annotations

Friday, September 27, 2024 1:26 PM

topic -

concept -

important concept to remember -

formula -

important note -

general info -

chapter - 1 (introduction and background)

Tuesday, September 24, 2024 11:19 AM

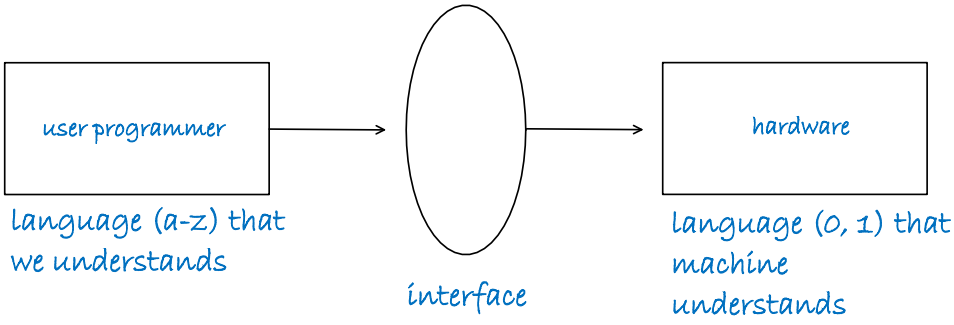
introduction and background

topic - understanding operating system

there are many different definitions of operating system :

(i) interface (intermediary) between user and hardware (computer)
(ii) resource manager : hardware + software
(iii) control programs.
(iv) set of utilities to simplify application development.
(v) acts (functions) like a government.

(i) interface (intermediary) between user and hardware (computer)



concept : micro operations

operations that are perfomed on the data stored in registers during a single clock cycle.

high level macro operations are converted into instructions and divided into micro operations

```
int a,b,c
c = a+b;
```

load r_1, a
store r_2, b
add r_1, r_2
store c, r_1

concept : memory

memory are of two types :
(i) primary/main/physical memory
(ii) secondary/auxiliary memory

- primary memory : RAM/ ROM/ cache

- secondary memory : disk, dvd

	primary	secondary
size	small	large
cost	high	low
speed	low access time	high access time

primary memory is volatile except ROM

volatility : once you switch off the system, you cannot access the content anymore. memory that only retains data while it's powered on.

concept : Von neunmann architecture (stored program concept)

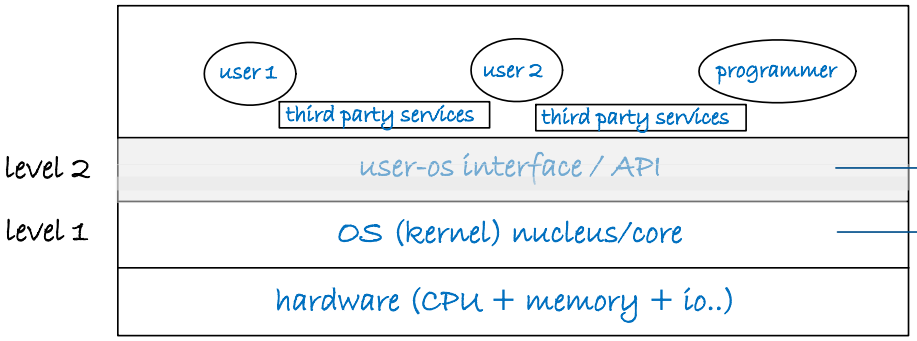
- main memory contain the instruction and data (jo bhi program processor par execute karana chahte hai that should be in main memory.
- ALU operating on binary data
- control unit interpret the instruction from memory and executes it
- Input/output equipment operated by control unit with the help of control signals.

general info : every computer system has its own architecture and har achitecture ke apne component honge har component ke apne functions hoga aur har architecture ka computing model hoga

operating system is dependent on architecture.
jab aap operating system develop karoge kisi particular archituecure ke lie toh vo usi pe work karega.

concept : command line interpreter

- CLI allows direct command entry



command interpreter

operating system is a collection of programs.
- process manager, memory manager, file manager, device manager, security manager

Operating system interface :
(a) level 2 : used by the user. (it cannot interact directly with OS)
user aur programmer ke lie alag alag interface hoga because they interact in different ways.
- user os interface : it is a interface for user .
- API/SCI : it is a interface for application programmer.
  API stands for application programmer interface.
  SCI stands for system call interface.
- third party services : we can access them without interface.

(b) level 1 : managing the core
this allows us to interact with the core. high level language to user OS interface.

command line interpreter in different operating systems :

(i) text based os : unix, dos (shell: command interpreter) no gui (graphics).

(ii) gui os : windows (dekstop: command interpreter)
window with all icons. in gui, desktop is a program that allow you to interact with os.

concept : booting

loading the operating system from disk to main memory.

there should be someone to initialise the disk, RAM and bring operating system to main memory.

this job is done by bootstrap, bootstrap is in ROM.

topic : function and the goals of operating system.

there is no universally accepted definition because there are different perception of operating system according to different peoples.

- kernel is the part that always runs on operating system.

functions :	goals :
(a) security	(a) convinience : easy to use
(b) protection	(b) efficiency : effective utilisation of resources.
(c) scheduling	(c) robustness : should not collapse easily.
(d) memory allocation	(d) reliability : should be reliable to use.
.	(e) scalability: ability to evolve.
.	(f) portability: able to run on different platforms.

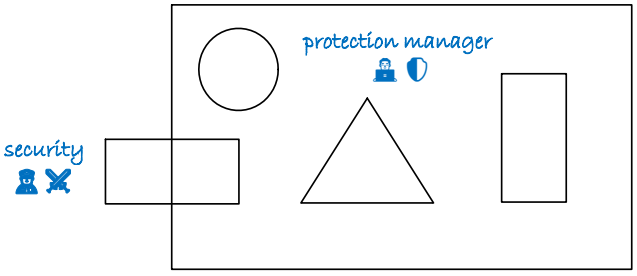
what is more important in operating system, convinience or efficiency?

convinience.

example :

windows is less efficient, reliable and robust compared to unix. unix is more efficient, reliable and robust but people still prefer windows because it is easy to use and it is GUI based.

concept : difference between security and protection



system

security : outside the system. only allows authorised programs,users,etc.

- firewalls, etc

- just like a security outside the campus only allows the authorised people to enter the campus.

protection manager : inside the system. checks if authorised users, programs are using the system or accessing the system in authorised way or not. it 's job is to giving the control access

- user(a) : only particular exe file, user(b) : read file but cannot write, user(c) : can write.

- just like the protection manager or the guard inside keep a watch if students are visiting permissible areas in campus and not trying to access thea restricted area like exam section, accounts section, hod office, etc.

topic : evolution of operating system.

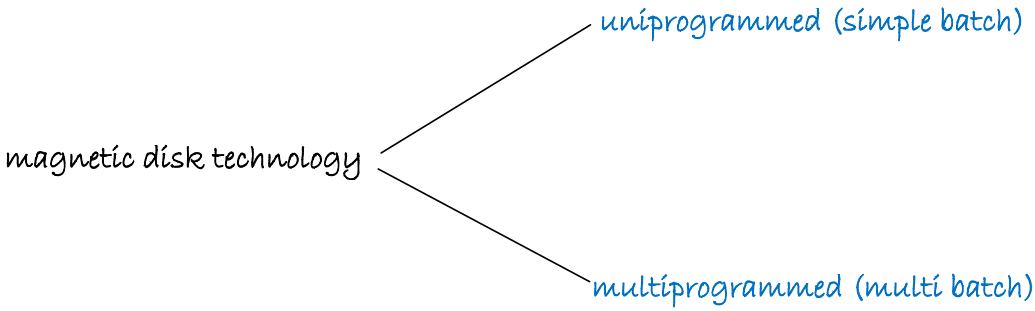
(i) first generation : 1930's to 1940's
- no operating system/manually
- vaccume tubes, punch cards (used by trained operators)

historical names :
(a) resident monitor
(b) supervisor
(c) operating system (os)

(ii) second generation : 1940's to 1950's
- magnetic types : batch processing

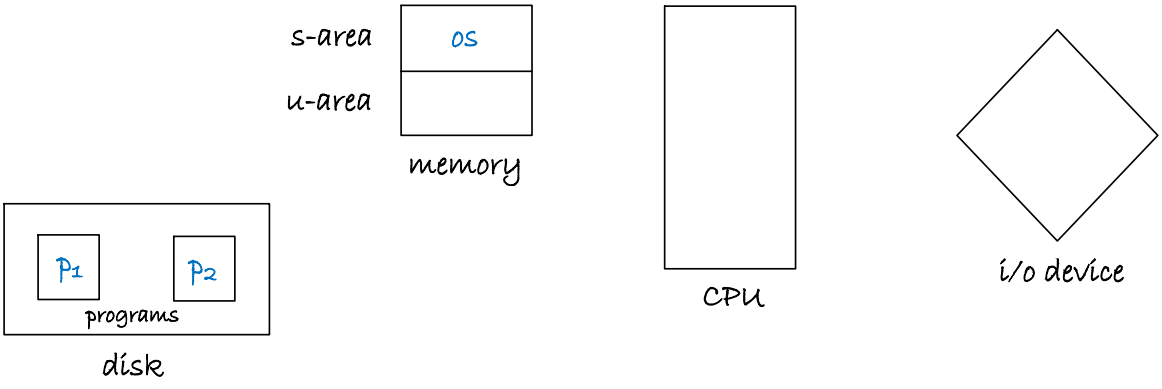
(iii) third generation : 1950's to 1960's
- evolved operating system

concept : uniprograming and multiprogramming

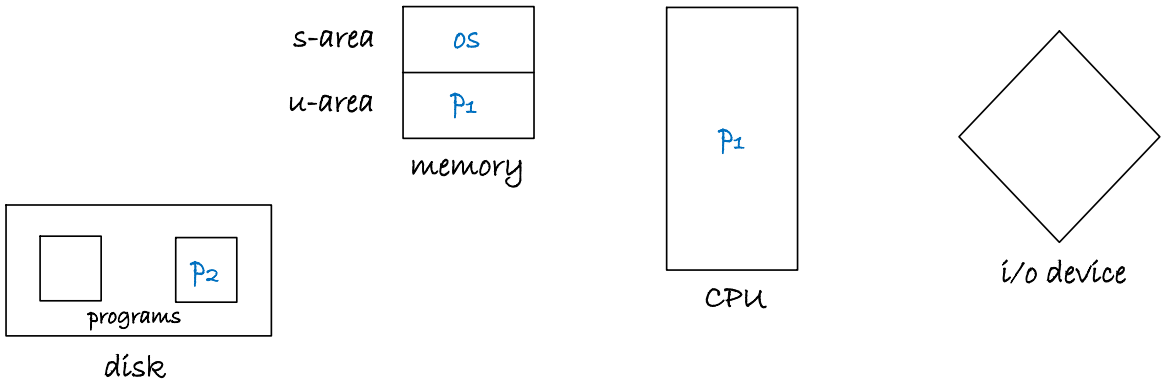


(a) uniprogramming : ability to local hold and manage a single program in memory

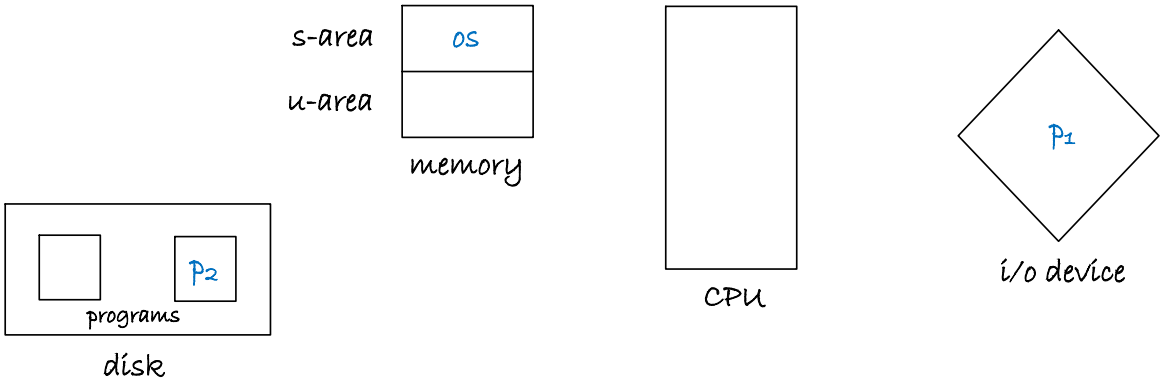
step (1) : os is loaded into system area through booting.



step (2) : program from the disk now goes to u-area and being executed in CPU.



step (3) : running program from the CPU now goes to i/o device for further operation.

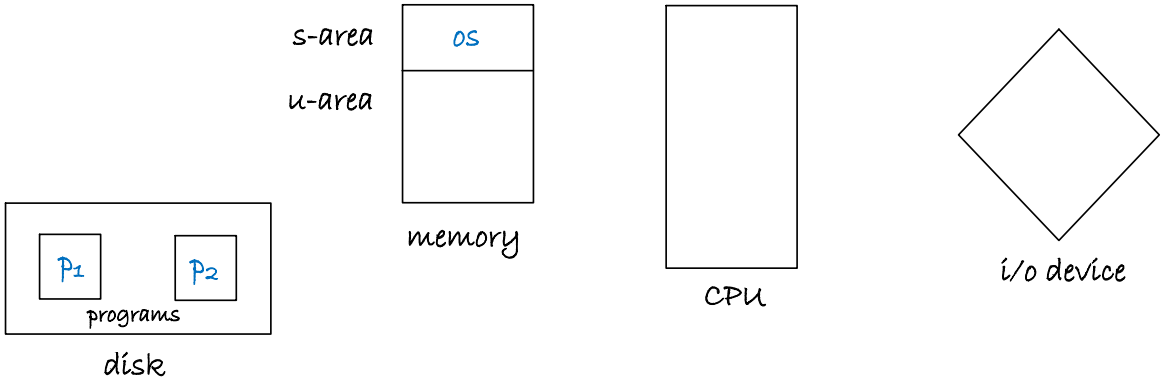


now, i want to load program P_2 for execution in CPU but it has only ability to hold a single program in memory, we have to wait for i/o task completion, meanwhile the CPU is idle.

