Operating Systems : Overview

Operating systems is one of the most used software

Few popular O.S :-

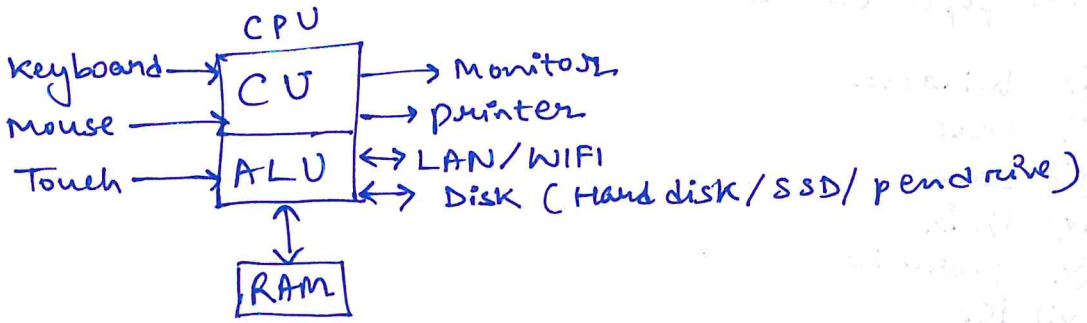
(i) Windows
(ii) Mac
(iii) Linux
(iv) Unix
(v) Android
(vi) iOS

Most of the laptops / desktops are driven by Windows / Mac. Most of the servers are powered by Linux. 99% of the mobile devices run on Android / iOS. We don't use Unix directly now-a-days. Unix is one of the earliest and very popular O.S. Mac and Linux have been inspired heavily by Unix. The core (kernel) of Mac is actually a Unix-like system. Linux is an open-source variation of Unix. The kernel of Android is a Linux or Unix based kernel.

E.g., If we are writing something to a file in c programming language, we do that using fprintf or fopen functions. Similarly we use printf to print something to the monitor. And we use scanf to get some input from the keyboard. The internal working of all these are powered by the Operating systems.

Q. What does an O.S do ?

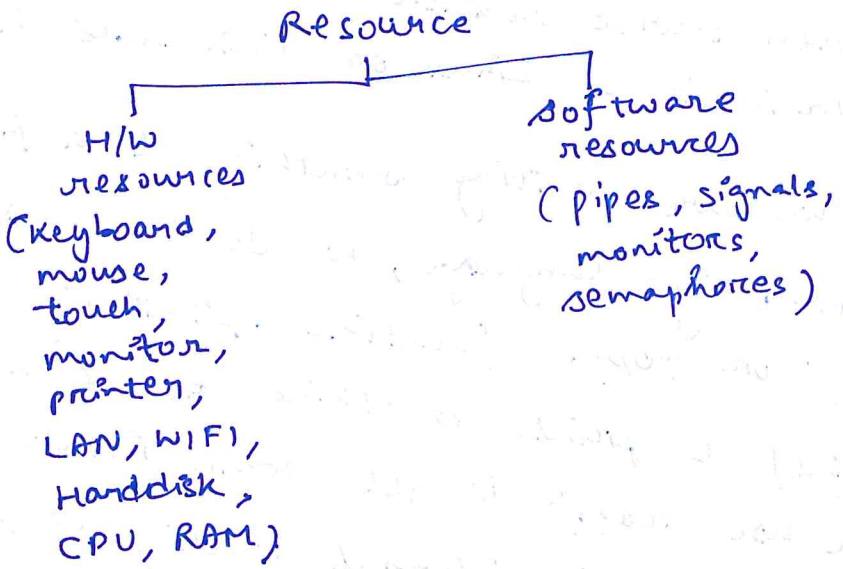
Operating systems can be thought of a resource manager.



CU is the control unit and ALU is the Arithmetic and Logic Unit.

CPU reads data from RAM effectively. It takes inputs from keyboard and executes somethings.

The CPU doesnot talk to these (keyboard/mouse,.etc) directly rather CPU talks to all of these via the RAM.



Operating systems provides convenience to the users of the operating systems. O.S wants to use all the resources very efficiently.
Operating systems provides a bunch of functionalities to the software engineer to simplify application development.

Most operating systems are pieces of software written in C/C++.

