**Process Creation**

Parent process creates children processes, which, in turn create other processes, forming a tree of processes. A process may be created by another process using fork(). The creating process is called the parent process and the created process is the child process. A child process can have only one parent but a parent process may have many children. Both the parent and child processes have the same memory image, open files and environment strings. However, they have distinct address spaces.

* Resource sharing
  + Parent and children share all resources.
  + Children share subset of parent’s resources.
* Execution
  + Parent and children execute concurrently.
  + Parent waits until children terminate.

**Process Termination**

Process termination occurs when the process is terminated The exit() system call is used by most operating systems for process termination.

Some of the causes of process termination are as follows:

1. A process may be terminated after its execution is naturally completed. This process leaves the processor and releases all its resources.
2. A child process may be terminated if its parent process requests for its termination.
3. A process can be terminated if it tries to use a resource that it is not allowed to. For example - A process can be terminated for trying to write into a read only file.
4. If an I/O failure occurs for a process, it can be terminated. For example - If a process requires the printer and it is not working, then the process will be terminated.
5. In most cases, if a parent process is terminated then its child processes are also terminated. This is done because the child process cannot exist without the parent process.
6. If a process requires more memory than is currently available in the system, then it is terminated because of memory scarcity.

**Types of Processes**

There are two types of operating system processes: independent and cooperating.

**Independent Processes**

These processes work independently of themselves. They do not affect and cannot be affected by other processes that are running within the operating system. These processes do not share any data with any processes or systems.

**Cooperating Processes**

These processes can be affected by other processes and they can, in turn, affect other processes within the operating system. These processes share their data with other processes and systems.

**Advantages of process cooperation:**

* Information sharing: Since several users may be interested in the same piece of information, we must provide and environment to allow concurrent access to these types of resources.
* Computation speed-up: 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.
* Modularity: We may want to construct the system in a modular fashion, dividing the system functions into separate processes.
* Convenience: Even an individual user may have many tasks on which to work at one time.

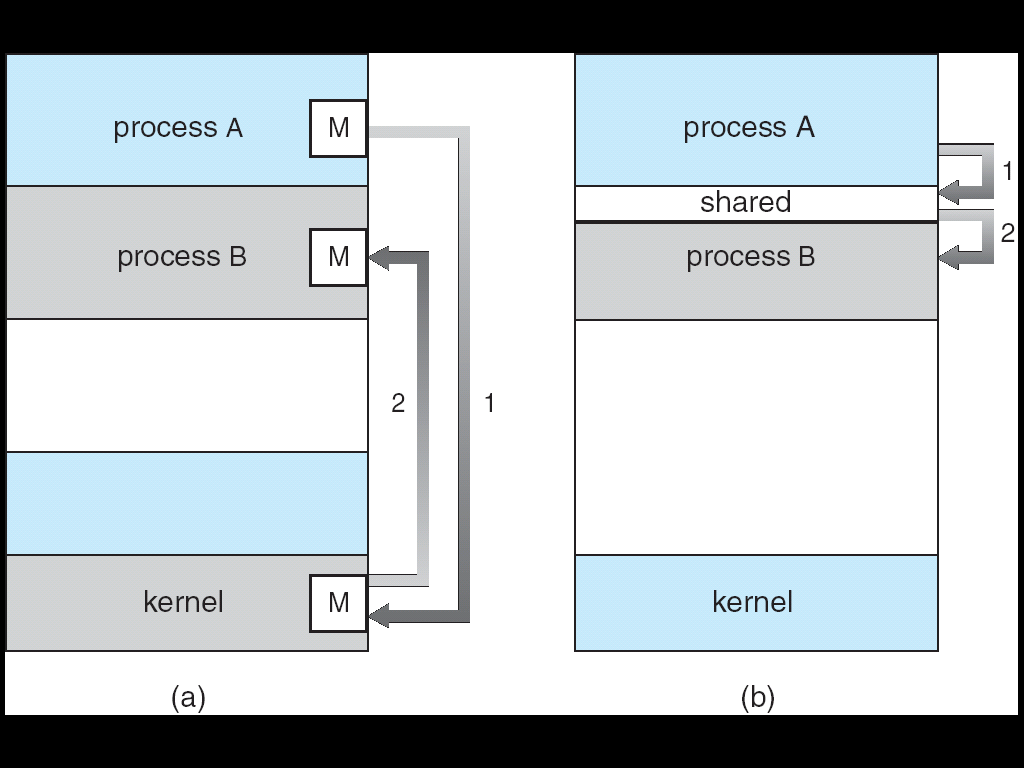
Cooperating processes require an **Interprocess Communication (IPC)** mechanism that will allow them to exchange data and information.

