CENG 322
LAB 5

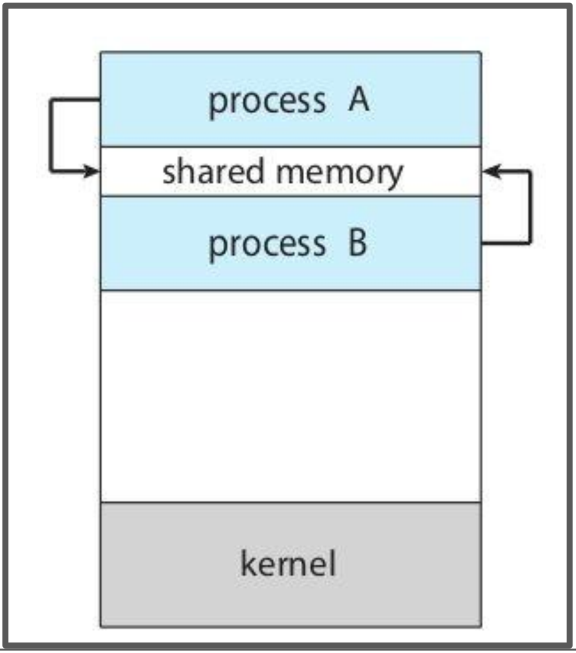
Processes 2

Inter Process Communication (IPC)

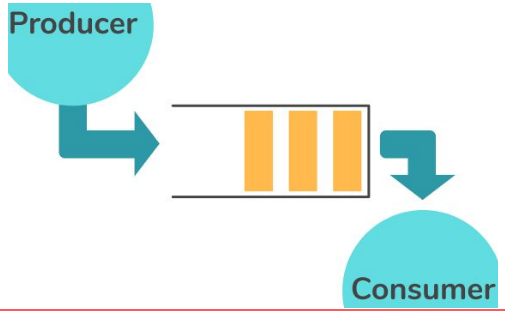
- A process can be a type of **independent** or **cooperating process**.
- There are several reasons to allow process cooperation:
    - **Information sharing:** Since several applications may be interested in the same piece of information (for instance, copying and pasting), we must provide an environment 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.

IPC - Shared Memory

- A **region** of memory that is shared by the **cooperating** processes is established.
- Processes can then **exchange information** by reading and writing data to the **shared region**.
- Let's consider the **producer–consumer problem**. (**producer** and **consumer** processes to run **concurrently**)
- The producer and consumer must be **synchronized**, so that the consumer **does not try to consume** an item that has **not yet been produced** (unbounded or bounded buffer).

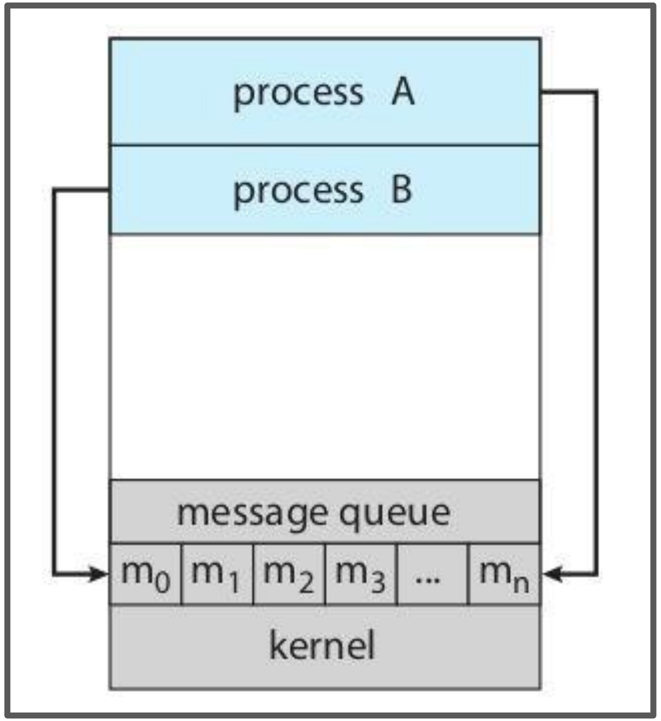


This figure was taken from your course book.



IPC - Message Passing

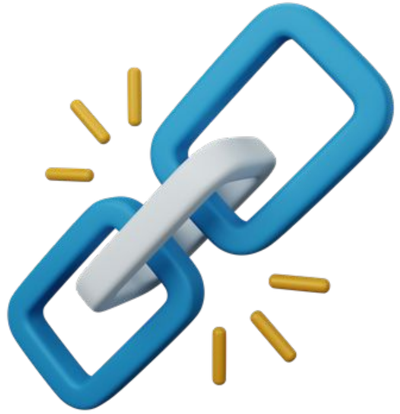
- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions **without sharing the same address space**.
- It is particularly useful in a **distributed environment**, where the communicating processes may **reside on different computers** connected by a network.
- If processes A and B want to communicate, they must **send** messages and **receive** messages from each other: a **communication link** must exist between them (m0, m1, m2 etc.).
- Communication between processes takes place through calls to **send()** and **receive()** primitives.

