Module 3

IPC

Interprocess Communication

- Processes within a system may be
  - ***independent*** or
  - ***cooperating***

Interprocess Communication

- ***Independent Process-***
  - *If it cannot affect or be affected by other processes executing in the system*
  - *Any process that does not share any data with any other process*

- **Cooperating process-**
  - can affect or be affected by other processes,
  - Any process that shares data with other process

Interprocess Communication

- Reasons for cooperating processes:
  - 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.

Interprocess Communication

- Reasons for cooperating processes:
  - 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.
    - A speedup can be achieved only if
      - the computer has multiple processing elements (such as CPUs or I/O channels).

Interprocess Communication

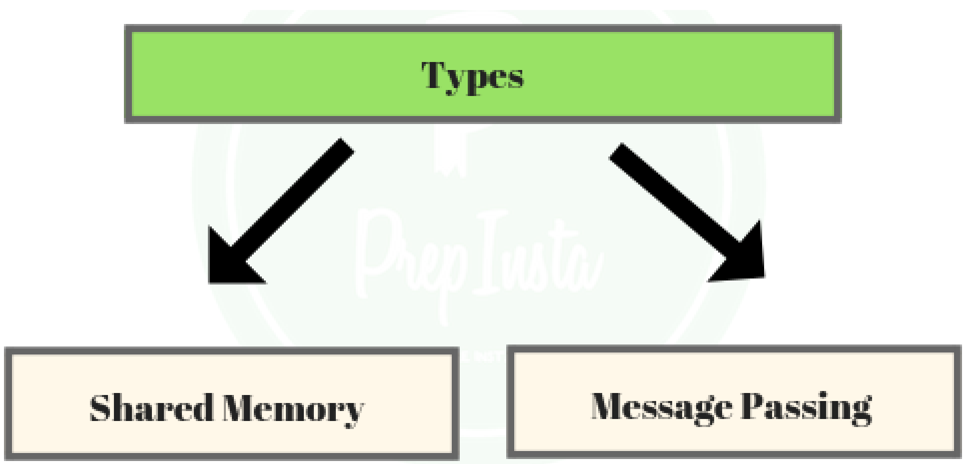
- Reasons for cooperating processes:
  - Modularity
    - We may want to construct the system in a modular fashion,
    - dividing the system functions into separate processes or threads,
  - Convenience
    - Even an individual user may work on many tasks at the same time.
    - For instance, a user may be editing, printing, and compiling in parallel.

Interprocess Communication

- Cooperating processes need **interprocess communication** (**IPC**) mechanism to exchange data and information

