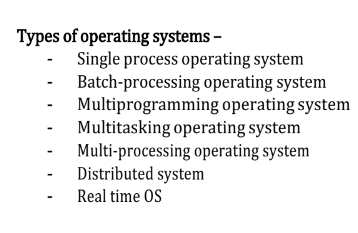
An operating system is a piece of software that manages all the resources of a computer system, both hardware and software, and provides an environment in which the user can execute his/her programs in a convenient and efficient manner by hiding underlying complexity of the hardware and acting as a resource manager.

Why OS is required:

* Access to the computer hardware only through OS. No hardware access given to external apps and software.
* Acts as interface between the user and the computer hardware
* Resource management (Aka, Arbitration) (memory, device, file, security, process etc) done by OS for all apps and software running on the system.
* Hides the underlying complexity of the hardware and (Aka, Abstraction) facilitates execution of application programs by providing isolation and protection.



**Single process OS**, only 1 process executes at a time from the ready queue. [Oldest]

**Batch-processing OS**,

1. Firstly, user prepares his job using punch cards.

2. Then, he submits the job to the computer operator.

3. Operator collects the jobs from different users and sort the jobs into batches with similar needs.

4. Then, operator submits the batches to the processor one by one.

5. All the jobs of one batch are executed together i.e. jobs are executed batch wise.

- Priorities cannot be set, if a job comes with some higher priority.

- May lead to starvation. (A batch may take more time to complete)

- CPU may become idle in case of I/O operations.

**Multi-programming** : If one job gets busy with I/o or wait state, CPU can execute other jobs meanwhile.

- Single CPU

- Context switching for processes.

- Switch happens when current process goes to wait state.

- CPU idle time reduced.

**Multitasking** is a logical extension of multiprogramming.

- Single CPU

- Runs more than one task simultaneously.

- Context switching and time sharing used.

- Increases responsiveness of each process.

- Priority processes executed first.

- CPU utilised the most.

**Multi-processing OS**- more than 1 CPU in a single computer.

- Increases reliability, 1 CPU fails, other can work

- Better throughput.

- Lesser process starvation, (if 1 CPU is working on some process, other can be executed on other CPU.

**Distributed OS**- OS manages many bunches of resources, >=1 CPUs, >=1 memory, >=1 GPUs, etc

- Loosely connected autonomous, interconnected computer nodes.

- Collection of independent, networked, communicating, and physically separate computational nodes.

**RTOS**- Real time operating system. When error free computation is required and no chance of error can be taken and fastest computation required. Eg: used in Air Traffic control (ATC)

**Program**: A Program is an executable file which contains a certain set of instructions written to complete the specific job or operation on your computer.

• It’s a compiled code. Ready to be executed.

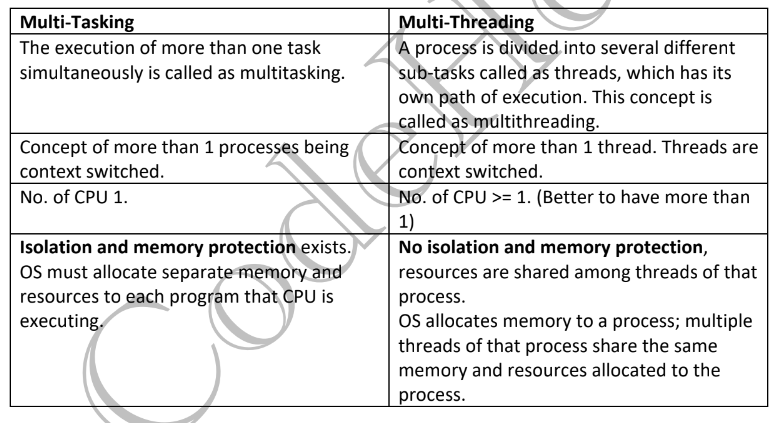
• Stored in Disk

**Process**: When a program/app comes in RAM, it is called a process. When we double click it, it’s .exe file executes and it comes in RAM.