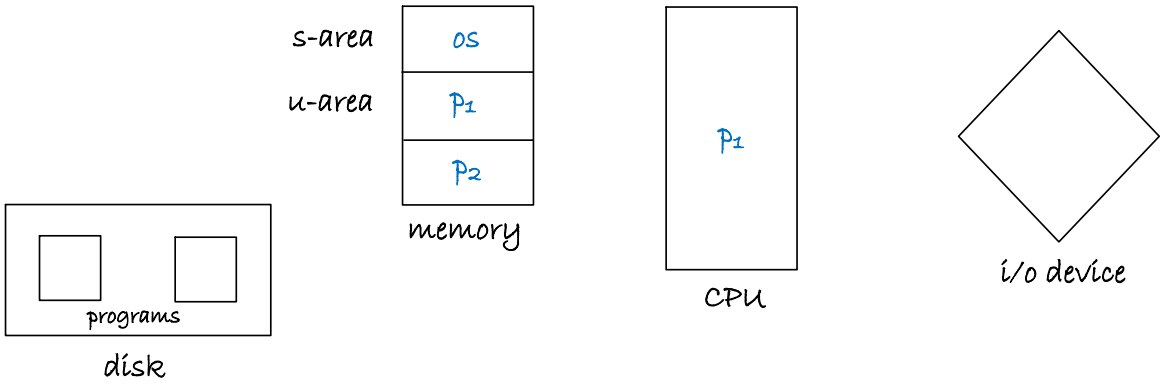
drawback of uniprogramming : idleness of CPU

(b) multiprogramming : ability to local hold and manage multi programs in memory

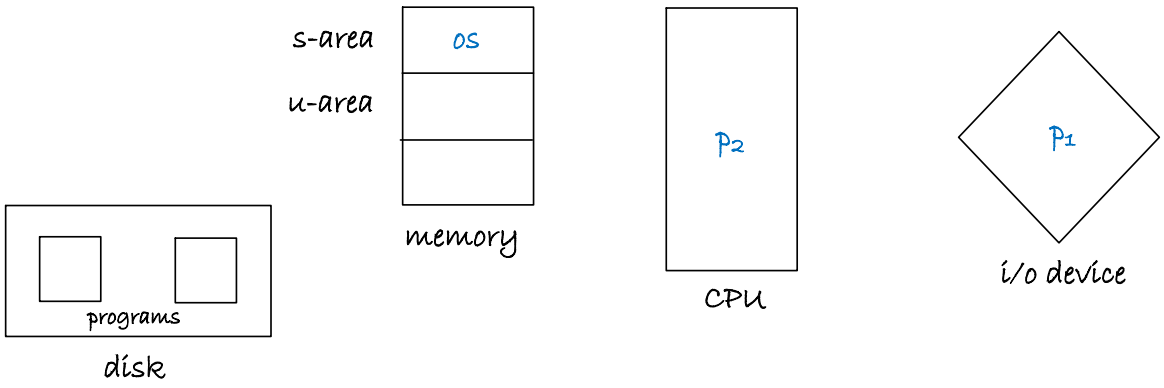
step (1) : os is loaded into system area through booting.



step (2) : program from the disk now goes to u-area and being executed in CPU.



step (3) : running program from the CPU now goes to i/o device for further operation and now p2 is running on CPU.



important point to remember : multiprogrammed os ability is to hold or manage multiple ready to run programs but the execution of programs are done once at a time on CPU, means CPU ek baar me ek hi program ko run karega.

objective : maximum CPU utilisation

concept : maximum CPU utilisation

---

maximum throughput (how many programs are completed under how much time)

$$\text{max throughput} = \frac{\text{number of programs completed}}{\text{unit time}}$$

it is a important performance matrix

topic : types of multiprogramming

CPU basically executes in two ways :

(a) non-pre-emptive : no forceful de-allocation

voluntary :

(a) it only leaves the CPU when it completes the execution.
(b) when it needs to go for i/O (or) system calls.
(c) if running program on CPU relinquished the CPU on its own (not forceful)

drawback :
(a) it is non-determinate.
(b) the other process is waiting to run on the CPU.
(c) lack of responsiveness.

(b) pre-emptive : forceful de-allocation

benefit :
(a) it enforces the interactiveness.
(b) it gives the chance to other program.

parameters for enforcing :
(a) timer : we can set the timer for program (even if it does not finish)
(b)priority : the other program with more priority will get the chance to execute.

topic : computing environments

(a) traditional : pc (convenient)

(b) mobile : battery/energy efficiency

(c) client server
(d) peer-to-peer

(e) cloud computing

(f) real time embedded : efficiency

real time system (RTS/RTOS) : the primary goal is efficiency, robustness and reliability.

real time systems (RTS) are of two types : characterised by strict deadlines constraints.

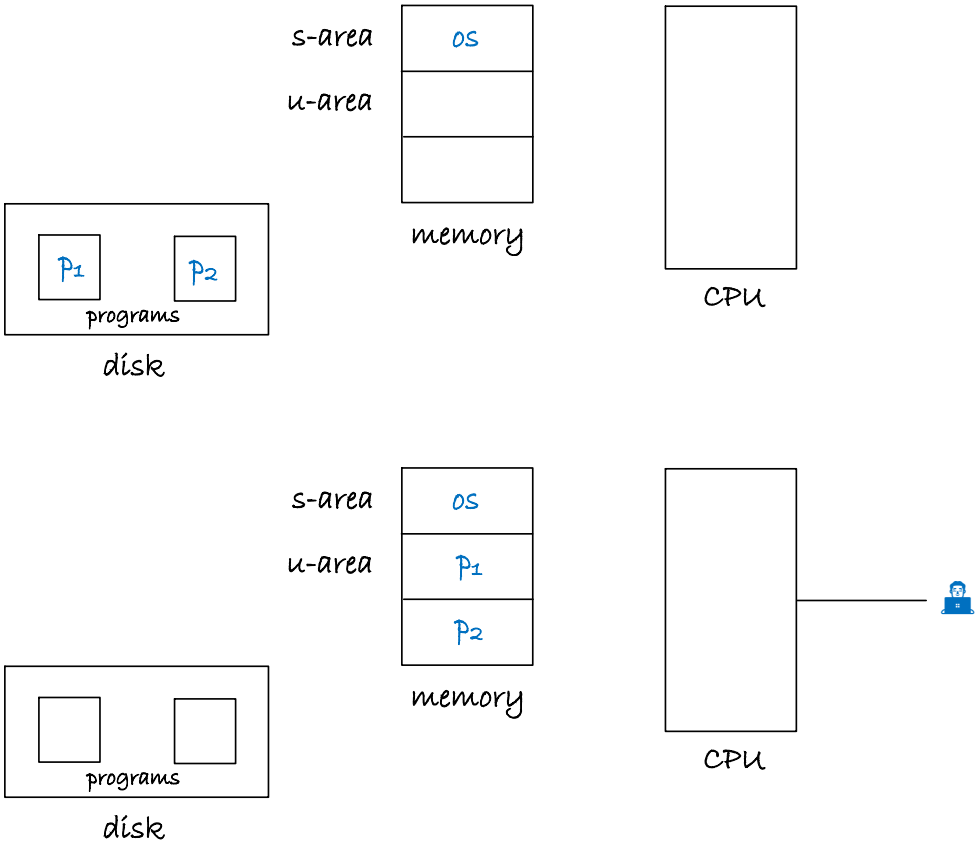
(a) hard : strict + loss
- example : nuclear system, satellite launch, missile control, etc (deadlines are strict because a little delay/miscalculation may cause a lot of trouble, loss and catastrophic failure in such cases.

(b) soft : loose + no loss
- example : ATM, weather forecasting, etc. (deadlines are losse because there is no loss or the chances of loss are very low like an ATM may take a extra minute to cash out but that does not cause any catastrophic failure)

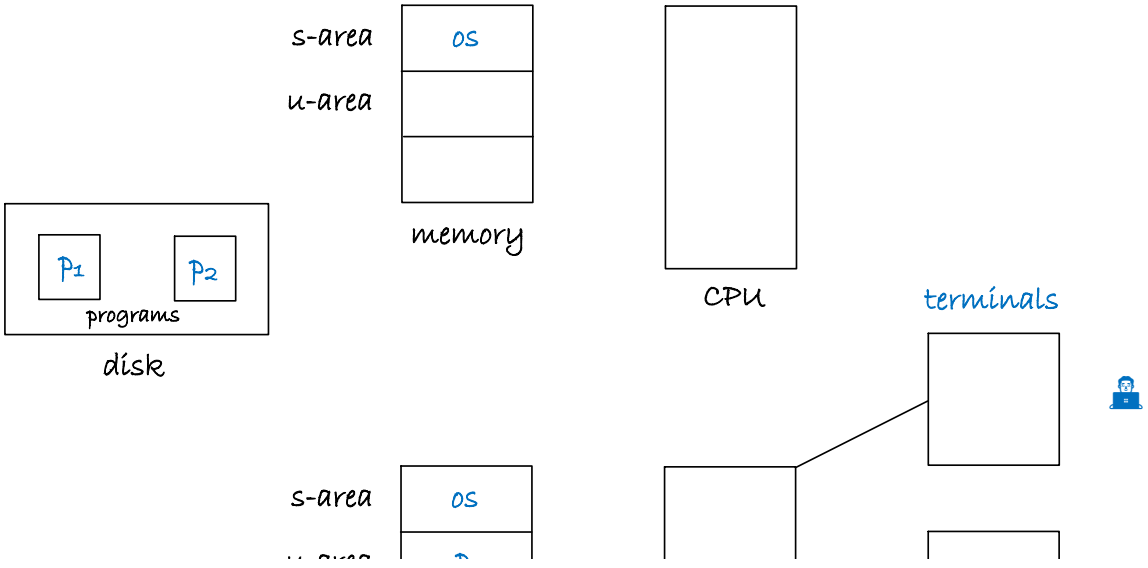
concept : multiprogramming

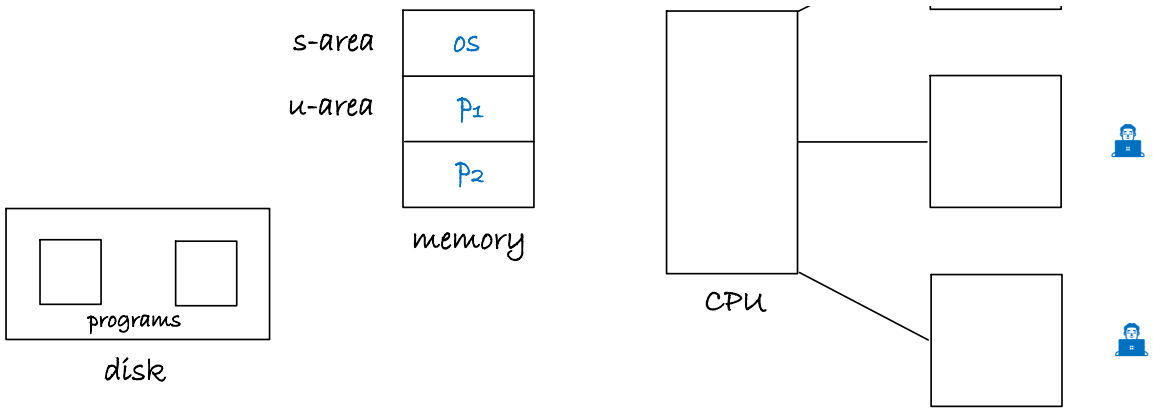
multiprogramming are of two types : the types are differentiated from user's perspective but from os perspective thereare no types.

(a) single user based : if a single user is working on the system.



(b) multi user based : if multiple users are working on the same system.





terminals are not the dedicated system but simply a dumb terminal in this sense is a text-based interface that allows users to input commands directly to the operating system (no process/no memory)

topic : architectural requirements for implementing a typical pre-emptive based multiprogramming in operating system.

[in laymen terms : what do you expect form hardware (requirement)]

(i) io (secondary storage) : in which mode data transfer will occur (direct memory access : DMA)

beside this there are different types of transfer :
(a) program control transfer : processor (CPU) is repsonsible for tranferring, no multiprogramming.

(b) interrupt driven : processor (CPU) is not directly involved for tranferring, but requires the permission hence processor is involed indirectly.

(c) direct memory access (DMA) : processor (CPU) will not be involved in the data transfer between the disk and memory, only the information goes to processor.

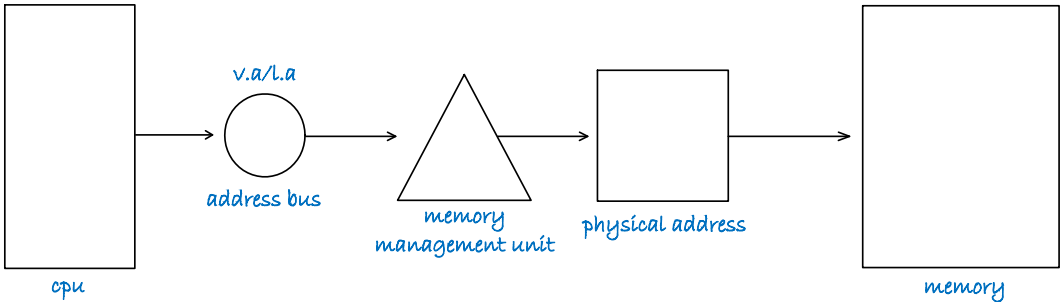
(ii) memory : address translation

- address generated by the running program on CPU : virtual address/logical address

- address needed to access memory : physical address

memory management unit : hardware (to take every generated logical address and convert to physical address)

if you allow the program running on cpu to access physical memory directly it can also access os, other programs.



(iii) CPU: dual mode operation

at-least two modes of operation of CPU :

(a) user mode : all user applications run in user mode.

- user mode is pre-emptive (forceful de-allocation).
- it is not important that all instructions are executed here.

examples : c, c++, java, database, compiler, google, etc.

- universal law : program/process running in user mode can get pre empted after execution of any instruction. (interleaved execution)

cause : anything: high priority program aa jaye, etc

(b) kernel mode : all routines/program/servies of OS runs in kernel mode.

- kernel mode is non-pre-emptive (no forceful de-allocation/atomic).
- it is important that all instructions are executed here.

examples : os programs, system routines, system calls, etc.

- universal law : program in kernel mode are non-pre emptive (atomic) once you start you complete the execution. so there is no corruption of data.

- privileged instruction : instruction that operates on hardware, that can change the data structures, that change the hardware settings are privileged (runs on kernel)

concept : mode shifting

shifting between kernel mode to user mode and user mode to kernel mode.

mode bit : it is a special bit in hardware/processor, one can distinguish between user mode and kernel mode.

- os is a service provider, os services are executed in kernel mode
- to avail os services from user applications, mode shifting is needed.

jab bhi aapko koi os ki service chaiye hogi toh we have to transfer user to control and then again we come back to user form kernel.

examples :

user mode

(i)
```
main ()
{
  int a,b,c ;
  c = a+b;
  f(c);
}

f(int k)
{
  printf ("%d", k);
}
```

this code is completely executed in user mode. there is no statement that is triggering system calls.