In this course we will go through the following concepts:

1. Process - Management (management of CPU)
   - scheduling
   - IPC & synchronization
   - concurrency
   - Deadlocks
   - Threads

2. Memory / RAM management
   - RAM organization
   - Techniques
   - virtual memory

3. File Systems & Device management
   - Interface
   - Implementation details

File Systems → stored in HDD, SSD, pendrive

Device management → keyboard, printers, monitors

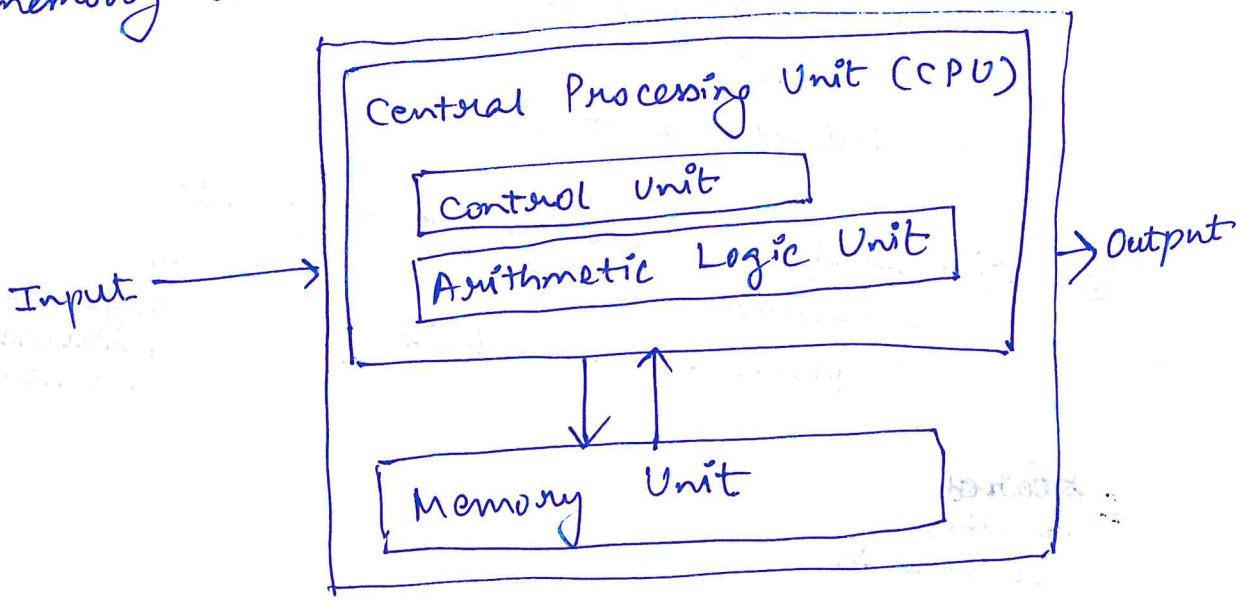
John von Neumann Architecture and History of Operating Systems.

Neumann Architecture :

John von Neumann is considered one of the greatest mathematician and computer scientist who came up with a simple and elegant architecture of what a computer is. Neumann architecture is also often referred to as Stored-program concept.

The stored program concept says any program that we want to execute will be stored in the RAM.

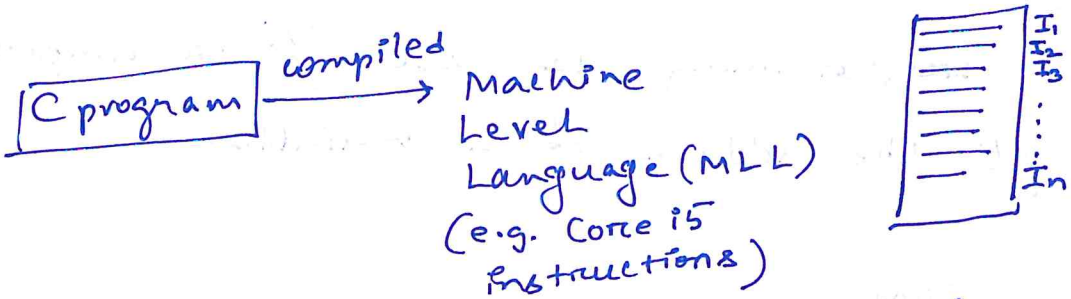
A program is a sequence of instructions that are there in machine level language. The CPU reads this instructions from the memory and executes one by one in a sequence.



Note: John Von Neumann came up with this concept in 1940s. It was revolutionary way of thinking about it.