Interprocess Communication

- Two models of IPC
  - **Shared memory**
  - **Message passing**



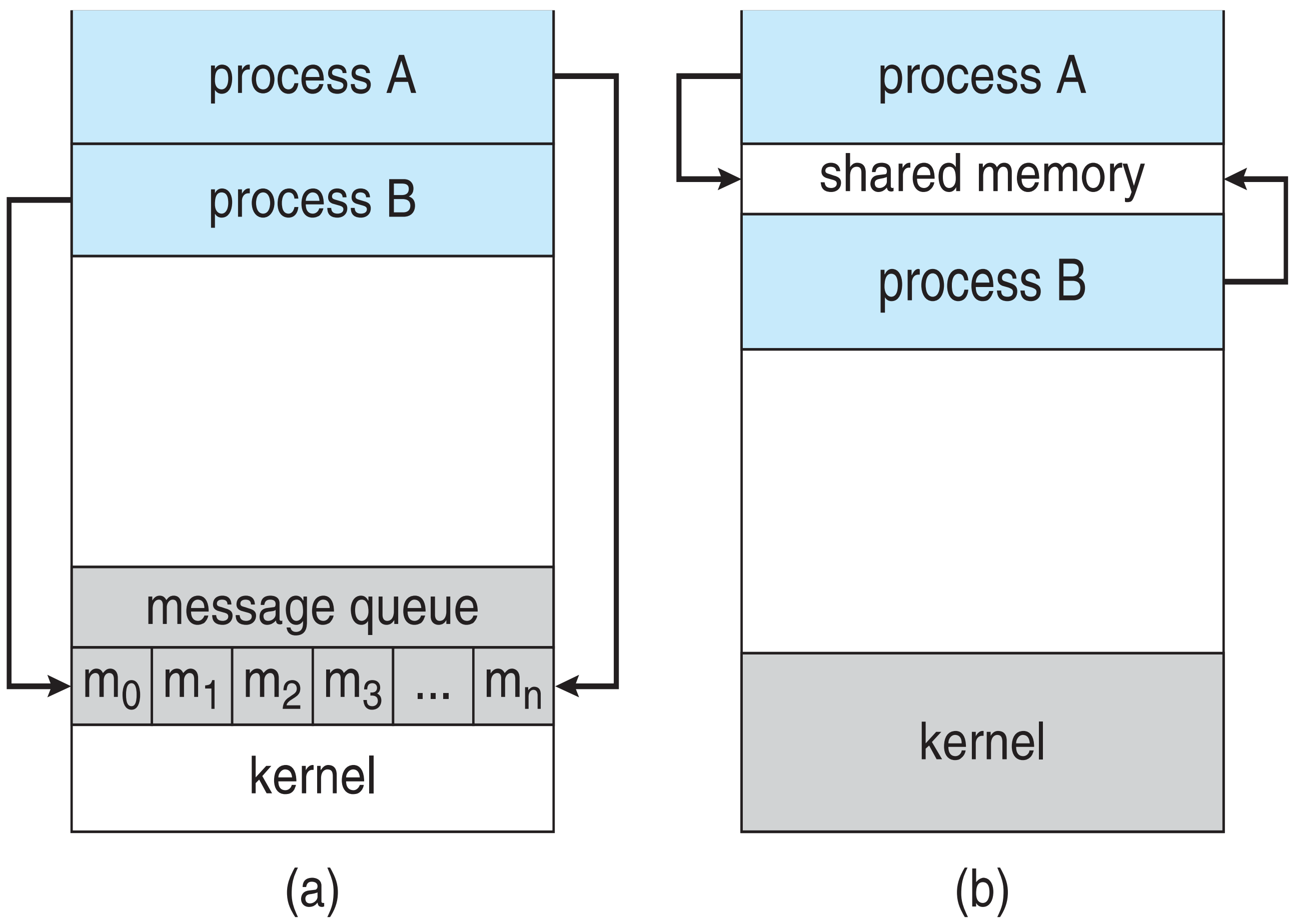
Interprocess Communication

- 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.