**Thread**: an independent process within a process. Also called light weight process(LWP). Used for achieving parallelism. Multiple threads within a process share same resources and multithreading is beneficial only with >1 CPUs (Multi core system). E.g., Multiple tabs in a browser, text editor (When you are typing in an editor, spell- checking, formatting of text and saving of text are done concurrently using multiple threads. Stack memory is separate for each thread in memory layout, heap memory is same.

**Benefits:**

1. Responsiveness – eg: in an interactive app, multiple things can be done simultaneously and give faster results.
2. If instead of diving a process into threads, it is divided into multiple processes, interprocess comm. Will be required to communicate b/t those processes, tedious task, time taking.
3. Creating threads is more economical than processes.



Thread Scheduling: Threads are scheduled for execution based on their priority. Even though threads are executing within the runtime, all threads are assigned processor time slices by the operating system.

**Differences:**

|  |  |
| --- | --- |
| Thread context switching | Process context switching |
| Resources, memory is not changed. | Resources are changed since each process has its own resources. |
| Fast switching | Slow switching |
| Cache is not cleared, since same cache is used by all threads of same process | Cache is cleared |

**Critical section**: refers to the segment of code where processes/threads access shared resources, such as common variables and files, and perform write operations on them.

**Race condition**: when two+ threads try to access and change the same data value simultaneously. Since thread scheduler can swap threads at any time, the order of accessing the shared resources changes and gives different results every time.

**Solutions to avoid race condition:**

* Atomic <int> at; variable in C++ stores the complete thread execution and only then thread switching happens. This avoids race condition.
* Locks/Mutex- mutex is a locking mechanism, Only one task can acquire the mutex. It means there is ownership associated with a mutex, and only the owner can release the lock (mutex).

Disadvantages:

1. **Contention**: if one thread acquired lock and died, the lock will not be released ever and other threads will wait forever.
2. Starvation of high priority threads forever.

**Semaphores:** A semaphore is an integer variable S that is initialized with the number of resources present in the system and used for process synchronization.  When multiple instances of same resource is there so multiple threads of a process can access the resources at the same time. Semaphore uses two atomic operations for process synchronisation:

* Wait (P)- check whether instances are available, if yes, decrement value of semaphore
* Signal (V)- when a thread leaves the memory instance, it signals the other threads.

**Kernel**: A kernel is that part of the operating system which interacts directly with the hardware and performs the most crucial tasks.

a. Heart of OS/Core component

b. Very first part of OS to load on start-up.

2. **User space:** Where application software runs, apps don’t have access to the underlying hardware. It interacts with the hardware through kernel.

a. GUI: graphical user interface

b. CLI: command line interface

Functions of kernel:

1. Process management:

* Scheduling processes and threads on CPU.
* Suspending and resuming processes.
* Proving mechanism for process synchronisation and communication

1. Memory management

* Allocating and deallocating memory
* Keeping track of which memory part is used by which process.

1. File management

* Create delete files/directories.
* Mapping files into secondary storage

1. I/O management:

* Buffering (copying data between 2 devices
* Caching: memory/ web caching
* Buffering : Youtube buffering

Two modes are there: user mode and kernel mode, transition between them happens through software interrupt. Eg: when typing on terminal: mkdir dir1- happens in user mode. After pressing Enter- it switches to kernel mode to actually create a directory. Mkdir is a system call.

**System calls**: With the help of system calls apps interact with kernel. A system call is a mechanism using which a user program can request a service from the kernel for which it does not have the permission to perform. System calls are written in C.

Types of System Calls:

1) Process Control

* end, abort- fork(), exit(), wait()
* load, execute
* create process, terminate process
* get process attributes, set process attributes
* wait for time
* wait event, signal event
* allocate and free memory

2) File Management

* create file, delete file- open(), read(), write(), close(), chmod()
* open, close
* read, write, reposition
* get file attributes, set file attributes

3) Device Management

* request device, release device
* read, write, reposition
* get device attributes, set device attributes
* logically attach or detach devices

4) Information maintenance

* get time or date, set time or date
* get system data, set system data
* get process, file, or device attributes- getpid()
* set process, file, or device attributes

5) Communication Management

* create, delete communication connection
* send, receive messages
* transfer status information
* attach or detach remote devices

pid- process id, chmod- change mode- read, write. vim hello.txt- creates a doc in terminal.