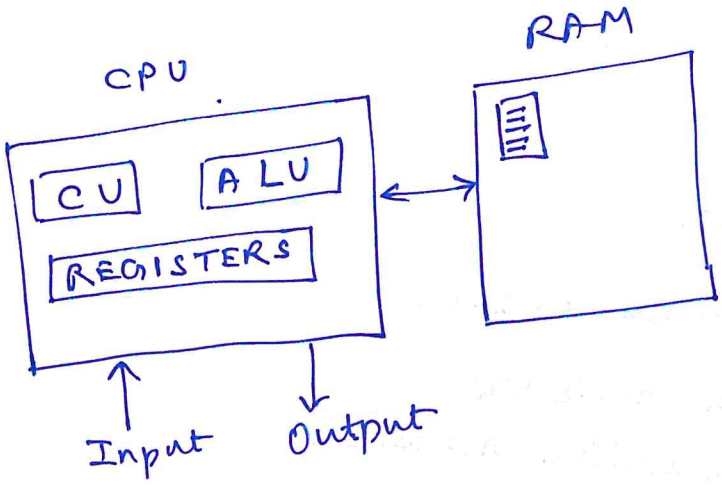
In the CPU, there are also other components like the registers.

Note:

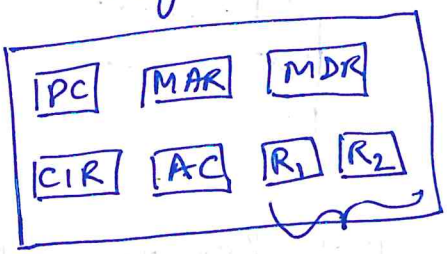


These sequence of ~~instructions~~ specific to a microprocessor is stored in the RAM.

Instruction Cycle:



Registers



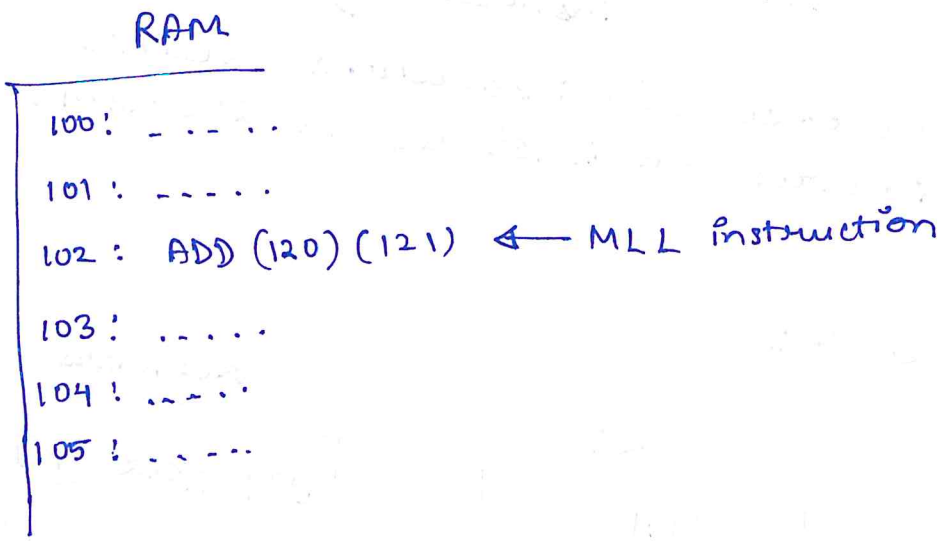
We can store values of variable in registers

A register is basically a bunch of memory which is part of the CPU chip itself, therefore, it is much faster to read data from the registers when compared to reading data from the RAM.

How does the CPU execute a bunch of instructions? That is what is explained by an Instruction cycle.

RAM is basically a sequence of addresses. We can store data in RAM and we can access them using addresses.

In C programming, we use '&' or ampersand to get the address of a variable.

