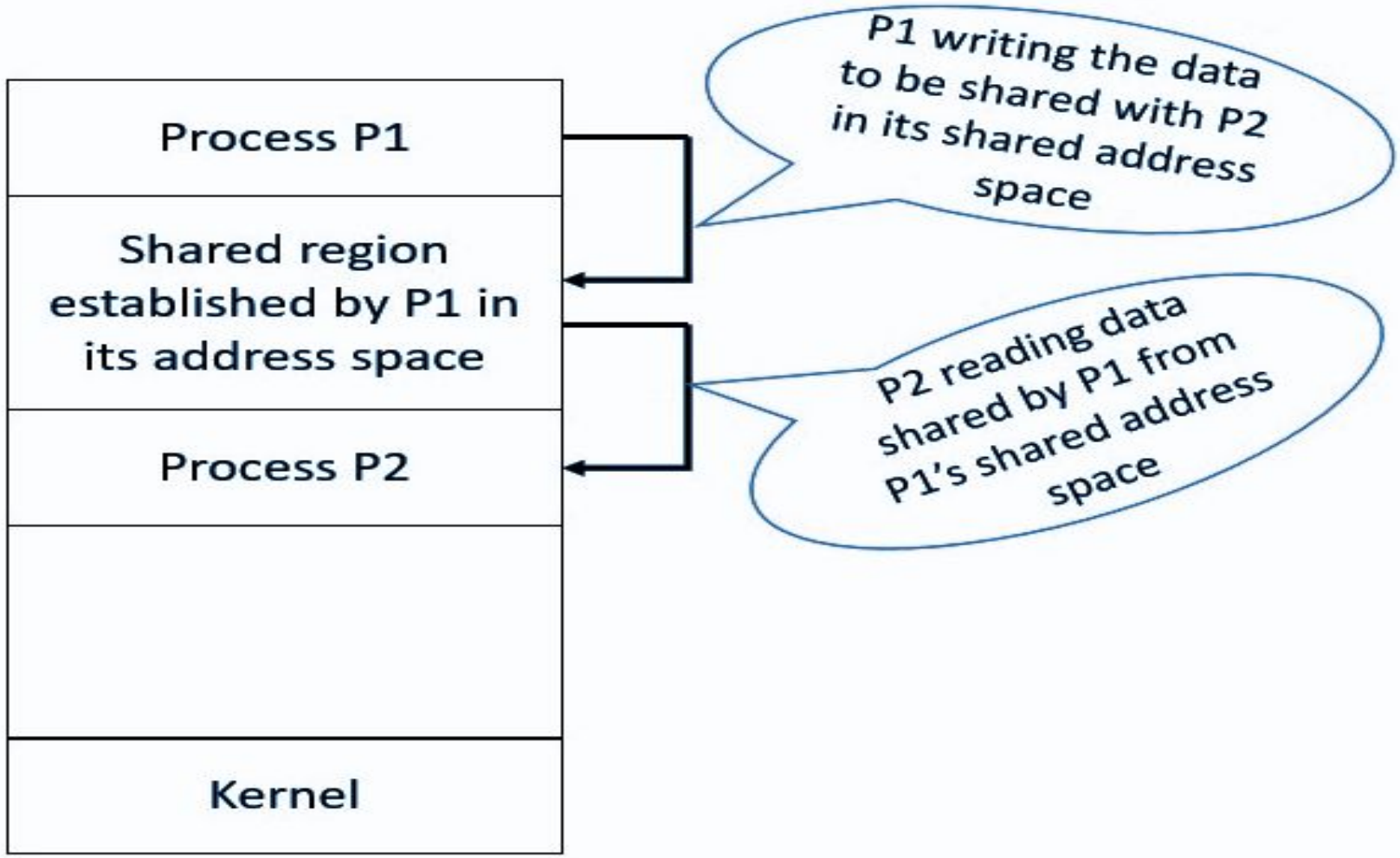
Shared-Memory Systems

- The form of the data and the location are determined by these processes and are not under the operating system's control.
- The **processes** are also **responsible** for ensuring that they are not writing to the same location simultaneously.
- A **producer** process produces information that is consumed by a **consumer** process.
- One solution to the **producer–consumer problem** uses shared memory.
- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.