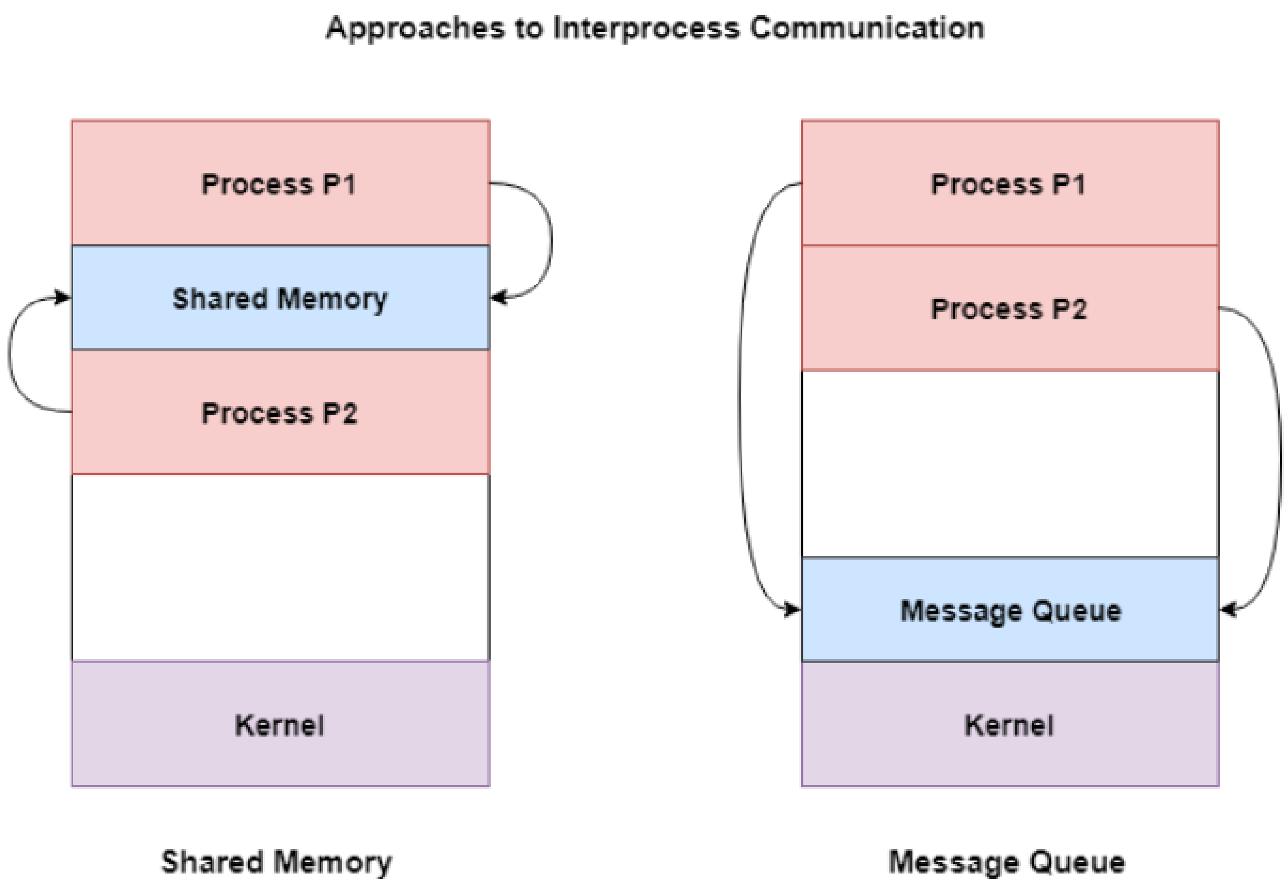
- In the message passing model,
  - communication takes place by means of messages exchanged between the cooperating processes.

Communications Models

(a) Message passing. (b) shared memory.



Communications Models



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 than
  - shared memory for interprocess communication.

Shared memory

- Shared memory allows
  - **maximum speed** and convenience of communication
- **Shared memory is faster**
  - than message passing.

Message Passing	Shared memory

- **Slower** As message passing systems are typically implemented
  - **using system calls and**
  - **thus require the more time-consuming task of kernel intervention.**

- Faster as 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.

Producer-Consumer Problem

- Paradigm for cooperating processes,
- A *producer* process
  - produces information
  - that is consumed by a *consumer* process.

Producer-Consumer Problem

For example,

1) A compiler may produce assembly code,
   - which is consumed by an assembler.
   - The assembler, in turn, can produce object modules,
   - which are consumed by the loader.
2) Server as a producer and a client as consumer.
   - For example, a Web server produces (that is, provides) HTML files and images,
   - which are consumed (that is, read) by the client Web browser requesting the resource.