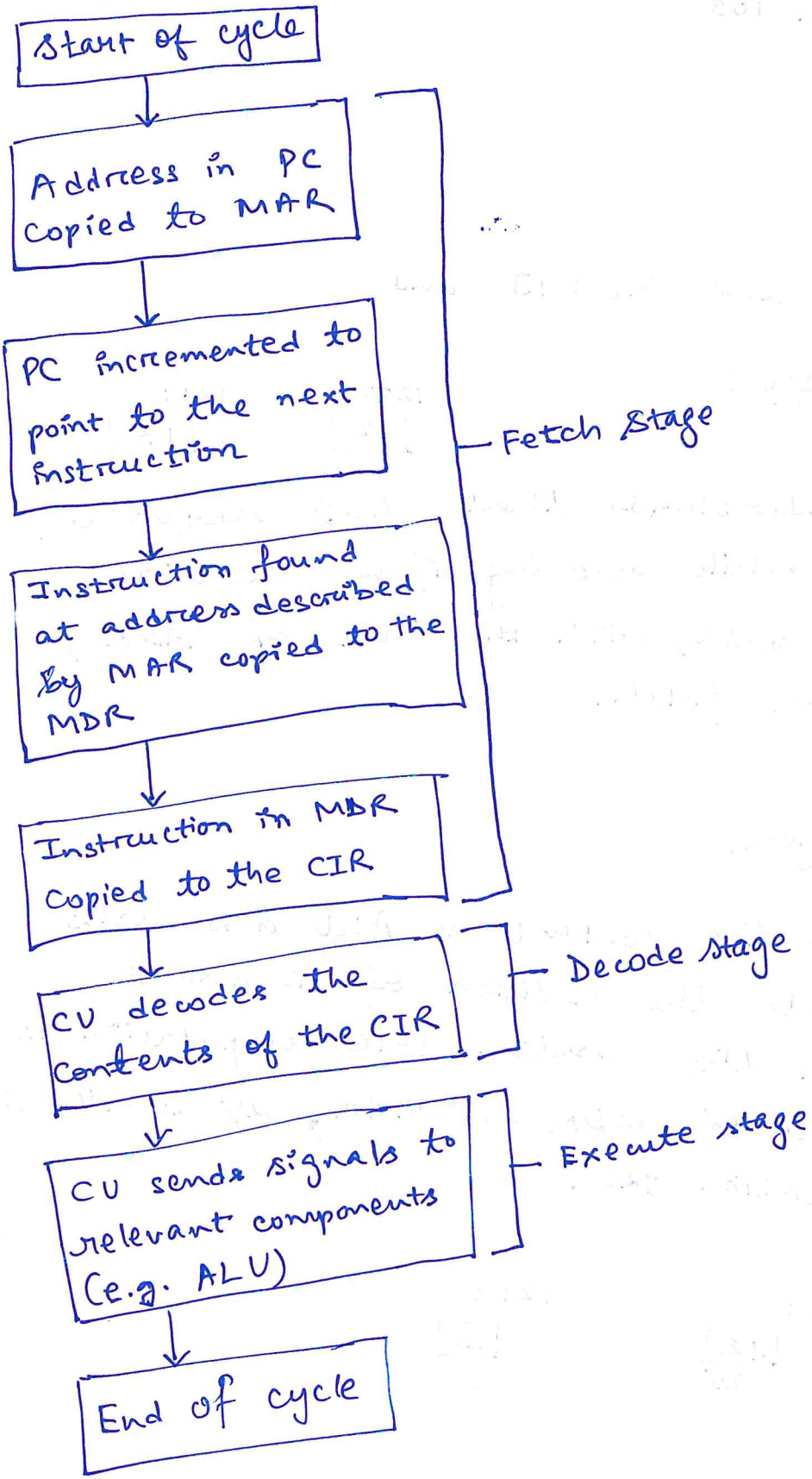


PC: Program Counter

MAR: Memory address register

MDR: Memory Data register

CIR: Current Instruction register


Start of cycle
Address in PC copied to MAR
PC incremented to point to the next instruction
Instruction found at address described by MAR copied to the MDR
Instruction in MDR copied to the CIR
CU decodes the contents of the CIR
CU sends signals to relevant components (e.g. ALU)
End of cycle
Fetch Stage
Decode stage
Execute stage

In Fetch cycle :

Let, PC 102 103

MAR : 102

MDR : ADD

CIR : ADD

Let, R_1 : 12 and R_2 : 13

In Decode cycle :

120 : 12 121 : 13

The CU understands that ADD requires the ALU. While decoding if we require additional data, with the help of MDR, we perform fetch.

In Execute cycle:

The CU gives the control to ALU and asks it to compute the addition of 12 and 13 and return the result. After computation is over, with the help of MDR and MAR, 25 is stored in the location 120.