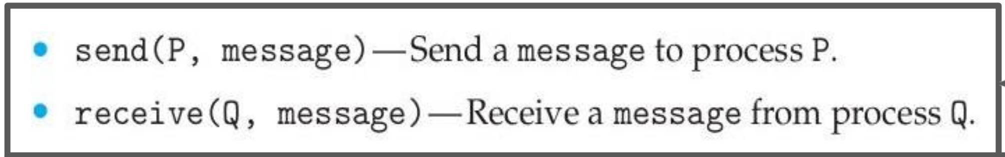


This figure was taken from your course book.

Message Passing - Direct

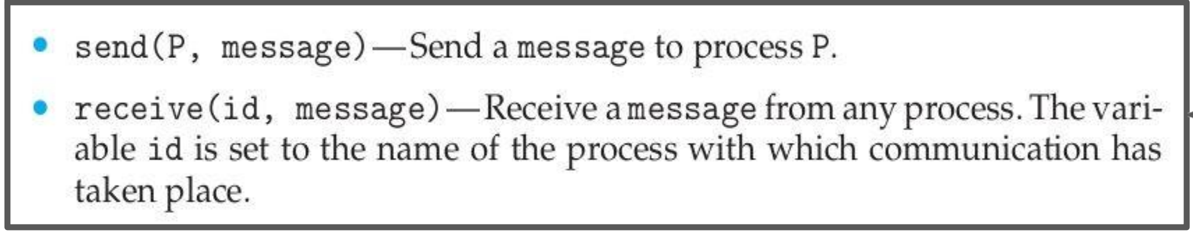
- In direct communication, each process that wants to communicate must **explicitly name** the recipient or sender of the communication.
- In this method, a **link** is established between one pair of communicating processes, and between each pair, **only one link exists**.





- send(P, message)—Send a message to process P.
- receive(Q, message)—Receive a message from process Q.

symmetric scheme

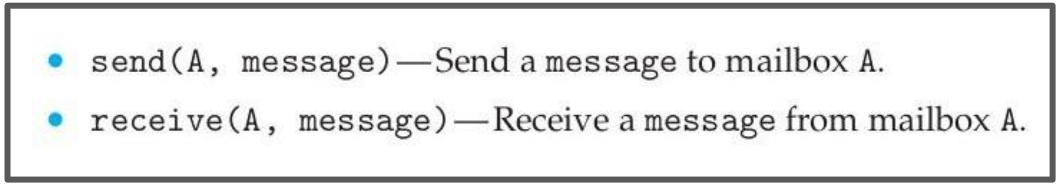


- send(P, message)—Send a message to process P.
- receive(id, message)—Receive a message from any process. The variable id is set to the name of the process with which communication has taken place.

asymmetric scheme

Message Passing - Indirect

- With indirect communication, the messages are sent to and received from **mailboxes**, or **ports**.
- Each pair of processes sharing **several communication links**.
- A link can communicate with **many processes**.
- The link may be **bi-directional** or **unidirectional**.



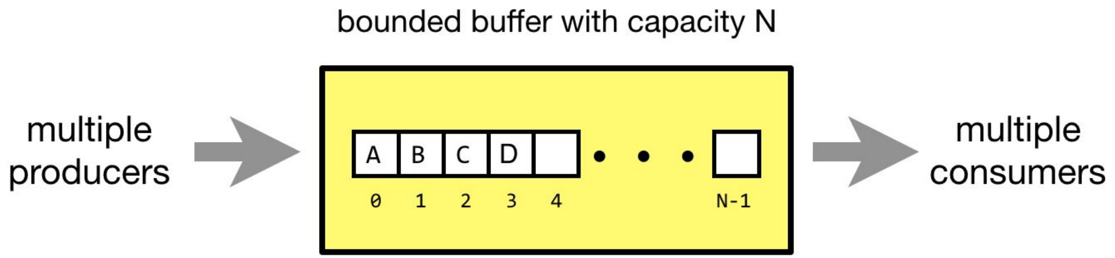
- `send(A, message)`—Send a message to mailbox A.
- `receive(A, message)`—Receive a message from mailbox A.

Message Passing - Synchronization

- Communication between processes takes place through calls to send() and receive() primitives. There are different design options for implementing each primitive. Message passing may be either **blocking** or **nonblocking** — also known as **synchronous** and **asynchronous**.
    - **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.