```
item next_produced;

while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
       ; /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

The producer process using shared memory.

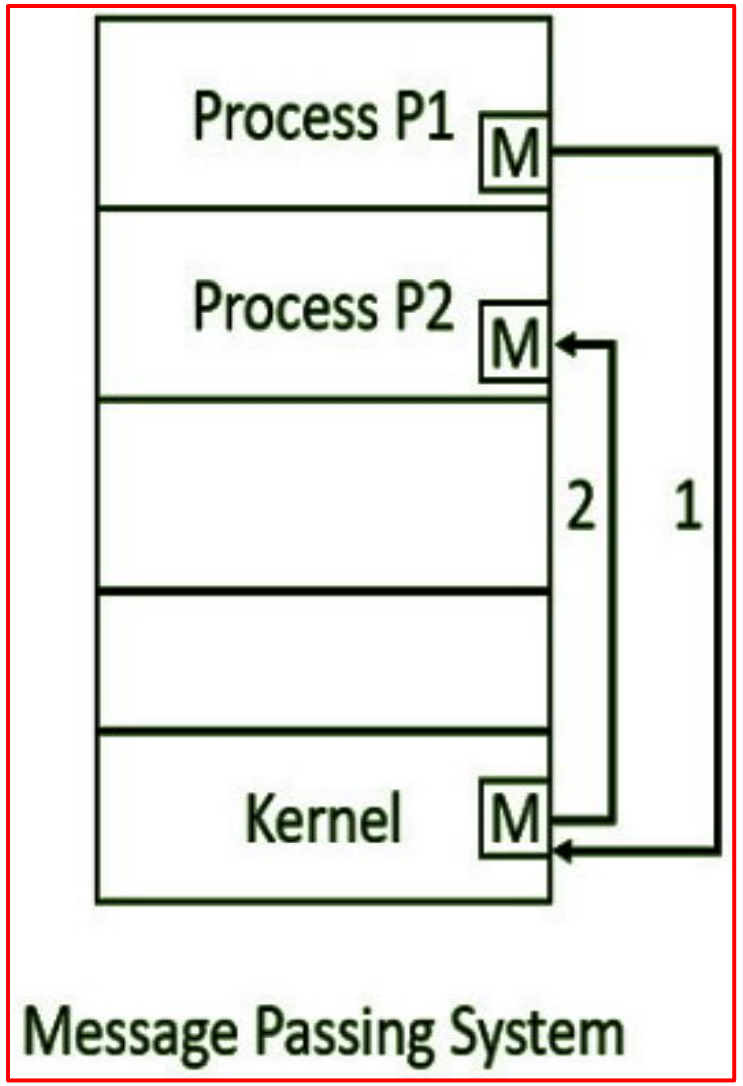
Shared-Memory Systems



Shared Memory System

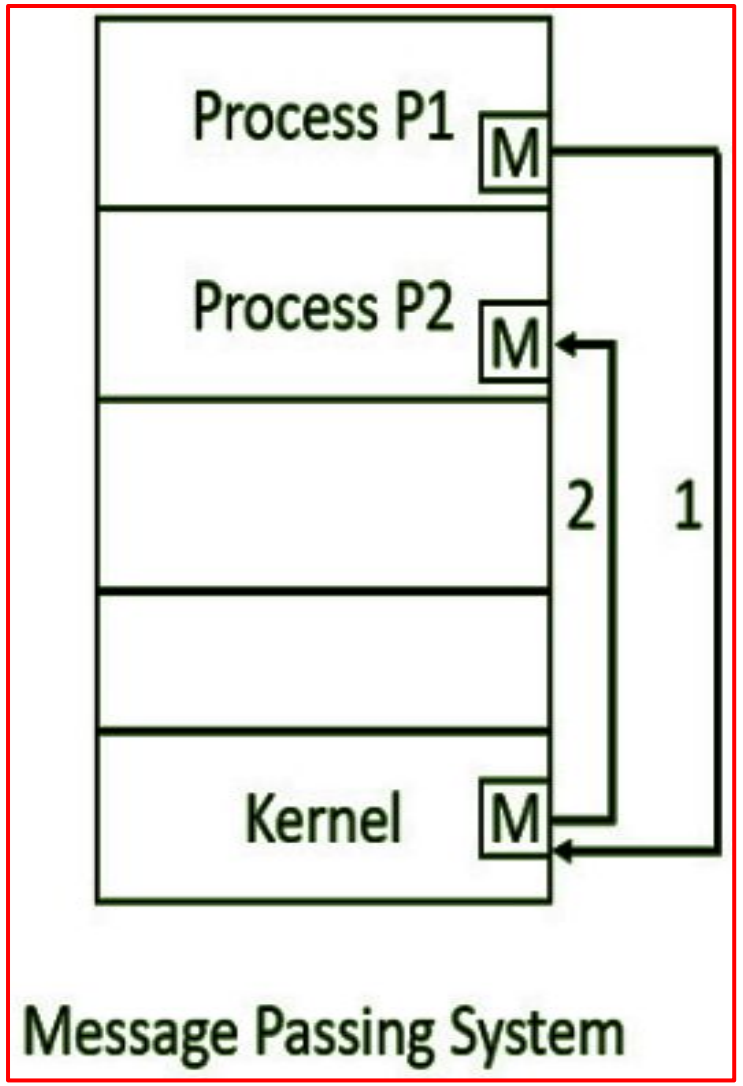
Message Passing Systems

- The **Operating System** to **provide** the means for cooperating processes to communicate with each other via a **message-passing** facility
- **Message passing** provides a mechanism to allow processes to communicate and to synchronize their actions **without sharing** the **same address space**.