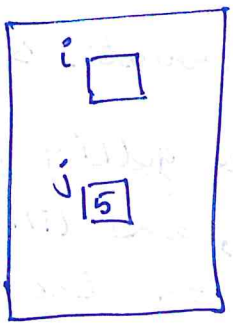
120 : 12 25 121 : 13

Interrupts: We take a C program example to understand interrupts.

```
Line 1:  main() {
Line 2:      int i;
Line 3:      int j = 3+2;
Line 4:      j = j*2;
Line 5:      scanf("%d", &i);
Line 6:      i = i/j;
Line 7:      printf("%d", i);
Line 8:  }
```

gets converted to → MLL instructions

In RAM:

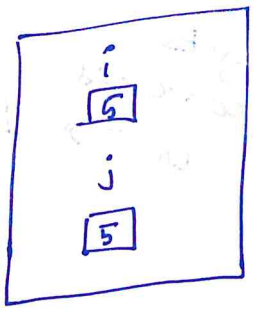


The lines 2, 3 and 4 and their corresponding machine level instructions gets executed.

Note that, each of lines 2, 3 and 4 will individually form (translated) into multiple instructions (MLL instructions) before they get executed.

Line 5 tells the CU and the CPU that we have to wait for a input from the keyboard. The becomes idle as it waits for the input. This creates an interrupt. Interrupt is basically interrupting the flow where in we are expecting some input to arrive into the system.

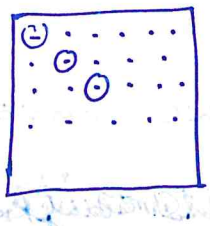
Therefore, the flow of execution has been interrupted waiting for something external to the CPU to work out.



Say the user entered input as 5. Therefore 5 gets copied into i.
Next line 6 will be executed without waiting. Line 7 has to print to the monitor but even here we can have interrupt. Say, there are a lot of things that are getting printed on the monitor right now, and till the time line 7 is printing i on the monitor, we want to wait. There can also be cases where outputs don't create interrupts.

History of Operating Systems:

(1) 1st-generation (40's & 50's): No O.S; punch cards were used as there was no secondary storage. People used to write codes in machine level language, punch holes into punch cards.



Every line would represent some set of instruction.
They would punch a hole and use electronic circuitry to read these and convert into a program.

Even the 1st generation systems were using the stored program because program was now stored on a punch card. And loaded into the memory for processing. Main memory was in the form of magnetic drum.

② 2nd - generation (60's and 70's): NO O.S; People started using magnetic tapes as the permanent way to store data. An example of magnetic tapes is walkman cassette tapes. Data would be copied from magnetic tapes to the RAM and then processed. So, instead of punch cards, we can now store more programs on the magnetic tape. This era was called as batch processing era.

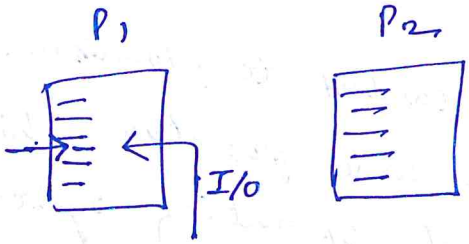
③ 3rd - generation (80's and 90's): O.S started to flourish. And also the disk technology became popular. We will focus on 3rd generation of operating systems in this course. O.S like MS-DOS, Unix, early Windows Os, early Linux Os etc, were built in this generation. Here we started to have the concept of Hard disk. The RAM size was increasing and the modern O.S were being designed in this age. Companies like Microsoft and Apple started during this era. Uniprogramming and multi-programming O.S concepts became popular.

④ 4th generation: These are OS like Network OS. (since 2000's)

Also called distributed OS, where we have hundreds of computers connected in the network. We also see Real time O.S (RTOS) like in IOT, mission critical applications like satellite launch. We even see parallel processing O.S, etc. We usually study about these O.S in a graduate level or research level.

Uniprogramming vs Multiprogramming

In uniprogramming environment, one program is executed at a time. It is a huge wastage of resources like CPU.



Suppose CPU is executing program P_1. At some instruction the program P_1 gets interrupted and is waiting for some keyboard input etc. In the uniprogramming environment, CPU is still running P_1 and program P_2 is waiting to get CPU.

The whole idea of multiprogramming is can we increase the efficiency by having multiple programs executed simultaneously?

In multiprogramming environment, the RAM will have multiple programs and CPU will allocated to another program if the currently running program gets blocked.

All the modern OS implements multiprogramming, e.g., Linux, Windows 10, Android etc.

Example of uniprogrammed OS is MS-DOS.

The shift from uniprogramming to multiprogramming happened as in the last 30-40 years the size of RAM has increased and the cost of RAM has decreased. Similarly, CPU speed and disk size has increased.

