What is an Operating System?

An Operating System (OS) is a collection of software that manages computer hardware and provides services for programs. Specifically, it hides hardware complexity, manages computational resources, and provides isolation and protection. Most importantly, it directly has privilege access to the underlying hardware.

Major components of an OS are the file system, scheduler, and device driver. You probably have used both Desktop (Windows, Mac, Linux) and Embedded (Android, iOS) operating systems before. There are three key elements of an operating system, which are: (1) **Abstractions** (process, thread, file, socket, memory), (2) **Mechanisms** (create, schedule, open, write, allocate), and (3) **Policies** (LRU, EDF).

There are two operating system design principles, which are: (1) **Separation of mechanism and policy** by implementing flexible mechanisms to support policies, and (2) **Optimization for common case:** Where will the OS be used? What will the user want to execute on that machine? What are the workload requirements?

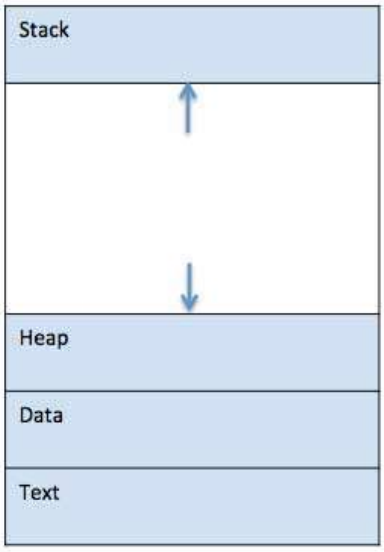
There are three types of Operating Systems commonly used nowadays. The first is **Monolithic OS**, where the entire OS is working in kernel space and is alone in supervisor mode. The second is **Modular OS**, in which some part of the system core will be located in independent files called modules that can be added to the system at run time. And the third is **Micro OS**, where the kernel is broken down into separate processes, known as servers. Some of the servers run in kernel space and some run in user-space.

Now let's get into those major concepts you need to understand in more detail.

1: Processes and Process Management

A process is basically a program in execution. The execution of a process must progress in a sequential fashion. To put it in simple terms, we write our computer programs in a text file, and when we execute this program, it becomes a process which performs all the tasks mentioned in the program.

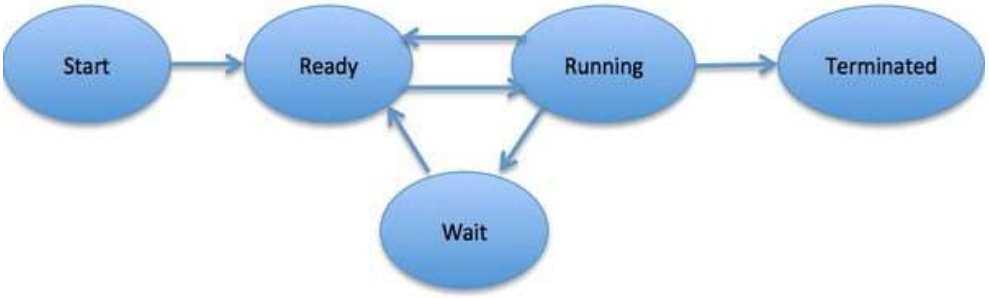
When a program is loaded into the memory and it becomes a process, it can be divided into four sections ─ stack, heap, text, and data. The following image shows a simplified layout of a process inside main memory



- **Stack:** The process Stack contains the temporary data, such as method/function parameters, return address, and local variables.

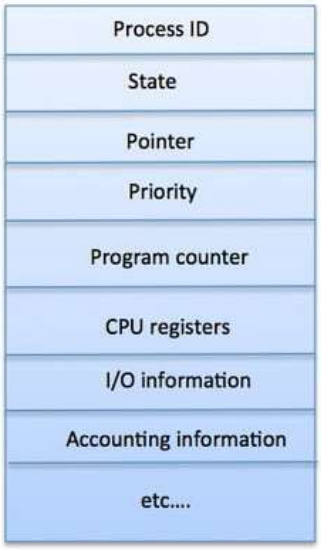
- **Heap:** This is dynamically allocated memory to a process during its run time.
- **Text:** This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.
- **Data:** This section contains the global and static variables.

When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized. In general, a process can have one of the following five states at a time:



- **Start:** The initial state when a process is first started/created.
- **Ready:** The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. A process may come into this state after the **Start** state, or while running it by but getting interrupted by the scheduler to assign CPU to some other process.
- **Running:** Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.
- **Waiting:** the process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.
- **Terminated or Exit:** Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.

A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID). A PCB keeps all the information needed to keep track of a process:



- **Process State:** The current state of the process — whether it is ready, running, waiting, or whatever.
- **Process Privileges:** This is required to allow/disallow access to system resources.
- **Process ID:** Unique identification for each of the processes in the operating system.
- **Pointer:** A pointer to the parent process.
- **Program Counter:** Program Counter is a pointer to the address of the next instruction to be executed for this process.
- **CPU Registers:** Various CPU registers where processes need to be stored for execution for running state.
- **CPU Scheduling Information:** Process priority and other scheduling information which is required to schedule the process.
- **Memory Management Information:** This includes the information of page table, memory limits, and segment table, depending on the memory used by the operating system.
- **Accounting Information:** This includes the amount of CPU used for process execution, time limits, execution ID, and so on.
- **IO Status Information:** This includes a list of I/O devices allocated to the process.

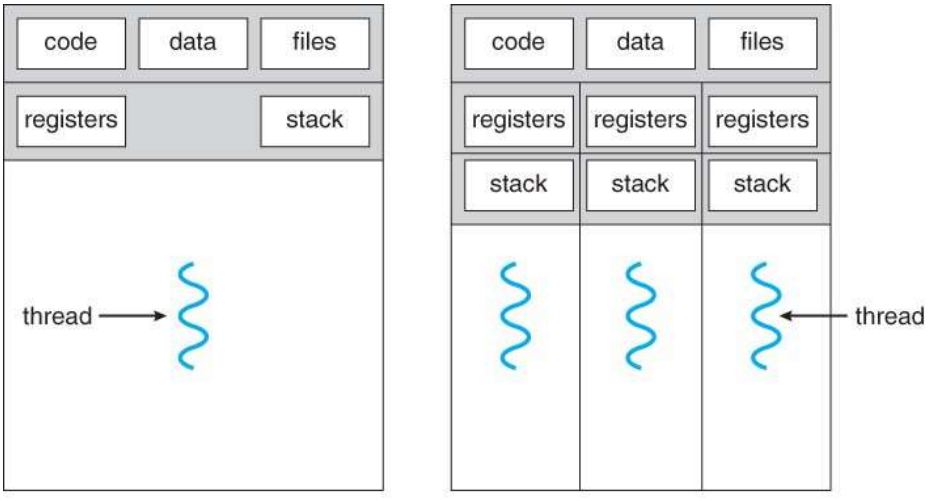
2: Threads and Concurrency

A thread is a flow of execution through the process code. It has its own program counter that keeps track of which instruction to execute next. It also has system registers which hold its current working variables, and a stack which contains the execution history.