Producer-Consumer Problem

- **Solution= Shared memory.**

- To allow producer and consumer processes to run concurrently,

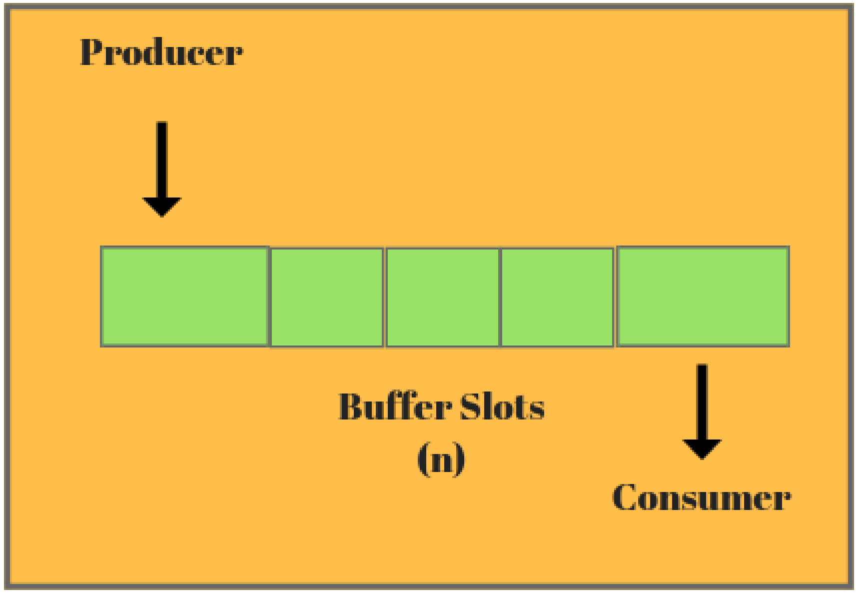
Producer-Consumer Problem

- An available buffer of items 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.

Producer-Consumer Problem

- A producer can produce one item
- While the consumer is consuming another item.



Producer-Consumer Problem

- **<u>The producer and consumer must be synchronized,</u>**
  - so that the consumer does not try to consume an item
  - that has not yet been produced.

Producer-Consumer Problem

Two types of buffers can be used.

- **unbounded-buffer**
- **bounded-buffer**

Producer-Consumer Problem

Two types of buffers can be used.

- **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.

- **bounded-buffer** assumes that there is a fixed buffer size
  - The consumer must wait if the buffer is empty,
  - and the producer must wait if the buffer is full.

Bounded-Buffer – Shared-Memory Solution

Shared data

- The following variables reside in a region of memory shared by the producer and consumer processes:

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

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

Bounded-Buffer – Shared-Memory Solution

- The shared buffer is implemented as a circular array
  - with two logical pointers: in and out.