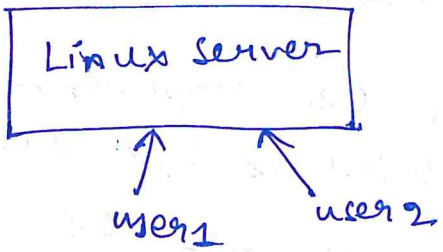
Multiprogramming vs Multitasking

Multiprogramming is a Unix terminology. Multitasking is a term used by Windows as a marketing tool. Both of these terminology actually mean the same thing.

Types of Multiprogramming :

(1) Single User vs MultiUser :

In a Linux server or a Unix server, multiple users can simultaneously login and use it.



In 70's and 80's, since everyone did not have personal computer, the companies would have powerful computers where multiple people can login, submit their work which will be executed and results would be returned back.

Incase of a single user O.S, only a single user can login and use the system.

(2) Preemptive vs Non-preemptive :

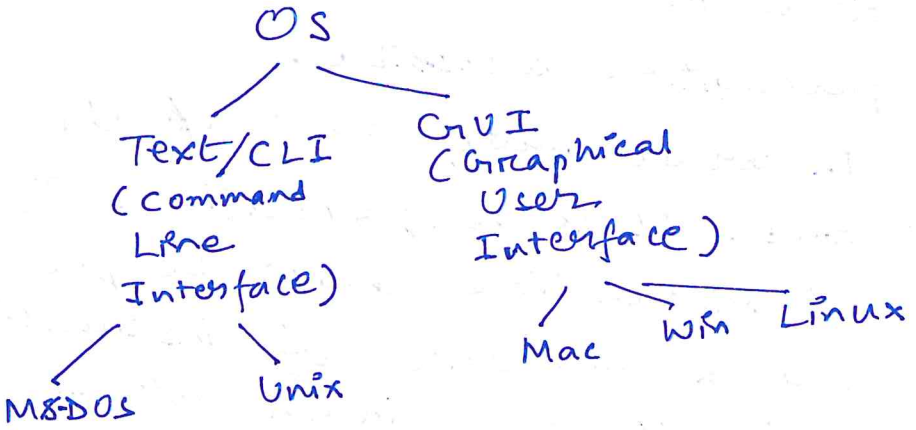
In a preemptive system, the O.S can force a process to give away control of some resources like CPU.

Preemption Basically means forcefully removing something.

In a non-preemptive system, the process has to give away control itself. The process might give away upon completion or if there is an I/O event.

Examples of Non preemptive O.s are windows 3.0 and windows 3.1. And examples of preemptive O.s are Windows 10, Linux, Android etc.

Note: There are also Text based Os and GUI based Os.



The GUI based Os also have support for CLI like command prompt (windows) and terminal (Mac, Linux).

Modes of CPU execution + FORK()

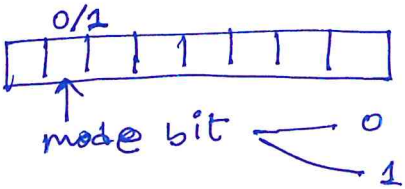
Modes of execution on a CPU:

There are two modes of execution — user mode and the kernel mode.
The kernel of an O.S is basically the core piece of the software which enables the O.S.

Every user submitted / user given program runs in the user mode. In a user mode, preemption is possible.
In kernel mode, most of the key functionality runs. It is non preemptive. If a sequence of instructions cannot be preempted, then such a sequence of instructions are said to run in an atomic fashion or atomic execution of instructions.

Mode bit in Process Status Word (PSW):-

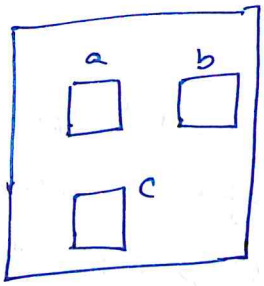
For every process, there is a data structure that the O.S maintains called the Process status word (PSW). Within the PSW, there is one bit called mode bit.



special Data structure

The other bits hold important information too.