- A message-passing facility provides at least two operations:
  - send(message)
  - receive(message)
- Messages sent by a process can be either **fixed** or **variable** in size.
- If only fixed-sized messages can be sent, the system-level implementation is straight forward.
- This restriction, however, makes the **task** of programming more **difficult**.



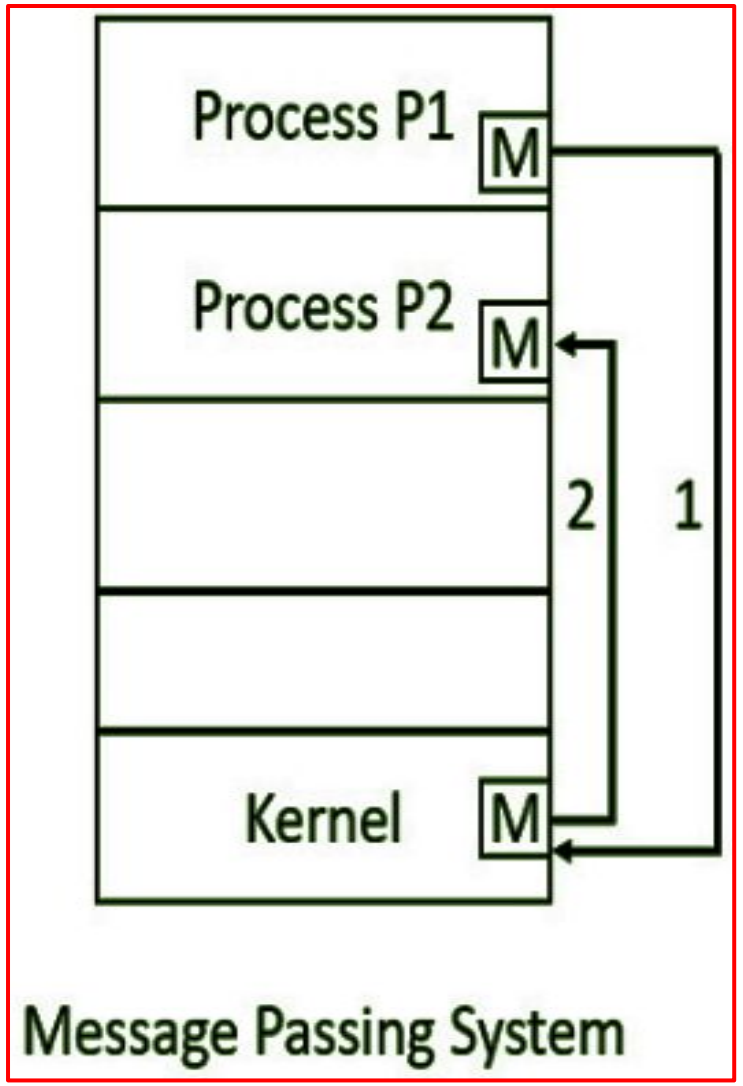
Message Passing Systems

- Variable-sized messages require a more **complex** system level **implementation**, but the **programming** task becomes **simpler**.