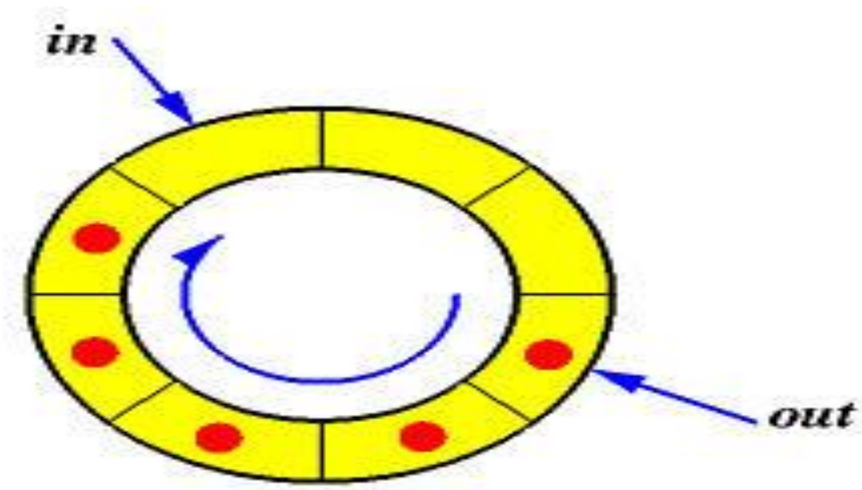
```
#define BUFFER_SIZE 10
          typedef struct {
            . . .
          } item;

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

Bounded-Buffer – Shared-Memory Solution

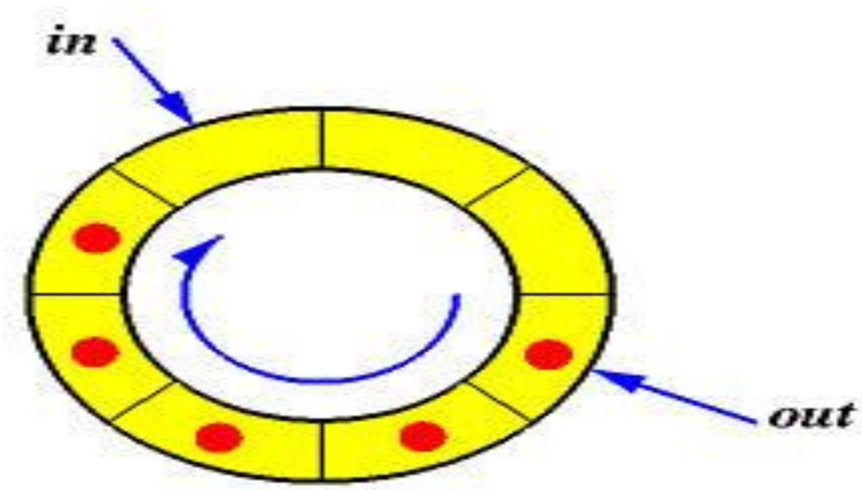
– in points to the next free position in the buffer;

– out points to the first full position in the buffer.



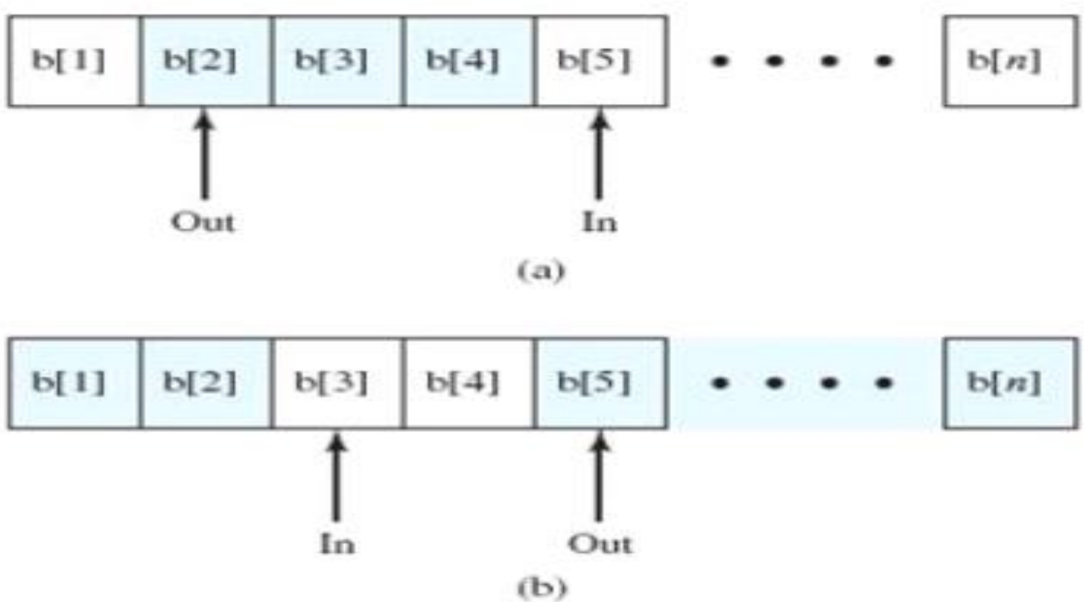
Bounded-Buffer – Shared-Memory Solution

- The buffer is empty when in== out;



Bounded-Buffer – Shared-Memory Solution

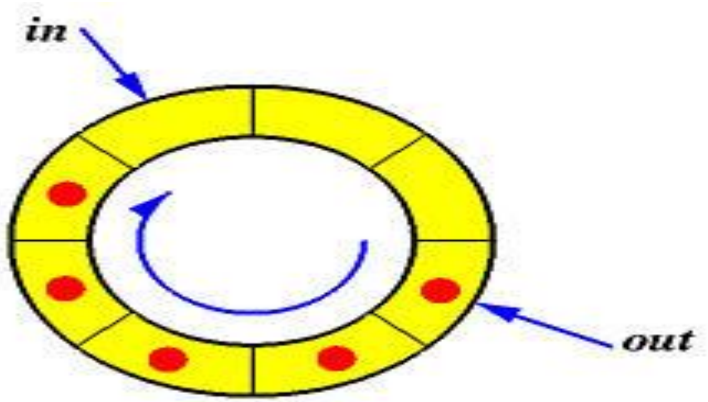
- The buffer is full when ((in+ 1)% BUFFER_SIZE) == out.



Bounded-Buffer – Producer

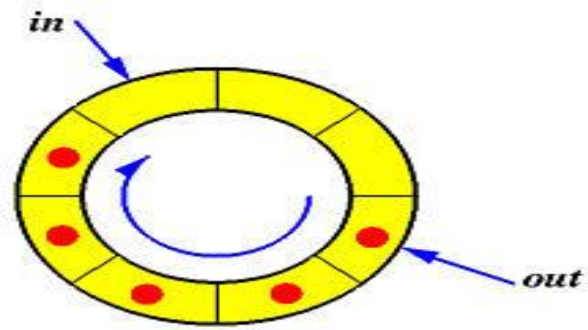
- The producer process has a local variable nextProduced
  - in which the new item to be produced is stored.