(ii)
```
main ()
{
  int a,b,c ;
  a=1, b=2;
  c = a+b;
  fork ();
  f(c);
}
```

kernel mode

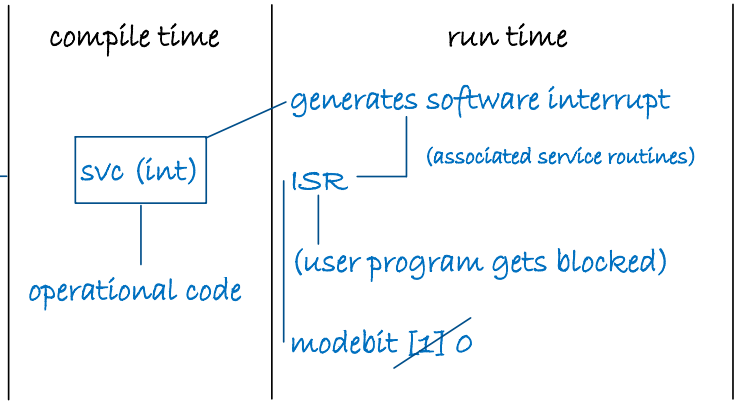
fork () ——— code/routine (in os kernel)

it is a call to operating system for availing some service

fork : to create child process
process creation is a pure os service.

concept : what happens when u call fork?

```
main ()
{
  int a,b,c ;
  a=1, b=2;
  c = a+b;
  fork ();
  f(c);
}
```



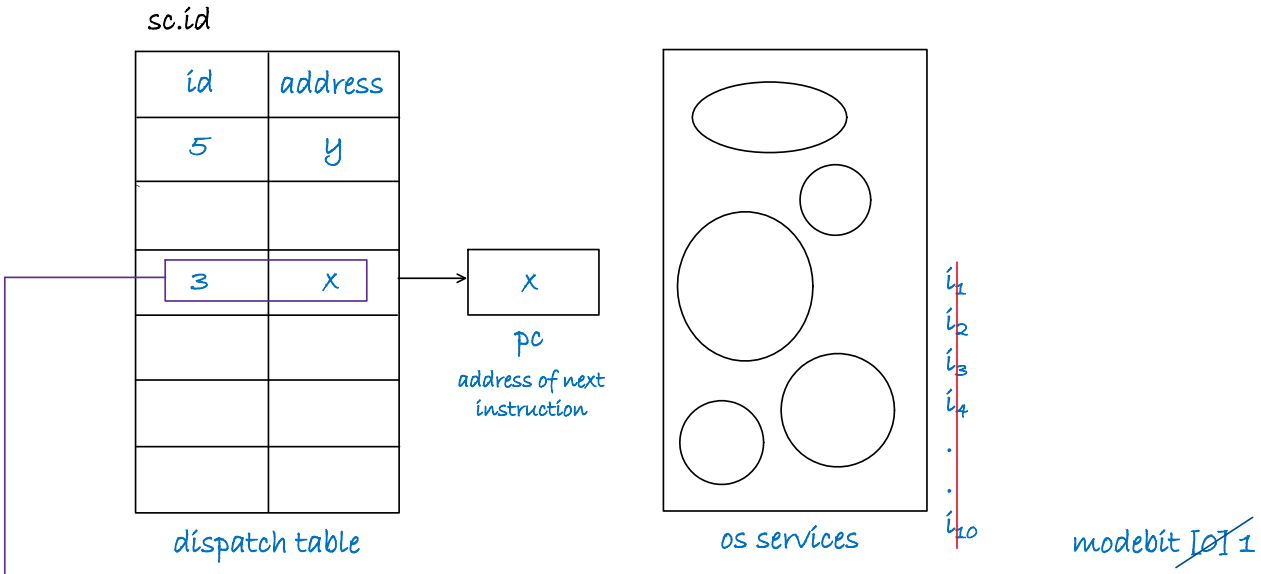
- compile time :

compiler will convert that fork system call to operation code(svc(int))
[svc : supervisory call, call to os. int : software interrupt instruction]
happens during compilation time.

- during run time :

execution of svc(int) at run time will generate the software interrupt
har ek interrupt ka ek associated service routine hogi jiska naam hoga ISR, this contain detail about how to service the interrupt.

ISR will be activated and the user program get blocked (ISR took the charge, CPU ko chod dega program, ISR will go to mode bit (to 0) to shows that it is in kernel mode (no interruptioms/saare interrupt mechanism closed)



dispatch table : information about all system calls that can be activated by user. every system call will have an entry in dispatch table.

har system call ka id hoga and in dispatch table ek system call id hoga jiske sath address hoga jaise fork call kia hai uska "sc.id" hai 3 jo ki dispatch table mai hai.

ISR aayega aur index 3 padega dispatch table mai aur jo address hai corresponding usko program counter mai dalega

- PC contains address of next instrution to be executed.

- CPU par ab fork execute karega, all inst of fork are being executed atomically.

system calls ke saare instruction execute hone ke baad last instruction (priviliged) jayega mode bit par vo wapas mode bit ko 0 to 1 krdega (user mode)

it will unblock the process, gets in ready state, runs on CPU an executes next statement.

(i) user application can directly make a system call.

(ii) user defined functions can also make a system call.

(iii) likewise pre-defined functions (printf, scanf) can also make system calls.

concept : header files

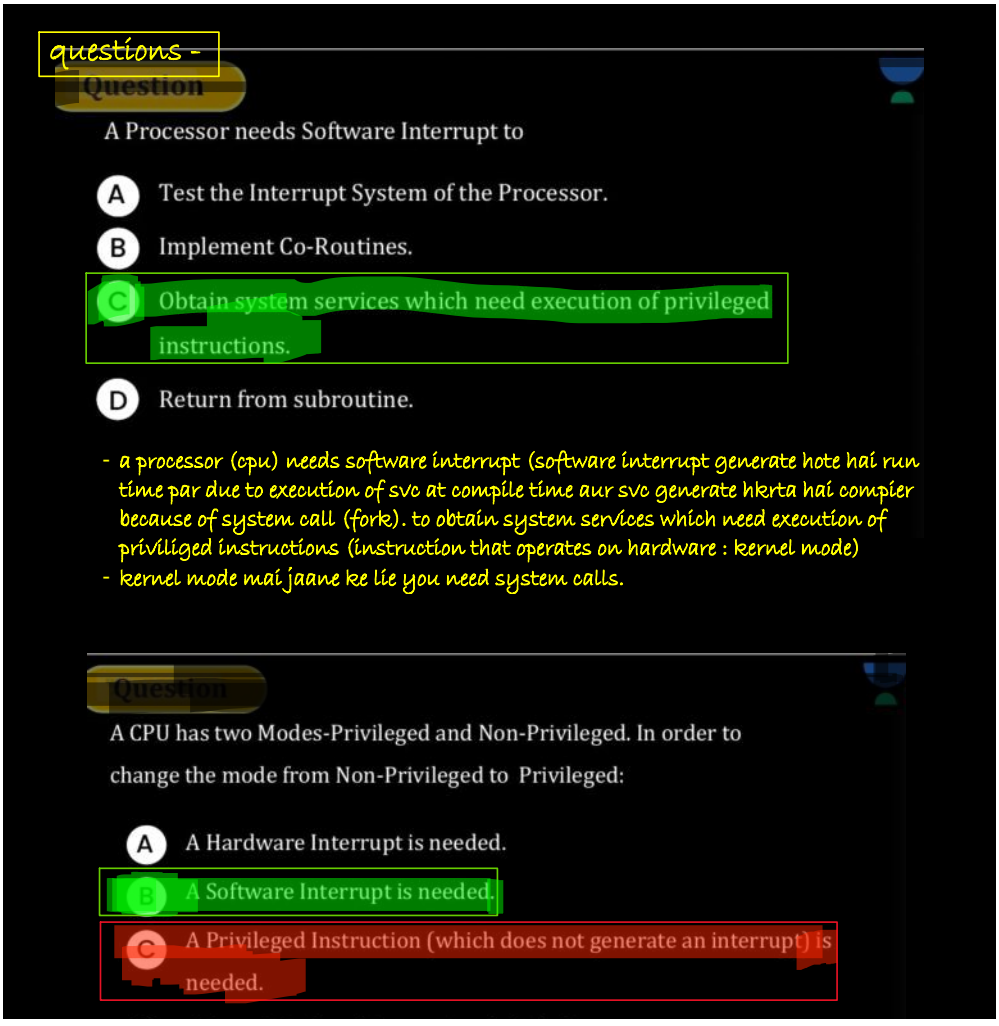
header files are used for type checking during compilation.

the code of printf will be in library file (.lib) it is in object file .exe executable code we cannot read it.

printf may call a system call by name "write" but printf is executing in "user mode". only on system call it switches to kernel.

system programs : whenever any program makes use of system calls.

printf : system program.



C A Privileged Instruction (which does not generate an interrupt) is needed.

D A Non - Privileged Instruction (which does not generate an interrupt) is needed.

- interrupt ki zarurt toh padti hi hai because that software interrupt basically will be services through ISR.
- a privileged instruction hota if only vo ek interrupt generate karta.
- a software interrupt is needed.

Question

System Calls are usually invoked by using:

A A Software Interrupt

B Polling - when a process continuously test the device.

C An Indirect jump

D A Privileged Instruction. a privileged instruction (ho sakta interrupt generate na kare)

Question

A computer handles several interrupt sources of which the following are relevant for this question:

- Interrupt from CPU temperature sensor (raises interrupt if CPU temperature is too high)
- Interrupt from Mouse (raises interrupt if the mouse is moved or a button is pressed)
- Interrupt from Keyboard (raises interrupt if a key is pressed or released)
- Interrupt from Hard Disk (raises interrupt when a disk read is completed)

Which one of these will be handled at the HIGHEST priority?

A Interrupt from Hard Disk

B Interrupt from Mouse

C Interrupt from Keyboard

D Interrupt from CPU temperature sen

- may cause catastrophic and permanent damage to processor components if temperature raises high.

Question

Which of the following services in least likely to be provided by an Operating System?

A accounting of resource usage

B Database Management System

C Memory Management System

D Protection modes for files

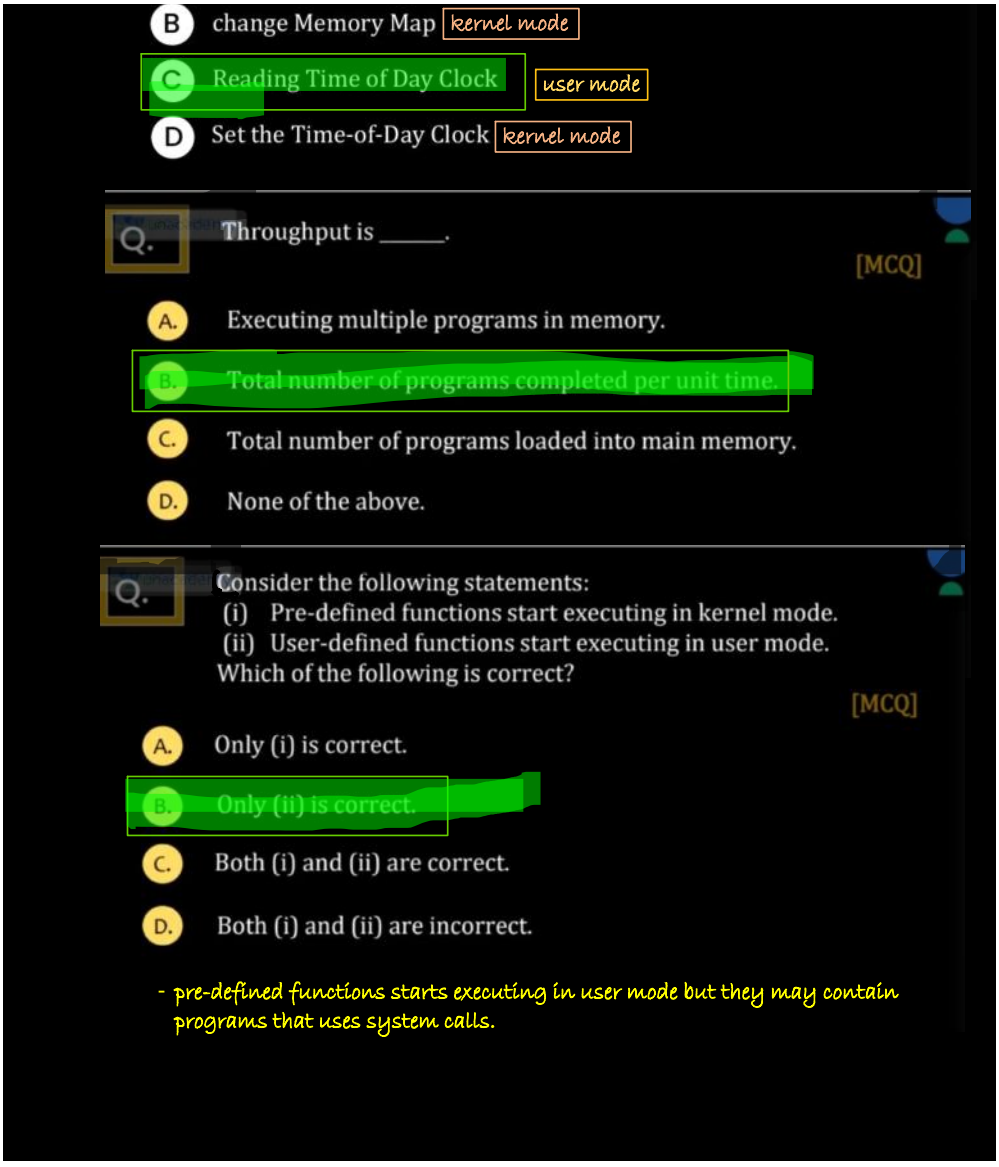
Question

All of the following Instructions should be allowed in Kernel mode except?

A Disable Interrupts kernel mode

B change Memory Map kernel mode

C Reading Time of Day Clock user mode



topic : system calls

- system calls are programming interface to the services provided by OS.
- accessed by programs via high-level language application program (API) rather than system call use.

interfaces to avail OS services :
(a) WIN - win 32 API
(b) UNIX/LINUX - POSIX API (posix : standardization for unix)
(c) JAVA - java API

how system call works?
- a number associated with easy system call (us number ke according system call interface ek dispatch table maintain karta hai)
- system call interface invokes intented system call in OS kernel and return status of the system call and any return values.
- the called need know nothing about how the system call is implemented. (abstraction)

concept : parameters passing mechanism

(a) call by value :

- receiving formal parameters should be of same type, size, category.

- value of the actual parameters are passed to the formal parameters and then the relationship between them are broken.

- changes are not reflected in actual parameters.

(b) call by reference:

- receiving formal parameters are not same as actual parameters, they are basically pointer variables storing address of actual parameters.

- changes are reflected in actual parameters.

concept : how to pass parameters in system calls

every process is attached with two stacks :

user stack : storing activation record of user . pre defined functions.

kernel stack : system call/parameter passing through stack.

there are three general methods use to pass parameters to the operating system : user mode

(a) simplest : pass the parameters in registers.

- system calls will take the value from register.
- drawback : parameters are more than the registers.

