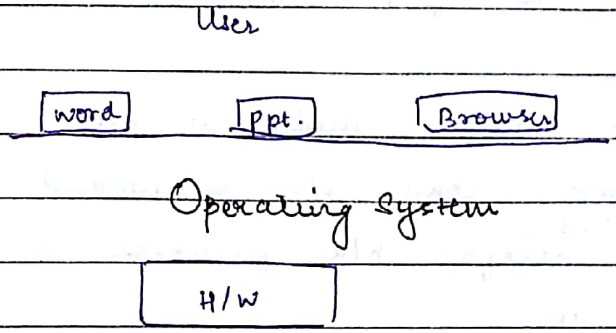
Operating Systems

Book :

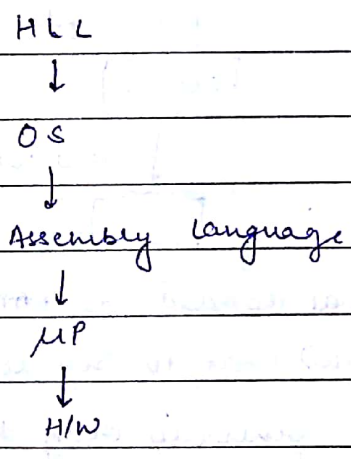
Operating Systems – Silberschatz, Galvin.

→ OS : interface b/w user and hardware.



→ Why do we need an interface ? Why can't we directly interact ~~work~~ with H/W

- Because of language diff.
- Because we want to run ~~program~~ app^n's independent of h/w



- In case you are malicious user, you can destroy h/w if OS is not ~~+~~nt.
- You may take control of HD
- Multiple users may use a machine. If you are given complete control of H/W, you can access & destroy other's progra

Teacher's Signature

That is why we need an interface

- We can destroy the flag
- No access to H/W or System Resources
  ↓
  CPU, Memory, i/o devices, file systems in secondary

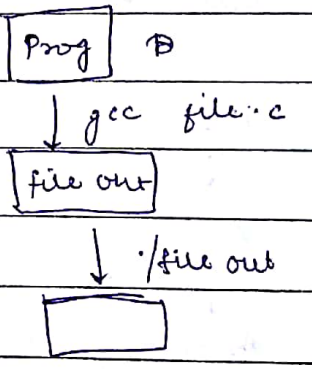
→ OS has to manage all System Resources.

→ Each device has a controller which is h/w. To interact with device some s/w is needed to provide common controller interface b/w peripherals & system. OS manages this all.

→ Kernel — very core part, contain very basic properties of OS

OS — Adding more functionality to Kernel.

→
```
main ( )
{
    pf ("Hello world");
}
```



When system was started, OS was loaded in RAM.
When compiling, compiler (s/w) has to be loaded into RAM. (can be part of OS generally only Kernel part is loaded initially)

→ When prog. is loaded into memory for execution, it becomes a process (earlier in sec. memory)

Teacher's Signature

std. i/p device : keyboard

o/p : monitor

→ ./a.out : again bring file in RAM

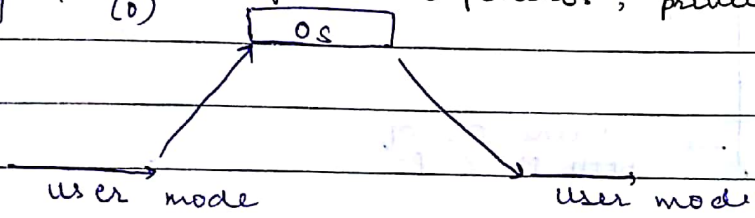
→ pf command is defined in stdio lib. It goes into lib. & sees def" of func". It then call OS to print ~~or to~~ write on monitor & go ~~take~~ back into user mode

(system call) ⇓ request OS to do something for it. User prog. can't directly access the monitor

↳ User mode : where prog. is running

↳ to access some func's system call

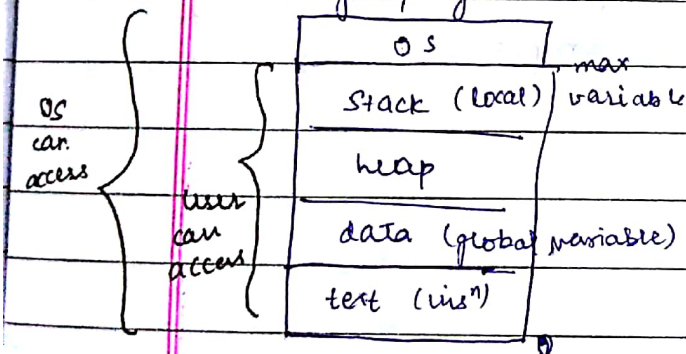
priority : (0) system (supervisor, privileged, kernel) mode



⇒ 2 levels of mode

priority: (1)

→ Every program is stored as :

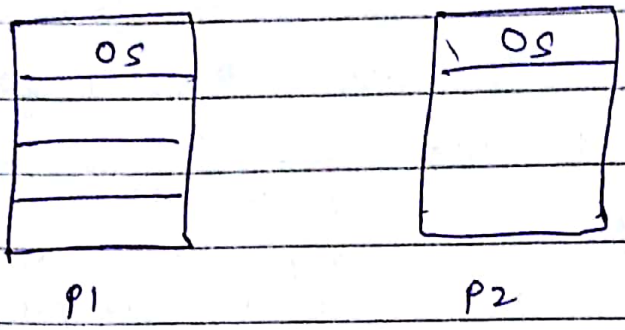


Each process has OS part also attached to it

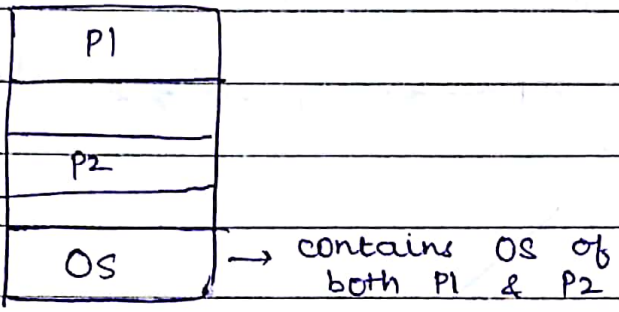
process virtual memory map (address map) (can be as large as possible)

→ When it'll be loaded to RAM, the OS part 'll be in lower part of memory and other ~~do~~ will be put wherever OS finds free space.

→ Suppose multiple process are going on

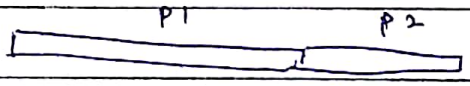


Again, OS of P2 will be loaded in lower part of RAM only.

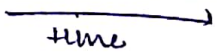


→ P1 is also running & P2 is also running.

1) 1st way to do it :



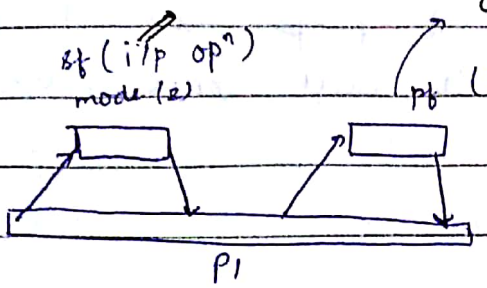
happened in earlier computers



P1:

→ main
{
int x ;
sf (" %d" , &x) ; → again it would be system call
pf (" ") ;
}

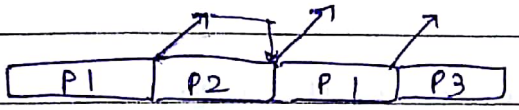
now, system is just waiting
CPU " " "



: at both times, CPU is sitting idle.

2.) Other way :

→ Whenever 1st program is waiting for i/o devices, OS can run other programs



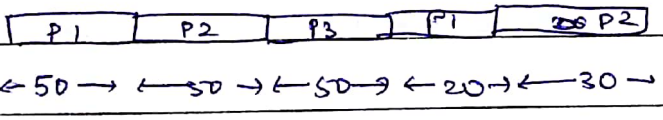
: effectively utilising resources
This is work of OS.
OS is Resource Manager

→ It is ~~mana~~ CPU manager here.

3.) Other way :

Fixed time slot for each process.

P1 : 70 ms
P2 : 80 ms



⇒ There is a need for timer also in OS ~~to~~ which give signal to OS to switch context : Context switch

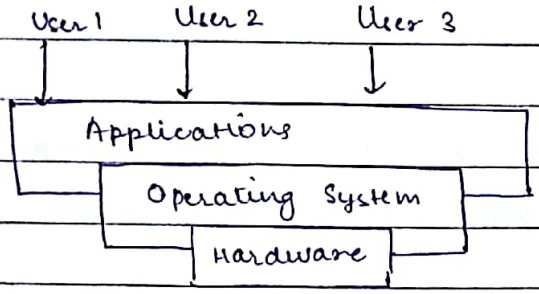
going to change process (fix time slot)

mode (1) to mode (2)

→ If prog. goes into infinite loop, ⇒ your prog. has taken control over system now.
So, timer will help now.