A thread shares with its peer threads various information like code segment, data segment, and open files. When one thread alters a code segment memory item, all other threads see that.

A thread is also called a **lightweight process**. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving the performance of operating systems by reducing the overhead. A thread is equivalent to a classical process.

Each thread belongs to exactly one process, and no thread can exist outside a process. Each thread represents a separate flow of control. Threads have been successfully used in implementing network servers and web servers. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors.



single-threaded process multithreaded process

Advantages of threads:

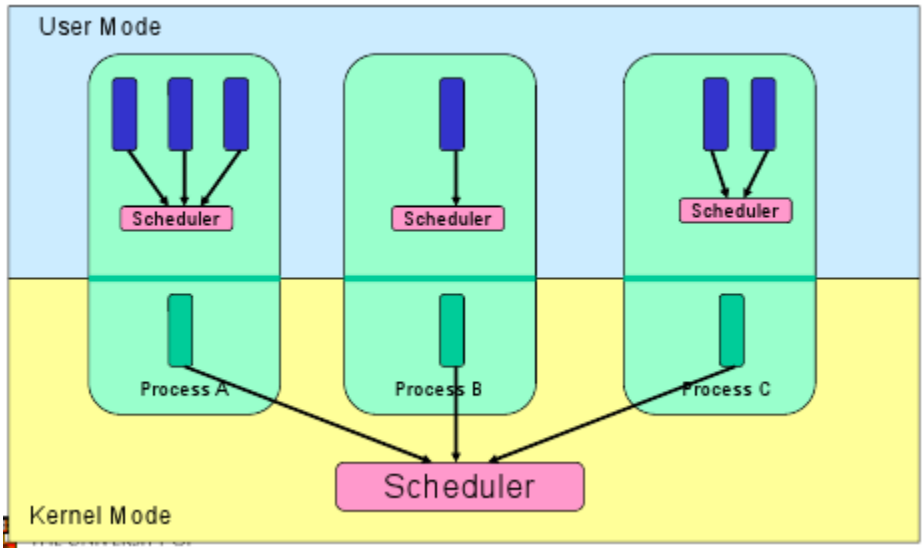
- They minimize the context switching time.
- Using them provides concurrency within a process.
- They provide efficient communication.
- It is more economical to create and context switch threads.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

Threads are implemented in the following two ways:

- **User Level Threads:** User-managed threads.
- **Kernel Level Threads:** Operating System-managed threads acting on a kernel, an operating system core.

User Level Threads

In this case, the thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing messages and data between threads, for scheduling thread execution, and for saving and restoring thread contexts. The application starts with a single thread.



Advantages:

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.
- Scheduling can be application-specific in the user level thread.
- User level threads are fast to create and manage.

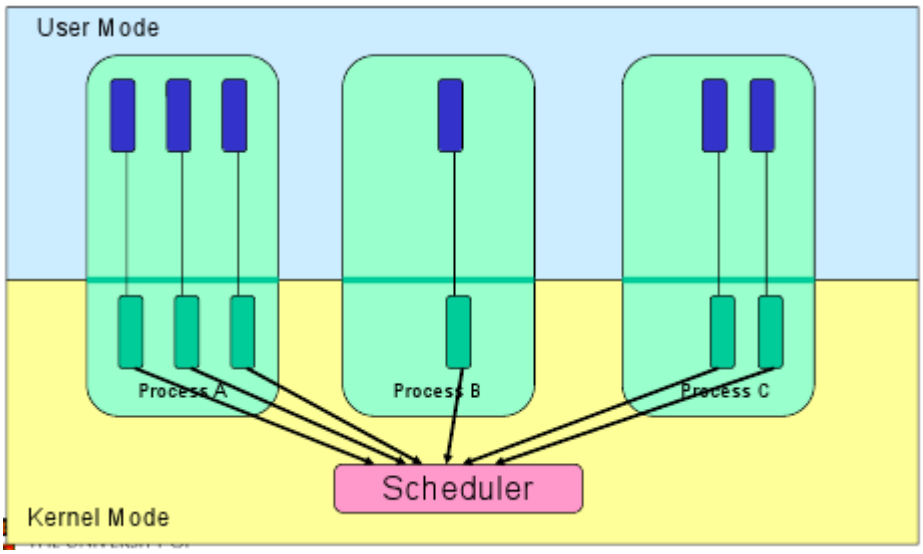
Disadvantages:

- In a typical operating system, most system calls are blocking.
- Multithreaded application cannot take advantage of multiprocessing.

Kernel Level Threads

In this case, thread management is done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling, and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.



Advantages

- The Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
- If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
- Kernel routines themselves can be multithreaded.

Disadvantages

- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

3: Scheduling

The process of scheduling is the responsibility of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

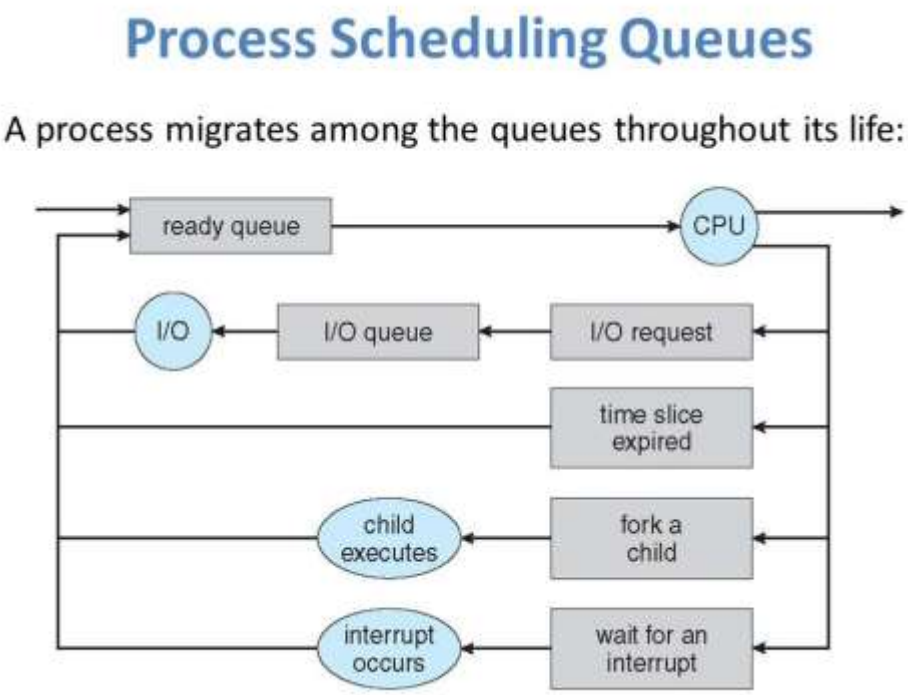
Process scheduling is an essential part of a Multiprogramming operating system. These operating systems allow more than one process to be loaded into the executable memory at a time, and the loaded process shares the CPU using time multiplexing.