(b) parameters stored in a block or table, in memory, and address of block passed as a parameter in register.
- unix (linux and solaris) user stack

(c) parameters places, or pushed, onto the stack by the program and popped off the stack by the operating system.

kernel stack

important types of system calls :

Some System Calls For Directory Management	
Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Process Management	
Call	**Description**
pid = fork()	Create a child process identical to the parent
pid = waitpid (pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File Management	
Call	**Description**
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Miscellaneous	
Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan 1,1970

	Windows	Unix
Process control	CreateProcess() ExitProcess()	fork() exit()

	Windows	Unix
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device manipulation	SetConsolMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information maintenance	GetcurrentProcessID() SetTimer() Sleep()	getpid() slarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSEcurityDiscriptorGroup()	chmod() umask() chown()

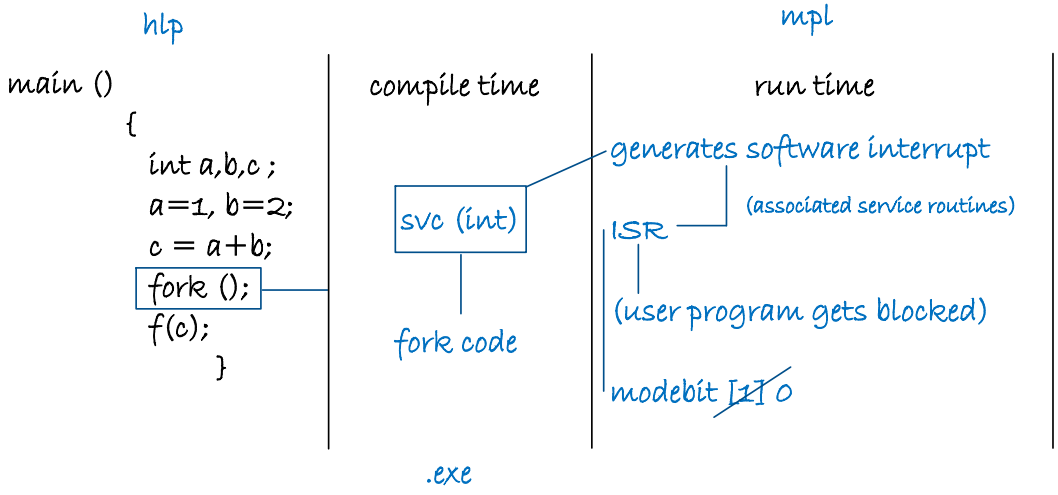
- access: checks if calling process has file access
- alarm: sets a process's alarm clock
- chdir: changes the working directory
- chmod: changes the mode of a file
- chown: changes the ownership of a file
- chroot: changes the root directory
- close: closes a file descriptor
- dup, dup2: duplicates an open file descriptor
- execl, execv, execle, execve, execlp, execvp: executes a file
- exit: exits a process
- fcntl: controls open files
- fork: creates a new process
- getpid, getpgrp, getppid: gets group and process IDs
- getuid, geteuid, getgid, getegid: gets user and group IDs
- ioctl: controls character devices
- kill: sends a signal to one or more processes
- link: links a new file name to an existing file
- lseek: moves read/write file pointer
- mknod: makes a directory, special or ordinary file
- mount: mounts a filesystem
- msgctl, msgget, msgsnd, msgrcv: message passing support
- nice: changes priority of a process
- open: opens a file for reading or writing
- pipe: creates an interprocess pipe
- plock: locks a process in memory
- ptrace: allows a process to trace the execution of another
- read: reads from a file
- semctl, semget, semop: semaphore support

- read: reads from a file
- semctl, semget, semop: semaphore support
- setpgrp: sets process group ID
- setuid, setgid: sets user and group IDs
- shmctl, shmget, shmop: shared memory support
- signal: control of signal processing
- sleep: suspends execution for an interval
- stat, fstat: gets file status
- stime: sets the time
- sync: updates the super block

topic : system programs

programs that make system call are known as system programs.

(a) fork system call : creates a new/child process.



(i) compiler generates .exe (bypassed linking phase)

(ii) executable file created and placed on disk.

- there is no existence of process without program. whenever you load any program for execution into main memory it become process.

general info - in unix/linux system if you compile a program and it compiles into .exe and if there are no errors then by default its name is "a.out"

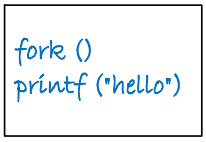
you can also rename it with the command

example :

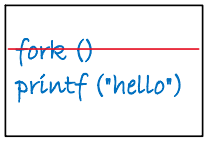
```
main ()
        {
         fork ();
         printf ("hello");
                        }
```

fork creates a child process

(i)



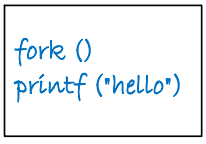
parent



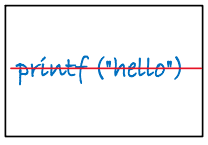
execution in child will start from next statement after fork

child

(ii)



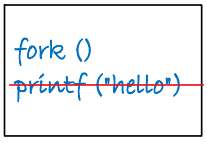
parent



hello

child

(iii)



hello

parent

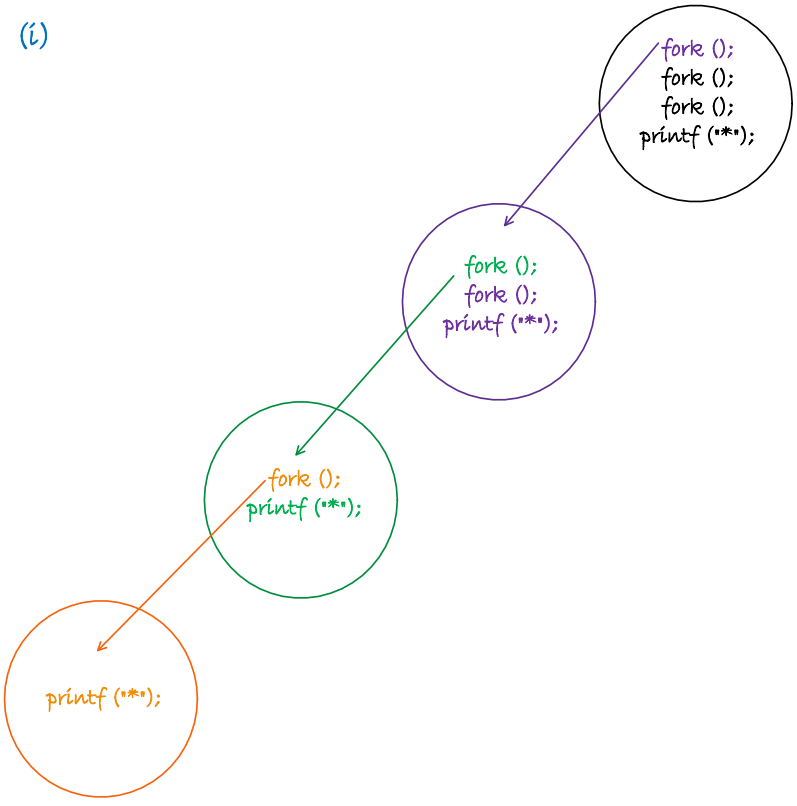
child

output : hello hello

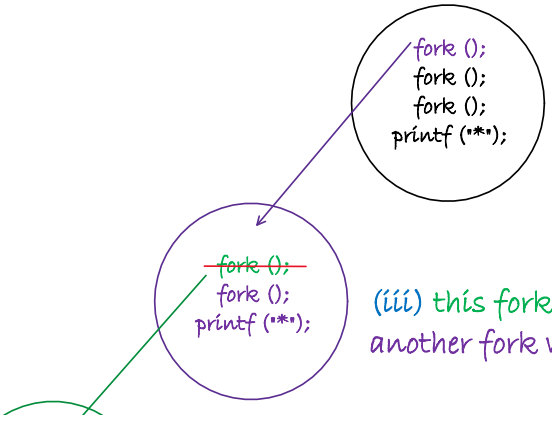
important point - child ka execution by rule will start from next statement after fork otherwise infinite child process create ho jayege.

```
main ()
        {
         fork ();
         fork ();
         fork ();
         printf ("*");
                        }
```

(i)



(ii)



(iii) this fork has already created another child another fork will execute.

printf ("*");

another fork will execute.

~~fork ();~~
printf ("*");

*

(ii) this fork has already created another child printf will execute.

~~printf ("*");~~

* (i) printf will execute

(iii)

fork ();
fork ();
fork ();
printf ("*");

~~fork ();~~
~~fork ();~~
printf ("*");

~~printf ("*");~~

*

*

*

(iv)

~~fork ();~~
fork ();
fork ();
printf ("*");

(i) this fork has already created another child.
(ii) this fork will execute.

*

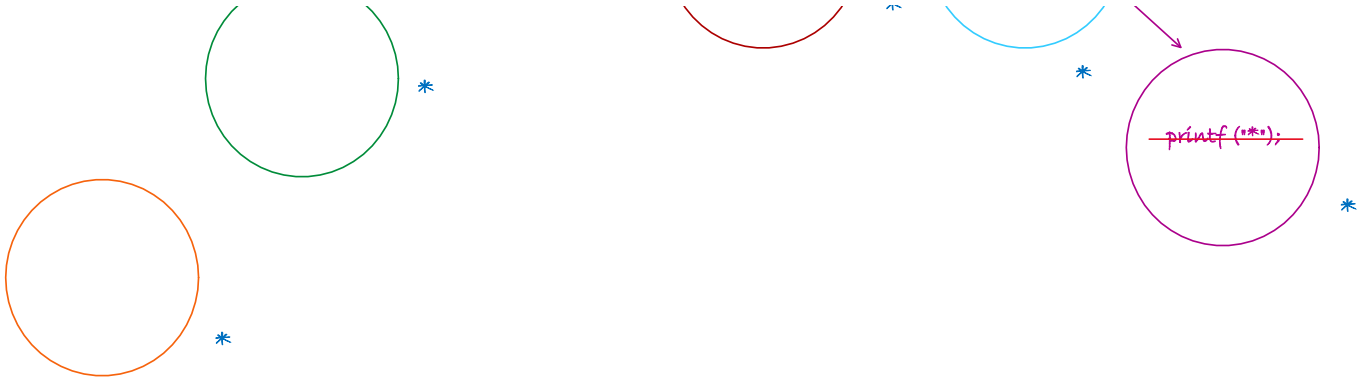
*

~~fork ();~~
printf ("*");

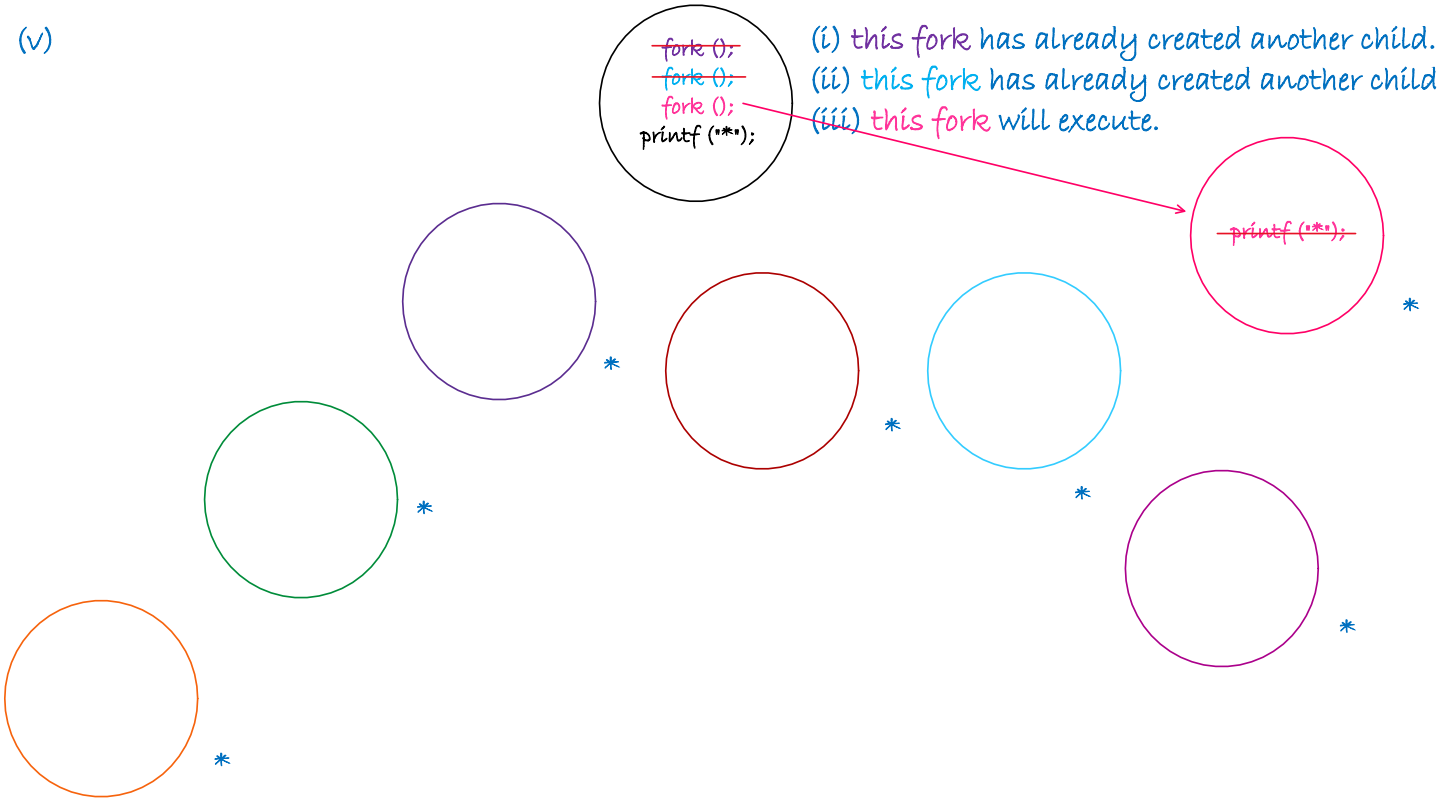
(ii) this fork has already created another child - printf will execute.

*

*

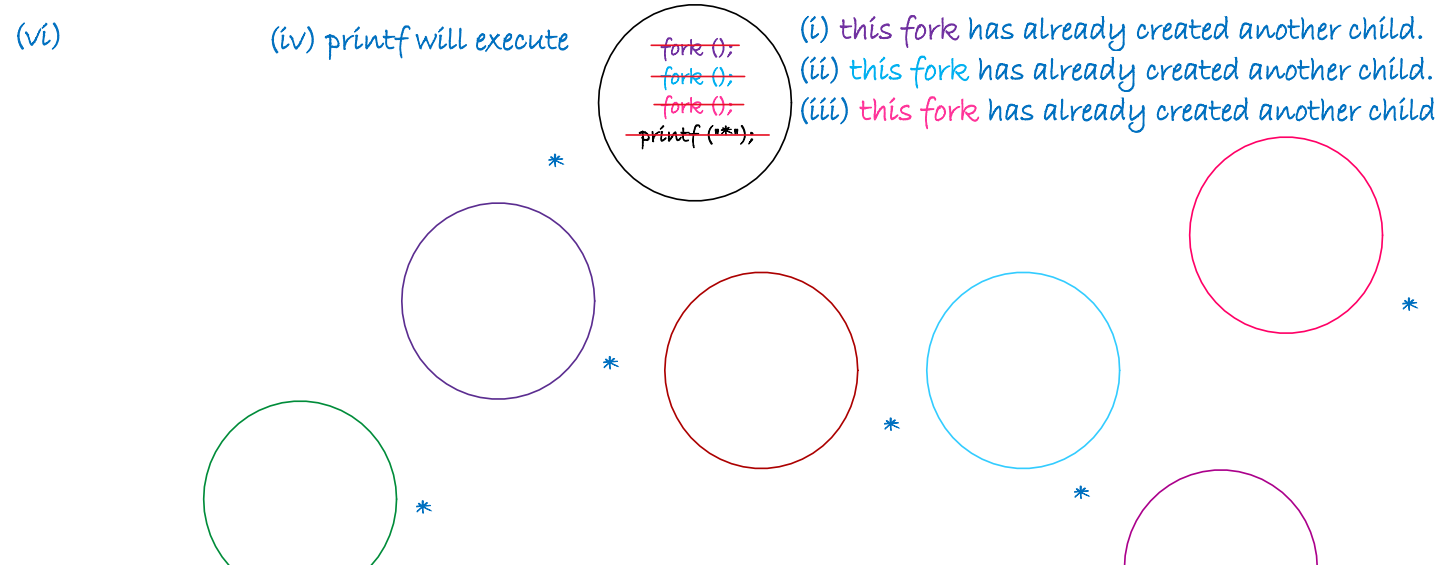


(v)



(i) this fork has already created another child.
(ii) this fork has already created another child.
(iii) this fork will execute.

