CSE308 Operating Systems

Inter-Process Communication (IPC)

Dr S.Rajarajan
SASTRA

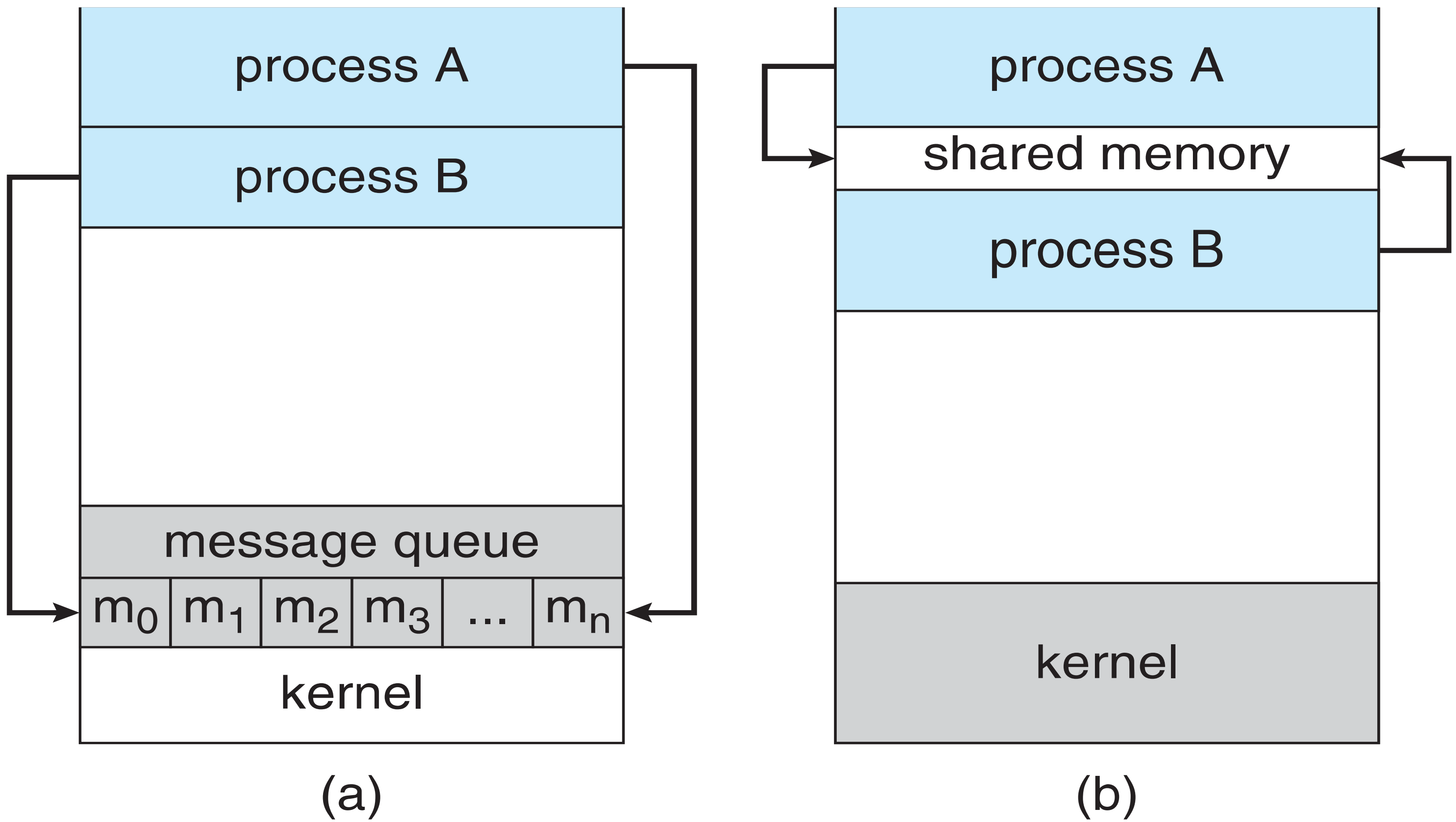
Interprocess Communication

- Processes within a system may be
  - ***independent***
  - or ***cooperating***
- Cooperating process can affect or be affected by other processes, including by sharing data
- Reasons for cooperating processes:
  - **Information sharing** - shared file
  - **Computation speedup** – break tasks into subtasks
  - **Modularity** – dividing the system functions into separate processes
  - **Convenience** – user may do several tasks at a time

- Cooperating processes have to communicate with each others.
- Two models of IPC
  - **Shared memory**
  - **Message passing**

(a) Message passing.