E.g.1)
```
#include <unistd.h>
#include <stdio.h>
main()
{
1. int a, b, c; — user mode
2. c = a + b; — user mode
3. fork(); — kernel mode
4. printf("Hi"); — user mode
}
```

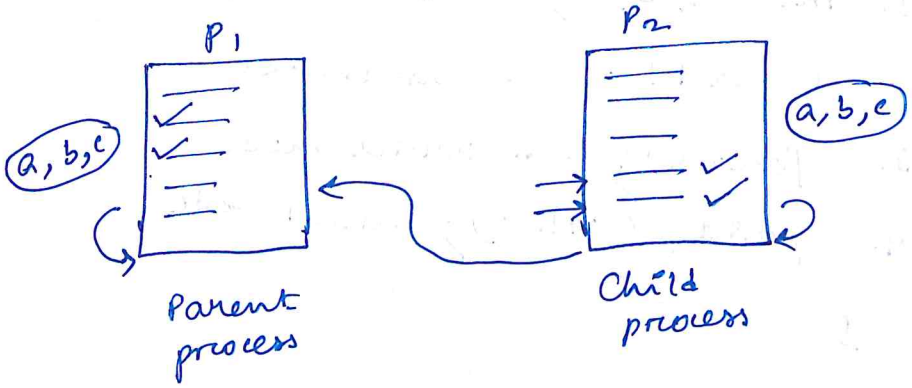


We declare 3 variables, ie., in the memory we have 3 places to a, b, c respectively.

fork() is often called a system call or sys call. The function is declared in unistd.h. This function eventually calls the O.S. Fork creates a new process. It copies the instruction and the variables.

While executing line 3. fork(), the O.S takes the control and it goes into kernel mode and takes all the instructions and memory associates with the parent and creates a copy (child process). And control is transferred to child process and child process execution begins on whatever is there after the fork() line (line 4). After finishing the execution of the program (child), control is

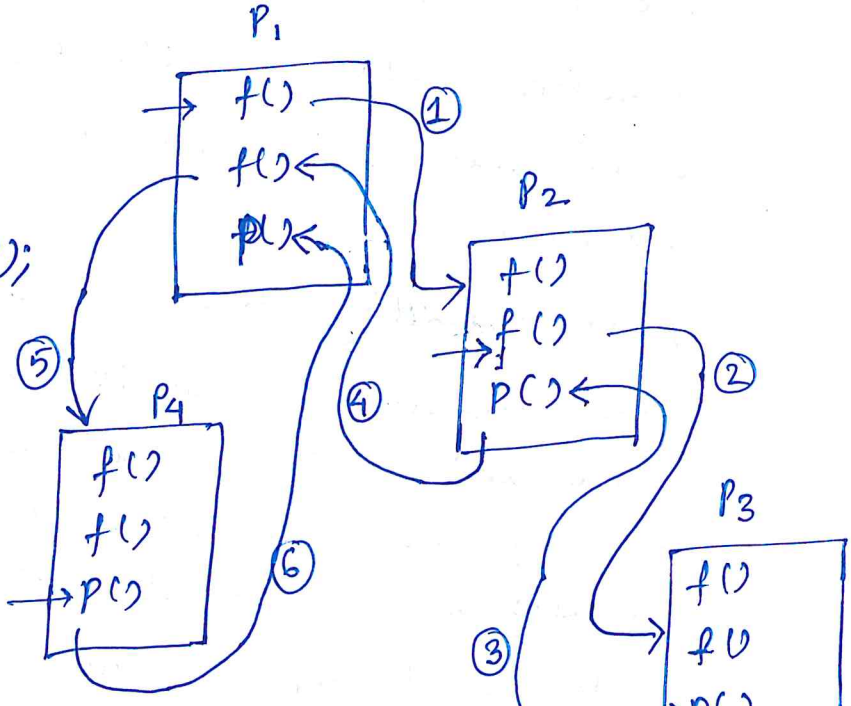
transferred back to the parent process (to the line after the fork()).



The child prints "Hi" and then when control is transferred back to the parent process, the parent prints "Hi".

E.g. 2)

```
#include <unistd.h>
main() {
  fork();
  fork();
  printf("Hi");
}
```



Output:

Hi
Hi
Hi
Hi

For 2 forks, 2^2 processes got created, including the parent process.

Similarly, if we have n forks, we will create 2^n processes including the 1st parent processes.