The OS maintains all Process Control Blocks (PCBs) in **Process Scheduling Queues**. The OS maintains a separate queue for each of the process states, and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

The Operating System maintains the following important process scheduling queues:

- **Job queue:** This queue keeps all the processes in the system.

- **Ready queue:** This queue keeps a set of all processes residing in the main memory, ready and waiting to execute. A new process is always put in this queue.
- **Device queues:** The processes which are blocked due to unavailability of an I/O device constitute this queue.



The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.). The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system. In the above diagram, it has been merged with the CPU.

Two-state process models refer to running and non-running states:

- **Running:** When a new process is created, it enters into the system in the running state.
- **Not Running:** Processes that are not running are kept in queue, waiting for their turn to execute. Each entry in the queue is a pointer to a particular process. Queue is implemented by using a linked list. The use of dispatcher is as follows: when a process is interrupted, that process is transferred in the waiting queue. If the process has completed or aborted, the process is discarded. In either case, the dispatcher then selects a process from the queue to execute.

A **context switch** is the mechanism that stores and restores the state or context of a CPU in the Process Control block. It allows a process execution to be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential feature of a multitasking operating system.

When the scheduler switches the CPU from executing one process to another, the state from the current running process is stored into the process control block. After this, the state for the next process is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.

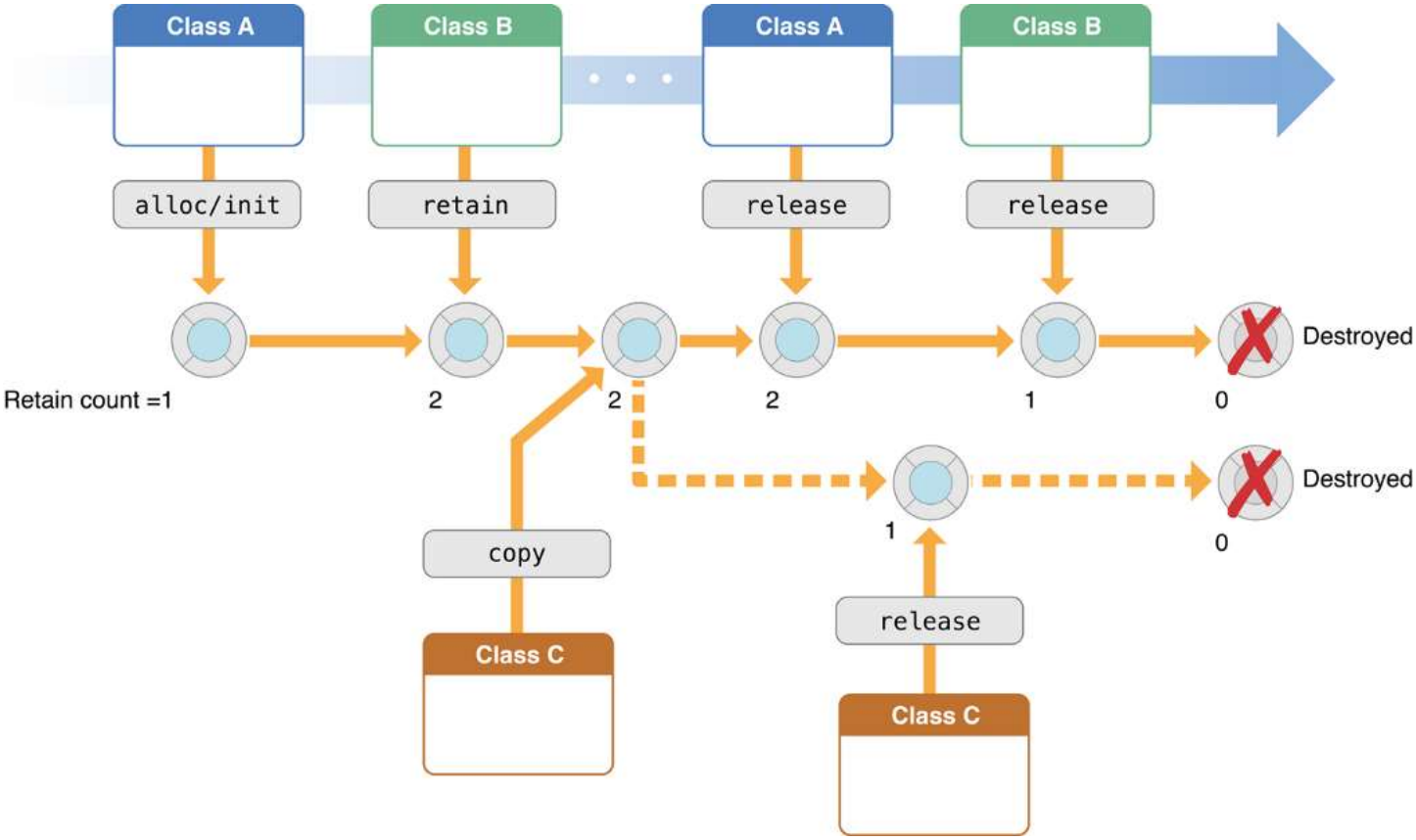
Context switches are computationally intensive, since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers.

When the process is switched, the following information is stored for later use: Program Counter, Scheduling Information, Base and Limit Register Value, Currently Used Register, Changed State, I/O State Information, and Accounting Information.

4: Memory Management

Memory management is the functionality of an operating system which handles or manages primary memory. It moves processes back and forth between the main memory and the disk during execution.

Memory management keeps track of each and every memory location, regardless of whether it is allocated to some process or free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. And it tracks whenever memory gets freed up or unallocated, and correspondingly updates the status.



The **process address space** is the set of logical addresses that a process references in its code. For example, when 32-bit addressing is in use, addresses can range from 0 to 0x7fffffff; that is, ^{231} possible numbers, for a total theoretical size of 2 gigabytes.

The operating system takes care of mapping the logical addresses to physical addresses at the time of memory allocation to the program. There are three types of addresses used in a program before and after memory is allocated:

- **Symbolic addresses**: The addresses used in a source code. The variable names, constants, and instruction labels are the basic elements of the symbolic address space.
- **Relative addresses**: At the time of compilation, a compiler converts symbolic addresses into relative addresses.

- **Physical addresses**: The loader generates these addresses at the time when a program is loaded into main memory.

Virtual and physical addresses are the same in compile-time and load-time address binding schemes. Virtual and physical addresses differ in execution-time address-binding schemes.

The set of all logical addresses generated by a program is referred to as a **logical address space**. The set of all physical addresses corresponding to these logical addresses is referred to as a **physical address space.**

5: Inter-Process Communication

There are two types of processes: Independent and Cooperating. An independent process is not affected by the execution of other processes, while a cooperating process can be affected by other executing processes.