(b) shared memory.



Two IPC methods

- 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.**
- In the message-passing model, communication takes place by **means of messages exchanged** between the cooperating processes.
- Both of the models just mentioned are **common in operating systems**, and many systems implement **both**.
- Once shared memory is established, all accesses are treated as routine memory accesses, and **no assistance from the kernel** is required.

Shared-Memory Systems

- **Shared-Memory Systems**
  - Communicating processes need to establish **a region of shared memory.**
  - 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**.
  - They can then exchange information by **reading and writing** data in the **shared areas.**
  - 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

Message passing

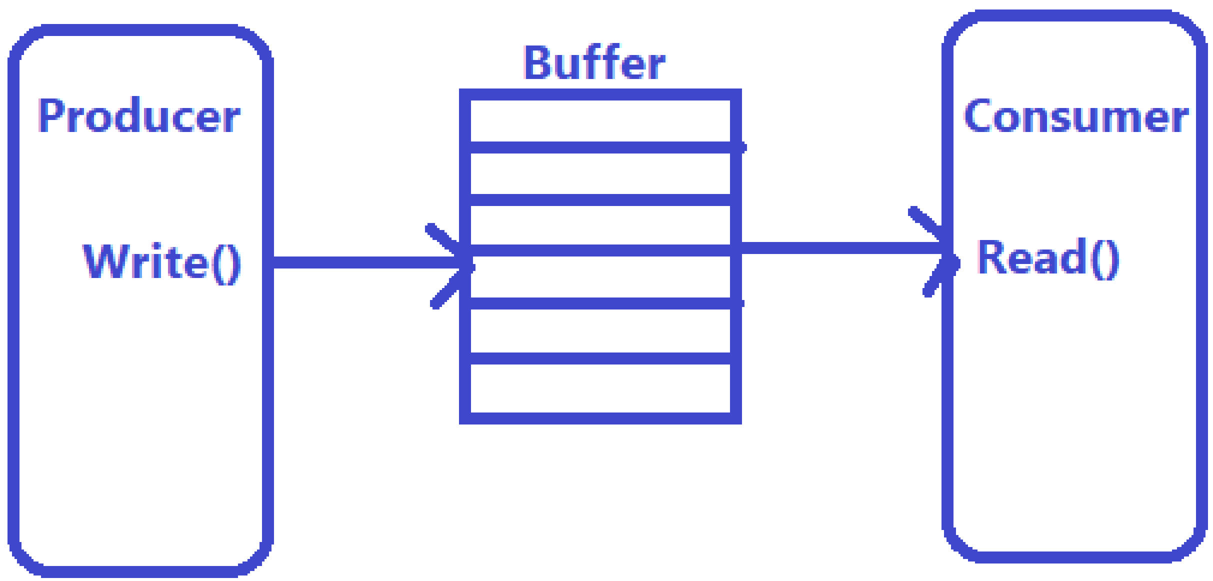
- **Message passing**
  - Message passing is useful for exchanging **smaller amounts of data**, because no conflicts need be avoided.
  - Message passing is also easier to implement in a ***distributed system*** than shared memory.
  - Shared memory can be **faster than message passing**, since **message-passing systems** are typically implemented using **system calls** and thus require more time-consuming task of **kernel intervention.**
  - In **shared-memory** systems, system calls are required **only to establish shared memory** regions.

Multi-core Systems

- Recent research on **multi-core systems indicates** that **message passing provides better performance** than shared memory on such systems.
- Shared memory suffers from **cache coherency issues**, which arise because shared data migrate among the several caches

Producer-Consumer Problem

- To illustrate the concept of cooperating processes, let's consider the producer–consumer problem.
- A **producer process** produces information that is **consumed by a consumer** process.
- For example, **a compiler may produce assembly code** that is **consumed by an assembler**.
- The producer–consumer problem also provides a useful metaphor for the **client–server paradigm**.


Producer
Write()
Buffer
Consumer
Read()

Shared memory based implementation

- To allow producer and consumer processes to run concurrently, we must have a **buffer** that can be **filled by the producer** and **emptied by the consumer.**
- This buffer will reside in a **region of memory** that is **shared** by the producer and consumer processes.
- A producer can **produce one item** while the consumer is **consuming another item**.
- **Shared memory** is used as the buffer
- The producer and consumer **must be synchronized**, so that the consumer **does not try to consume** an item that has not yet been produced.