→ When a program is run, if proper return is not returned to OS, it will know something is wrong.

5/1/17

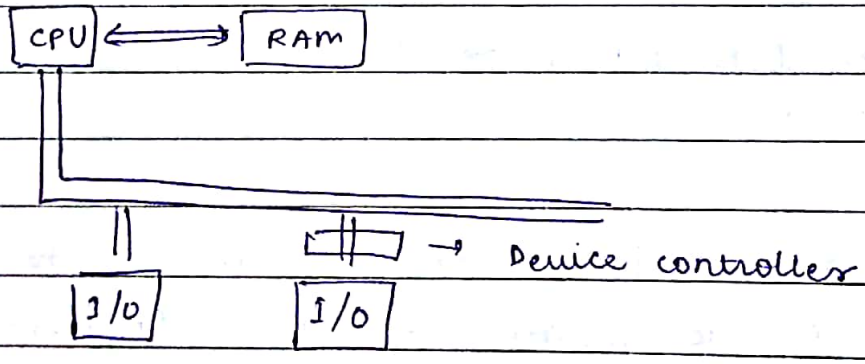


Why user don't directly interact with h/w.

→ difficult for user to write prog. to interact with diff. h/w

→ security issues

↳ Suppose we let user interact directly with h/w



→ Device driver is prog. which interacts with device controller.

→ ~~Interrupt vector~~ Device controller sends an interrupt to CPU, find add. of ISR corresponding to vector.

→ All device controller have buffer

Embedded system : have microcontrollers

→ If we need to tranfer bulk of data, we initialize DMA.

→ Suppose user is allowed to interact with h/w directly.
⇒ He has to write a prog. to interact with h/w.
If device changed, prog. won't run.
So, OS does provides Hardware Abstraction

OS :

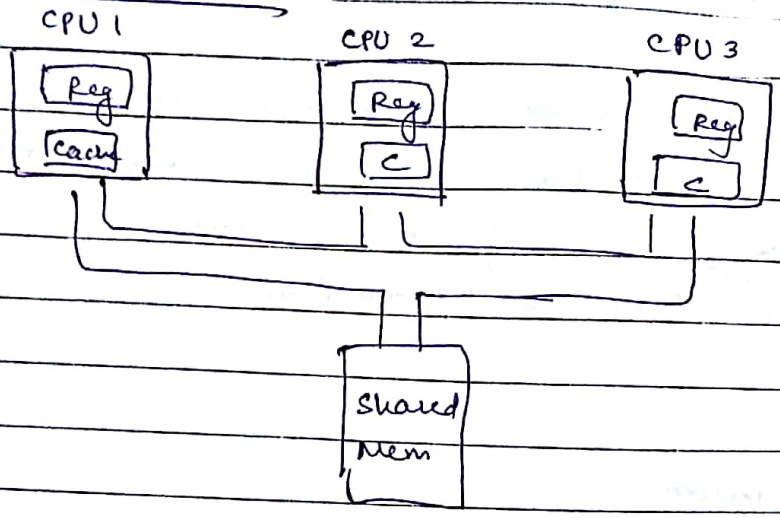
1.) Hardware Abstraction
2.) Resource Manager

Operating System :- is a prog. which is running all the time. It is in the Kernel mode.

→ While designing an OS, we've to take care what kind of system I'm designing it for
(diff. for embedded machines : have memory constraints
" mobile phones : power constraint + ")
OS has its own overheads (using resources like memory)
Context switching :
In real time systems, we can't afford time required context switching. (air bags in cars have to inflate quickly at time of accident)
OS for this will be diff. For PC, it'll be diff
For PC, expectations from OS : (User P.O.V.)

1) Easy to use
2) " " switch from 1 app" to another
3) run my prog. very fast
we're last bothered how resources are allocated.

→ Multiprocessor :



(Quad core)

(may have cache)

Multiprocessor

↓ Symmetric

Everyone is equal

OS has to be neutral

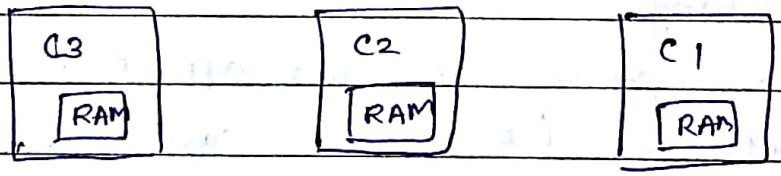
All will do all tasks

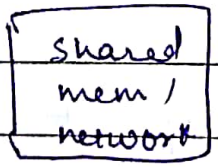
↓ Asymmetric (master - slave relationship)

↓ Processor is boss.

It decides which processor will do what

→ Multicomputer (~~Middle~~ clustered)





→ ↑ possibility of some shared memory & shared network

Asymmetric

1 of comp. is on standby.

As soon as comp. fails working is finished comp. on standby will take over

Symmetric

All are working, but are monitoring each other. If one fails, other takes over

3) Mainframe : Server

OS work :

- maintain isolation b/w users
- efficiency of Resource Management

4) Workstation :

High End Machine

Good mathematical capabilities

* can have cluster of Workstations (COW)

Expects from OS working for COW :

→ Individual usability

→ Efficient communication b/w processes running on Workstations

* Expectations from OS differ.

↳ When you power on, OS is in hard disk. Initially, basic kernel (Boot Strap Loader) is kept in ROM. On switch on, BSL starts working. Work of BSL :

- It initializes CPU registers. Gives add. of kernel
- Checks all devices
- Load Kernel into Memory (lower add.)

Teacher's Signature

- Creates process init & then it waits for an event to occur.

* OS always waits for an event / interrupt to occur & then only it comes into action.
  H/W interrupt : I/O device want to communicate
  S/W " : some prog. has wrong instn : trap
  System call within a prog.

→ You don't do anything, nothing 'll happen to the system.

→ When you click, you give H/W interrupt. When your ISR runs, it is in OS space of memory.

→ Suppose you click 'New' in Word, It is just an API. It is a system call. You are asking OS to start a new process to open a new file.

→ Copying data : WAP to copy data Option
1) write name of both files
2) drop down menu
Pf : again, a system call. Mode : from user to kernel
Sf : as soon as we write, again, it is h/w interrupt. So, again, it is system call.

Now, it has to open 1st file. Opening 1st file is again system call (file brought from HD into RAM)
If file D.N.E. → system call will print an error message : "File D.N.E."
If file exists : create new file. If already a file exists with same name for same user (backhead check)

Teacher's Signature

OS :→ keeps track of all users, all their data, validating their passwords processes

either it'll terminate or it will ask you to rename/replace exist file. These all things OS has to do

* At any point of time, Kernel is running.
→ Apart from Kernel, prog. running :

1.) System Prog : helping Kernel, related to OS
Compiler, File Management Process, Interpreter Device Management

2.) App^n Prog : Not related to OS
• Browser, Word

OS :

↳ interface to interact with
↓

1.) Command line Interpreter (CLI)
2.) Graphic User Interface (GUI)

1.) CLI : It is executed line by line ⇒ Shell
Write a command, it's executed.

2 approaches

Shell has commands
do ⇒ what is needed
→ want efficient comm^n b/w I/O devices, Inter-process

Shell don't do anything. All prog. are written in files in OS. space. It has all commands in diff-diff files
Advantage : To create new comm and, simply create new file & add to repository. Otherwise, had to change S/W of shell.

Teacher's Signature

→ Provide h/w abstraction

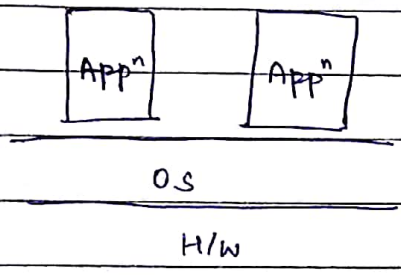
→ Resource Manager
↙ ↓
HD RAM

→ don't allow user to access h/w directly
Why ? ↙

→ isolate users

→ security

→ don't want single user to monopolize the resources



→ Everytime, when app^n want to interact with h/w, they have to request OS → System Call
System Call : OS is going to call some Kernel func^n which will interact with h/w & it'll do something & return from there.