Eg 3)

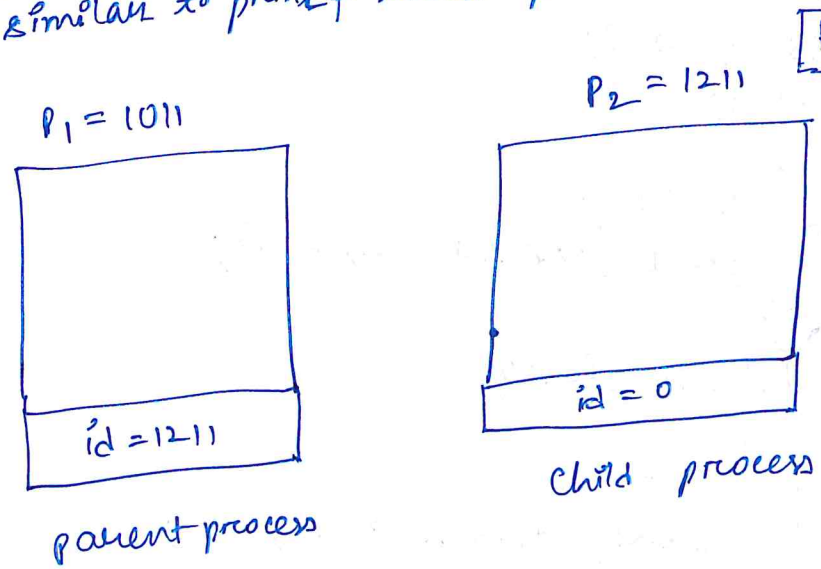
```
main ( ) {
int id;
id = fork ();
if ( id == 0 )
{
  ==
}
else if ( id > 0 )
{
  ==
}
(when id < 0) {else
{
perror ("Unable to fork");
}
```

executed by the child process

executed by the parent process

executed when O.S could not fork a new process.

similar to printf which prints errors.



Let the process id of parent process $P_1 = 1011$ and the process id of the child process $P_2 = 1211$.

The variable id in the parent process gets the value from value returned by the fork(). Basically, the process id of the child is updated in the variable id of the parent.

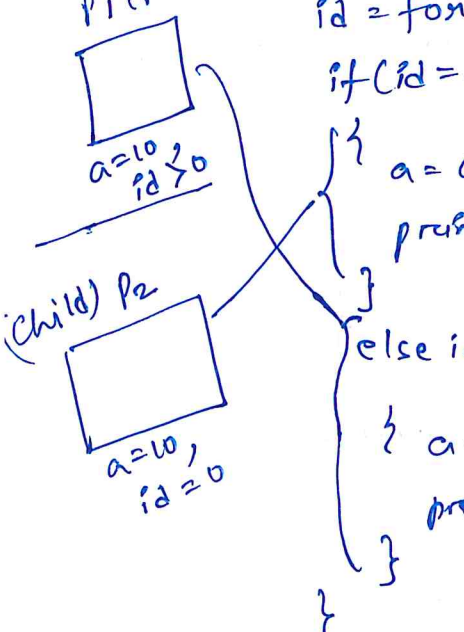
∴ id in parent will become 1211.

While variable id in the child will be updated to 0.

Note: The process ids are always greater than 0.

PID > 0

Eg 4:)




```
main(){
int id, a=10;
id = fork();
if(id==0)
{
  a = a+5;
  printf("%d %d", a, &a);
}
else if (id>0)
{
  a = a-5;
  printf("%d %d", a, &a);
}
}
```

If u & v are values printed by parent and x & y by child, then

a) $u = x + 10$; $v \neq y$

b) $u = x + 10$; $v = y$

c) $u + 10 = x$; $v = y$

d) $u + 10 = x$; $v \neq y$

u v
Child: 15 , & a

parent: 5, & a
x y

fork() returns 0 in the child process and process id of the child process in parent process.
In child process, $a = a + 5$ is executed.
In parent process, $a = a - 5$ is executed.

$\therefore x = u + 10$.

The physical addresses of 'a' in parent and child must be different. But the program accesses virtual addresses (assuming we are running on an OS that uses virtual memory. The child process gets an exact copy of parent process and virtual address of 'a' doesnot change in child process. Therefore we get same addresses in both parent and child. $\therefore v = y$. Option (C) is correct.

Solved Problems

Q. The following C program is executed on a Unix/Linux system.

```
#include <unistd.h>
int main()
{
  int i;
  for(i=0; i<10; i++)
    if( i%2 ==0)
      fork();

  return 0;
}
```

0 9

0, 2, 4, 6, 8 } 5 times fork is called
fork();

The total number of child processes created is 31.

Since fork() will be called 5 times, we will have 2^5 processes in total and $(2^5 - 1)$ child processes.

And $2^5 - 1 = 31$.