Bounded buffer and Unbounded buffer

- 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.

Bounded-Buffer – Shared-Memory Solution

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- The shared buffer is implemented as a **circular array** with two logical pointers: **in and out.**
- The variable **Rear** points to the next **free position** in the buffer; **Front** points to the **first full position** in the buffer.
- The buffer is **empty**
  - when **Front == Rear;**
- the buffer is **full**
  - when ((**Rear + 1) % BUFFER SIZE) == Front**.

Producer

```
item next_produced;
while (true) {
        /*  store produced data into buffer */
        while (((Rear + 1) % BUFFER_SIZE) == Front); // do
                                          //nothing since full buffer
                buffer[Rear] = next_produced;
                Rear = (Rear + 1) % BUFFER_SIZE;
}
```

Consumer

```
while (true) {
    while (Rear==Front) ;/*do nothing since empty buffer */
    next consumed = buffer[Front];
    Front= (Front + 1) % BUFFER SIZE;
/* consume the item in next consumed */
}
```

Support on Unix-like systems
(for lab experiment)

- System V provides system calls for using shared memory
- System V interprocess communication includes the shared-memory functions ***shmat***, ***shmctl***, ***shmdt*** and ***shmget***
- For programming languages with System V bindings (say, C/C++), shared memory regions can be created and accessed by calling the functions provided by the operating system

Synchronization

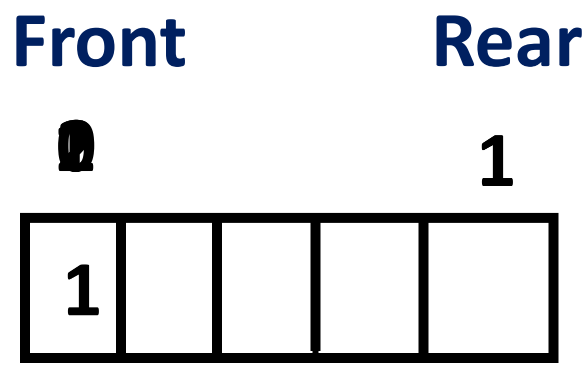
- One issue this illustration does not address concerns the situation in which both the producer process and the consumer process **attempt to access the shared buffer at a time**.
- It may result in a **race condition** and **incorrect result**
- We will discuss how synchronization among cooperating processes can be implemented effectively

Consumer 1

```
while (true) {
   while (Rear==Front) ;
   next consumed = buffer[Front];
   Front= (Front + 1) % BUFFER SIZE;
}
```

Consumer 2

```
while (true) {
   while (Rear==Front) ;
   next consumed = buffer[Front];
   Front= (Front + 1) % BUFFER SIZE;
}
```

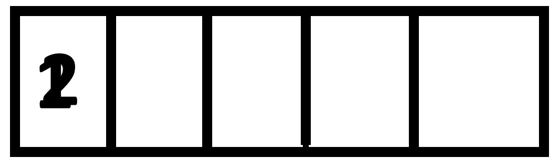


Producer 1

```
item next_produced;
while (true) {

      while (((Rear + 1) % BUFFER_SIZE) == Front);
            buffer[Rear 0] = next_produced;   1
            Rear = (Rear + 1) % BUFFER_SIZE;
}
```

Front 0   Rear 0



Producer 2

```
item next_produced;
while (true) {

      while (((Rear + 1) % BUFFER_SIZE) == Front);
            buffer[Rear 0] = next_produced;   2
            Rear = (Rear + 1) % BUFFER_SIZE;
}
```

Message-Passing Systems

- Shared memory scheme requires that these processes **share a region of memory** and that the code for accessing and manipulating the shared memory be **written explicitly** by the application programmer.
- Another way to achieve the same effect is for the **operating system to provide the means** for cooperating processes to communicate with each other via a **message-passing facility**
- Implementation of message passing does not require the communicating processes to be sharing the same memory

- Message passing provides a mechanism to allow processes **to communicate** and to **synchronize** their actions without sharing the same address space.
- It is particularly useful in **a distributed environment**, where the communicating processes may **reside on different computers** connected by a network.
- For example, an **Internet chat program** could be designed so that chat participants communicate with one another by **exchanging messages.**

Operations

- A message-passing facility provides at least two operations:
  - **send(message)**
  - **receive(message)**

Fixed Vs Variable sized messages

- 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 straightforward**.
- This restriction, however, makes the task of programming more difficult.
- Conversely, variable-sized messages require a **more complex system** level **implementation**, but the **programming task becomes simpler**.