**What happens after we turn on the computer?**

1. PC on
2. Cpu initalises a BIOS/UEFI program stored in bios chip.
3. BIOS -tests and initializes system hardware. Bios loads configuration settings. If something is not appropriate (like missing RAM) error is thrown and boot process is stopped.
4. BIOS searches for a program which initialises OS. is a small program that has the large task of booting the OS stored in MBR/EFI. Windows, mac and linux have different bootloaders.

* 1 byte= 8 bits
* 4 byte register= 32 bit register
* 232=4 GB
* A 32 bit systems adds 64 bit data in 2 cycles.
* A 32-bit OS has 32-bit registers, and it can access 2^32 unique memory addresses. i.e., 4GB of physical memory.
* A 64-bit OS has 64-bit registers, and it can access 2^64 unique memory addresses. i.e., 17,179,869,184 GB of physical memory.
* 64 bit os can run both 32 AND 64 bit OS.

Memory storage in CPU:

* **Register**: Smallest unit of storage. It is a part of CPU itself. A register may hold an instruction, a storage address, or any data (such as bit sequence or individual characters). Registers are a type of computer memory used to quickly accept, store, and transfer data and instructions that are being used immediately by the CPU.
* **Cache**: Additional memory system that temporarily stores frequently used instructions and data for quicker processing by the CPU.
* **Main Memory**: RAM.
* **Secondary Memory**: Storage media, on which computer can store data & programs.

1. Cost:

* Primary storages are costly.
* Registers are most expensive due to expensive semiconductors & labour.
* Secondary storages are cheaper than primary.

2. Access Speed:

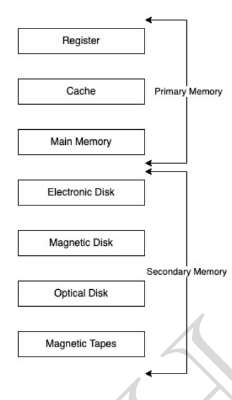
* Primary memory (cache) has higher access speed than secondary memory (main memory).
* b. Registers has highest access speed, then comes cache, then main memory.

3. Storage size:

* Secondary has more space.

4. Volatility:

* Primary memory is volatile.
* Secondary is non-volatile.



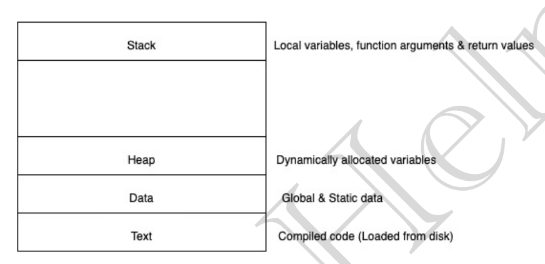
**Process:**

How OS creates a process? Converting program into a process.

STEPS:

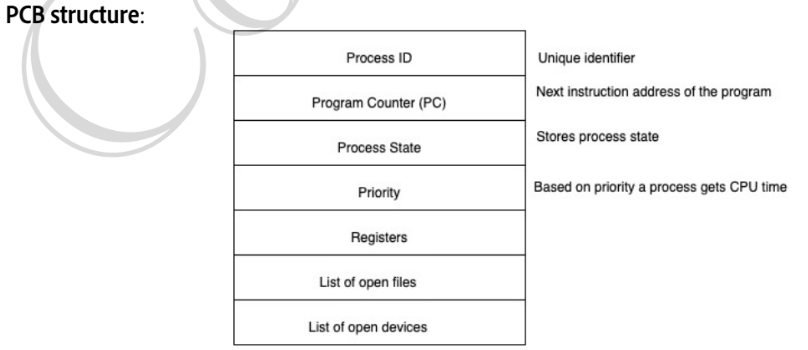
* + Load the program & static data into memory.
  + Allocate runtime stack- for local variable, runtime functions
  + Heap memory allocation- for dynamically allocating memory.
  + IO tasks- input, output, error handling.
  + OS handoffs control to main ().