Types of OS : (OS Structure)

1. Simple (MS-DOS) — Earlier, no concept of multiprocessor. When users ↑ed, they segregated the user
2. Layered (UNIX) ; very bulky, has many issues. Debugging, updating very hard → Kernel small.
3. Microkernel : shell developed as user prog
4. Modular

Users
Shells
compilers ; ass

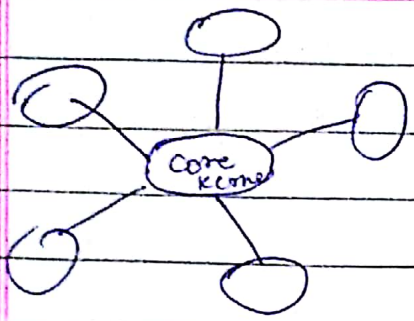
Kernel { filesystem, page rep
CPU scheduling virt. mem
device drivers

h/w

Teacher's Signature

Layered OS

Modular :



Create a module & attach to core kernel to add new functionality

Kinds of System calls required ?

→ Interrupts / Traps :
↓ need for

1.) Device Management :- HD, network, i/o devices, peripheral devices

diff. types of devices : i/o, network

If multiuser : 2 people need to use same resource, OS has to manage who uses first (avoid loss of data)

User will send request for device, and release, send & receive data from device, should get device attribute & set it.

In UNIX, all device info. is in /dev

2.) An executable in disk : program

A running program : process (prog. in execution)

Process Control :-

- Create process (find space in memory, add. map : Kernel created along with add. map)
- Load
- Execute
- Terminate (could be normal / abnormal)
- Error checking ↓ error msg needs to be sent, OS might ~~keep~~ dump for debugging.
- Get / Set Process attributes

3) File Management :-

File → Text
→ Binary

File system : On disk data structure

Teacher's Signature

PAGE NO.:

Types of System calls needed for File System

- Create
- Delete
- Read / Write
- Append
- Open / close
- Get / Set file attributes
  (name, date of creation, etc.)

4) Protection :-

5) Information Maintenance :- System time / date.
Current version of OS

* ~~6)~~ Memory Management :- System call is not used in this.

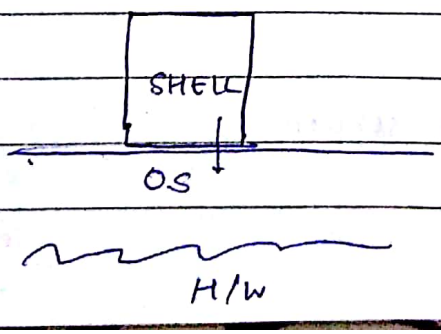
→ CLI : has a shell

$ ls... Interprete line by line
< > ↓ executing line by line
$

Batch Interface : Take whole & each command keeps executing in background.

Shell : has CLI

Nowdays, Shell is an app^{n} which runs over OS.
This is a process.



Teacher's Signature

shell is running our kernel now. Whenever it needs something, it will give system call

$ ls

You've written ls, it will send request to OS. ls is again a program. It will run, list directories, return back, again show $

→ Inside a shell program, an infinite loop is going on. Taking something from keyboard, executing it & again writing $ command

→ Program :

```
while (1)
{
write (STDOUT, "$", 2);
```

where I want to write → 1 → file description (fd)
what → "$"
length of string → 2

1st ly, it has to show $ on console. ∴ has to access h/w. It will send system call to OS. write command is needed

write ← system call that I want to write

* There will be some FD associated with STDOUT

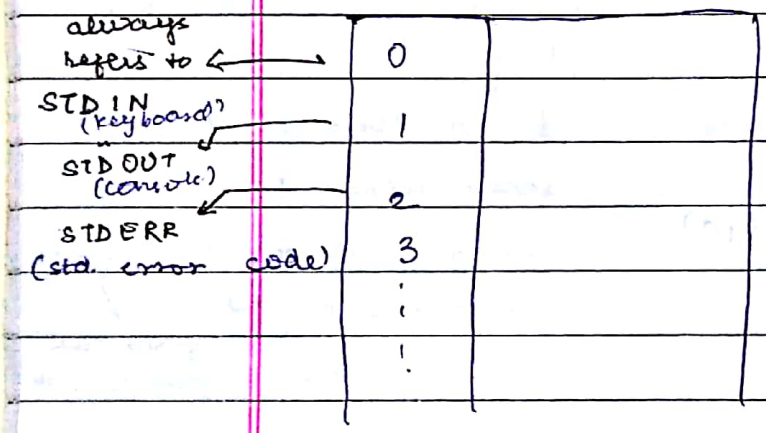
→ Each devices / file is represented via a no. →
file descriptor ⇒ identifier for a file

→ The way you access i/o devices & files are very similar. So, i/o devices are also considered as files.

→ Info. of all fd are kept in a fd table.

→ With each process that is created, a fd table is created. (in the OS part, process won't be able to access it)

fd table :



It don't know what is STDIN, ---.
OS will recognize
All contents are addresses
pointer to devices (or offset to device)
which device & address are put in the table.

Write (1 , "$" , 2)
STDOUT from fd table

fd : actually pointer to fd table

pointer to base of file + some offset

10/1/18

→ When you power on, it gives interrupt to CPU. It will reset. Start looking for 1st ins".

x-86 BIOS : going to check peripherals, initialize reg, locate BSL and BSL 'll now take over, find & Kernel in HD & load

Teacher's Signature

2-step Boot loader

1) BIOS
2) BSL
} → job of both is all the things mentioned.

All CPU loads are set to 0 except PC.

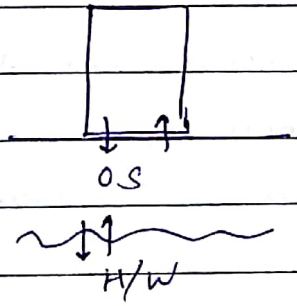
→ If Not x-86 system : only BSL exists, not BIOS is there.

→ Kernel is loaded into lower space of RAM.

→ 1st booting takes place in 16-bit mode (backward compatible) & then it activates 32-bit mode

→ Either we can have GUI
or CLI (use shell)

- Shell : also a s/w . It is also written as a prog. in HD either OS will open it ~~terminal~~ automatically or a command has to be given.
- Shell runs as an app^n in user space



→ We can run multiple programs at same time using single processor.

Possible ways :

1) Using timer to slice up program. If a prog. is running, OS halts it based on priority

2) OS can preempt a program, then starts another program.

non-preemp process : No one can stop it once it has started

3) When waiting for i/o devices, go for another prog

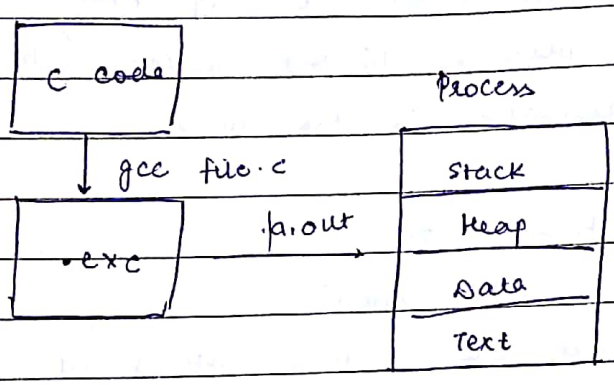
Virtual Mem.

DATE : / /
PAGE NO. :

Virtualization : OS

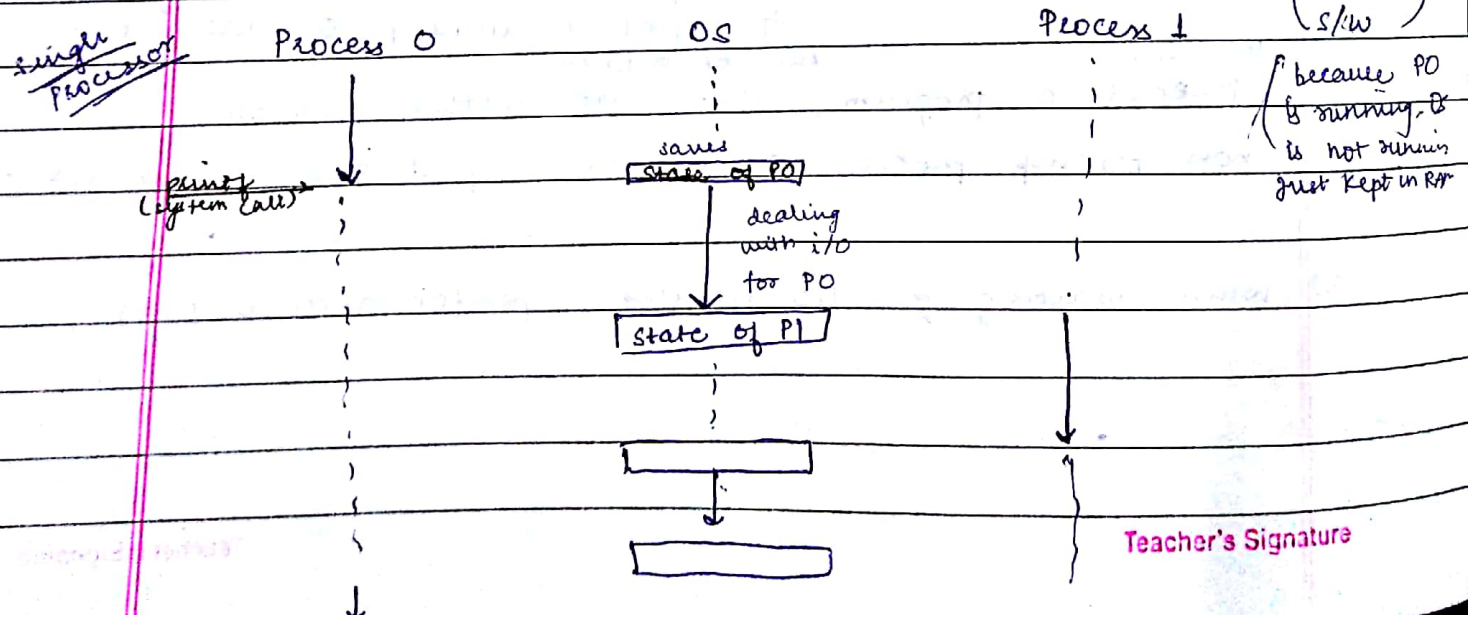
Everyone feels it has CPU at its own mind (has its own h/w)