- There are several methods for implementing the send()/receive() operations:
  - **Direct or indirect communication**
  - **Synchronous or asynchronous communication**
  - **Automatic or explicit buffering**

Synchronous vs Asynchronous

- Synchronous message passing occurs between objects that are **running at the same time.**
- With asynchronous message passing the **receiving object can be down or busy** when the requesting object sends the message
- Messages are **sent to a queue** where they are stored until the receiving process requests them
- Asynchronous messaging requires additional capabilities for storing and retransmitting data - **buffer**

Direct Communication

- Under direct communication, each process that wants to communicate must **explicitly name the recipient or sender** of the communication **`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**

Symmetry vs Asymmetry

- This scheme exhibits ***symmetry in addressing; that is, both the sender*** process and the receiver process must name the other to communicate.
- A **variant of this scheme** employs ***asymmetry in addressing.***
- *Here, **only the sender*** **names the recipient**; the recipient is not required to name the sender.
- In this scheme, the send() and receive() primitives are defined as follows:
  - **send(P, message)—**Send a message to process P.
  - **receive(id, message)—**Receive a message **from any process**. The
- variable id is set to the name of the process with which communication has taken place

Indirect Communication

- Messages are sent to 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**

- 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

Conflicts

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends a message to mailbox A
  - Both P_2 and P_3 execute receive () from A
  - 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**. The system may **define an algorithm** for selecting which process will receive the message (for example, **round robin**) Sender is notified who the receiver was.

Who owns mailbox ?

- A mailbox may be owned **either by a process** or by the **operating system.**
- If the mailbox is owned by a process (that is, the mailbox is **part of the address space of the process**), then we distinguish between the **owne**r (which can only receive messages through this mailbox) and **the user** (which can only send messages to the mailbox).
- Since each mailbox has a unique owner, there can be **no confusion about which process should receive a message** sent to this mailbox.
- When a process that **owns a mailbox terminates**, the **mailbox disappears**.
- Any process that subsequently sends a message to this mailbox must be notified that the **mailbox no longer exists.**

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**

Producer-Consumer

- The solution to the producer–consumer problem becomes simple when we use **blocking send() and blocking receive()** statements.
- The producer merely invokes the blocking send() call and waits until the message is delivered to either the receiver or the mailbox.
- Likewise, when the consumer invokes receive(), it blocks until a message is available.

Buffering

- Whether communication is direct or indirect, **messages** exchanged by communicating processes **reside in a temporary queue**.
- Basically, such queues can be implemented in three ways:
- **Zero capacity.**
- **Bounded capacity**
- **Unbounded capacity**
- The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it.
- In this case, the sender must block until the recipient receives the message.

- **Zero capacity.** The queue has a **maximum length of zero**; thus, the link cannot have any messages waiting in it.
- In this case, the sender must block until the recipient receives the message.
- **Bounded capacity.** The queue has finite **length *n*;** *thus, at most n messages* can reside in it. If the link is full, the **sender must block** until space is available in the queue.
- **Unbounded capacity.** The queue's length is **potentially infinite**; thus, any number of messages can wait in it. The sender never blocks.

```
message next_produced;

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

     send(next_produced);
}
```

The producer process using message passing.

```
message next_consumed;

while (true) {
     receive(next_consumed);

     /* consume the item in next_consumed */
}
```

The consumer process using message passing.

Support on Unix-like systems
(for lab experiment)

- Unix **System V** provides system calls for using message queue
- System V interprocess communication includes the shared-memory functions ***msgsnd***, ***msgrcv***, ***msgctl*** and ***msgget***
- For programming languages with System V binding(say, C/C++), message queue can be created and accessed by calling the functions provided by the operating system

Summary

- What are concurrent processes ?
- What are the reasons for their communication?
- What are the methods of IPC?
- Differences between Shared memory and Message passing ?
- How to choose a method ?
- Producer consumer problem
- Message passing design issues
- Implementation of Shared memory and message passing