Operating Systems Fundamentals

Process Communications

Process Classification

Processes executing concurrently in the OS may be either Independent processes or Cooperating processes.

- **Independent processes** are processes which run and operate without interacting or sharing data with other processes that are running
- **Co-Operating processes** are processes which exchange information to perform computations and operations
- We are concerned in this lecture with Co-Operating processes
- A Cooperating process can affect or be affected by other processes
  - Any process that shares data with other processes are cooperating processes.

Cooperating Processes

- **Reasons for /Advantages of cooperating processes:**
  - Information sharing:
    Cooperating processes can share information directly through shared memory or indirectly through IPC (Inter-Process Communication) mechanisms like messages, pipes, and sockets. This capability enables applications that require multiple processes to access and manipulate the same set of data or for tasks that are naturally divided into subtasks, improving data coherence and consistency.
  - Computation speedup:
    By dividing a task into multiple processes that can run concurrently on multiple CPUs or cores, cooperating processes can reduce the time to complete the task. This parallel execution takes advantage of multi-processor or multi-core systems, leading to a more efficient use of computational resources and faster application performance.

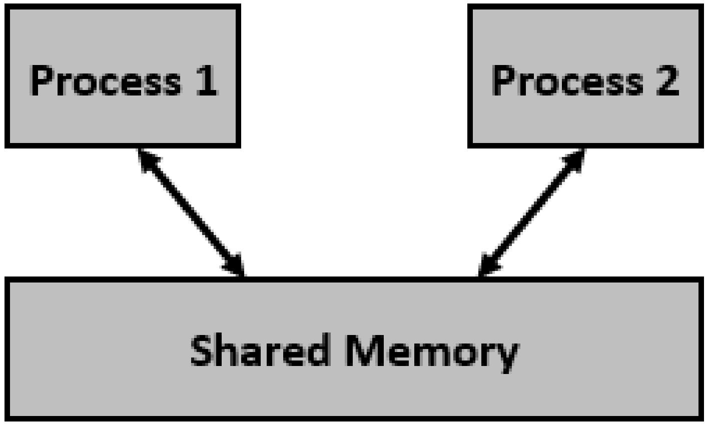
Cooperating Processes

- **Reasons for /Advantages of cooperating processes:**
  - <u>Modularity</u>:
    Cooperating processes allow for a modular approach to software development, where different modules or components of a larger application can be developed independently as separate processes. This modularity facilitates easier development, debugging, maintenance, and understanding of complex systems. Each module can be designed to perform a specific function and communicate with other modules as needed.
  - <u>Convenience</u>:
    Certain problems are inherently easier to solve when multiple cooperating processes are used. For example, serving multiple clients in a networked application can be more efficiently handled through a process per client model or a utility may invoke multiple components, which interconnect via a pipe structure that attaches the stdout of one stage to stdin of the next, etc This approach simplifies the design of the application and can improve responsiveness and service quality.

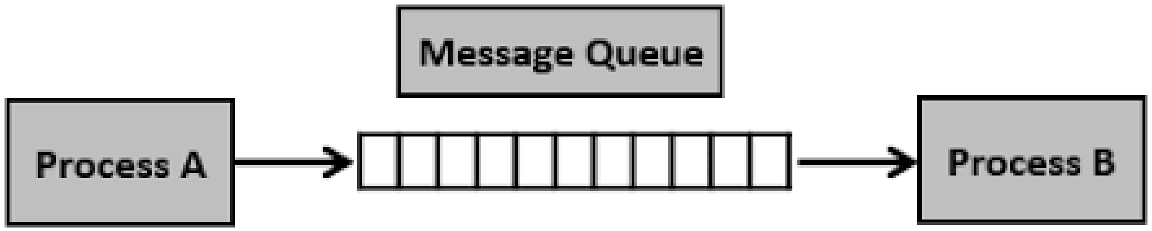
Cooperating Processes

- Cooperating processes need **Inter-Process Communication (IPC)**

- **Two models of IPC**
  - Shared memory:



  - Message passing:



Producer-Consumer Problem

- **The Producer-Consumer model is an example of Shared Memory IPC**

- **The Producer-Consumer Problem** is a paradigm for cooperating processes; the *producer* process produces information that is consumed by a *consumer* process

- **Producer – Consumer model** is similar to the **client – server concept.**

Example: multiple applications printing files (print spooling).