Address space : Each process is given some area under memory. No other process can access other's area.



→ MMU : converts logical add. into physical add. need to refer to page table.

⇒ Each process needs something associated with it ⇒ Kernel space (Kernel stack) containing info. about process. When system call occurs, OS will start running



Teacher's Signature

String is a pointer

STDOUT : could be anything : console or anything

→ State of process : values of Reg, PC, Stack Pointer. It'll be stored in kernel stack associated with that process.

→ Till P0 is waiting, OS will load state of P1
As soon as it is loaded, P1 'll start running

→ Creating a Process : Reserving some part also in OS which will store info. about it, providing add. space, etc.

⇒ What a Shell will look like
$ will come up, then command is written, executed & again $ will come up.

① → have to create window

~~Shell~~ Shell Program :-

```
while (1)
{
    write (1, "$", 2);
    read (0, buf, 100);
    pid = fork ();
    if (pid < 0)
    {
        // error → stderror
        write (2, "error", 5);
    }
    else if (pid == 0)
    {
        //child
        exec ("ls");
    } // exit (0); → terminated
    else { // parent // sleep();
        wait ( );
    } close (0); → close STDIN connec" to device
```

write (fd, "buffer", length)
write on this device; from this buffer; length of the buffer
buffer: pointer to a memory area
(assuming 1 → console)

- write is a system call because there is a h/w involved & shell can't directly interact with h/w.
- When ls is written, everything is listed
  - either it is a func" written in shell prog & shell is going to call it, run (not done because to add, have to change shell prog)
  - All are separate prog. Adding / debugging becomes easy. ls is another prog. stored as executable file

0 → STDIN
1 → STDOUT
2 → STDERR

fork () → till here, code was running for parent only; both pids will be checked from here

if child was spawned, pid = 0, this part won't be executed by child

exit (0) → can also be used

Teacher's Signature

- This means ls has to be loaded. New process has to be created.
- But there is no command to create a process in LINUX. But who created shell then ?
  Initially, 1st process created by kernel is : init which in turn, created shell

init ⟶ parent of shell

→ you wrote ls which shell has to read (from device into buffer)

→ we could give arguements also in the command.

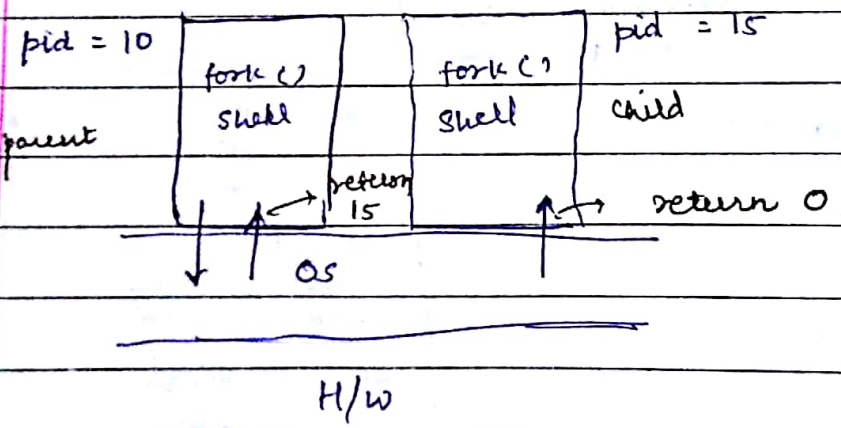
Eg → ls - l

→ read : i/p has gone into buffer. shell has to read from buffer., parse it & get some command and arguement. Shell has to create process It knows some command has to be run

Way of creating process :

fork () ; ⇒ fork is creating a child process (clone)

⇒ an exact copy of this shell would be created



* If any process is calling fork, an exact copy of its

Teacher's Signature

parent will be created.

→ How to distinguish b/w parent & child ?

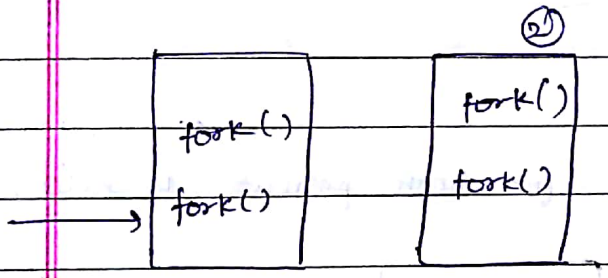
Each process has an identifier associated to it : Process id.
(OS assigns pid)

fork is going to return 2 diff. pid's now.
it'll
To child , return pid = 0

To parent, return value will be pid = pid of child

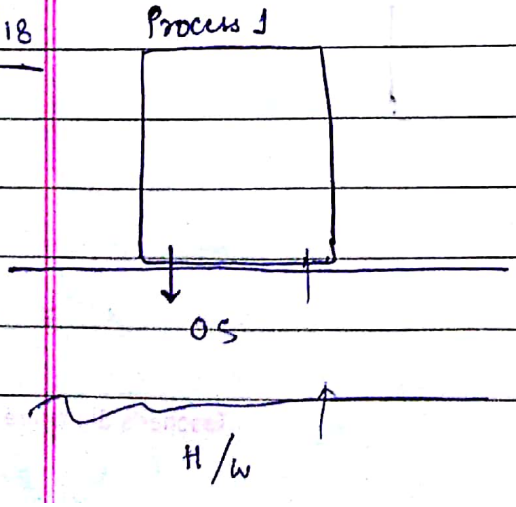
In system that exist, pid has not changed. Only it returns these value.

⇒ fork ( ) ; is called, child is getting back 0 from fork
State is also same. If fork was running ⇒ PC will point to next line ⇒ all the values stored in Kernel
⇒ Everything is duplicated in child : so, both processes will start executing from next command.



If again fork is written, the ~~child will be~~ will have child,
② ka bhi ek child hoga
⇓
4 processes will be created.

12/1/18



In what all circumstances there would be a system call ? (Process 'll give system call)?

1.) there is an i/o request

2) s/w exception : something gone wrong with the program (maybe h/w will tell OS something is wrong

Teacher's Signature

→ If interrupt is from h/w, then also OS will give process a msg.

Creating Process (Cont.)