- This is a common kind of tradeoff seen throughout operating-system design.
- If processes P and Q want to communicate, they must send messages to and receive messages from each other: a **communication link** must exist between them.
- This link can be implemented in a variety of ways.
- We are concerned here **not** with the link's **physical** implementation such as shared memory, hardware bus, or network, but rather with its **logical** implementation.



Message Passing Systems

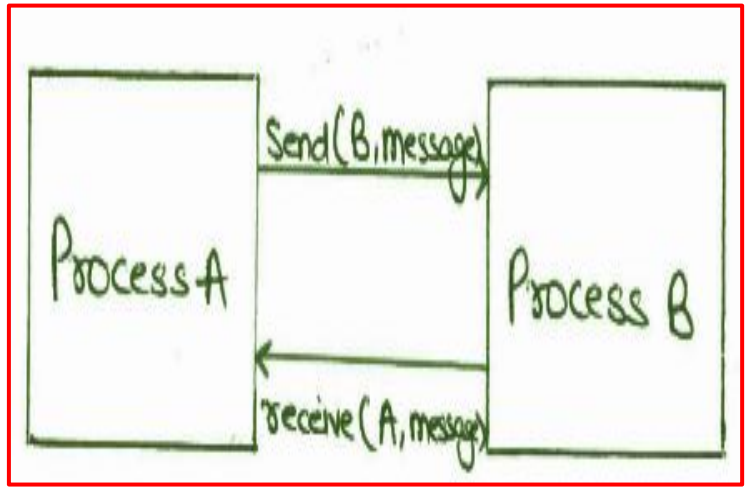
- Here are several methods for **logically** implementing a link and the send() and receive() operations:
  - **Direct** or **indirect** communication
  - **Synchronous** or **asynchronous** communication
  - **Automatic** or **explicit** buffering



Message Passing Systems - Direct Communication

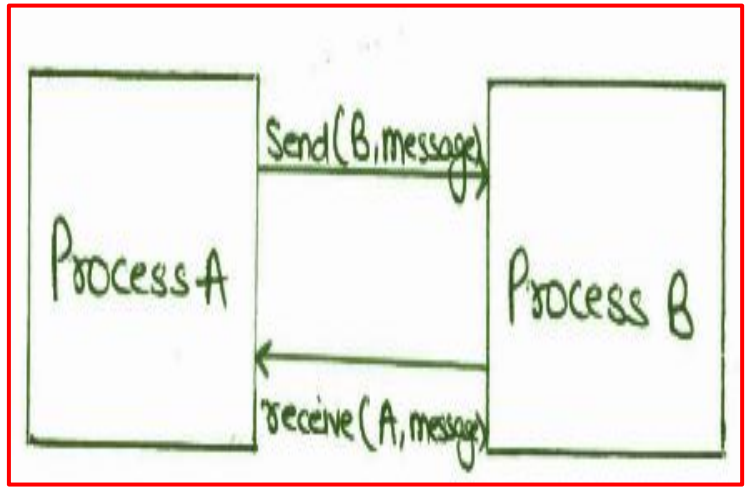
- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from process Q

- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional



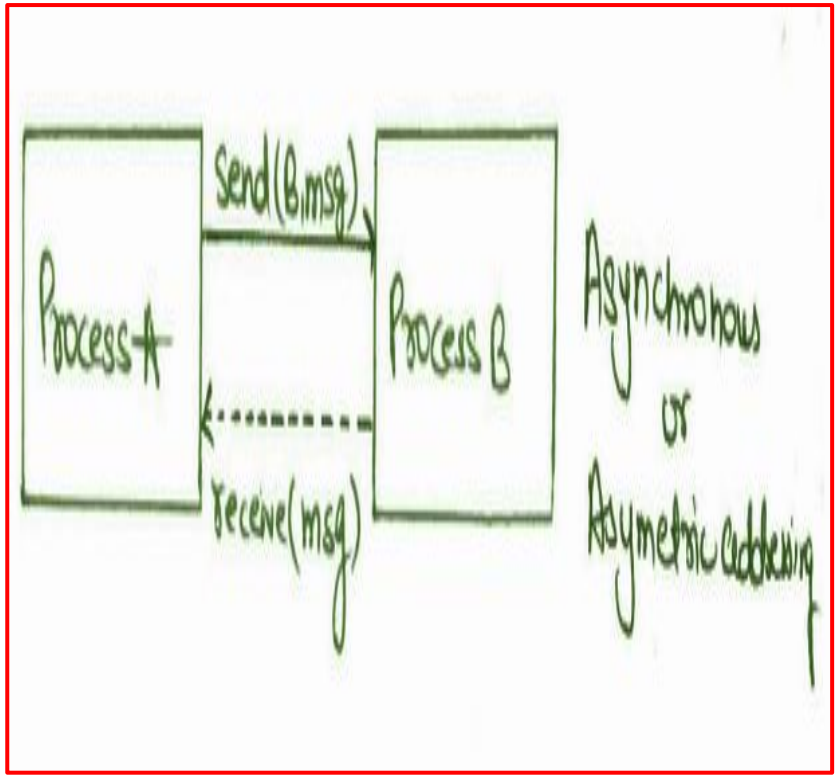
Message Passing Systems - Direct Communication - Symmetric Addressing

- In this type that two processes need to name other to communicate. This becomes easy if they have the same parent.
- If process A sends a message to process B, then send(B, message); Receive(A, message);
- By message passing a link is established between A and B.
- Here the receiver knows the Identity of sender message destination.
- This type of arrangement in direct communication is known as **Symmetric Addressing**.



Message Passing Systems - Direct Communication - ASymmetric Addressing

- Another type of addressing known as **asymmetric** addressing where receiver is **not aware** of the ID of the sending process in advance.



Message Passing Systems - Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox

- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Message Passing Systems - Indirect Communication

- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox

- Primitives are defined as:

  send(*A, message*) – send a message to mailbox A

  receive(*A, message*) – receive a message from mailbox A

Message Passing Systems - Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?

- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was

Message Passing Systems - Indirect Communication

- The following types of communication link are possible through mailbox.