Architecture of process:



**Attributes of a process:**

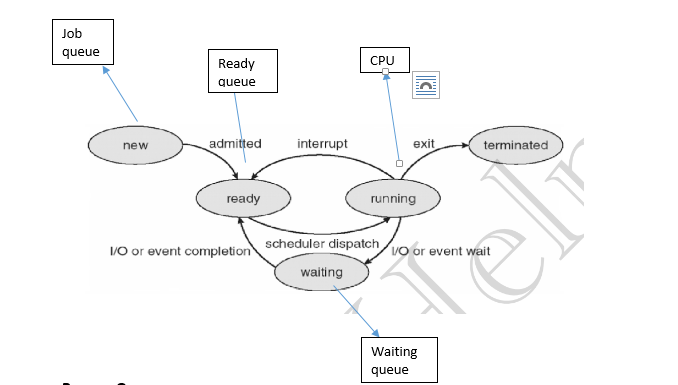
* Each process attributes are stored in PCB (process control block). OS identifies each process with its attributes in PCB.



Process states:

As process executes, it changes state. Each process may be in one of the following states.

* New: OS is about to pick the program & convert it into process. OR the process is being created.
* Run: Instructions are being executed; CPU is allocated.
* Waiting: Waiting for IO.
* Ready: The process is in memory, waiting to be assigned to a processor. Picking the job from ‘New’ or from ready queue to ready state is done by job scheduler.
* Terminated: The process has finished execution. PCB entry removed from process table.



**Process queues**

a. Job Queue:

* Processes in new state.
* Present in secondary memory.
* Job Scheduler (Long term scheduler (LTS)) picks process from the pool and loads them into ready queue for execution.

b. Ready Queue:

* + Processes in Ready state.
  + Present in main memory.
  + CPU Scheduler (Short-term scheduler) picks process from ready queue and dispatches it to CPU.

c. Waiting Queue:

* + Processes in Wait state.

LTS- controls degree of multi programming.

STS- Short-term scheduler

MTS- medium term scheduler

**Swapping:**

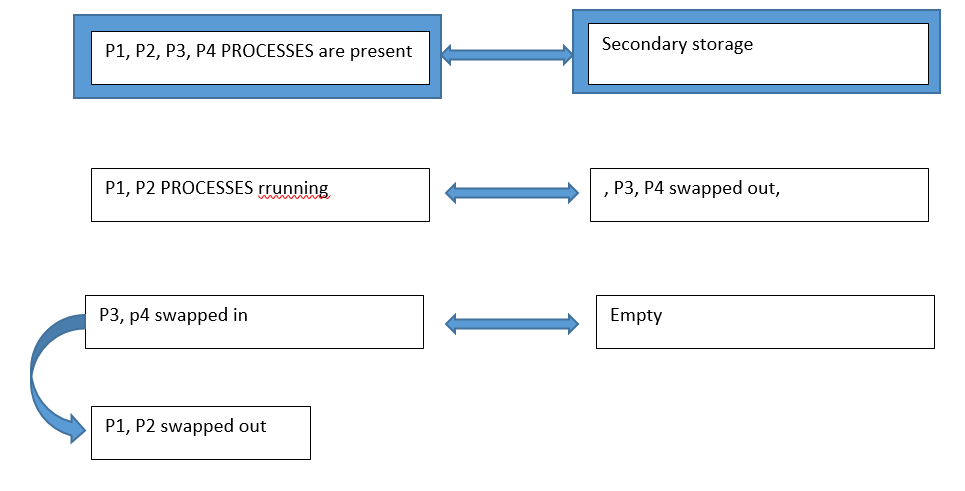
Done by MTS to avoid ready queue from overloading/ reduce degree of multiprogramming.

Done when ready queue gets exhausted due to many memory intensive processes in it.

Swapping is done by swapping some processes which are not been currently used, they are swapped with secondary storage memory. Swapping is done while their preserving their state/work.

Once the work of currently used processes is done, they are removed from the queue and rest of the processes from the secondary storage are swapped in ready queue again.