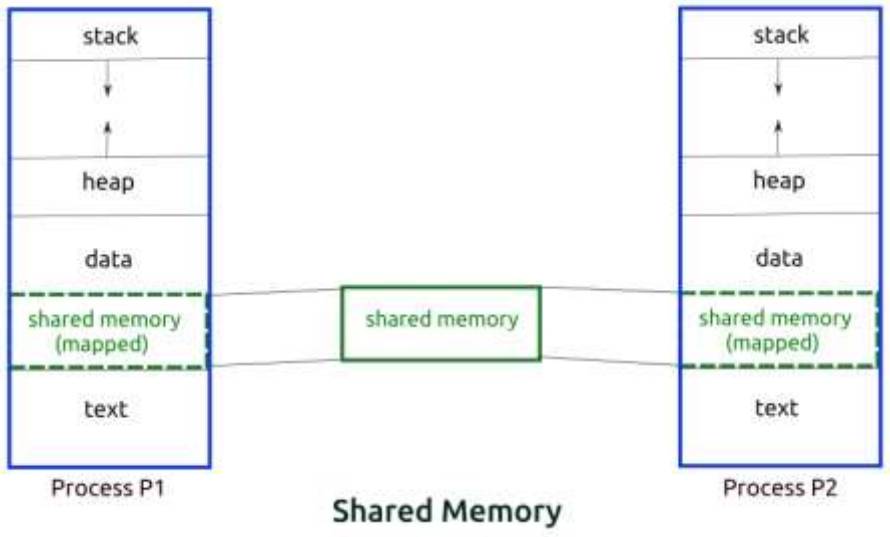
You might think that those processes, which are running independently, would execute very efficiently. But in reality, there are many situations when a process' cooperative nature can be utilized for increasing computational speed, convenience, and modularity. **Inter-process communication (IPC)** is a mechanism which allows processes to communicate with each other and synchronize their actions. The communication between these processes can be seen as a method of cooperation between them.

Processes can communicate with each other in two ways: Shared Memory and Message Parsing.

Shared Memory Method

Let's say there are two processes: the Producer and the Consumer. The producer produces some item and the Consumer consumes that item. The two processes shares a common space or memory location known as the "buffer," where the item produced by the Producer is stored and from where the Consumer consumes the item if needed.

There are two versions of this problem: the first one is known as the unbounded buffer problem, in which the Producer can keep on producing items and there is no limit on the size of the buffer. The second one is known as the bounded buffer problem, in which the Producer can produce up to a certain number of items, and after that it starts waiting for the Consumer to consume them.



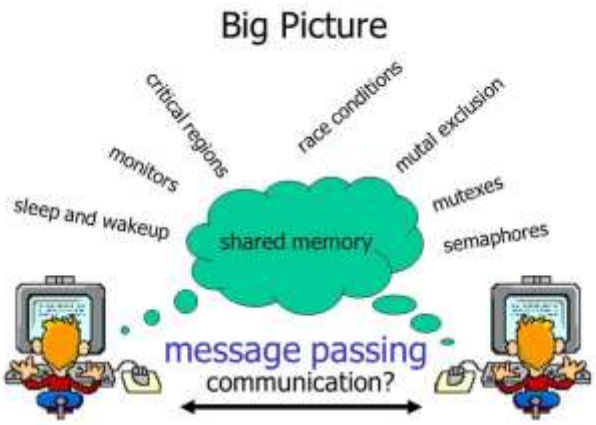
In the bounded buffer problem, the Producer and the Consumer will share some common memory. Then the Producer will start producing items. If the total number of produced items is equal to the size of buffer, the Producer will wait until they're consumed by the Consumer.

Similarly, the Consumer first checks for the availability of the item, and if no item is available, the Consumer will wait for the Producer to produce it. If there are items available, the Consumer will consume them.

Message Parsing Method

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

- 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) and **receive**(message, host) or **receive**(message)



The message size can be fixed or variable. If it is a fixed size, it is easy for the OS designer but complicated for the programmer. If it is a variable size, then it is easy for the programmer but complicated for the OS designer. A standard message has two parts: a **header** and a **body.**

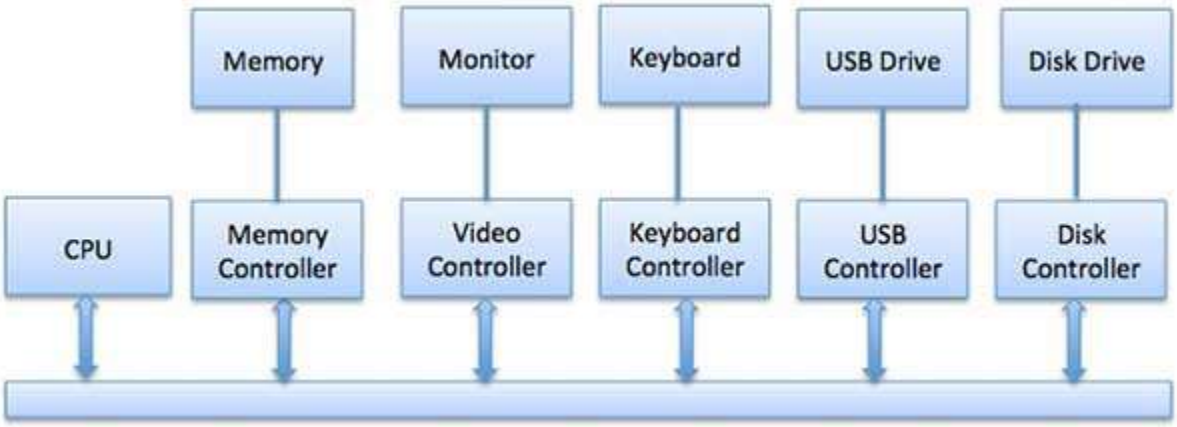
The **header** is used for storing the Message type, destination id, source id, message length, and control information. The control information contains information like what to do if it runs out of buffer space, the sequence number, and its priority. Generally, the message is sent using the FIFO style.

6: Input/Output Management

One of the important jobs of an Operating System is to manage various input/output (I/O) devices, including the mouse, keyboards, touch pad, disk drives, display adapters, USB devices, Bit-mapped screen, LED, Analog-to-digital converter, On/off switch, network connections, audio I/O, printers, and so on.

An **I/O system** is required to take an application I/O request and send it to the physical device, then take whatever response comes back from the device and send it to the application. I/O devices can be divided into two categories:

- **Block devices:** A block device is one with which the driver communicates by sending entire blocks of data. For example, hard disks, USB cameras, Disk-On-Key, and so on.
- **Character Devices:** A character device is one with which the driver communicates by sending and receiving single characters (bytes, octets). For example, serial ports, parallel ports, sounds cards, and so on.