(vi) (iv) printf will execute



(i) this fork has already created another child.
(ii) this fork has already created another child.
(iii) this fork has already created another child.

total times printed : 8
total processes : 8 (7 child and one parent)

generalisation :

number of forks in series	total processes	number of child
1	2	1
2	4	3
3	8	7
4	16	15
5	2^n	(2^n-1)

```
main ()
        {
         printf ("hi");
         fork ();
         fork ();
         printf ("hello");
         fork ();
                }
```

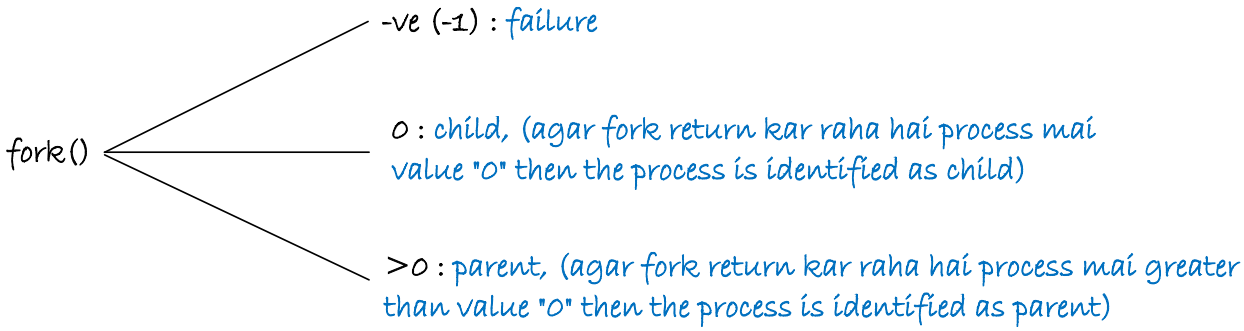
whichever processes are created iwth this last fork, they will not print anything.

```
main ()
        {
         int i, n;
         for (i=1, i<=n; ++i)
         fork ();
                }
```

total number of processes = 2^n

number of child processes = 2^n-1

concept : return value of system call



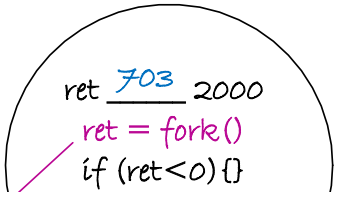

```
main ()
        {
         int ret;
         ret = fork ();
         if (ret<0)
         {
          printf ("failure");
          exit ();
                 }
         {
           if (ret==0)
                          }
         else (ret>0)
         {
          --------
                 }
           --------
                  }
```

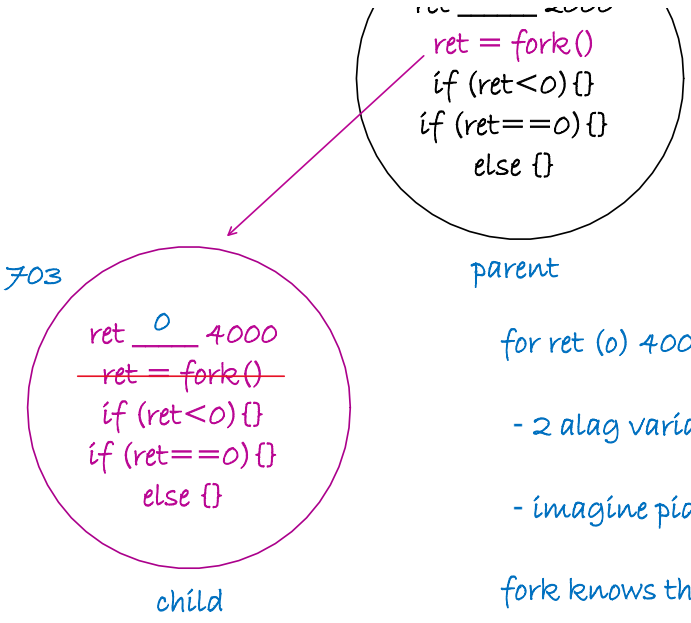
```
main ()
        {
         int ret;  parent
         ret = fork ();
         if (ret<0)
         {
          printf ("failure");
          exit ();
                 }
                 {
          if (ret==0)
          {
            --child---
                          }
          else (ret>0)
          {
            parent
                 }
          child + parent
                  }
```

value lesser than 0 (-1)islie failure

agar fork return kar raha hai process mai value "0" then the process is identified as child

agar fork return kar raha hai process mai greater than value "0" then the process is identified as parent





for ret (0) 4000 likhega

- 2 alag variables with same name and different locations.

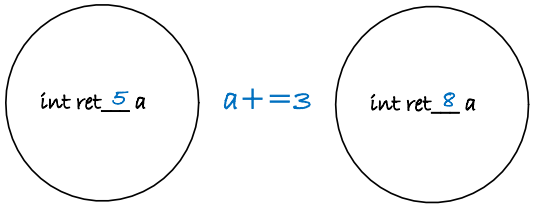
- imagine pid is 703 toh vo 703 parent me likhega

fork knows the process id, now fork leaves.

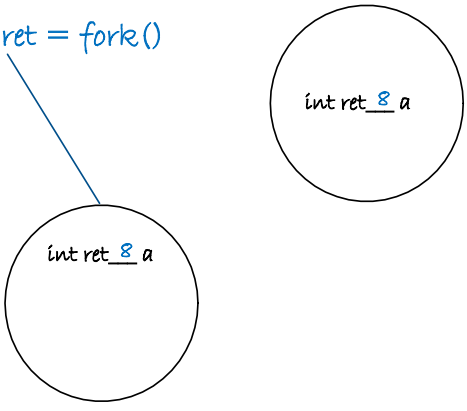
assume fork always succeed :

```
main ()
        {
         int ret, a=5;
         a+=3; pf(a);
         ret=fork();
         if (ret==0)
         {
          a+=10;
          pf(a);
                }
         else
             {
               a*=5;
               pf(a);
                     }
          a++;
          pf(a);
                }
```

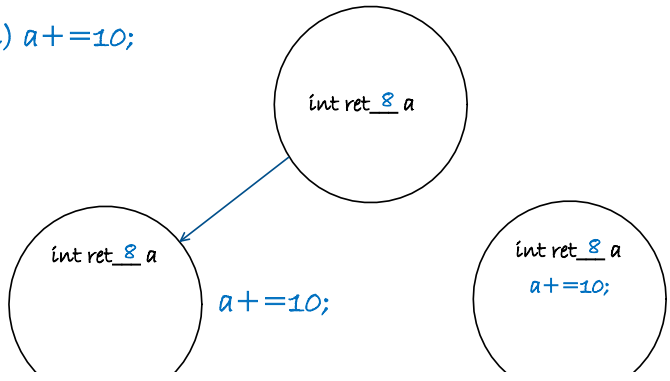
(i)



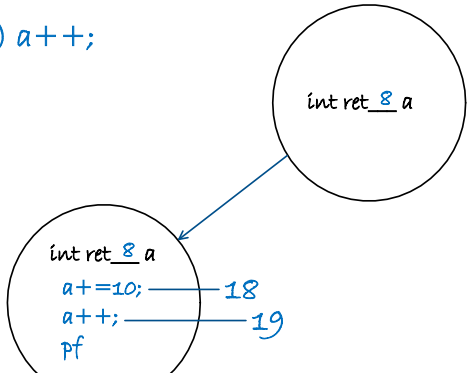
(ii) ret = fork()

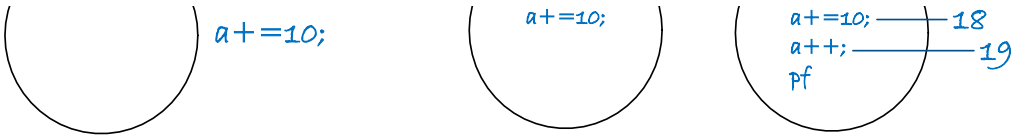


(iii) a+=10;

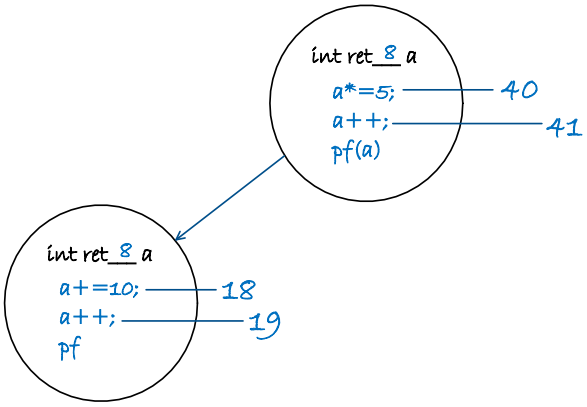


(iv) a++;




a+=10;
a+=10;
a+=10;
18
a++;
19
pf

(v) a*=5;


int ret_8 a
a*=5;
40
a++;
41
pf(a)
int ret_8 a
a+=10;
18
a++;
19
pf

Q.

```
main ()
{
    int i, n;
    for (i = 1; i < = n; + + i)
    if (fork ( ) = = 0)
            print ("*");
}
```

every

```
int i, n;
for (i=1; i<=n; ++i)
    if (fork ()==0);
    printf ("*");
```

outside for loop
(scope till first semi colon)

```
int i,n ;
for {
     (i=1; i<=n; ++i)
     if (fork()==0)
                    }
     printf ("*");
```

print statement is outside for loop

this will be executed by all processes, child as well as parent.

n times calling fork
aur fork ==0 means child processes
every child will print this.

ans : 2^n
every child : 2^{n-1}

process management

topic : process concepts

concept : program vs process

(i) program

program ⟶ process

instructions
(program code)

data
2 types
(a) static
- known/fixed - size
- load time : when you want to execute it. (not compile time)

(b) dynamic
- maybe fixed/variable - size
- run time allocation

as an adt (abstract data type):
(a) definition
(b) structure
(c) operations
(d) attributes

int n, a[n] (defining n, a is array)
scanf ("%d", &n); (run time par n value read karke array declare karenge)

(not possible) (if even possible then dynamic because size not known, allocation is taking place at run time)

dynamic array of size n integers : using pointers and dma.
int n, *p; (define n, pointer because integer array)
scanf ("%d", &n);
p = (int*) malloc (size of(int)xn);

pointer jisko point kar raha vo hai dynamic

dynamic

static static 0 1 2 3 . . . . . n-1

run time par malloc se n bytes allocate honge

point to remember :
you can compile a file but may or may not execute. so why do we have to allocate memory on compile time? so instead the memory allocation happens during load time.

high level language consist of two things :

(a) instructions
(b) data

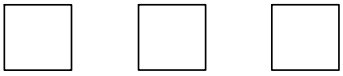
high level language → compiler → .exe (low level program)

```
int a, b, c
a=1;
b=2;
c=a+b;
```

instructions :

```
load a
load b
load r1, a
load r2, b
add r1, r2
store c, r1
```

data:

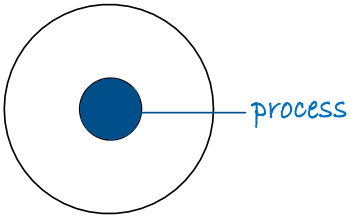


(3 variables)

you cannot add the memory locations directly in processor, you have to load the values in register.

(ii) process/task

(a) program under/in execution : utilising the resources of system
(b) instance of a program : multiple processes (by using fork we create multiple processes in a single program)
(c) unit of a CPU utilisation: threads, process.
(d) schedulable unit : unit which can make decision/selecting the process (4-5 process hai unme se which process to select).
(e) dispatchable unit : unit which can dispatch to CPU/giving control of CPU process.
(f) active entity : utilising the system resources.
(g) locus of control : it makes the whole operating system alive. control point that makes the whole OS.



(h) animated spirit :
dead body : no spirit
alive : spirit

program on disk : dead body
program in main memory : spirit

we cannot see spirit, same way we also cannot see the process running on cpu.
- if there is no user process, there will be no activity for os.

point to remember :

- how can we create multiple process from a single program?
by using fork system call, fork se multiple process create kar sakte.

concept : developer view of process

as an adt (abstract data type)

it will always be associated with four common parameters :

(a) definition
(b) structure
(c) operations
(d) attributes

concept : process structure in memory (block diagram)

every process in memory attached with stack :

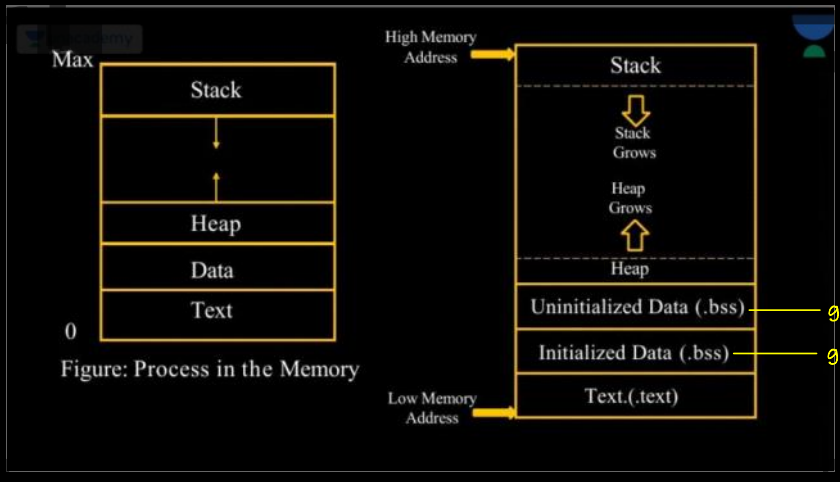
formals parameters, local variables
- return address (hamesha rakhega)

activation records of function calls
- user defined
- pre-defined
- not system calls, they are on kernel stack (run time stack)

dma (dynamic memory allocation)

static and global data : initialised and uninitialised (those variables that are type case as static)

instructions store karega programs ke

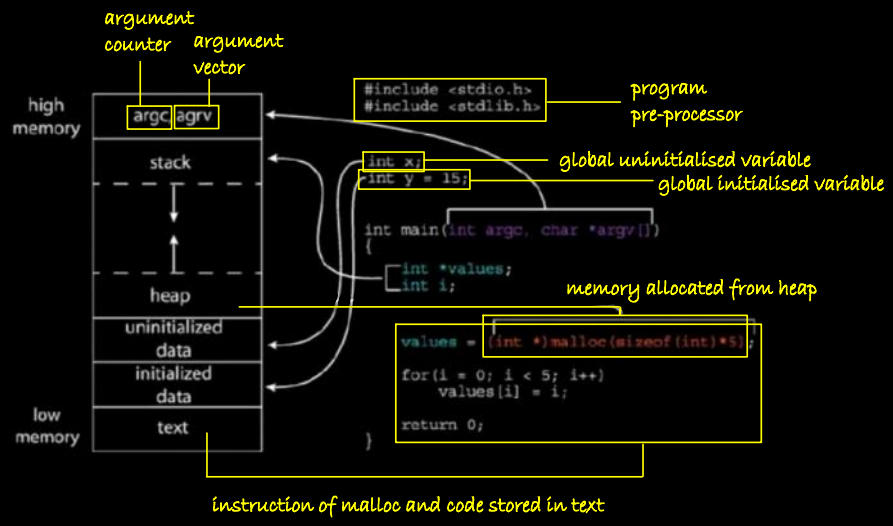


global (not initialised)

global (initialised)

concept : memory layout of c program

agar aap main function se input program ko dena chah rahe toh
that is stored in argv and their count is specificed in argc



- An operating system executes a variety of programs that run as a process.
- Process – a program in execution; process execution must progress in sequential fashion. No parallel execution of instructions of a single process
- Multiple parts
  - The program code, also called text section
  - Current activity including program counter, processor registers
  - Stack containing temporary data
    - Function parameters, return addresses, local variables
  - Data section containing global variables (uninitialised + initialised)
  - Heap containing memory dynamically allocated during run time
  - Program is passive entity stored on disk (executable file); process is active
    - Program becomes process when an executable file is loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- One program can be several processes (fork)
  - Consider multiple users executing the same program

concept : process operations (done by os)

(i) create () : resource allocation + pcb create (jitne bhi resource process ko chaiye)
(ii) schedule() : selecting the process.
(iii) dispatch () : giving control of cpu to process.
(iv) run() : execution of instructions on cpu.
(v) block() : performing i/o operation/system call (execution of system call).
(vi) suspend() : push out the process from memory to disk.
(vii) resume() : bring the process back to memory.
(viii) terminate() : resource deallocation.

concept : process attributes (characterisitics)

process as an entity in operating system is associated with several attributes.