→ No process can be created on its own

→ Every process is spawned

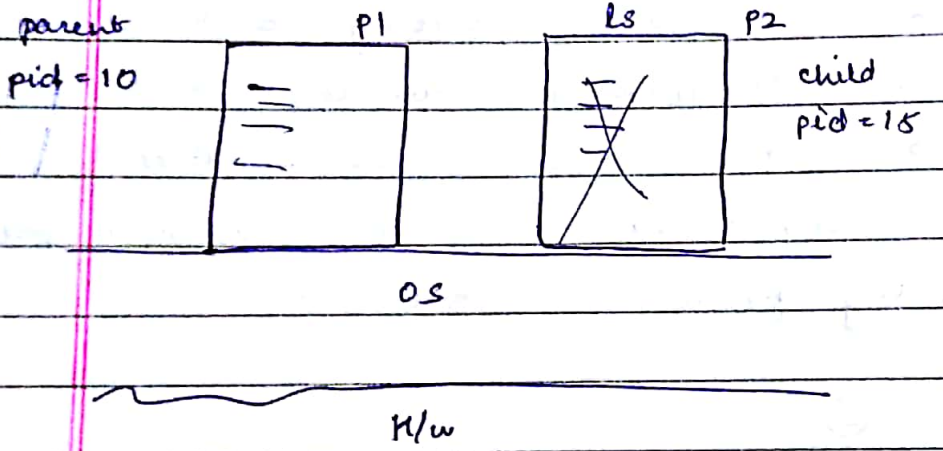
→ System call for creating a new process : fork();

Write (1, Buff, leng);

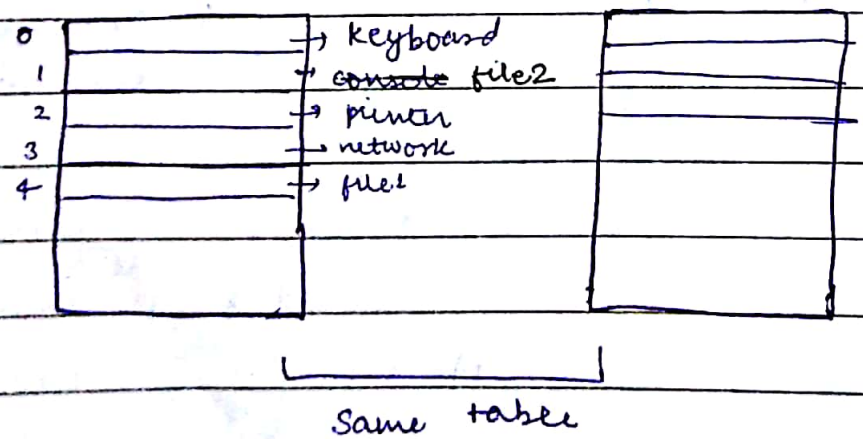
Read (0, Buff, leng);

Buffer gets whatever is written in i/p device & further read

pid = fork();



→ fd table will also be same for both parent & child.



→ whenever there is a fault in system, a -ve value is returned.

DATE : / /
PAGE NO. :

- open ("file1", r) ; → system call.

It'll check fd table. Starting from beginning, 1st unused no. will be given to file1.

→ OS will locate file1 & open in read mode & use 1st unused fd (like if 1st 3 are occupied, then it will assign 4)

- close (1) ; → System call

It'll remove the mapping. 1 in fd table won't point to console anymore.

If then ;

open ("file2", r) ;

since 1 is not mapped to anyone, it'll be assigned to file2. STDOUT has now changed.

fd of file2 : 1

Now this becomes STDOUT.

→ The fd table are part of kernel space. They are kept on stack of kernel space

→ When it spawns a new process, fdtable also gets copied.

⟹

→ What if there was an error ?

Suppose OS isn't able to create a process because :

↳ exhausted no. of processes

↳ no enough memory

→ pid should always be > 0

→ if pid < 0 ⇒

→ OS is written in C as well as in Assembly Language.

→ if pid == 0 ⇒ we're on child process.
We've created one process because we want to do something.
In child process, ls needs to be run (In eg. mentioned earlier)
ls is another file on HD. If it needs to bring ls, a child has to be created & it has to be executed in environment of child.

→ exec ("ls"); → command (gets from reading i/p)
Child has also started running ~~from~~ after fork() :
As soon as exec is written, the program will become a process. It'll be picked up from HD & put in child.
Now, it's not running same code anymore. ls will start executing now.

→ if pid > 0 ⇒ I'm again on parent ⇒ I have to go back to $ prompt.
But it 1st waits for 'ls' to complete then only $ is printed.

$ ls

<.----->

$ ls
↓ ↳ child $
main program

That means parent waits for the child to complete

Teacher's Signature

either there is an explicit exit command, or it terminates. As soon as process is completed, it is wiped off (all things associated with it gets wiped off).

→ Possibility :

- Open 1st browser, 2nd browser, ... all browsers are independent & don't wait for another (Parent don't wait for the child).

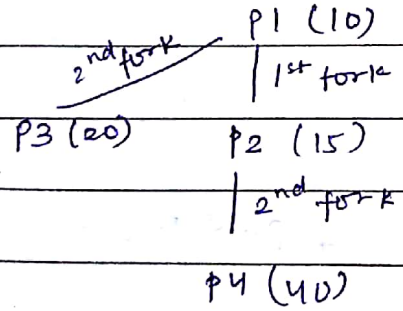
→ In Shell, it waits for child using wait command.
↓
system call

either I can specify pid of child for which it has to wait.

→ If I an error ⇒ child wasn't created only ⇒ -ve value of pid will be returned to parent only.
So, (pid < 0) part will be executed by parent only.
& (pid = 0) part will be run by child only because parent gets pid of child

→ wait (____);
↳ when child finishes, it sends some signal to parent to not ~~if~~ wait further.

→ 2 forks ( ) ;



Now, we 've 4 processes

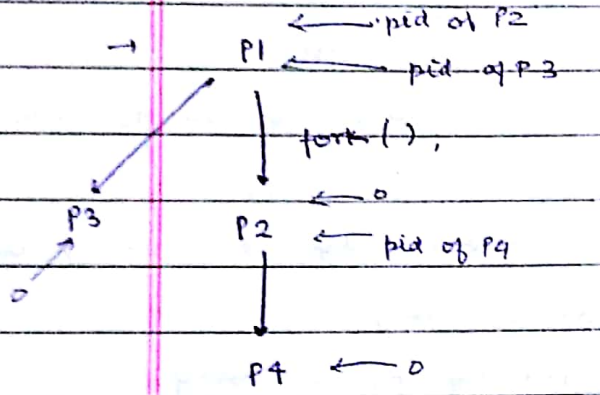
→ List of all processes are kept by OS in Process Table.
→ Parents keeps track of its children
→ With each process, I a data structure, Process Control Block (PCB)

Teacher's Signature

including PC

→ It has info about pid, current state, registers (content), list of open files, children also. (pids will be stored) PCB will be stored in Kernel space. It is associated with each process. (scheduling info, priority of process, user of process, ~~total~~ memory limit)

These PCBs could be linked with each other using linked list / doubly linked list.



22-01-18

→ If it is single processor : either child or parent can run. If we want child to run first, may be sleep(); can be used in parent's case.

→ fd table in parent process will be duplicated in child process also.

→ In code close(0); → closed
Open("file.c", "r"); → point to file 1

→ If open in ~~read~~ write mode & connect to STDIN ⇒ give error.

→ When child process terminates, it can return a value to parent process.

→ Context Switch : involves storing complete info. about previous process in PCB

PCB : could be in linked list.

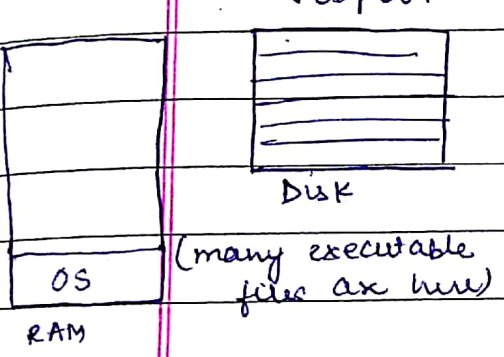
States of Process

→ When a child is created ⇒ new process is created

↑ state of process : 1) New :]

→ OS is finding an ~~use~~ unused int. & providing pid to that process, creating PCB for that state

→



"New" ~~fo~~ will also join Jobpool

2) "Ready" state : Only when picked up from Jobpool & put in RAM.

There is many processes in RAM.

Assuming single processor, (single core) only 1 process is running at a time

→ may have time slice

3.) "Running" state : As soon as it gets CPU & its ins^n are being executed.

Possibility while running

1.) either it gets completed ⇒ terminated

2) ~~process with high~~

2.) execution time finished / (time slice) ⇒ goes back to Ready state
high priority process came

3.) may go for an I/O ⇒ " " "

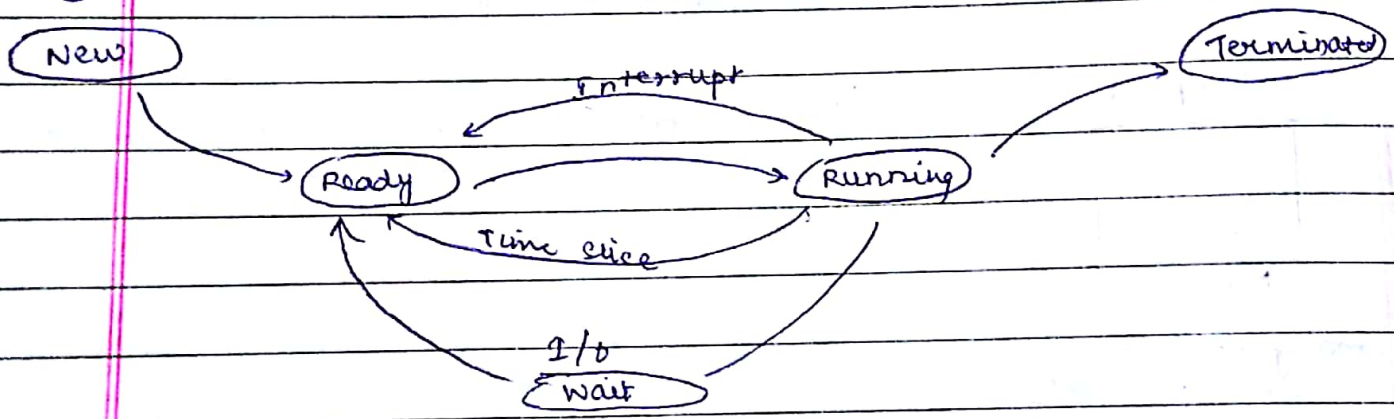
4.) Interrupt by h/w ⇒ " "
→ fork() ⇒ " " "

(context switching)

→ Wait/ state associated with I/O
Block

(Possibility that it could go back to disk also)

— (1)



Everywhere, there is a queue.