One process, the web server is **producing** (or providing) data (files and images), which are **consumed** (or “read”) by the client web browser that is requesting the resource.

Producer-Consumer Problem

- **The Producer-Consumer Problem** is an example of a multi-process synchronization issue, used to illustrate the need for coordinated access to a shared resource, typically a buffer or storage area, in concurrent programming environments.
- There are two types of processes: producers and consumers.
- Producers are responsible for generating data or retrieving it from an external source, and then placing it into a shared buffer.
- Consumers take this data from the shared buffer for further use, processing, or consumption.
- The shared buffer can be of a fixed size, and it is crucial that this size is not exceeded (overflow) by producers when it is full or underflowed by consumers when it is empty.

Producer-Consumer Problem

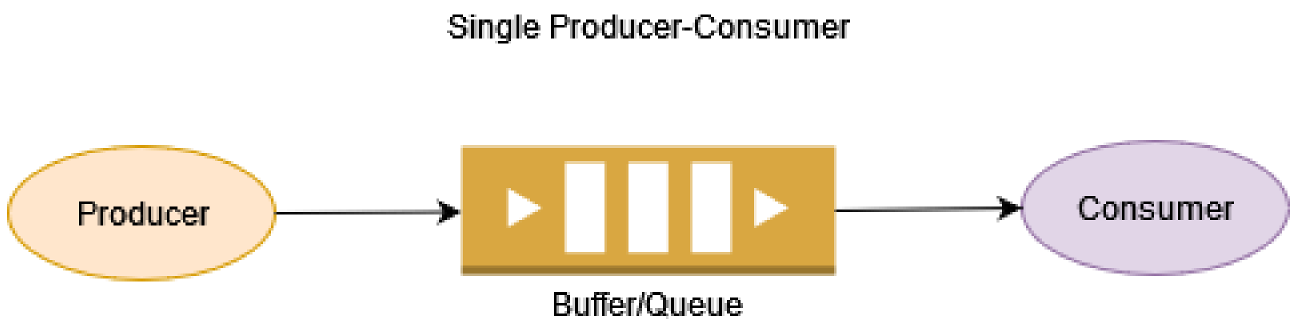
Key aspects of the Producer-Consumer problem include:

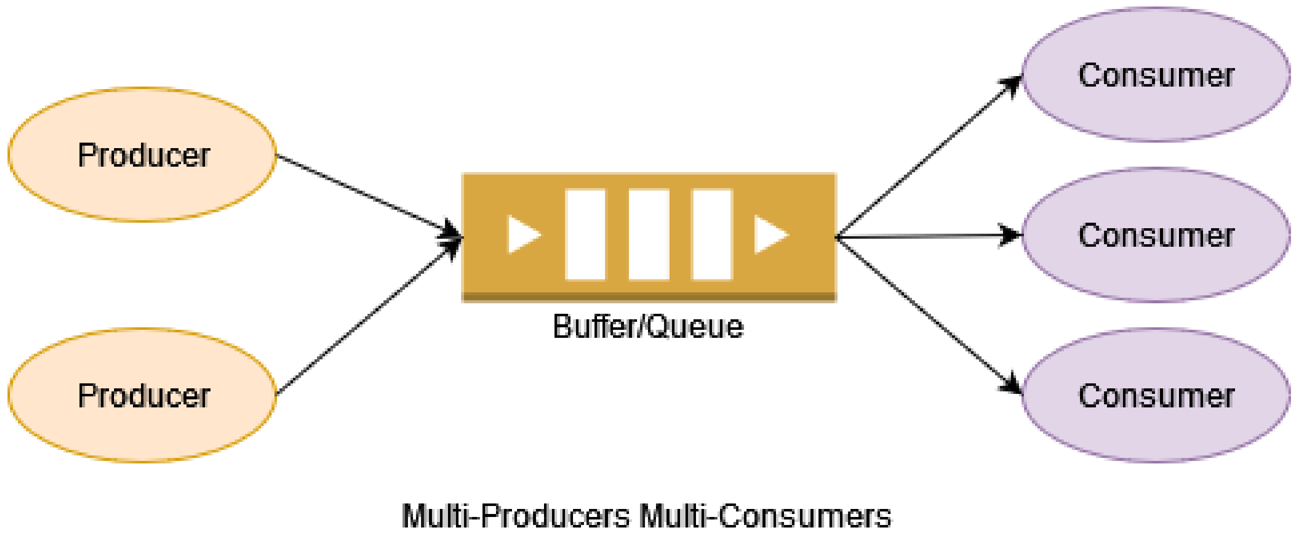
- Mutual Exclusion: Ensuring that only one (or more if specified) producer or consumer accesses the buffer at any given time to prevent data corruption or inconsistent read/writes.