Eg: p1, p2,p3, p4 processes are there.



**Context switching**

a. Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process.

b. When this occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.

c. It is pure overhead, because the system does no useful work while switching.

d. Speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied etc.

e. done by kernel.

**Orphan process**

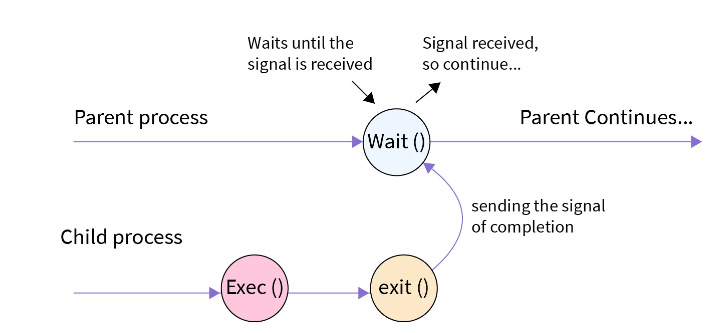
Every process is generated from a parent process through fork() command. A parent only exists when its child process exits or ends. It reads it return statement to know whether it has executed successfully or not. If it is executed successfully then only its process id is removed from parent’s process table.

In a case when the parent process exits arbitrarily, the child process becomes orphan. In this case OS assigns init() (PID=1) as its parent.

**Process table is a data structure in RAM to store information about a process.**

**Zombie process**

* + It is a process whose execution is completed but still has its entry in the process table.
  + When a child process waits for its parent process to call wait() (system call) and read its return status, so that it can exit from the process table.
  + As entry in the process table can only be removed, after the parent process reads the exit status of child process. Hence, the child process remains a zombie till it is removed from the process table.
  + If in case, parent process does not call wait to read the child process return statement, or parent table exits without reading child process, process table gets over loaded.



Zombie processes in OS are harmful as:

* OS has one process table of finite size only, lots of zombie processes will result in a full process table.
* A full process table means that OS cannot create a new process when required and Zombie processes in OS are of no use as the process has died already but its entry is present.

**Process scheduling in OS**

**CPU Scheduler**

1. Whenever the CPU become ideal, OS must select one process from the ready queue to be executed.
2. Done by STS.

**Non pre-emptive scheduling**

1. Once the CPU is allocated to a process, process keeps the CPU until it is terminated or process switches to wait state.
2. Starvation to other processes can happen if CPU intensive process has taken the CPU.
3. Low CPU utilization.

**Pre-emptive scheduling**

1. CPU is taken away from processes after time quantum expires along with terminating and switching to wait state.
2. More CPU utilization

Goals of CPU scheduling

* Max cpu utilization
* Minimum time spent for each process from ready queue to getting executed. (turn around time)(TAT)
* Min wait time
* Min response time for each process- time when first time CPU is allotted to any process
* Max throughput of system: max number of processes executed.
* Burst time: time required by the process to complete
* Arrival rime: time when process arrives at ready queue.
* Turn around time= completion time- arrival time
* Wait time: Turn around time – burst time

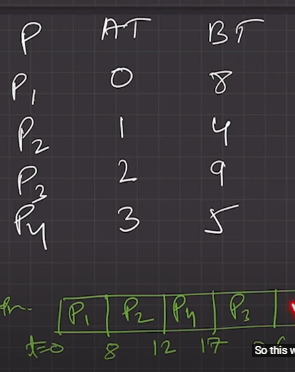
**Scheduling algorithms in operating systems:**

**First come First serve(FCFS)**

* Whichever process comes first will be given the CPU.
* This can cause **convoy effect**, where CPU intensive job comes first and takes the CPU for long time and rest processes which doesn’t require that much CPU, will have to wait for long.

**Shortest Job First (SJF) [Non-preemptive]**

* 1. Process with least Burst Time will be dispatched to CPU first.
  2. Must do estimation for BT for each process in ready queue beforehand, correct estimation of BT is an impossible task (ideally.)
  3. Criteria for SJF algo- Arrival Time + Burst Time. (first come first serve+ lowest burst time)
  4. Convoy effect and Process starvation might happen if first process which came in Ready state is having a large BT.