  - **One to One link**: one sender wants to communicate with one receiver. Then single link is established.
  - **Many to One link:** Multiple Sender want to communicate with single receiver.Example in client server system, there are many client processes and one server process. The mailbox here is known as PORT.
  - **One to Many link:** One sender wants to communicate with multiple receiver, that is to broadcast message.
  - **Many to many:** Multiple sender want to communicate with multiple receivers.

- Message passing may be either **blocking** or **non-blocking**.
- **Blocking** is considered **synchronous**
- **Non-blocking** is considered **asynchronous**
- **send** and **receive** primitives may be either **blocking** or **non-blocking**

Message Passing Systems - Indirect Communication - Synchronisation

- **Synchronous** receive
  - Receiving process blocks until message is copied into user-level buffer by the sender

- **Asynchronous** receive
  - Receiving process issues a receive operation (specifying a buffer) and then carries on with other tasks
  - It either polls the OS to find out if the receive has completed or gets an interrupt when the receive has completed
  - Threads (discussed later) allow you to program an asynchronous receive in a synchronous way
  - Issue a synchronous receive with one thread while carrying out other tasks with other threads

Message Passing Systems - Indirect Communication - Synchronisation

- OS view vs Programming Languages (PL) view of synchronous communication

- OS view
  - synchronous send ⇒sender blocks until message has been copied from application buffers to kernel buffer
  - Asynchronous send ⇒sender continues processing after notifying OS of the buffer in which the message is stored; have to be careful to not overwrite buffer until it is safe to do so
- PL view:
  - synchronous send ⇒sender blocks until message has been received by the receiver
  - asynchronous send ⇒sender carries on with other tasks after sending message (OS view of synchronous communication is asynchronous from the PL viewpoint)

Message Passing Systems - Indirect Communication

- 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**

Message Passing Systems - Buffering

- Queue of messages attached to the link
- Implemented in one of three ways.
  - **Zero capacity** – 0 messages Sender must wait for receiver (rendezvous).
  - **Bounded capacity** – finite length of n messages or N bytes. Sender must wait if link full.