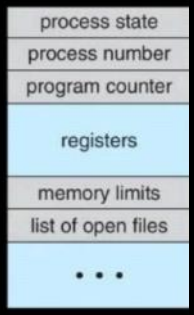
(i) identification : (process id (pid), parent process id (ppid), group id (gid) kaunse group ko process belong karta hai.
(ii) cpu related : type, state, program counter, general registers.
(iii) memory related : size, limits....
(iv) file related : list of open files.
(v) device related : list of open devices.
(vi) accounting related : resources.
(vii) resume() : bring the process back to memory.
(viii) terminate() : resource deallocation.

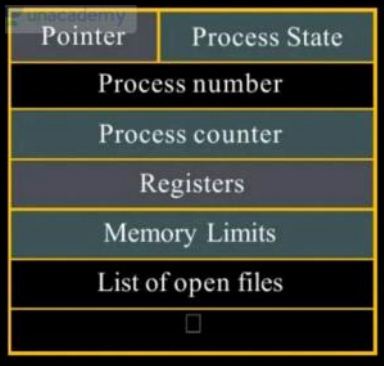
concept : process control block (pcb)

- all the attributes are stored/organised in a table known as process control block.
- representative of process.

Information associated with each process(also called task control block)

- Process state – running, waiting, etc.
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



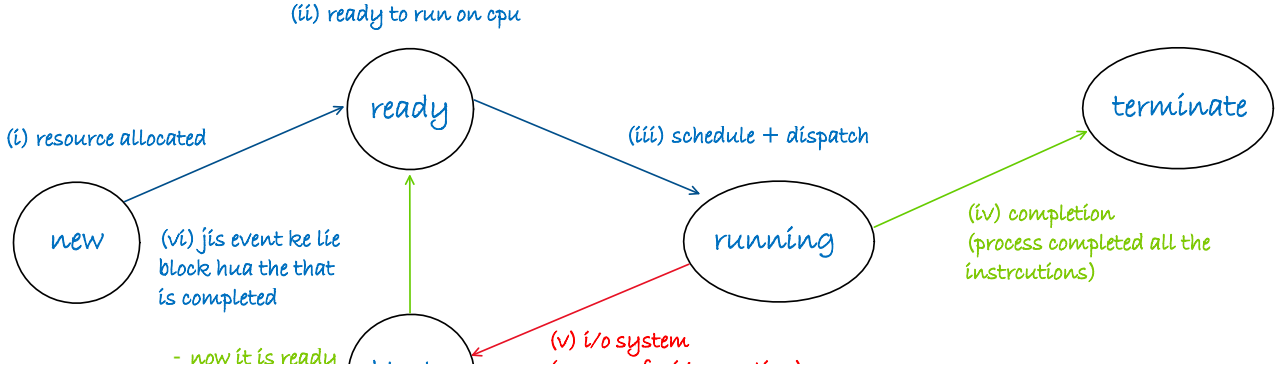


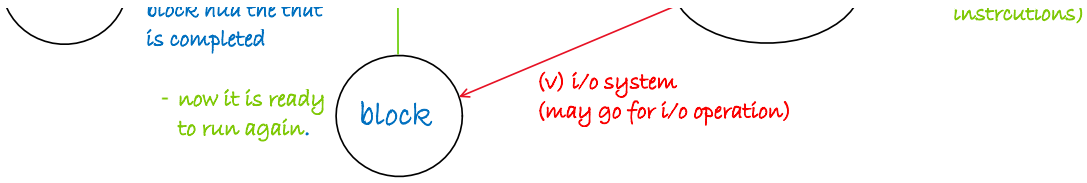
- pcb is in memory
- content of pcb is known as process context/process environment

concept : process state and state transition diagram

states :

- new
- ready
- running
- block
- terminate



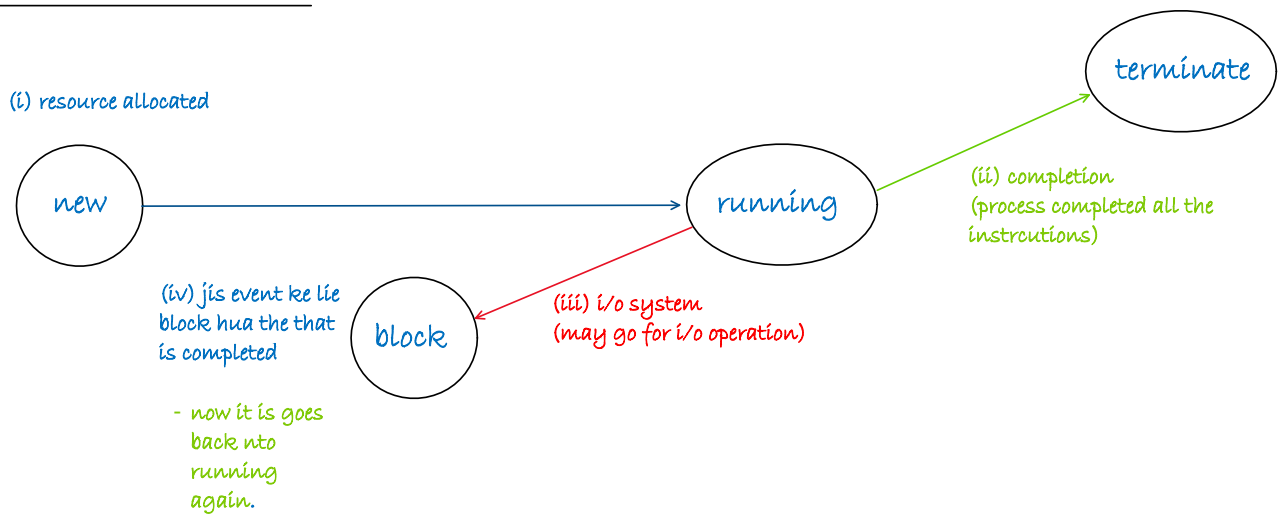


point to remember :

because of ready state process is classified as multiprogramming.

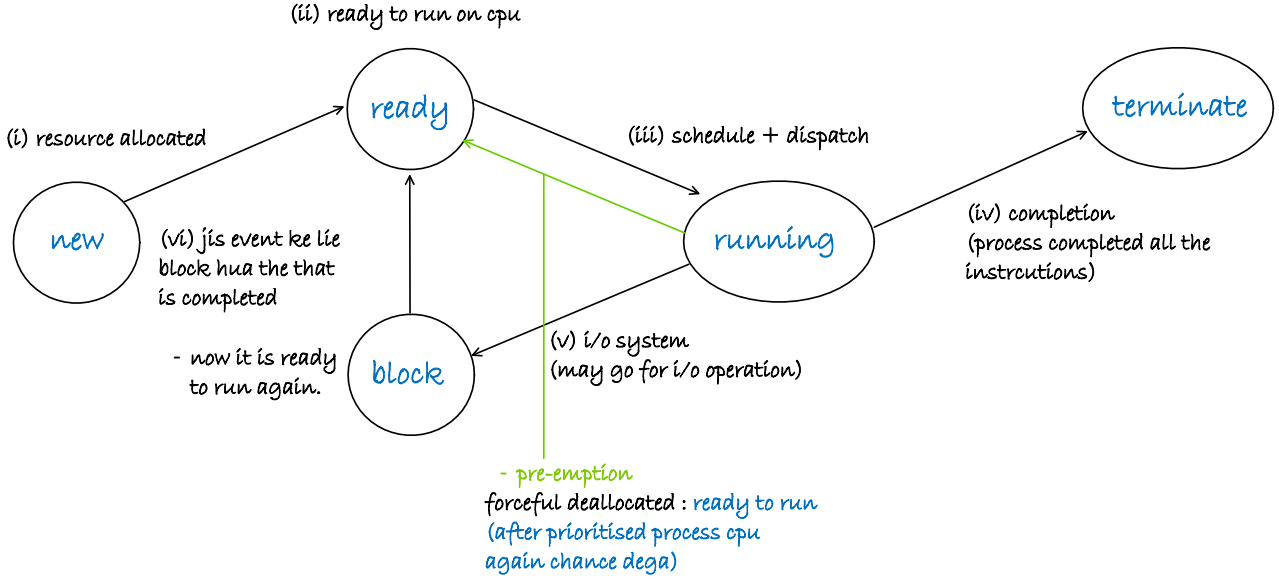
in upr you will not have ready state.

in uniprogrammed os :



back to multiprogramming os :

- if i want to add pre-emptive (forceful de-allocation) in this diagram :



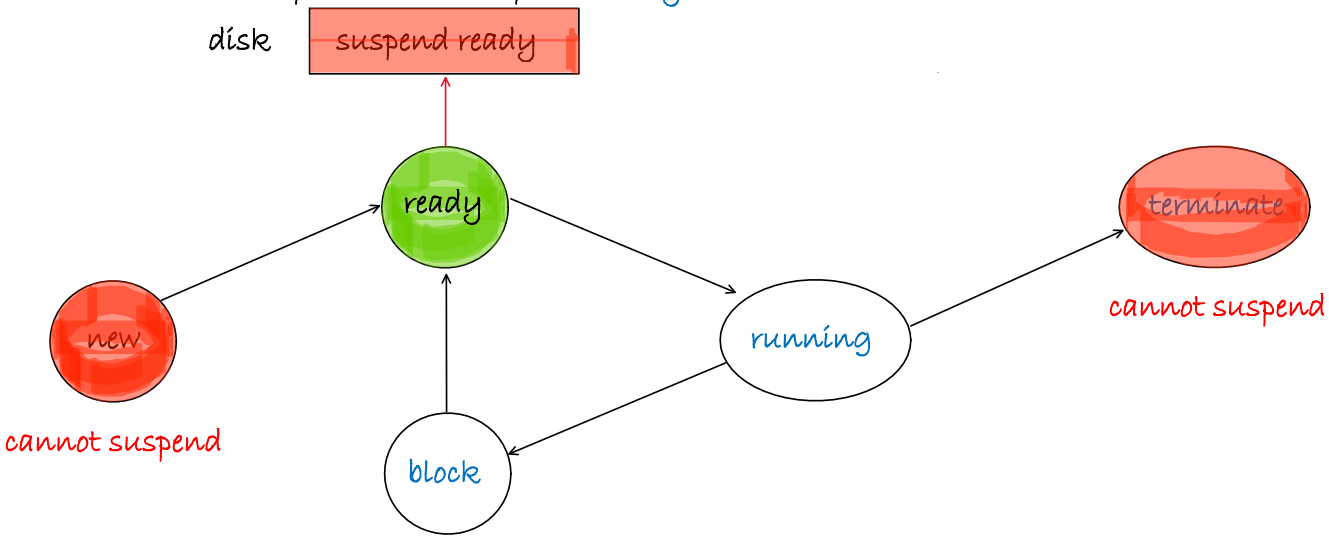
- time
- priority

running to ready
- memory to memory

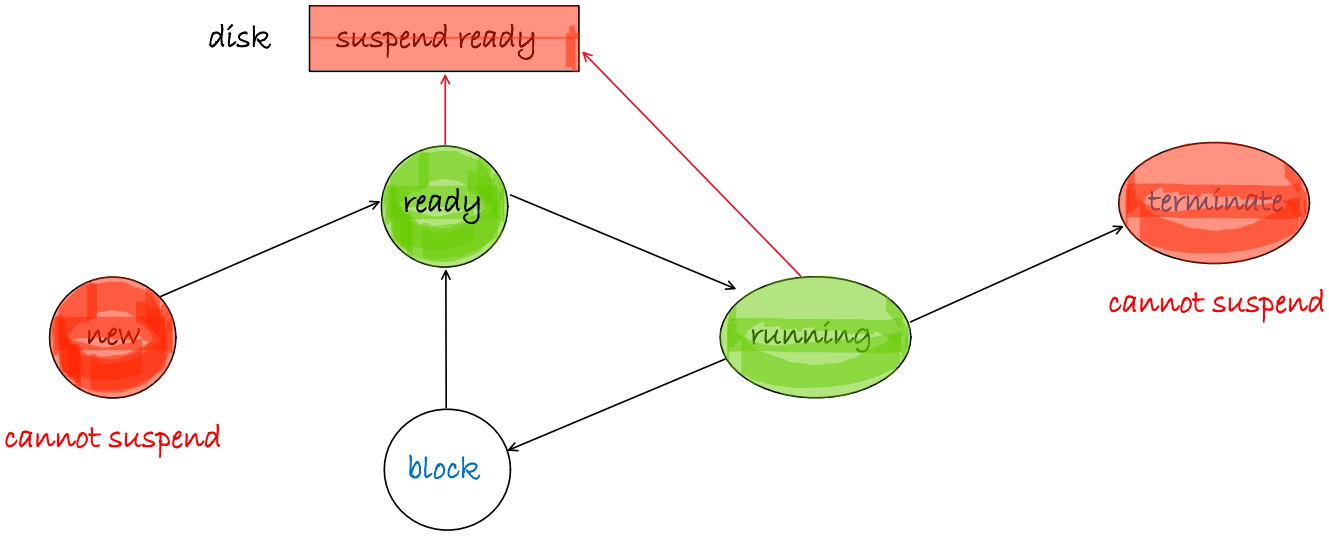
if ek cpu ho toh i can have only one running state but many ready and block processes.

during the course of execution, many times the os may have to push the process from main memory to disk because of no resources, performance improvement, etc. so that operation is called "suspend".