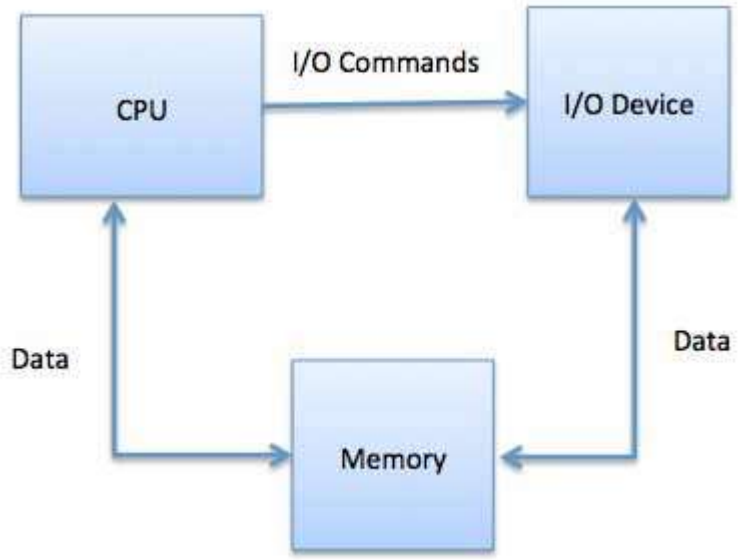
The CPU must have a way to pass information to and from an I/O device. There are three approaches available to communicate with the CPU and Device.

1. Special Instruction I/O

This uses CPU instructions that are specifically made for controlling I/O devices. These instructions typically allow data to be sent to an I/O device or be read from an I/O device.

2. Memory-mapped I/O

When using memory-mapped I/O, the same address space is shared by memory and I/O devices. The device is connected directly to certain main memory locations so that the I/O device can transfer block of data to/from the memory without going through the CPU.



While using memory mapped I/O, the OS allocates buffer in the memory and informs the I/O device to use that buffer to send data to the CPU. The I/O device operates asynchronously with the CPU, and interrupts the CPU when finished.

The advantage to this method is that every instruction which can access memory can be used to manipulate an I/O device. Memory-mapped I/O is used for most high-speed I/O devices like disks and communication interfaces.

3. Direct memory access (DMA)

Slow devices like keyboards will generate an interruption to the main CPU after each byte is transferred. If a fast device, such as a disk, generated an interruption for each byte, the operating system would spend most of its time handling these interruptions. So a typical computer uses direct memory access (DMA) hardware to reduce this overhead.

Direct Memory Access (DMA) means the CPU grants the I/O module authority to read from or write to memory without involvement. The DMA module itself controls the exchange of data between the main memory and the I/O device. The CPU is only involved at the beginning and end of the transfer and interrupted only after the entire block has been transferred.

Direct Memory Access needs special hardware called a DMA controller (DMAC) that manages the data transfers and arbitrates access to the system bus. The controllers are programmed with source and destination pointers (where to read/write the data), counters to track the number of transferred bytes, and various settings. These include the I/O and memory types and the interruptions and states for the CPU cycles.

7: Virtualization

Virtualization is technology that allows you to create multiple simulated environments or dedicated resources from a single, physical hardware system. Software called a **hypervisor** connects directly to that hardware and allows you to split one system into separate, distinct, and secure environments known as **virtual machines** (VMs). These VMs rely on the hypervisor's ability to separate the machine's resources from the hardware and distribute them appropriately.

The original, physical machine equipped with the hypervisor is called **the host**, while the many VMs that use its resources are called **guests**. These guests treat computing resources — like CPU, memory, and storage — as a hangar of resources that can easily be relocated. Operators can control virtual instances of CPU, memory, storage, and other resources so that guests receive the resources they need when they need them.

Ideally, all related VMs are managed through a single web-based virtualization management console, which speeds things up. Virtualization lets you dictate how much processing power, storage, and memory to give to VMs, and environments are better protected since VMs are separated from their supporting hardware and each other. Simply put, virtualization creates the environments and resources you need from underused hardware.

Types of Virtualization:

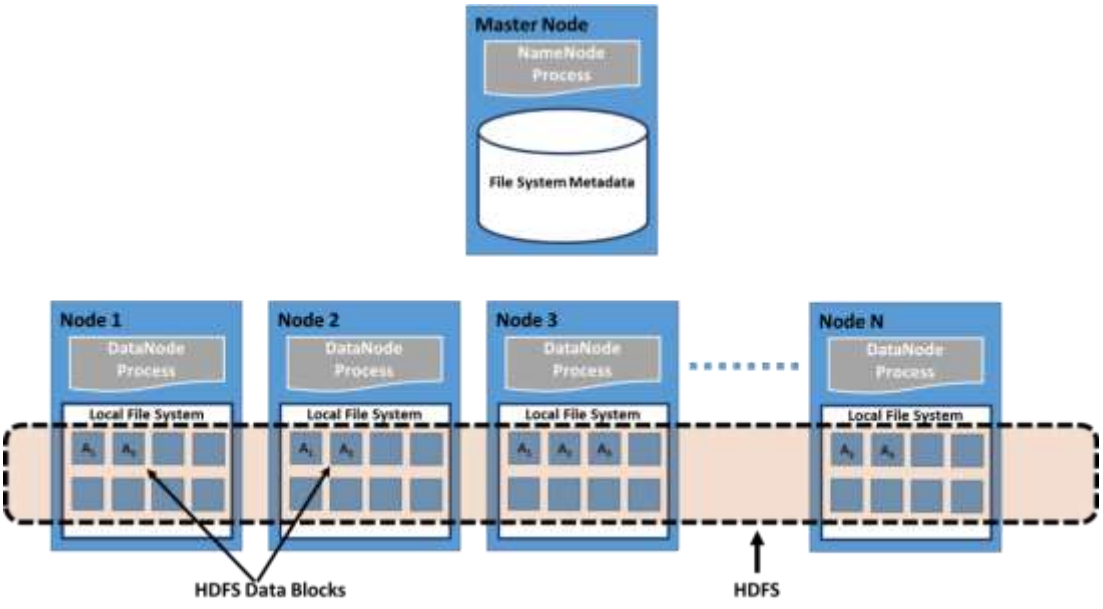
1. **Data Virtualization:** Data that's spread all over can be consolidated into a single source. Data virtualization allows companies to treat data as a dynamic supply — providing processing capabilities that can bring together data from multiple sources, easily accommodate new data sources, and transform data according to user needs. Data virtualization tools sit in front of multiple data sources and allow them to be treated as single source. They deliver the needed data — in the required form — at the right time to any application or user.