(waiting in queue for next stage)

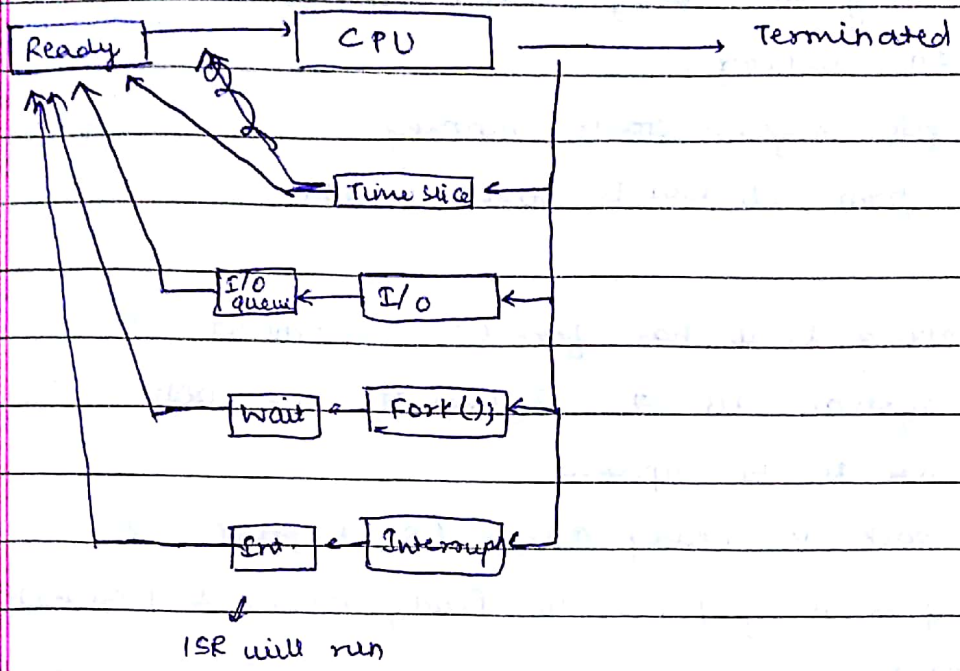
→ I/O queue → queue for keyboard

→ ~~Queue of~~

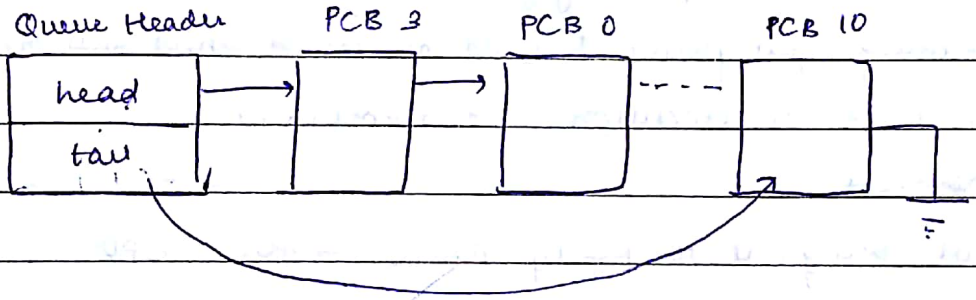
→ CPU queue : Ready Queue

→ Queue : can be represented as in linked list

Queue :



→ CPU Queue :



→ Device Queue (similar as above)

→ Ready Queue :- list of processes which are waiting for CPU They are in RAM.

→ Info about processes (in kernel space) are making Ready queue.

→ Out of these processes, 1 process will be picked up & PC will get its ~~*~~ add.

→ If i/o (printf) process is executing → system call → Kernel has to run OS will access device & print ~~no~~ on monitor,

Teacher's Signature

→ whatever was there in buffer is printed

→ CPU was sitting idle bcoz device controller was writing something on monitor.

→ so, some else program starts running.
All info. from PCB will be loaded in CPU.

⇒ P3 is running & it has fork(); command.

•) fork is system call ⇒ OS has to run now

•) P3 PCB has to be updated

•) It'll go back in Ready Queue / Block state.

•) It is going to bring process in Ready Queue & (maybe) assign CPU

•) child process is in running state
↳ could be interrupted by another process
↳ assuming just parent & child process ⇒ child will run, complete its execution ⇒ terminated.

~~OS will~~

•) OS will bring it to Ready Queue → give CPU

when child terminated ⇒ all its Resources removed. However, its PCB is still there (to give its exit status to parent)
↳ all return info.

When parent waiting for child & child has exited, it'll take info. from child's PCB & → ~~a~~ reaped. Then only, PCB of child is completely removed.

⇒ If parent isn't waiting for child
As soon as child goes for an I/O, parent gets CPU.

1.) Child process completed. PCB of child still exist. But parent is not worried about it. :- Zombie Process
{process whose return status hasn't been read by parent}

Teacher's Signature

2) If child is running & parent terminated first ?
then, child is Orphan now.
Now, ~~init~~ init() process will take care of . (adopts)

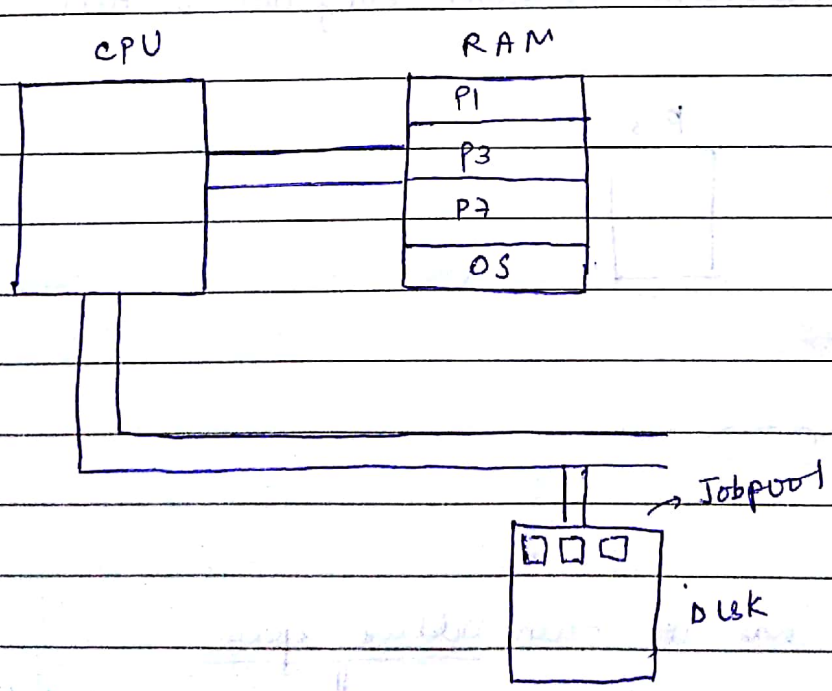
Diagram ① is taken care by CPU scheduler.

1) One scheduler is going to take program from JobPool to RAM :- longterm scheduler.
(how many program to shift, ...)
decides degree of multiprogramming.

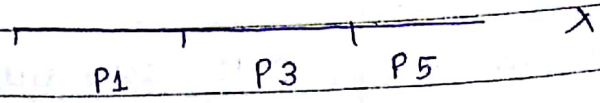
2) Out of program in RAM, which will get CPU :- shortterm Scheduler.