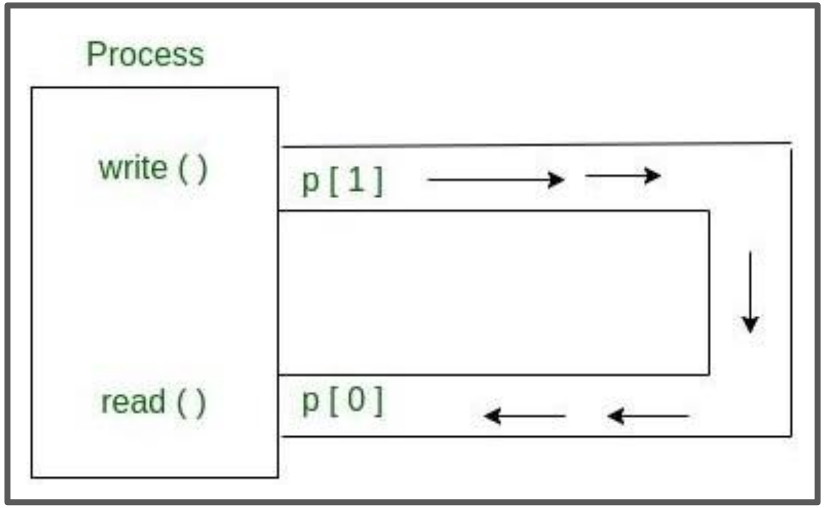
Message Passing - Buffering

- **Zero capacity:** The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. Processes must "**handshake**" in order to communicate.
- **Bounded capacity:** The queue has finite length N; thus, at most N messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The link's capacity is finite, however. 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.



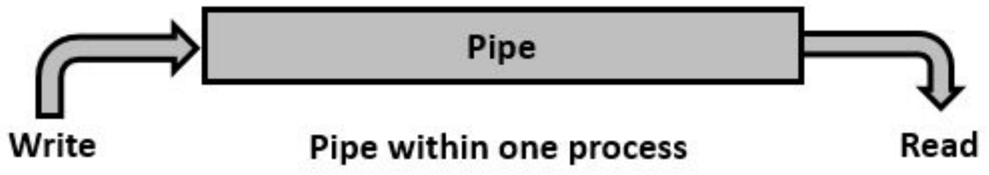
Pipes

- Conceptually, a pipe is a connection between two processes, such that the **standard output** from one process **becomes the standard input** of the other process.
- Pipe is **one-way communication** only i.e we can use a pipe such that one process **write to the pipe**, and the other process **reads from the pipe**.
- If a process tries to **read before something** is written to the pipe, the **process is suspended** until something is written.
- Pipes behave **FIFO** like a **queue** data structure.
- Size of read and write **don't have to match**. We can write 512 bytes at a time but we can read only 1 byte at a time in a pipe.



https://www.geeksforgeeks.org/pipe-system-call/

Pipes



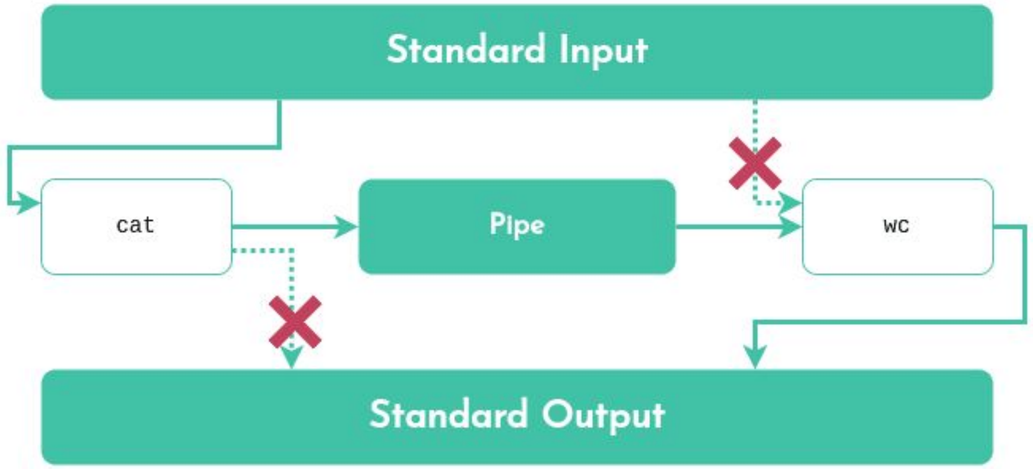
Advantages:

- **Simplicity:** Pipes are a simple and straightforward way for processes to communicate with each other.
- **Efficiency:** Pipes are an efficient way for processes to communicate, as they can transfer data quickly and with minimal overhead.
- **Reliability:** Pipes are a reliable way for processes to communicate, as they can detect errors in data transmission and ensure that data is delivered correctly.
- **Flexibility:** Pipes can be used to implement various communication protocols, including one-way and two-way communication.

Pipes are a useful IPC technique for simple and efficient communication between processes **on the same computer.** However, they **may not be suitable for large-scale** distributed systems or situations where bidirectional communication is required.

Pipes

- We do **cat test.txt | wc -l**, the contents of the file **don't appear at all**. So what is this **"|"** operator doing here exactly?
- The shell creates **a pipe** and **two child processes**, one for the **cat command** and **one for wc**. Then, it **redirects** cat's **standard output** towards wc's **standard input**.
- Therefore, the cat command does not write its output in the standard output (our terminal), but rather in the pipe. Then, the wc command will **go looking for the data in that pipe** rather than the standard input.


Standard Input
cat
Pipe
wc
Standard Output