- Synchronization: Coordinating the access to the buffer so that consumers do not attempt to consume data that has not yet been produced (underflow) and producers do not produce more data than the buffer can hold, leading to overflow.

- Communication: Often involves signaling between producers and consumers to indicate when the buffer is full or empty, enabling consumers to wait when there is nothing to consume and producers to wait when the buffer is full until space becomes available.

Producer-Consumer Problem





Producer – Consumer, Buffers

- There are **limitations** on how much data can be exchanged in the Producer – Consumer model

- Data that is exchanged may need to be stored after it is produced but before it is consumed. Why?
  - If data is produced too fast, we need a means to store / slow down the exchange of data

- The producer and consumer must be synchronised, so that the consumer does not try to consume an item that has not yet been produced.

- We need a **buffer** for items that can be **filled by the producer** and **emptied by the consumer** – this will allow producer and consumer processes to run concurrently
- For Example:
  - Think of online videos and the need for a certain amount to be downloaded before they start playing, this is the idea of buffers / buffering.

Buffers for cooperating Processes

- Buffers may be thought of as **arrays for storing data** - they store the data that is being exchanged between the co-operating processes

- Two buffer approaches are used
  - **Unbounded-buffer:** places no practical limit on the size of the buffer. This is mainly an ideal, unrealisable approach – consumer may have to wait but producer can always produce.

  - **Bounded-buffer** assumes that there is 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.
    - Bounded Buffer is more common as there is a physical limit to the amount of RAM in a computer

- **In both cases, the buffer is <u>shared memory</u> – shared by both processes**

Inter-process Communication – Message Passing

- The Producer-Consumer Model shows how cooperating processes communicate in a shared-memory environment – i.e. by sharing a common buffer pool

- Another way to achieve the same effect is for the operating system to provide the means for cooperating processes to cooperate with each other via an inter-process-communication facility

  - This approach is called the **Message-passing** model

The **Message-passing** model

The **Message-passing** model:

- Message passing provides a mechanism to allow processes to communicate and to synchronise their actions **without sharing the same address space**, and is particularly useful in a distributed environment, where the communicating processes may reside on 2 different computers.