```
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;
}
```



Bounded Buffer – Consumer

- The consumer process has a
  - local variable next Consumed in which the item to be consumed is stored.




```
item next_consumed;
while (true) {
       while (in == out) ; /* do nothing */
       next_consumed = buffer[out];
       out = (out + 1) % BUFFER_SIZE;

       /* consume the item in next consumed */
}
```

Interprocess Communication – Shared Memory

What ?

- An area of memory shared among the processes that wish to communicate

Interprocess Communication – Shared Memory

- The communication is under the control of the
  - users processes not the operating system.
- The code for accessing and manipulating the shared memory be written
  - explicitly by the application programmer.

Interprocess Communication – Shared Memory

- Major issues is to provide mechanism that will allow the user processes to
  - synchronize their actions when they access shared memory.

Interprocess Communication – Message Passing

Interprocess Communication – Message Passing

- Another means for cooperating processes to communicate with each other
  - **via a message-passing facility.**

- Processes communicate with each other
  - **without resorting to shared variables or shared address space**

Interprocess Communication – Message Passing

- Mechanism needed for processes
  - to communicate and
  - to synchronize their actions

Interprocess Communication – Message Passing

- Why?

Interprocess Communication – Message Passing

- Particularly useful in a distributed environment,
  - where the communicating processes may reside on different computers
  - connected by a network.
- For example, a **chat** program used on the World Wide Web could be designed so
  - that chat participants communicate with one another by exchanging messages.

Interprocess Communication – Message Passing

- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size is either
  - fixed or
  - variable

Interprocess Communication – Message Passing

- If only fixed-sized messages can be sent,
  - the system-level **implementation is straightforward.**
  - the task of programming more difficult.

- Conversely, variable-sized messages require
  - a more **complex system-level implementation,**
  - but the programming task becomes simpler.

Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive

Message Passing (Cont.)

- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Message Passing (Cont.)

- Implementation issues:
  - A link has some capacity that determines the number of messages that can reside in it temporarily
  - for which every link has a queue associated with it

Message Passing (Cont.)

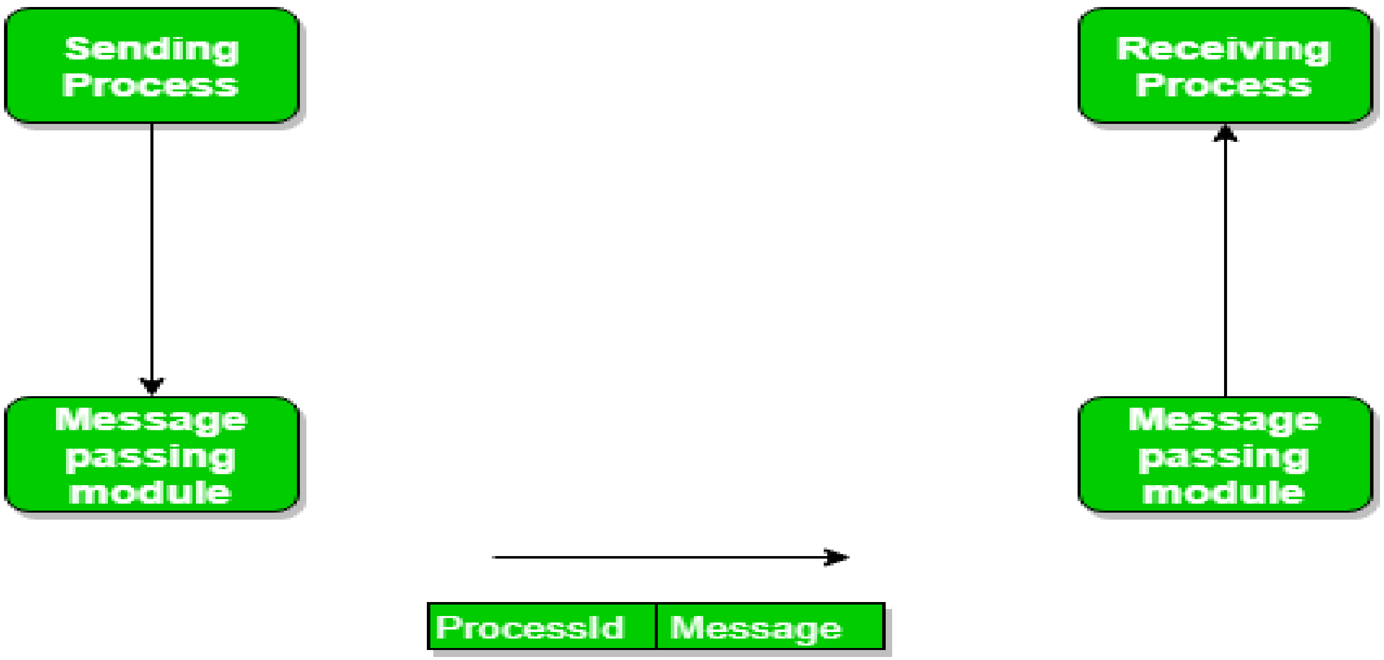
- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

Message Passing (Cont.)

- Processes that want to communicate must have a way to refer to each other.
- They can use either
  - direct or
  - indirect communication.

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



Direct Communication

Properties of communication link

- Links are established automatically between every pair of processes that want to communicate.