  - **Unbounded Capacity** - Infinite length of n messages or N bytes. Sender never waits

Inter-Process Communication (IPC) OS dependent Mechanisms

Pipes – Communication between two related processes. The mechanism is half duplex meaning the first process communicates with the second process. To achieve a full duplex i.e., for the second process to communicate with the first process another pipe is required.

FIFO – Communication between two unrelated processes. FIFO is a full duplex, meaning the first process can communicate with the second process and vice versa at the same time.

Message Queues – Communication between two or more processes with full duplex capacity. The processes will communicate with each other by posting a message and retrieving it out of the queue. Once retrieved, the message is no longer available in the queue.

Shared Memory – Communication between two or more processes is achieved through a shared piece of memory among all processes. The shared memory needs to be protected from each other by synchronizing access to all the processes.

Semaphores – Semaphores are meant for synchronizing access to multiple processes. When one process wants to access the memory (for reading or writing), it needs to be locked (or protected) and released when the access is removed. This needs to be repeated by all the processes to secure data.

Signals – Signal is a mechanism to communication between multiple processes by way of signaling. This means a source process will send a signal (recognized by number) and the destination process will handle it accordingly.

Inter-Process Communication (IPC) - Pipes

- **Pipe** is a communication medium between two or more related or interrelated processes.
- It can be either within **one** process or a communication between the **child** and the **parent** processes.
- Communication can also be multi-level such as communication between the **parent**, the **child** and the **child of a child**, etc.
- **Communication** is achieved by one process **writing** into the **pipe** and other **reading** from the pipe.
- To achieve the pipe system call, create two files, one to write into the file and another to read from the file.

Inter-Process Communication (IPC) - Pipes

- Pipe mechanism can be viewed with a real-time scenario such as filling petrol with the pipe into some vehicle, say a petrol tank, and someone retrieving it, say with a nozzle.

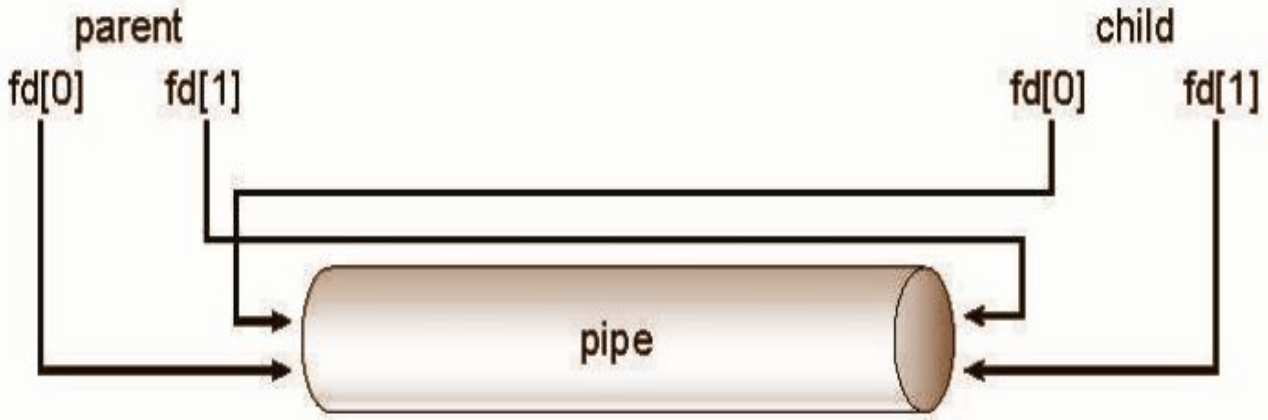
- The filling process is nothing but writing into the pipe and the reading process is nothing but retrieving from the pipe. This implies that one output (petrol) is input for the other (tank).

Inter-Process Communication (IPC) - Pipes

- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?
- Ordinary pipes – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- Named pipes – can be accessed without a parent-child relationship

Inter-Process Communication (IPC) - Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes



- Windows calls these **anonymous pipes**

Topics Uncovered in this Session

- Interprocess Communication
  - Shared Memory
  - Message Passing
- Named and Unnamed Pipes

THANK YOU

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

nitin.pujari@pes.edu

For Course Deliverables by the Anchor Faculty click on www.pesuacademy.com