- For example:

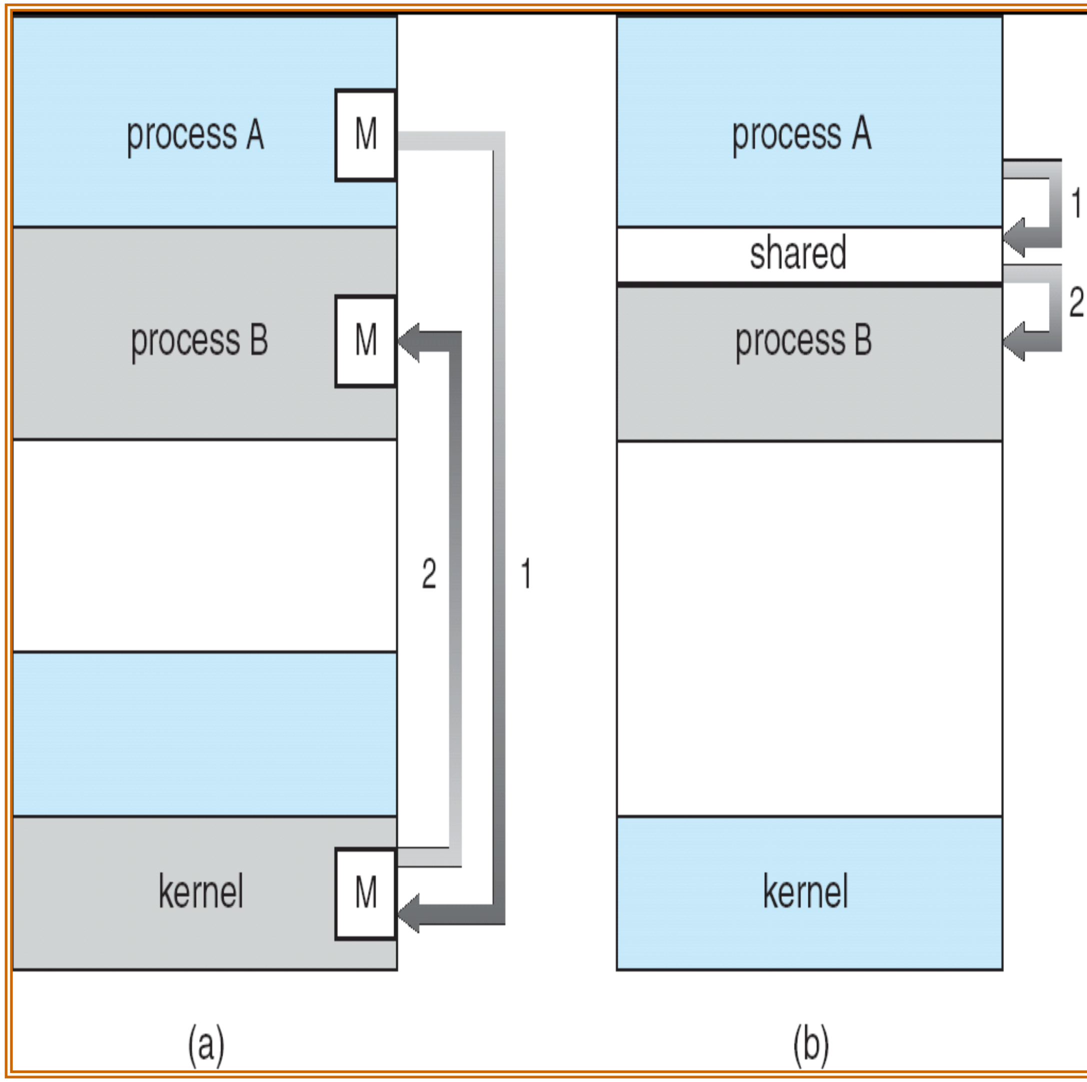
  A chat program use on the web could be designed so that participants communicate with one another by exchanging messages.

Inter-Process Communication: 2 Approaches

Mechanisms for processes to communicate and to synchronize their actions

- **(a) Message Passing approach.** Here process A sends a message to process B

- **(b) Shared Memory approach.** A region of shared memory is established. Processes can then exchange information by reading and writing data to the shared region. The producer-consumer model illustrates the shared-memory approach.