  - The processes need to know **only each other's identity to communicate.**
  - 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**

Direct Communication

- This scheme **exhibits symmetry in addressing**; that is,
  - both the sender process and the receiver process **must name the other to communicate.**

Direct Communication- A variant

- A variant of this scheme employs *asymmetry* in addressing.
  - Only the sender names the recipient;
  - the recipient is not required to name the sender.

A variant

- 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.

Disadvantage in symmetric and asymmetric schemes

- Limited modularity of the resulting process definitions.

- Changing the identifier of a process may necessitate **examining all other process definitions.**

- All references to the **old identifier must be found**, so that they can be modified to the new identifier.

- Any such **hard-coding** techniques, where identifiers must be explicitly stated, are **less desirable than techniques involving indirection.**

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

Indirect Communication

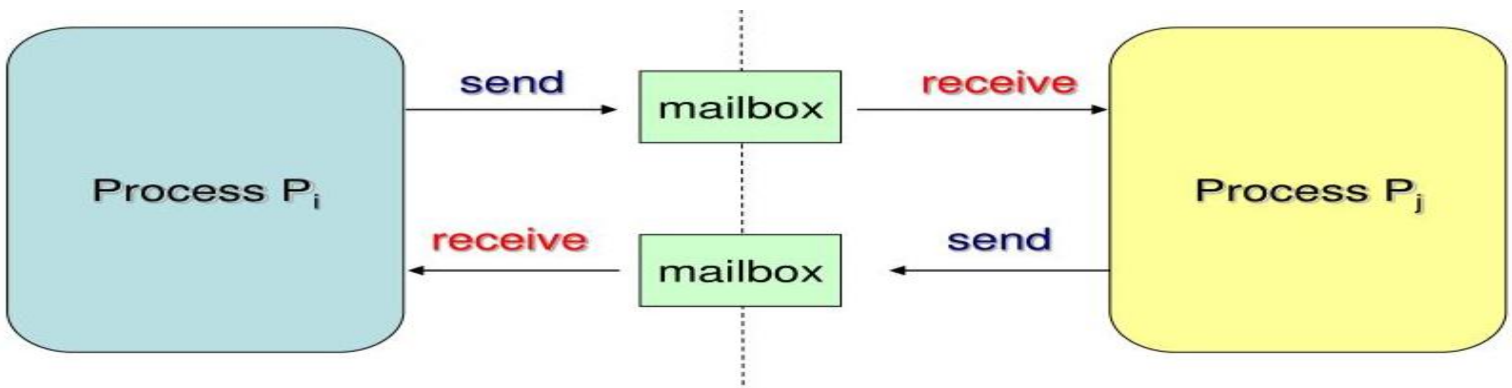
- A mailbox can be viewed abstractly as an
  - object into which messages can be placed by processes and
  - from which messages can be removed.
  - Each mailbox has a unique identification

Fig: Indirect Addressing

Indirect Communication-mailboxes

- **A process can communicate with some other process <u>via a number of different mailboxes.</u>**

- Two processes can communicate only if the processes have a shared mailbox,



Indirect Communication

- Properties of communication link
  - **Link established only if processes share a common mailbox**
  - A link may be associated with many processes
  - **Between Each pair of communicating processes, there may be a number of different links with each link corresponding to one mailbox**
  - Link may be unidirectional or bi-directional

Indirect Communication

- Primitives are defined as:

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

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

Indirect Communication

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

Indirect Communication

Solutions

- The answer depends on which of the following methods we choose:
  - This can be solved by either
  - **Enforcing that only two processes can share a single mailbox**

or

  - **Allow only one process at a time to execute a receive operation**
  - That is, either P2 or *P3,* but not both, will receive the message

or

  - **Allow the system to select arbitrarily the receiver**
  - The system also may define an algorithm for selecting which process will receive the message
    - that is, *round robin,* where processes take turns receiving messages
  - Sender is notified about the receiver.

- Who owns the Mailbox??

- A mailbox may be owned either
  - by a process or
  - by the operating system.

If the mailbox is owned by a process

- Mailbox is part of the address space of the process
- **The owner can only receive messages through this mailbox**
- **The user 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.