Two fundamental models:

* Shared memory
* Message passing

In **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 message exchanged between the cooperating processes.



**Shared memory system**

Communication between processes using shared memory requires processes to share some variable and it completely depends on how programmer will implement it. One way of communication using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously and they share some resources or use some information from other process, process1 generate information about certain computations or resources being used and keeps it as a record in shared memory. When process2 need to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from other process as well as for delivering any specific information to other process.

Let’s discuss an example of communication between processes using shared memory method.

**Producer-Consumer Problem**

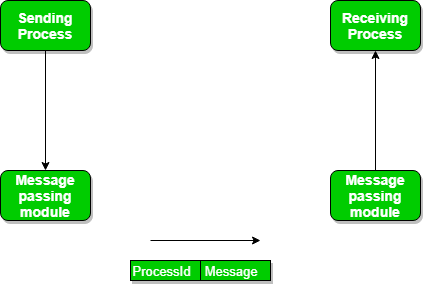
There are two processes: Producer and Consumer. Producer produces some item and Consumer consumes that item. The two processes shares a common space or memory location known as buffer where the item produced by Producer is stored and from where the Consumer consumes the item if needed. There are two version of this problem: first one is known as unbounded buffer problem in which Producer can keep on producing items and there is no limit on size of buffer, the second one is known as bounded buffer problem in which producer can produce up to a certain amount of item and after that it starts waiting for consumer to consume it. We will discuss the bounded buffer problem.

The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer. The solution for the producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

**Message passing system**

In this method, processes communicate with each other without using any kind of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follow:

* Establish a communication link (if a link already exists, no need to establish it again.)
* Start exchanging messages using basic primitives.  
  We need at least two primitives:  
  – **send**(message, destination) or **send**(message)  
  – **receive**(message, host) or **receive**(message)



The message size can be of fixed size or of variable size. If it is of fixed size, it is easy for OS designer but complicated for programmer and if it is of variable size then it is easy for programmer but complicated for the OS designer. A standard message can have two parts: **header and body.**  
The **header part** is used for storing Message type, destination id, source id, message length and control information. The control information contains information like what to do if runs out of buffer space, sequence number, priority. Generally, message is sent using FIFO style.

**Logical implementation**

* Direct or indirect communication
* Synchronous or asynchronous communication
* Automatic or explicit buffering

**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 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.
* This scheme is symmetric in addressing since both the sender and receiver process must name to other to communicate.
* In asymmetric addressing only sender names the recipient; the recipient is not required to name the sender.

**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.
* Operations
  + create a new mailbox
  + send and receive messages through mailbox
  + destroy a mailbox
* The messages are sent to and received from **mailboxs or ports**.
* A mailbox can be viewed abstractly as an object into which messages can be places by processes and from which messages can be removed.
* Each mailbox has a unique identification
* Two processes can communicate if they have a shared mailbox.
* The send() and receive() primitives are defined as follows:
* send(A, message)---Send a message to mailbox A
* receive(A, message)---Receive a message from mailbox A.

**Automatic and explicit buffering**

Automatic buffering provides a queue with indefinite length, thus ensuring the sender will never have to block while waiting to copy a message. There are no specifications on how automatic buffering will be provided; one scheme may reserve sufficiently large memory where much of the memory is wasted. Explicit buffering specifies how large the buffer is. In this situation, the sender may be blocked while waiting for available space in the queue. However, it is less likely that memory will be wasted with explicit buffering.

**Buffering**

* Queue of messages attached to the link; implemented in one of three ways.

1. Zero capacity – 0 messages  
   Sender must wait for receiver.
2. Bounded capacity – finite length of *n* messages  
   Sender must wait if link full.
3. Unbounded capacity – infinite length   
   Sender never waits.