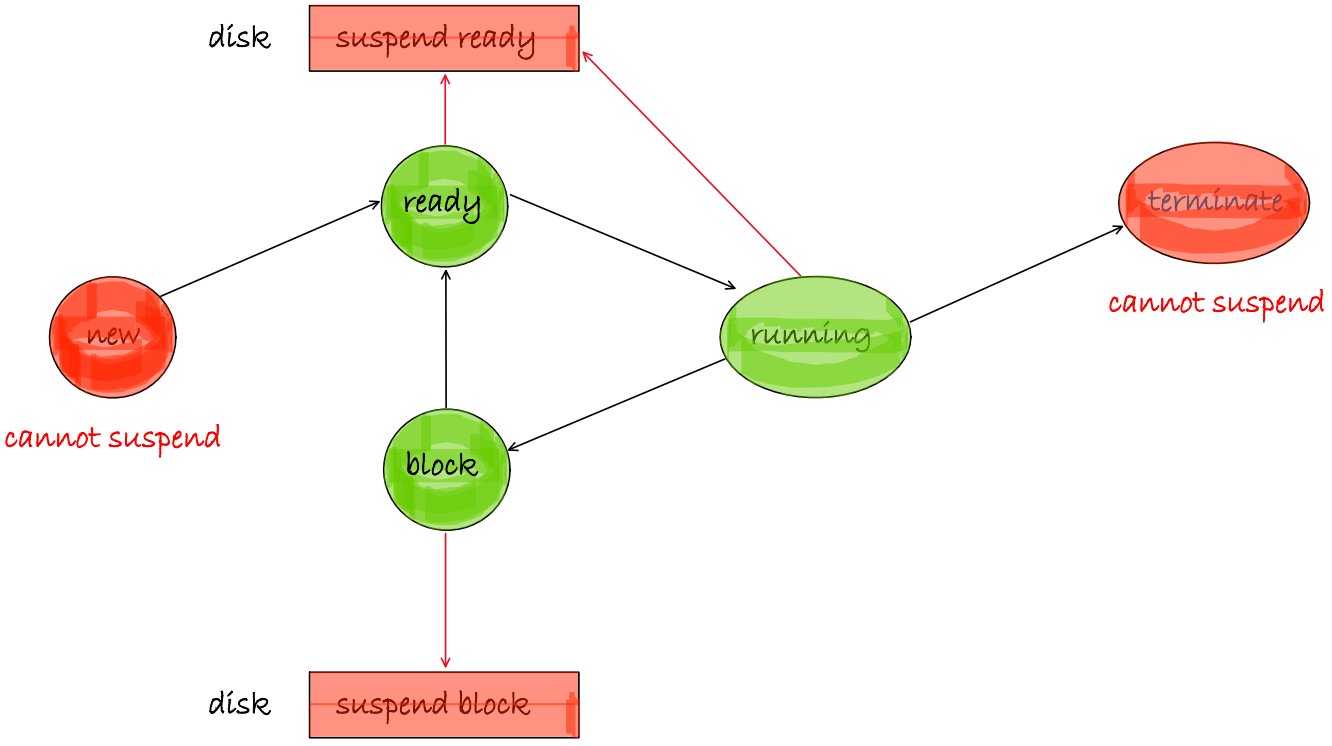
- most desirable process to be suspend : ready



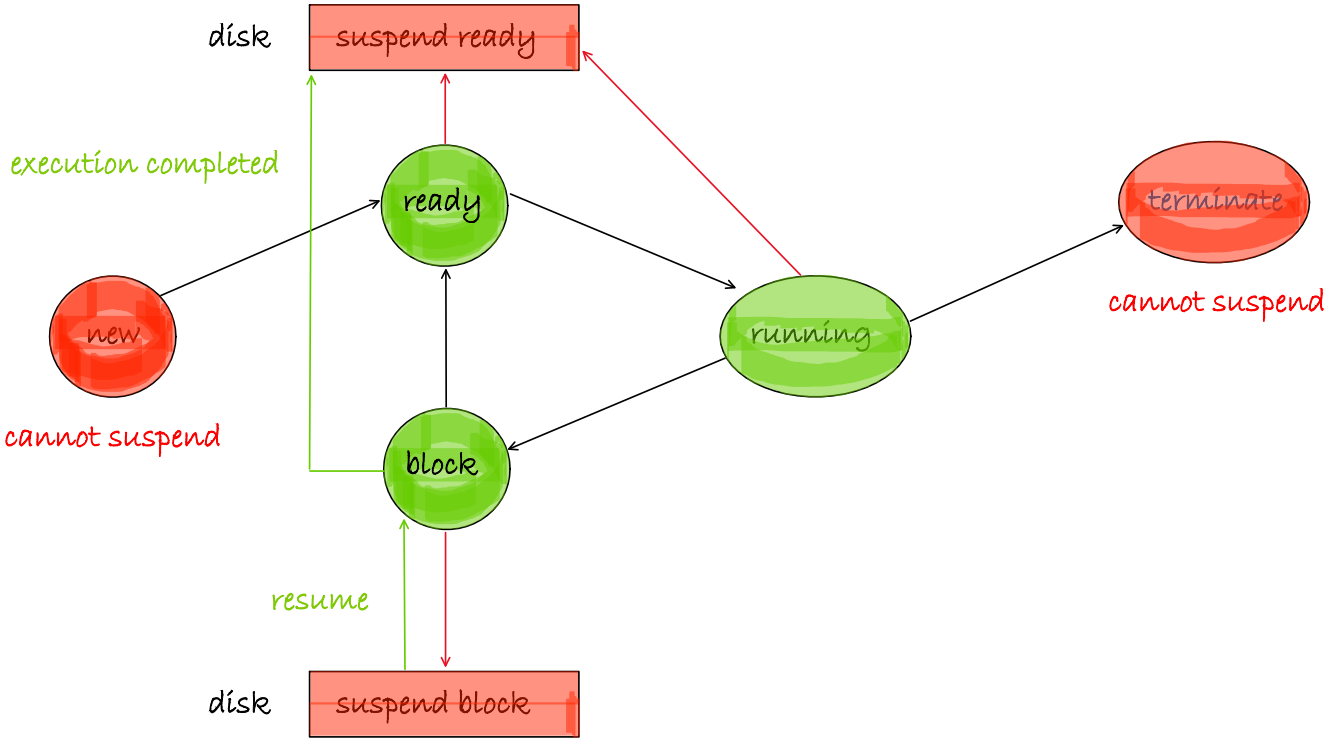
if the process get suspended from running : memory to disk



if the process get suspended from block state : suspend block



- however, if the suspended block program gets the chance to resume then after the execution where will program go : suspend ready



in suspend block state, if somehow the process completed its i/o (satisifes the event), then the process makes a transition to suspend ready state.

concept : scheduling queues and queueing diagram

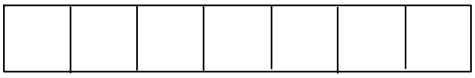
(a) on disk

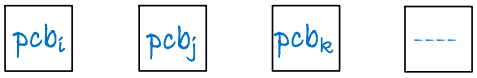
(i) suspend : contains all the suspended program

(ii) job (i/p) queue : programs to be loaded in memory

(b) in-memory

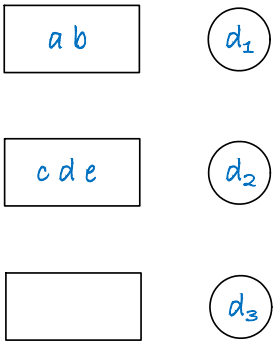
(i) ready : (ready state) ready to run programs.

ready queue : 

ready queue : 

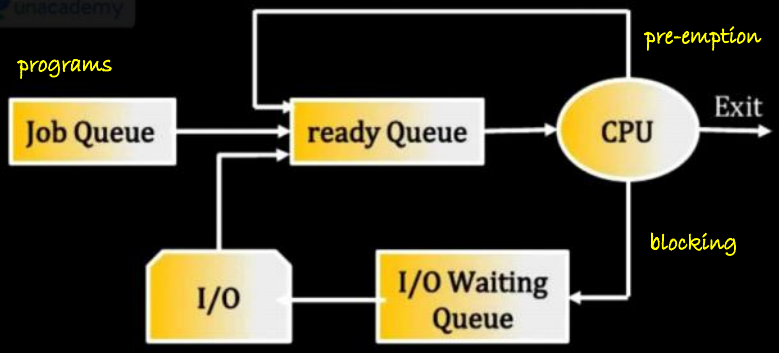
linked lists of pcb$_s$ that are ready to run

(ii) block queue: (block state)performing i/o operation/system call .

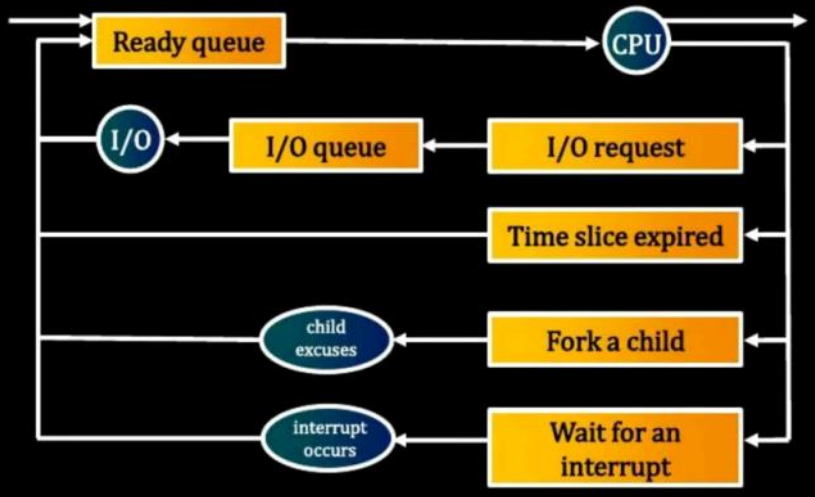


point to remember

- there is no queue for running because on cpu only one process can run



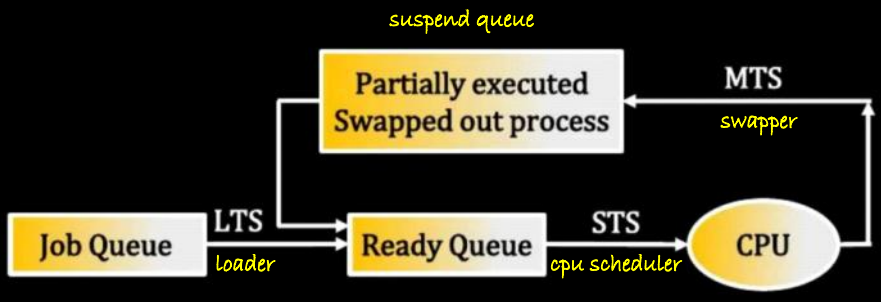
state-queueing diagram



if operating system is an abstract machine then queues are its components

- queue are implemented through linked lists.

how one process moves from one to another queue



concept : context switching

context switching is an activity that is carried out during a process switch on CPU; that involves saving

PCB and loading PCB onto CPU.

concept : scheduler and dispatcher

(i) scheduler : scheduler are those programs of process manager that makes decision on queue.

(a) job queue : long term scheduler (LTS)

job queue : which program should get into memory

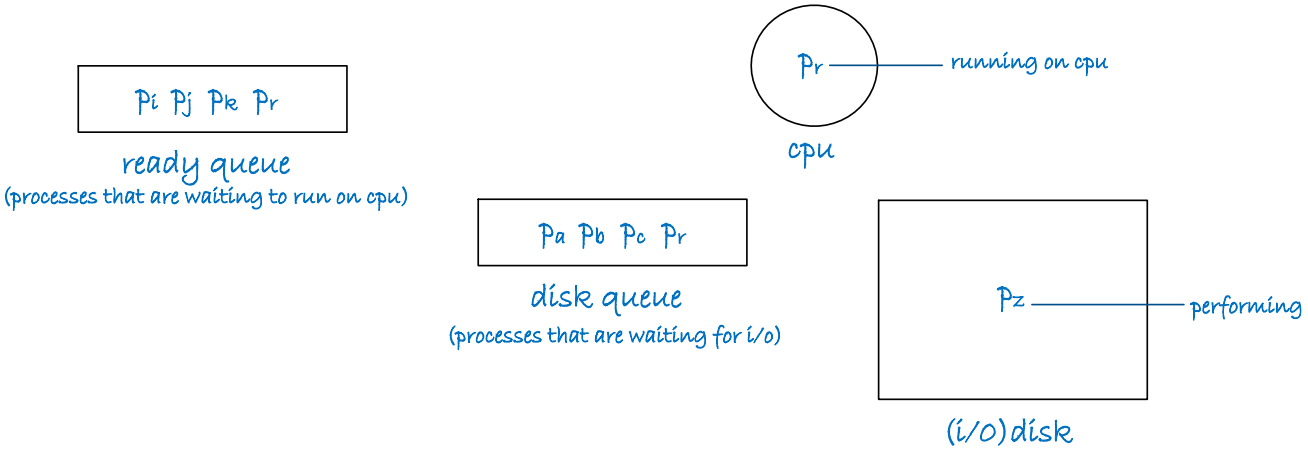
(b) ready queue : short term scheduler (STS)

ready queue : which ready process runs on cpu.

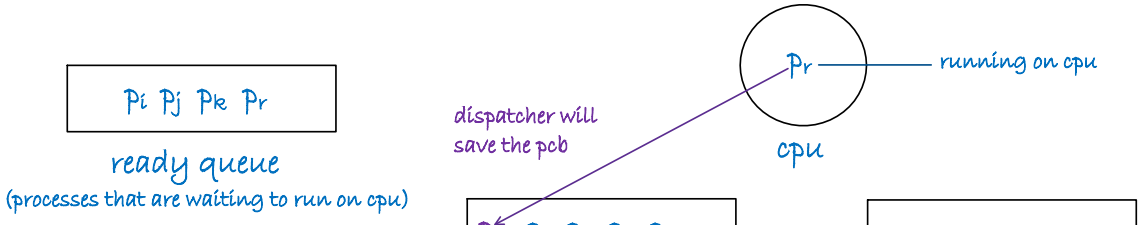
(c) suspend queue : medium term scheduler (MTS)

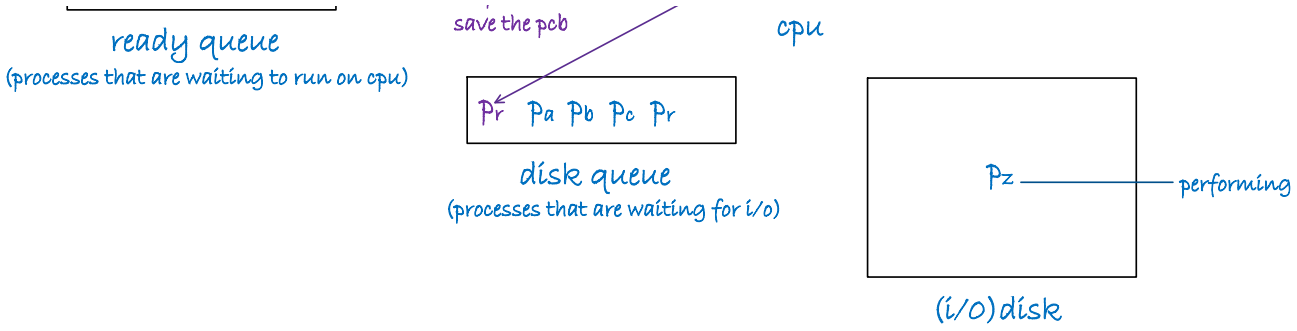
process suspension and resumption

(ii) dispatcher : module of OS that carries out the activity of context switching.