- What happens when the process that owns mailbox terminates??

What happens when the process that owns mailbox terminates??

- **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.

If the mailbox is owned by OS

- A mailbox that is owned by the operating system has an existence of its own.
- It is independent and is not attached to any particular process.

If the mailbox is owned by OS

- If OS owns the mailbox, then:
  - It is independent process, not attached to a process.
  - Then the OS must allow the process to:

1) create a mailbox.
2) send and receive messages through mailbox.
3) delete the mailbox.

- Process becomes the owner of new mailbox by default.
- Owner process can only receive messages through this mailbox.
- Ownership and receiving privileges can be passed using system calls to other processes. This will result in multiple receivers for each mailbox.

Process Synchronization

Synchronization

- IPC takes place via send() and receive () primitives.
- There are different design options for implementing each primitive.

Synchronization

- Message passing may be either
  - Blocking/synchronous or
  - non blocking/ asynchronous

Synchronization

- Blocking send
- Non blocking send
- Blocking receive
- Non blocking receive

Synchronization

- Blocking send
  - The sending process is blocked until the message is received by the receiving process or by the mailbox.
- Non blocking send
  - The sending process sends the message and resumes operation.
- Blocking receive
  - The receiver blocks until a message is available.
- Non blocking receive
  - The receiver retrieves either a valid message or a null.

Synchronization

- When both send() and receive() are blocking,
  – Rendezvous between the sender and the receiver.

- The solution to the producer-consumer problem becomes trivial –
  – use blocking send() and receive()

Solution =Producer-consumer problem

- Producer invokes
  - the blocking send() call and
  - waits until the message is delivered to either the receiver or the mailbox.

- 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.

Buffering

- Queues can be implemented in three ways:
  - Zero capacity
  - Bounded capacity
  - Unbounded capacity.

Buffering

- Zero capacity
  - Maximum Queue Length=0
  - The link cannot have any messages waiting in it.
  - The sender must block until the recipient receives the message.

Buffering

- Bounded capacity
  - Queue has finite length *n;*
  - At most *n* messages can reside in it.
  - The link's capacity is finite.

Buffering

- Bounded capacity
  - If the queue is not full
    - the message is placed in the queue
    - either the message is copied or a pointer to the message is kept
    - the sender can continue execution without waiting.
  - If the link is full,
    - the sender must block until space is available in the queue.

Buffering

- Unbounded capacity.
  - The queue's length is potentially infinite;
  - Any number of messages can wait in it.
  - The sender never blocks.

Buffering

- The zero-capacity = message system with no buffering;
- The other cases =systems with automatic buffering.