2. **Desktop Virtualization:** Easily confused with operating system virtualization — which allows you to deploy multiple operating systems on a single machine — desktop virtualization allows a central administrator (or automated administration tool) to deploy simulated desktop environments to hundreds of physical machines at once. Unlike traditional desktop environments that are physically installed, configured, and updated on each machine, desktop virtualization allows admins to perform mass configurations, updates, and security checks on all virtual desktops.
3. **Server Virtualization:** Servers are computers designed to process a high volume of specific tasks really well so other computers — like laptops and desktops — can do a variety of other tasks. Virtualizing a server lets it to do more of those specific functions and involves partitioning it so that the components can be used to serve multiple functions.
4. **Operating System Virtualization:** Operating system virtualization happens at the kernel — the central task managers of operating systems. It's a useful way to run Linux and Windows environments side-by-side. Enterprises can also push virtual operating systems to computers, which: (1) Reduces bulk hardware costs, since the computers don't require such high out-of-the-box capabilities, (2) Increases security, since all virtual instances can be monitored and isolated, and (3) Limits time spent on IT services like software updates.
5. **Network Functions Virtualization:** Network functions virtualization (NFV) separates a network's key functions (like directory services, file sharing, and IP configuration) so they can be distributed among environments. Once software functions are independent of the physical machines they once lived on, specific functions can be packaged together into a new network and assigned to an environment. Virtualizing networks reduces the number of physical components — like switches, routers, servers, cables, and hubs — that are needed to create multiple, independent networks, and it's particularly popular in the telecommunications industry.

8 — Distributed File Systems

A distributed file system is a client/server-based application that allows clients to access and process data stored on the server as if it were on their own computer. When a user accesses a file on the server, the server sends the user a copy of the file, which is cached on the user's computer while the data is being processed and is then returned to the server.

Ideally, a distributed file system organizes file and directory services of individual servers into a global directory in such a way that remote data access is not location-specific but is identical from any client. All files are accessible to all users of the global file system and organization is hierarchical and directory-based.



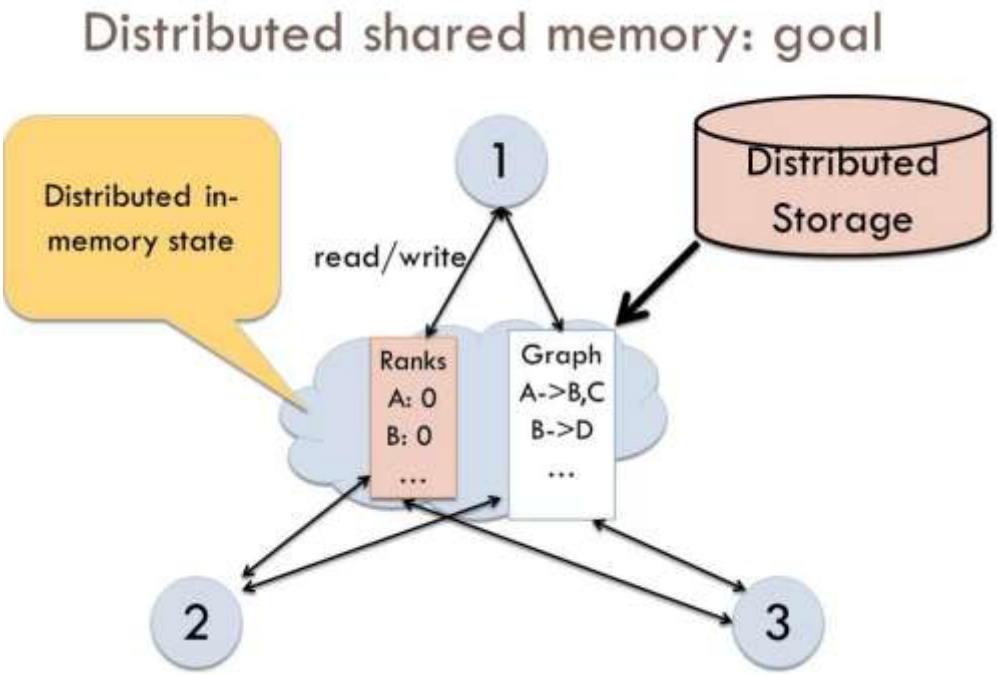
Since more than one client may access the same data simultaneously, the server must have a mechanism in place (such as maintaining information about the times of access) to organize updates so that the client always receives the most current version of data and that data conflicts do not arise. Distributed file systems typically use file or database replication (distributing copies of data on multiple servers) to protect against data access failures.

Sun Microsystems' Network File System (NFS), Novell NetWare, Microsoft's Distributed File System, and IBM's DFS are some examples of distributed file systems.

9 — Distributed Shared Memory

Distributed Shared Memory (DSM) is a resource management component of a distributed operating system that implements the shared memory model in distributed systems, which have no physically shared memory. The shared memory provides a virtual address space that is shared among all computers in a distributed system.

In DSM, data is accessed from a shared space similar to the way that virtual memory is accessed. Data moves between secondary and main memory, as well as, between the distributed main memories of different nodes. Ownership of pages in memory starts out in some pre-defined state but changes during the course of normal operation. Ownership changes take place when data moves from one node to another due to an access by a particular process.



Advantages of Distributed Shared Memory:

- Hide data movement and provide a simpler abstraction for sharing data. Programmers don't need to worry about memory transfers between machines like when using the message passing model.
- Allows the passing of complex structures by reference, simplifying algorithm development for distributed applications.
- Takes advantage of "locality of reference" by moving the entire page containing the data referenced rather than just the piece of data.

- Cheaper to build than multiprocessor systems. Ideas can be implemented using normal hardware and do not require anything complex to connect the shared memory to the processors.
- Larger memory sizes are available to programs, by combining all physical memory of all nodes. This large memory will not incur disk latency due to swapping like in traditional distributed systems.
- Unlimited number of nodes can be used. Unlike multiprocessor systems where main memory is accessed via a common bus, thus limiting the size of the multiprocessor system.
- Programs written for shared memory multiprocessors can be run on DSM systems.

There are two different ways that nodes can be informed of who owns what page: invalidation and broadcast. Invalidation is a method that invalidates a page when some process asks for write access to that page and becomes its new owner. This way the next time some other process tries to read or write to a copy of the page it thought it had, the page will not be available and the process will have to re-request access to that page. Broadcasting will automatically update all copies of a memory page when a process writes to it. This is also called write-update. This method is a lot less efficient more difficult to implement because a new value has to sent instead of an invalidation message.

10 — Cloud Computing

More and more, we are seeing technology moving to the cloud. It's not just a fad — the shift from traditional software models to the Internet has steadily gained momentum over the last 10 years. Looking ahead, the next decade of cloud computing promises new ways to collaborate everywhere, through mobile devices.

So what is cloud computing? Essentially, cloud computing is a kind of outsourcing of computer programs. Using cloud computing, users are able to access software and applications from wherever they need, while it is being hosted by an outside party — in "the cloud." This means that they do not have to worry about things such as storage and power, they can simply enjoy the end result.

Traditional business applications have always been very complicated and expensive. The amount and variety of hardware and software required to run them are daunting. You need a whole team of experts to install, configure, test, run, secure, and update them. When you multiply this effort across dozens or hundreds of apps, it isn't easy to see why the biggest companies with the best IT departments aren't getting the apps they need. Small and mid-sized businesses don't stand a chance.