IPC Message Passing

Naming: Direct vs Indirect Communication

If processes P and Q want to communicate, a **communication link** must exist between them. Processes must have a way to refer to each other. They can use either **direct** or **indirect** communication.:

Direct Communication

- A message passing facility provides at least 2 operations – to **send a message** and to **receive a message.**
- Communications occur using process IDs.
- Processes must name each other explicitly as part of the Message:
  - **Send** (*P*$_{pid}$*, message*) – send a message to process P
  - **Receive** (*Q*$_{pid}$*, message*) – receive a message from process Q

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.

Direct vs. Indirect

Think of the differences this way!

- **Direct -** go directly to a person and tell them the message you want them to hear
- **Indirect** - the same as posting a letter / email and the person receives it when they are there / able to receive it

Message Passing: Synchronization

- Message passing may be either **blocking** or **non-blocking,**
  - **Blocking** causes either the sender or the receiver to halt at that point until the communication is completed
  - **Non-Blocking** allows the sender or the receiver to proceed but care must be taken to test the communication has occurred
    - Producer can still perform computations

Blocking Synchronisation

- **Blocking** is considered **synchronous**
  - **Blocking send/Blocking receive** Both the sender and the receiver are blocked until the message is delivered –sometimes called a *rendezvous. Allows for tight synchronization between processes.*

- Considered **synchronous** as <u>both sides must work together to perform the message exchange</u>
  - Think of telling someone a story in person.
  - You both need to be present before the story will be told/exchanged.

- **Blocking receive** has the receiver block until a message is available.

Non-blocking Synchronisation

- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send**, the sender sends the message and continues
  - **Non-blocking receive**, the receiver receives a valid message or may receive nothing.

- Considered asynchronous because data can be exchanged regardless of what the other is doing
  - Think of leaving a voice mail or text message on somebody’s phone
  - They receive and interact with it when they are ready while you are able to keep doing things that you need to do

- **Nonblocking send / nonblocking receive** – Neither party is required to wait.

Combinations..

- **Nonblocking send, blocking receive** – Although the sender can continue on, the receiver is blocked until the requested message arrives.

  This is probably the most useful combination as it allows:

  – a process to send one or more messages to a variety of destinations as quickly as possible (for example: printing files).

    - One potential danger of the nonblocking send is that an error could lead to a situation in which a process repeatedly generates messages. Because there is no blocking to discipline the process, these messages could consume system resources including processor time and buffer space, to the detriment of other processes and the OS

  – A process that must receive a message before it can do useful work needs to be blocked until such a message arrives.

    - However, if a message is lost, which can happen in a distributed system, or if a process fails before it sends an anticipated message, a receiving message could be blocked indefinitely

Communications and Buffers

- As we know, buffers support the exchange of data between processes
  - Whether the communication is direct or indirect, messages are exchanged by communicating processes residing in a temporary queue (Buffer).

- Such a queue can be implemented in three ways.

Three Implementations of buffers

1. **Zero capacity** – 0 capacity in queue/buffer

   Sender must wait for receiver (rendezvous) to receive msg

2. **Bounded capacity** – finite length of *n* messages in buffer.

   Sender must wait if buffer full until space becomes available.

3. **Unbounded capacity** – infinite length in buffer.

   Sender never waits (because sender never blocks)

The zero-capacity case is sometimes referred to as a message system with no buffering; the other cases are referred to as automatic buffering.

Summary: Inter-Process Communication (IPC)

- Cooperating processes (Producers and consumers) require an inter-process mechanism that will allow them to exchange data and information.

- The 2 fundamental models of Inter-Process Communication (IPC) are **Shared memory** and **Message Passing**.

  - **Shared Memory-** the producer consumer paradigm illustrates this

  - **Message Passing**
    - A message passing facility provides a ***send*** operation and a ***receive*** operation.
    - 2 processes wishing to communicate need a **communication link** – can be implemented as **direct** or **indirect**,
    - Message passing can be **blocking/synchronous** or **nonblocking/asynchronous** e.g. ***nonblocking send/blocking receive***
    - Messages exchanged can reside in a **temporary queue** of which there are 3 types: **zero capacity, bounded capacity** and **unbounded capacity**.