**SJF [Preemptive]**

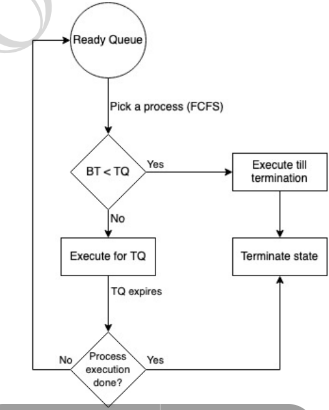
a. Less starvation.

b. No convoy effect.

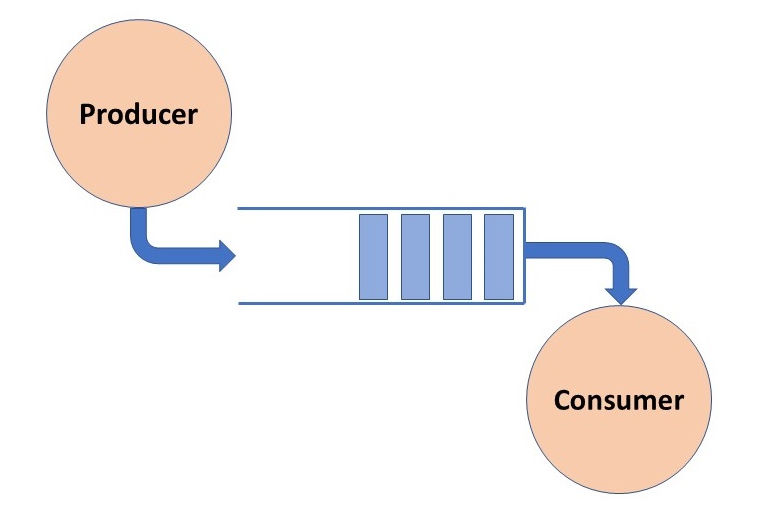
c. Gives average WT less for a given set of processes as scheduling short job before a long one decreases the WT of short job more than it increases the wait time of long jobs.

**Round robin scheduling**

1. Most popular
2. Like FCFS but pre-emptive
3. Designed for time sharing system
4. Criteria: AT+TQ, doesn’t depend on BT.
5. Low starvation since no process is going to wait forever
6. Is TQ is small , more context switching.( overhead more )
7. Process in ready queue is picked and run for a time quantum duration of time. If TQ >BT, it is again fed to ready queue, if TQ<BT, process executed till termination.



**Producer consumer problem**



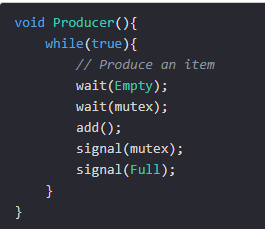
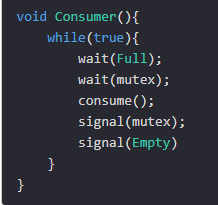
**Conditions to be met:**

1. The Producer process must not produce an item if the shared buffer is full.
2. The Consumer process must not consume an item if the shared buffer is empty.
3. Access to the shared buffer must be **mutually exclusive**; this means that at any given instance, only one process should be able to access the shared buffer and make changes to it.

**Solution**

The solution to the Producer-Consumer problem involves three *semaphore* variables.

* **semaphore Full**: Tracks the space filled by the Producer process. initialized 0 as the buffer will have 0 filled spaces initially.
* **semaphore Empty**: Tracks the empty space in the buffer. initially set= **buffer\_size**.
* **semaphore mutex**: Used for mutual exclusion so that only one process can access the shared buffer at a time.
* **Signal()** – increases the value by 1
* **Wait()**- decreases the value by 1

Producer waits if the empty value is 0- i.e. if buffer is full, it cannot produce and add more.

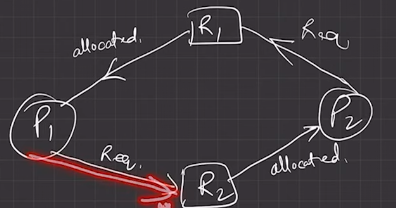
* Wait(mutex) ensures only one thread is accessing the resources, it locks the source.
* Add() – adds in the buffer.
* Signal(mutex) releases the mutex so that other threads can access.
* Signal(full)- increases the value of full by 1.