Source: https://data-notes.co/how-operating-systems-work-10-concepts-you-should-know-as-a-developer-8d63bb38331f

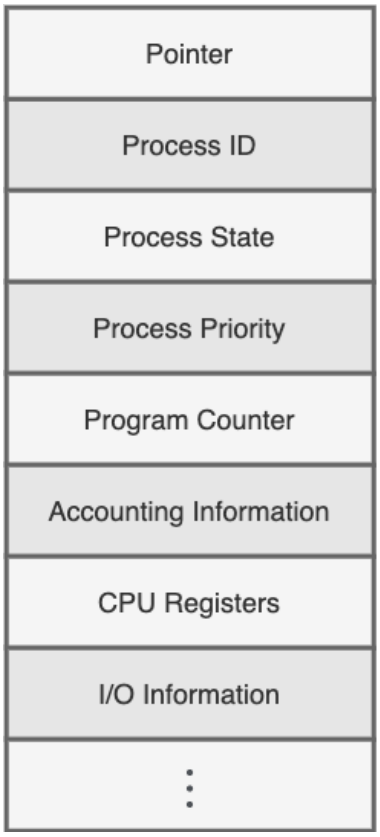
What's a process? https://mattchw.medium.com/overview-of-operating-system-cc3f6f6bb062

A **process** is a program in execution. It contains every information of that running program:

- Current program counter
- Accumulated running time
- List of files that are currently opened by that program
- Page table

What's Process Control Block(PCB)?

Every process is represented by a data structure called **Process Control Block(PCB)**. A PCB keeps all the important information of a process.

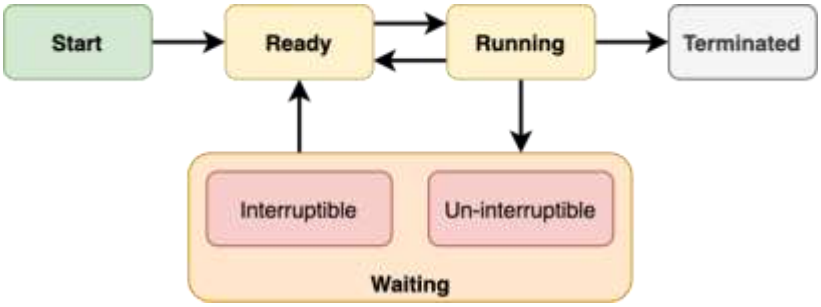


Process Control Block

- **Pointer**: A pointer to the parent process.
- **Process ID**: Unique ID number representing a process.
- **Process State**: The state any process currently is in (Ready, Wait, Exit, etc).
- **Process Priority**: Numeric value which states how urgent a process is. Process with the highest priority will be allocated to the CPU first.
- **Program Counter**: Address of the next instruction in line of the processes.
- **Accounting Information**: The amount of CPU used, time limits, job or process numbers, etc.
- **CPU Registers**: Accumulators, index registers, general-purpose registers, etc.
- **I/O Information**: An array of open files and I/O devices allocated to the process.

What's Process Lifecycle?

There is a lifecycle of every process. The states of a process are Start, Ready, Running, Waiting and Terminated.



Process Lifecycle

Start: The birth of a process.
Except the first process "init", every process is created using **fork()**.

Ready: The process is ready.
It means it's ready to run but is not running. A process may become "ready" (runnable) after:

- It is just created by **fork()**
- It has been running on the CPU for some time and the OS chooses another process to run (scheduled context switch)
- Returning from blocked states

Running: The process is running.
The OS chooses this process to be running on the CPU and changes its state to "Running".
Waiting/Blocking: The process is blocked.
While the process is running, it may wait for something (e.g. getc(), wait()). There are 2 types of wait, interruptible and un-interruptible.

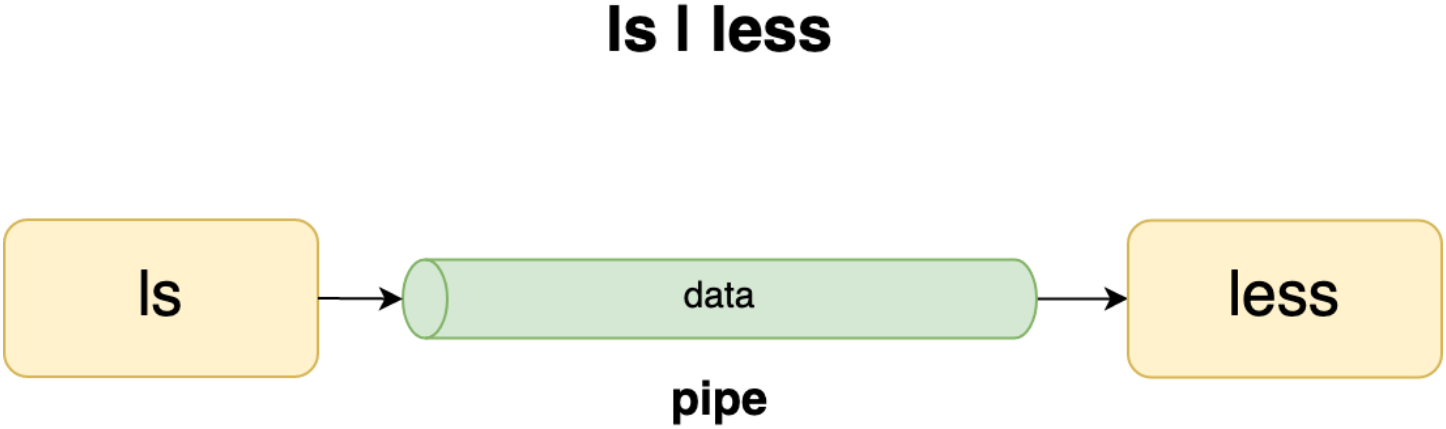
- **Interruptible wait**: Sometimes, a process has to wait for the response from a device and, therefore, it is blocked. This blocking state is interruptible. It means pressing "Ctrl + C" can get the process out of the waiting state.
- **Un-Interruptible wait**: Sometimes, a process needs to wait for a resource until it really gets the response. It's in an Un-interruptible status. It means it won't be "Ctrl + C" interruptible.

Terminated: The process is going to die.
The process may choose to terminate itself or force to be terminated.