3) Out of all program, If it feels, 1 requires much memory. It removes 1 process for short term (kept in swap space designated as in HD) in hard disk & put that program in RAM :- Medium term scheduler.
It takes care of swapping.
1) maybe process is in block state
2) higher priority process has come

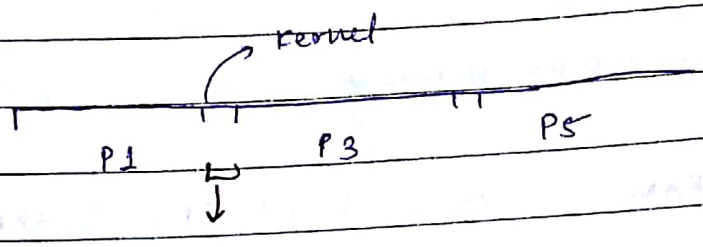
24-01-18



Context Switching



First, state of P1 will be saved & state of P3 will be loaded ⇒ some context switching taking place, OS running

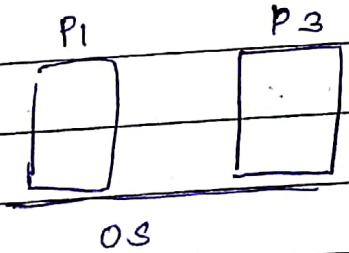


Context switch Overhead

→ We want to minimize context switching overhead.

⇒ Short term scheduler is working more frequently (has to work whenever there is time slice)

→ Sometimes, Mid term scheduler may not be there.



Each process has its own address space.

⇓

space assigned to particular process in a memory

Teacher's Signature

→ No process can access ∅ add. space of another process

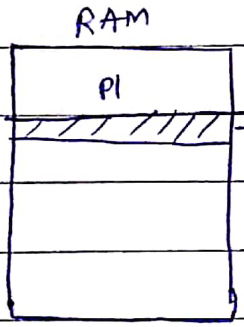
→ If they share parent - child relationship, even then, they can't access each other's add. space.

~~• If child~~

• If it is written wait (& x) ⇒ then some value can be returned from child to parent process.

Ways of Communication b/w 2 processes

1) Shared Memory



→ Shared mem. area b/w P1 & P2
Whatever, P1 wants to send, it'll send through this area only. Similar to Consumer - Producer problem.

~~maybe to process~~

Compiler (Producer) —asc. lang. code→ Assembler (Consumer)

→ A process in taking in info. about sensor (Producer) ⟶ Another process is checking whether given info. is in range or not (Consumer)

→ Circular Queue : Bounded Buffer (size of buffer is fixed)

in : → to position ∅ where producer can put data

out : → 1st consumer get

Queue is empty : in = = out

full : (in + 1) % buffer size = = out

One process is producer, other is consumer

Producer :

```
while (1) {

                                    // Producer        // don't do anything if full
while (((in+1) % bufsize == out));
    buf [in] = nextproduced;        ] → produce
    in = (in+1) % bufsize;

}
```

Consumer :

```
while (1) {

while ( in == out) ;                // don't do anything if empty
    nextConsume = buf[out];
    out = ( out + 1 ) % bufsize ;
}
```

4) System call in UNIX which creates shared memory :

```
shmid = shmget (key, size, flags);   ← area is created, P3 now needs to attach this area to its add. space
ptr   = shmat (shmid, NULL, 0);      ← by 1
                                     → It'll return ptr to shared mem
sprintf (ptr, "Hello", 5);           → P1 writes this msg. P3 is able to read it
```

shmid ← id of shared memory

shmdt ← Once comm^n is finished, shared mem. can be detached.

Teacher's Signature

mem _____ : shell (System call)

shmdt ( ) ; ⇒ definitely, it'll detach from P3. P1 will have the memory but P1 won't be able to access it.

* Once this shared mem. is created. Kernel plays some part in creating mem. After that Kernel has no control over msg passing
If bulk of msgs : this is good

2) Message Passing

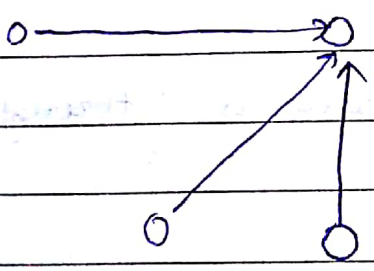
P1 will send msg in * some part of Kernel & P3 will get it from Kernel. Kernel is involved in whole msg. passing.
If limited msgs : this is good

→ Send (A, message)
Receive (B, message)

One-to-One, Symmetrical direct link has been associated to both process
→ Link is going through Kernel

→ 

One-to-One, Symmetrical



Asymmetrical :

receive (id, message)

→ Or we can create a mailbox. Send msg to mailbox & other person receives from mailbox

Mailbox : in Kernel Space

DATE : / /
PAGE NO. :

Things to be taken care of :

1.) Synchronisation : either I will receive, or all will receive ($P_1, P_2, P_3 \rightarrow P_1$ sends, who receives : only P_2 ? only P_3 ? both)

Type of synchronisations :

↳ Blocking send : I've sent a msg to you. I'll be blocked till you receive

↳ Non- " " : I've sent " "
not blocked (I'm doing my own work)

↳ Blocking Receive : blocked till I receive some msg

↳ Non- " " : Receive msg, Otherwise doing my own work

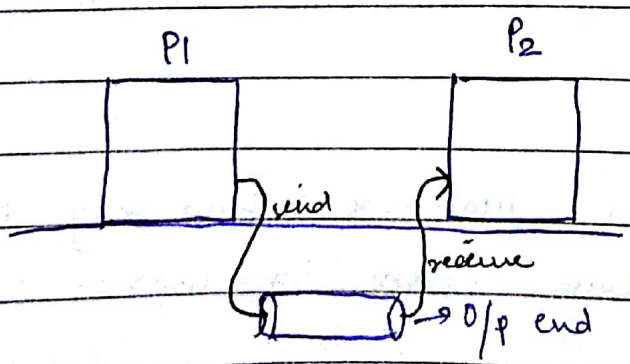
2.) Capacity of Mailbox:

Zero capacity : Nothing can be stored (Immediately send msgs as received)

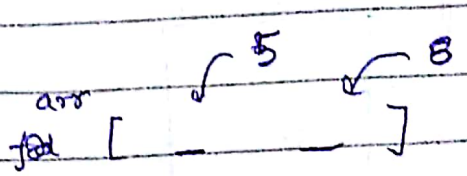
Bounded buffer : fixed capacity

Infinite " : No limit (can send as many msgs as we need)

* Other way of Comm" b/w processes is : through Pipe



Pipe : in Kernel space




```
int fd[2];
pipe (fd);
```

arr

It'll check fd table. Suppose 1st unused fd was 5, it'll assign 5 ~~0000~~, next unused was 8

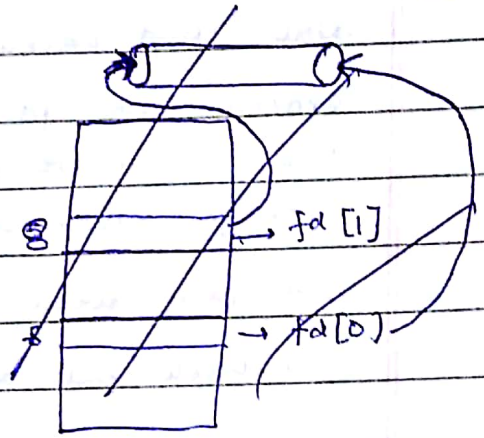
arr

fd[0] → output end of pipe → (read)

fd[1] → input end of pipe → (write)

arr

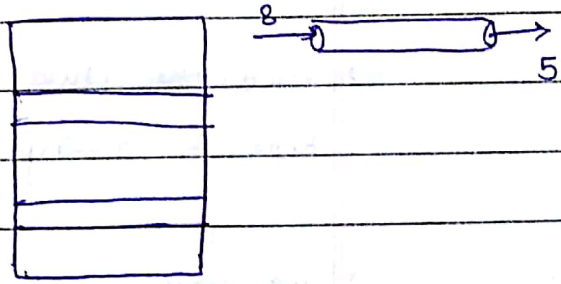
not fd 0 in table



29-01-18

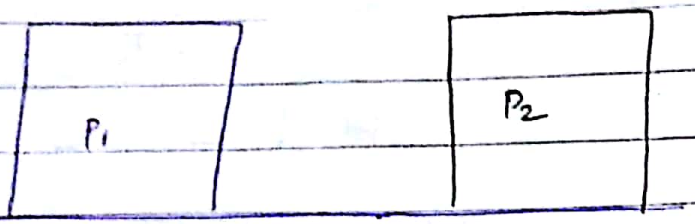
```
pid = fork ( ) ;
if (pid == 0)
{
    close (0) ;
    dup (arr[0]) ;
    read (arr[0], buf, len) ;
    close (arr[1]) ;
}
```

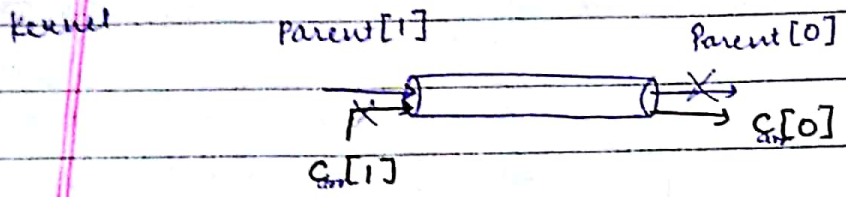
This assigning will occur after pipe ( ) is executed




```
if (pid > 0)
{
    // parent
    close (1) ;
    dup (arr[1]);
    close ( arr[0]);
    write (arr[1], buff, len);
```

→ after fork();





.Now, child process is created

Exactly same fd table is made for the child (all variable are passed on to the child. After that, if you change something in parent, it won't be reflected in child & vice versa ☺. However, if some common code is running variable will be same)

pid = fork();

⇓

it's not process id, it is value returned by fork.