**Deadlock**

1. When a process has requested a resource but the resource is occupied already and the process goes into infinite wait since the resource never gets free.
2. DL is a bug present in the process/thread synchronization method.
3. Processes never finish executing, and the system resources are tied up, preventing other jobs from starting.

**Conditions resulting into deadlock:**

1. Mutual exclusion
2. Hold and wait:when p1 took r1 but is waiting for r2, and p2 held r2 but is waiting for r1.



1. No preemtion: until one process is complete, it does not release a resource, no preemtion should happened.

**Deadlock avoidance algo.- Bankers algo**

**Deadlock Avoidance:** Idea is, the kernel be given in advance info concerning which resources will use in its lifetime. By this, system can decide for each request whether the process should wait.

Info given to kernel: **total resources available, resources allocated, remaining need, max need of every process.**

When a process requests resources, the system should check whether after allocating these resources will the system be in safe state, if yes only then allocate these resources otherwise wait till other processes release resources.

**Safe state**- when system is able to allocate resources to the process and still avoid DL.

**Memory management**

**Logical Address**

* + An address generated by the CPU.
  + The logical address is basically the address of an instruction or data used by a process.
  + User has indirect access to the physical address through logical address.
  + Range: 0 to max

**Physical address**

* Address loaded into memory address register of physical memory.
* Physical address is only accessible through logical address.
* Physical address is in memory unit.
* Memory management unit maps physical to logical address.

**How OS manages free space:**

**Defragmentation/Compaction**

* + Dynamic partitioning suffers from external fragmentation- free spaces get divided into small-small separate parts.
  + All the free partitions are made contiguous, and all the loaded partitions are brought together- called compaction
  + By applying this technique, we can store the bigger processes in the memory. The free partitions are merged which can now be allocated according to the needs of new processes. This technique is also called defragmentation.
  + The efficiency of the system is decreased in the case of compaction since all the free spaces will be transferred from several places to a single place.

How free space is stored/represented in OS?

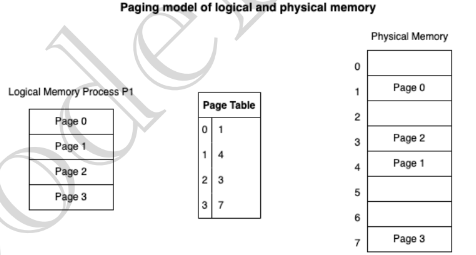
* + Free fragments of memory are represented as linked list.

**Various ways of finding empty blocks of memory and assigning them to processes:**

* First fit
* Allocate first big enough free block
* Easy and less complicated and faster
* Next fit
* Same as first fit but starts search from where the last block was allocated.
* Fast and less complicated
* Best fit
* Allocates the smallest big enough block
* Slow since it iterates all
* Can create many external fragmentation
* Has less internal fragmentation
* Worst fit
* Finds largest big enough block to allocate
* Less external frag.
* More internal frag.

**Paging**

* Paging is a memory-management scheme that permits the physical address space of a process to be non-contiguous.
* Idea is to divide the physical memory into **fixed-sized** blocks called **Frames**. Also divide the logical address of all processes into **fixed size** called **pages**.
* Page size is usually determined by the processor architecture.
* Page Table- Data structure stores which page is mapped to which frame.



* Paging avoids external fragmentation by non-contiguous allocation of the pages of process is allowed in the random free frames of the physical memory.
* To increase the speed of paging- Translation Look-aside buffer (TLB) cache is used where key value pairs are used

**Segmentation**

The chunks that a program is divided into which are not necessarily all of the exact sizes. Segmentation gives the user’s view of the process which paging does not provide. It provides more flexibly to user since he can segment processes according to his needs and reuse of segments of code can be done whereas in paging the hardware decides the page size. Done to improve performance of OS, faster than paging and takes less space than page table.