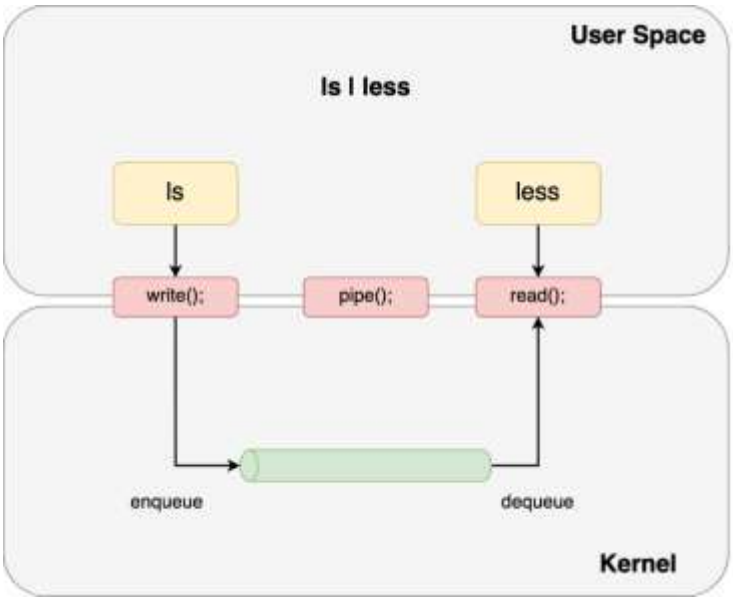
Inter-process communication (IPC) is simply a mechanism for the processes to manage shared data. The most commonly used object to share data between two processes is pipe.

What's a pipe?

Pipe is a shared object between two processes which data from one to another. It is unidirectional, which means data can only flow from left to right.



Example of Inter-process Communication
Deep down in the kernal, the pipe() system call creates a piece of shared FIFO queue in the kernel space.



In this example, the “ls” process is a producer and it writes data into the pipe. The “less” process is a consumer and it reads the data from the pipe.
However, you may realise that there could be a problem. The buffer’s size is limited. What if “ls” produces too fast while “less” consumes too slow?

It’s called **producer-consumer problem** and it’s a synchronization problem. Producer-consumer problem is one of the classic IPC problems and I will talk about it later.
As user space memory is not sharable between processes even they are parent and child, shared objects are therefore provided at the kernel level. There are other IPC problems beyond pipe. You may have to use shared memory and shared files directly. The concurrent access may cause unpredictable outcomes and Kernel won’t help you to take care of it.

Race condition

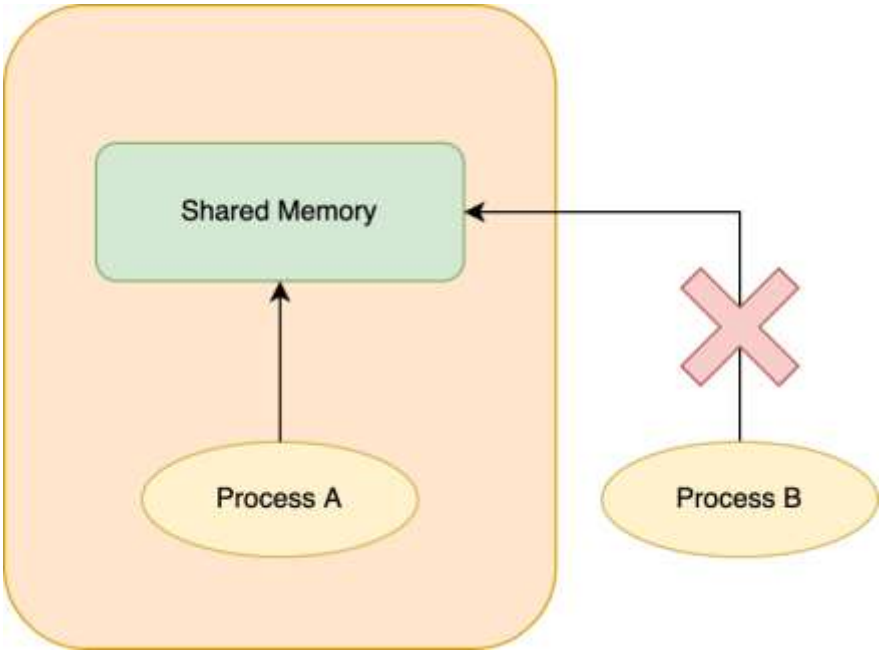
When processes share a same memory together, we could not skip talking about race condition. It could become a bug if we don’t take a good care of it.
A race condition means the outcome of an execution depends on a particular order in which the shared resource is accessed. It may happen when:

1. there are shared objects
2. there are multiple processes
3. the processes are concurrently accessing shared objects

Race condition is always a nightmare as it’s hard to debug.

How to resolve race condition on a shared object?

Mutual Exclusion



Mutual Exclusion

When I'm playing with the shared memory, no one could touch it. A set of processes would not have the problem of race condition if *mutual exclusion is guaranteed*.
Bear in mind. Shared object is still sharable, but:

- the shared object could not be shared **at the same time**

- it has to be shared one by one

To Achieve mutual exclusion, the program code could be divided into 3 sections:

1. **Section Entry**: start of critical section. Telling other process that the shared memory is in used.
2. **Critical Section**: code segment that is accessing the shared object. It should as tight as possible.
3. **Section Exit**: end of critical section. Telling other process that the shared memory is now free.

In short, mutual exclusion is achieved by blocking other processes from entering a critical section if a process is already accessing the critical section.

How to achieve mutual exclusion by programming?

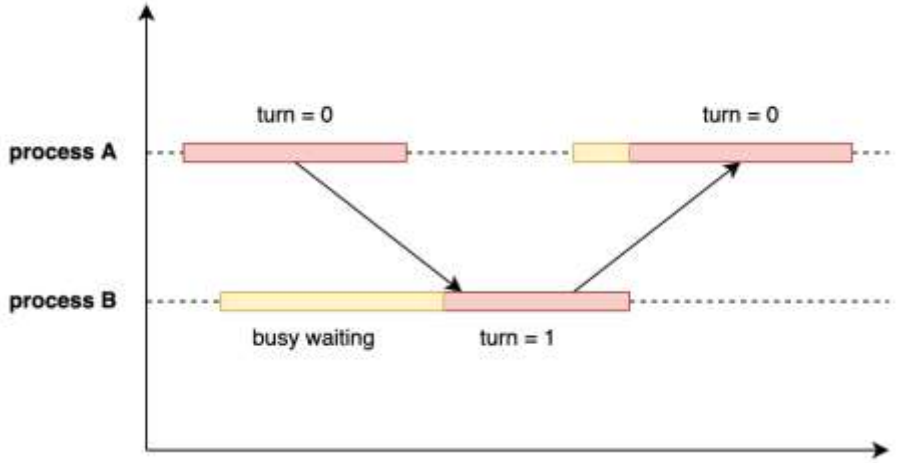
Spin lock

It is to loop on another shared object, "**turn**", to detect the status of other processes.
For example, Process A:

```
while (TRUE) {
  while( turn != 0 ) /* busy waiting */  ----------------------
  critical_section();  ----------------------

  turn = 1;

  remainder_section();
}
```

Process B:

```
while (TRUE) {
  while( turn != 1) /* busy waiting */----------------------
  critical_section();----------------------

  turn = 0;

  remainder_section();
}
```



Example of spin lock