→ Even the child's array [1] will point to input end of pipe & array [0] to output end

→ we want an interprocess comm^{n}.

(Assuming only parent is sending msgs to child)

So, I have to read from C_{arr} [0] & write from P_{arr} [1]

↓

child is reading from std. i/p.

→ As a child, I've to close my own std. i/p

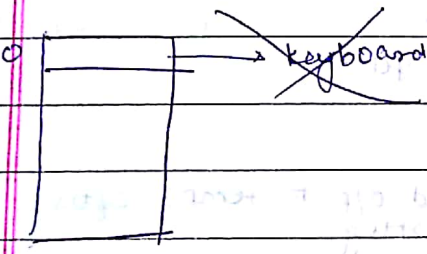
- close (0);

→ I want to connect that std. input to arr[0]
(child will read from o/p end of pipe (not the keyboard))
→ connect o/p of pipe to std. i/p of child.

dup (arr[0]); ⇒ now, it is connected.

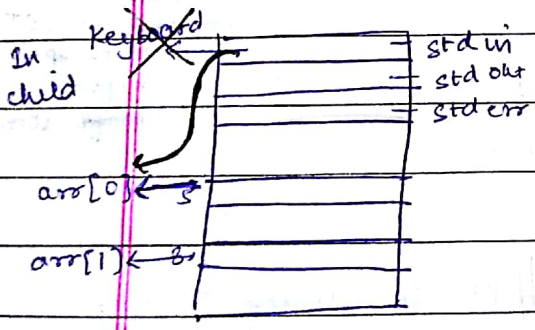
read ( ) : read from array

So, the child is ready to read. Also, we're not interested in child writing something (sending msg)
⇒ close (arr[1]);



close ( ); will remove ptr

→ In parent environment



⇐ close (0);

dup (arr[0]); Now, 1st unused fd is :0 it will duplicate ptr. to arr 0

* It only reads from STDIN

→ In parent : close STDOUT, connect arr[1] to STDOUT & close arr[0]

* When we close arr[0] in parent env., it'll change only in parent fd table

* After fork(); they are 2 diff. processes.

Shell Commands :

↳ $ sort → like ls

It will : fork() + exec ("sort")

i/p : my
hello
hi

o/p : hello
hi
my

↳ $ sort < file.txt

STDIN : file.txt
(take i/p from file & then sort)

↓

input redirection

→ fork() ;

* go into child process where sort will occur
This will create fd table,
↑ pointing @ fd 0

Internally : fork() + close(0) + Open ("file.txt", r) + exec ("sort")
(look 1st unused, get fd 0 & become STDIN)

↳ $ sort < file.txt > temp1.txt

Send o/p to temp1 after sorting.

↓

output redirection

* not written after exec() because have to connect STDOUT to temp1 before sorting

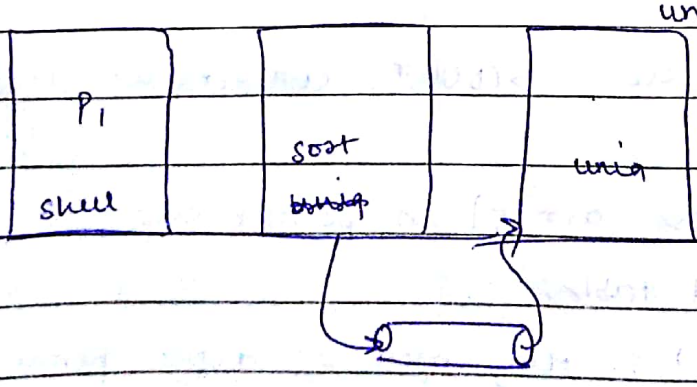
Internally :

fork() + close(0) + Open ("file.txt", r) + close(1) +
Open ("temp1.txt", w) + exec ("sort") :

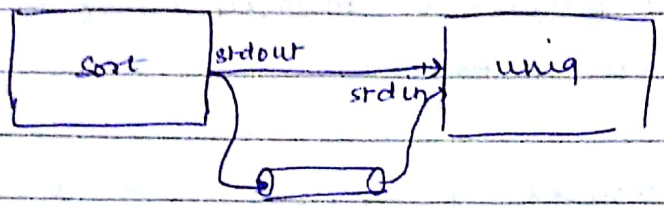
↳ $ uniq

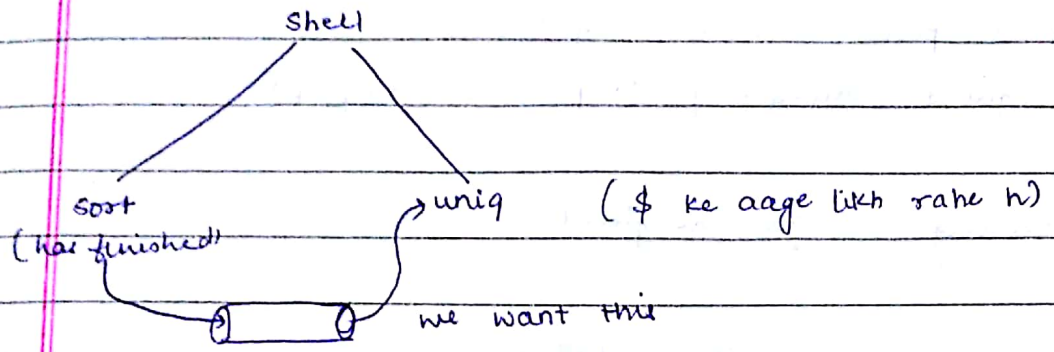
// Kernel utility to find unique words : remove duplicate words

uniq : taking i/p from STDIN & display on STDOUT



We want from sorting to do uniq





① Sort process khatm ho gaya but o/p aa gaya h pipe me.

~~After sort has already run.~~

→ create arr[2] in shell, do a pipe ⇒ array values are defined (they will get some fd)

then we will fork.

→ We'll come to sort.

Connect stdout of sort to i/p of pipe & sort the file

~~① → When forked again~~

$ sort < file.txt | uniq

① Shell me hi pipe kar diya h.

Sort me: Std out will be ~~do~~ connected to i/p end of pipe

STDIN: file.txt

executed → process finished.

However, o/p is available in pipe (fd)

NOW, I've to do fork to run 'uniq'.

Again, check if pid = 0 : close STDIN : connect to o/p end of pipe.

Multiprogramming :
We keep multiple processes in memory ready to execute.
CPU won't sit idle in this case.

→ Multi thread :. Instead of copying whole source code, only data part will be duplicated (done in Multiprocessing)
* light weighted (amount of occupied memory ↓)

→ Multiprocessing : high - weighted

→ System's performance will be better in case of multithreading.

→ Any multithreaded program uses 'pthread.h' library file
It is compiled as : gcc thread.c -lpthread

<pthread.h>

```
int main ( )
{   int n = 5 ;
    pthread_t tid ;
    pthread_attr_t attr ;
    pthread_attr_init (&attr)
    pthread_create (&tid, &attr, printChild, (void *)&n)
    // parent will execute this part
    // after this, we'll have 2 threads : one parent and one child
```

keeps track of different threads using their id's

memory allocated, priority of thread, etc

initialize attr. with default values

exact add. will be assigned only after thread is created but other attributes can be initialised here (like priority)

NULL (if don't want to pass any arguement)

actual add. will come in this 4th arguement ; to be passed in print Child

which threads you want to join

is pass a variable, whatever value returned by child will come

```
pthread_join (tid, NULL);          // for making parent wait till child is executing
for (i = 1 ; i <= n ; i++)
    pf ("in parent %d \n", i);
    return 0;
}
```

```
void *printchild ( void *param)
{   // child is executing this part
    int i ;
    int n = (int) *param ;
    for (i=1 ; i <= n ; i++)
        pf ("in thread %d \n", i) ;
```

There should be some func^n printchild
↳ void pointer

Once thread is created, control will shift to this func^n