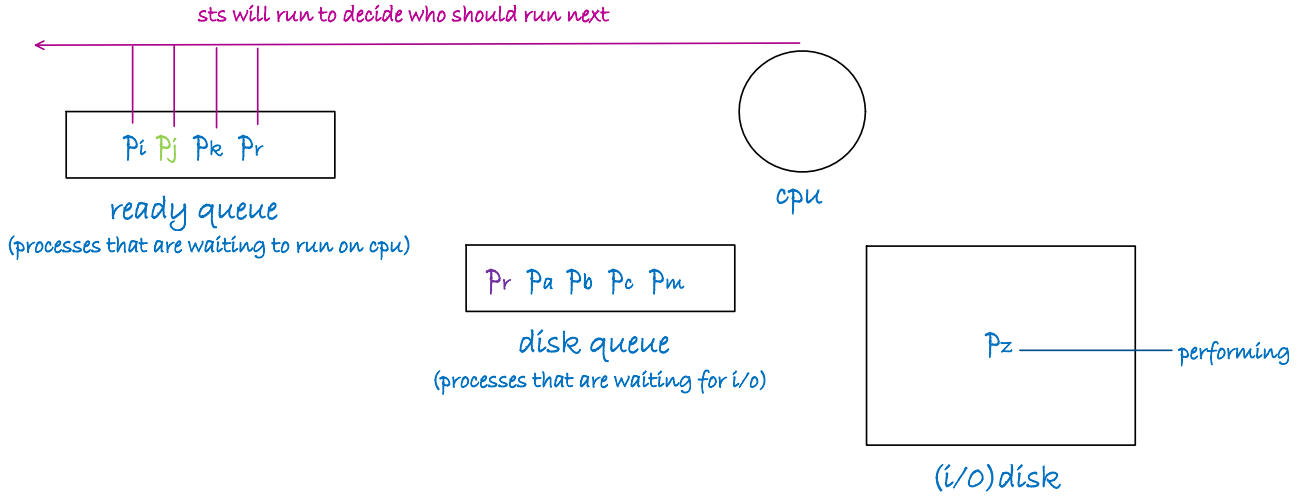


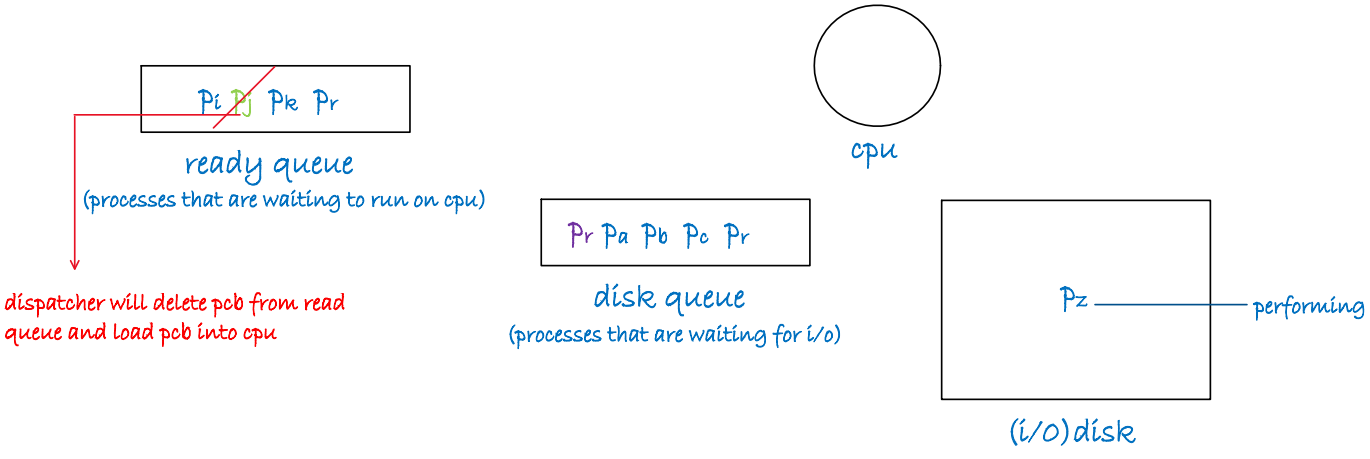
(ii) p_r wants to got for i/o

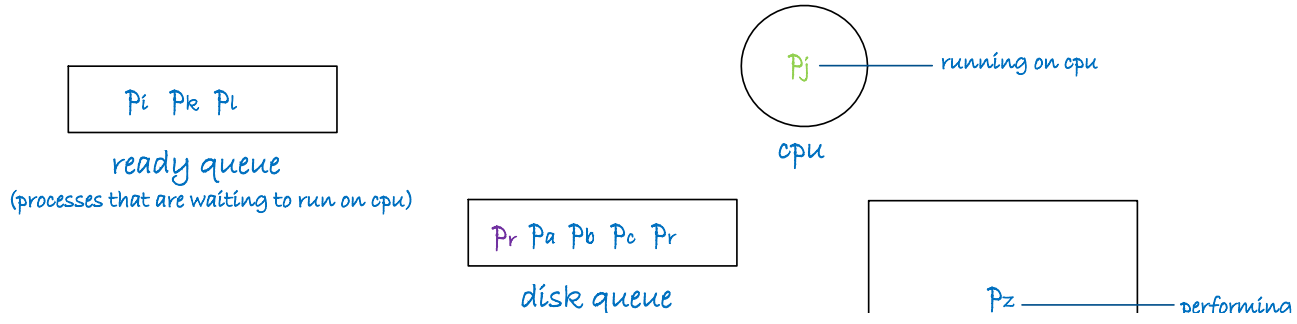


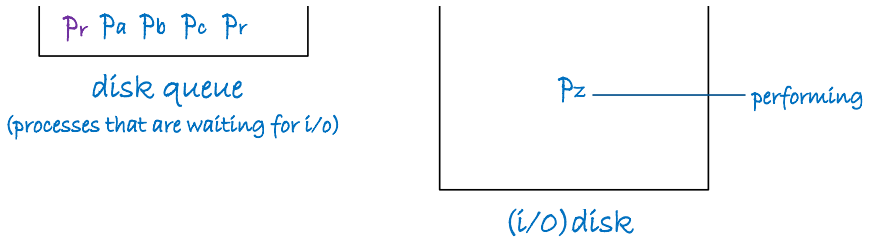


(iii) P_j is waiting to run on cpu









it is desriable to have less context swtich jitna jada hoga utna jada overheat and utna hi impact hoga throughput par

question :

- who control the degree of multiprogramming?
ans. long term scheduler

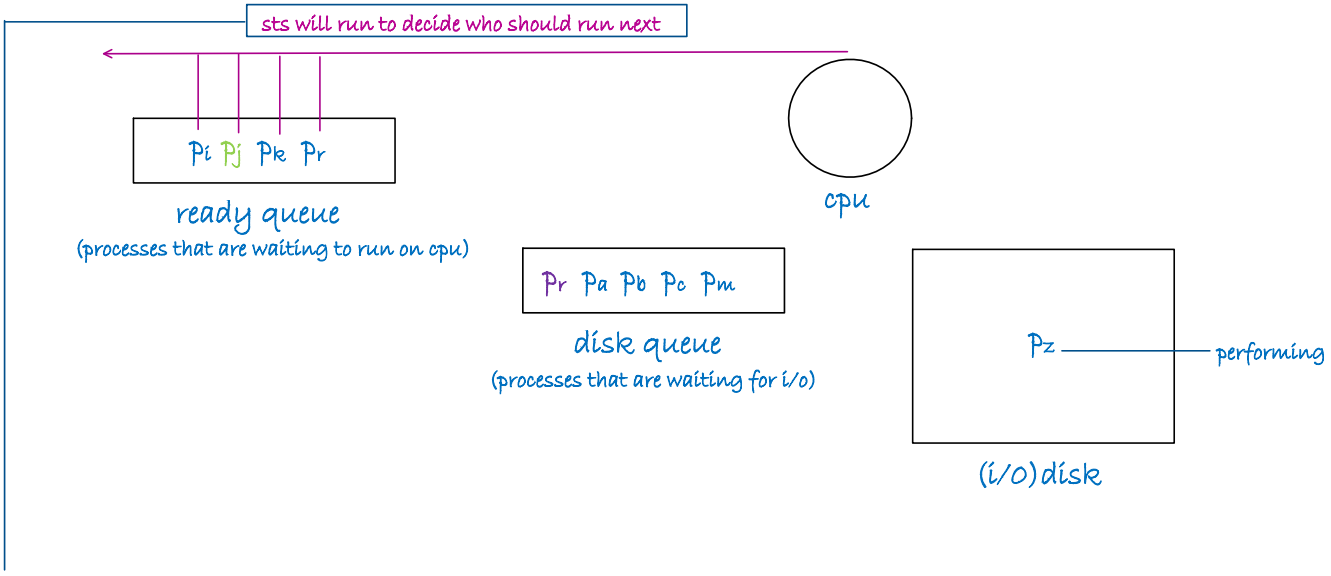
(i) CPU scheduling/process scheduling

Friday, October 11, 2024 3:30 AM

topic : CPU scheduling/process scheduling

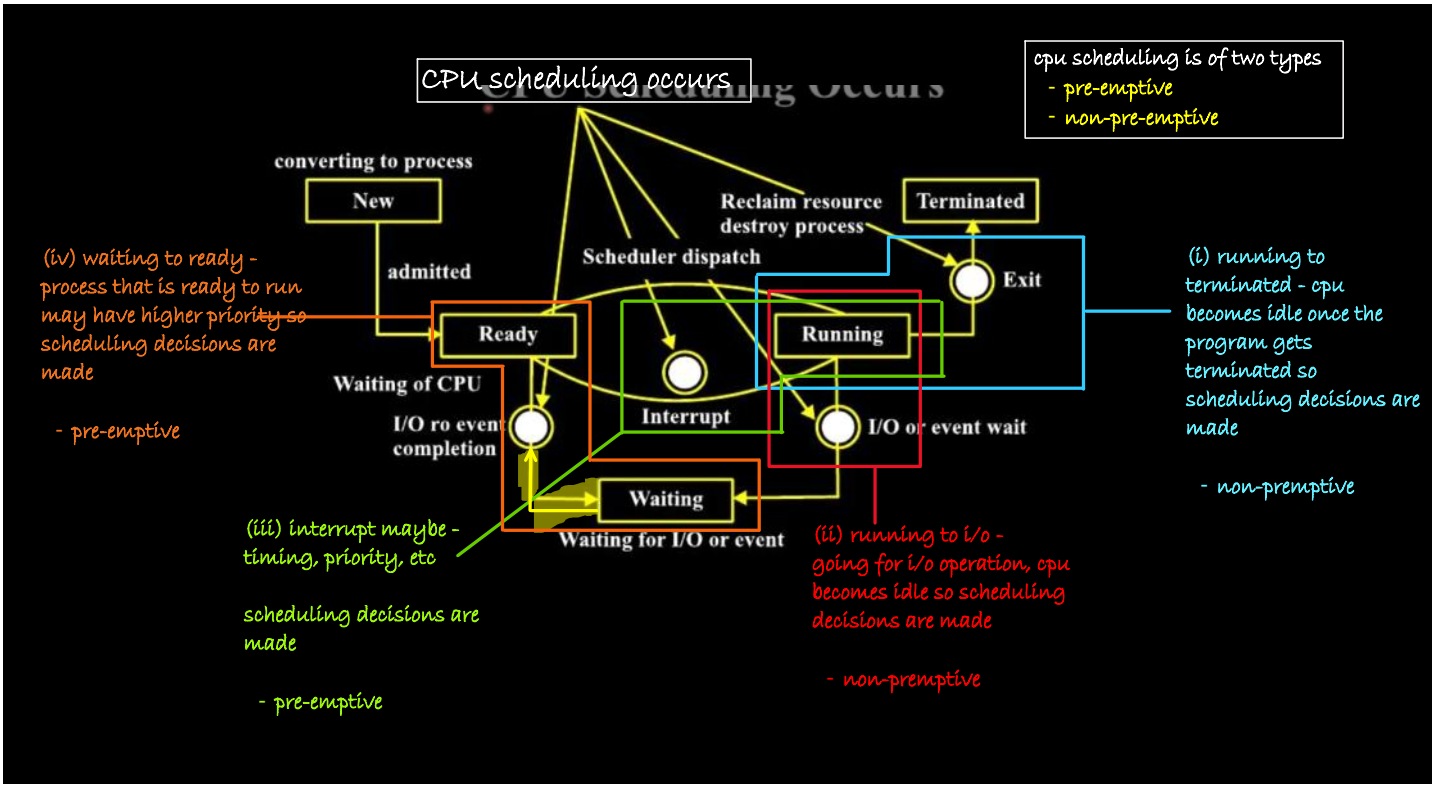
concept : design of a short term scheduler

short term scheduler is resposible to choose which ready process should run on cpu next.



how sts will actually decide?

there are some criteria and selection parameters that is known as cpu scheduling algorithm.



concept : scheduling criteria

criteria are the parameters that are used to compare different scheduling algorithms to decide which scheduling technique is best, optimal and efficient.

- like a problem may have different solutions but we compare solutions to find which is best, efficient and optimal,

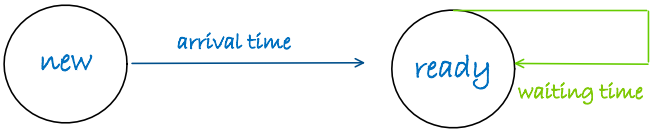
(i) maximize :
- cpu utilisation
- throughput (efficiency) ——— maximum throughput scheduling must be done in a way, it's objective is to keep processor busy all the time

(ii) minimize
- turnaround time
- waiting time
- response time

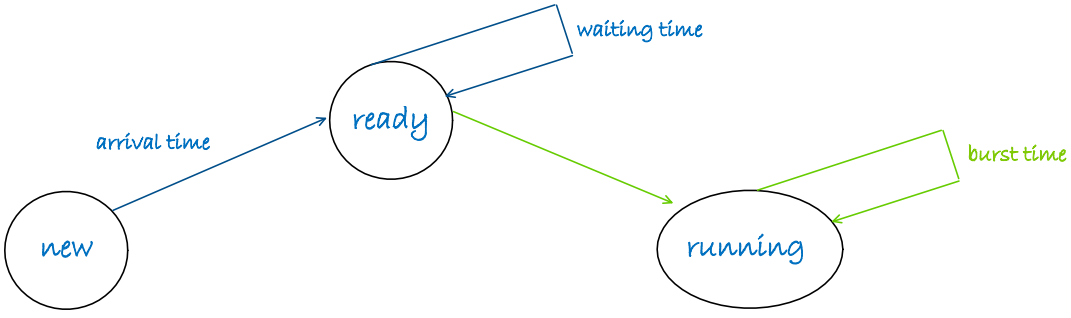
concept - process times : what are the various times a process is associated with during its lifetime.

(i) arrival time (A.T.) : sabse pehle time jo process associated rahega (from new to ready state).

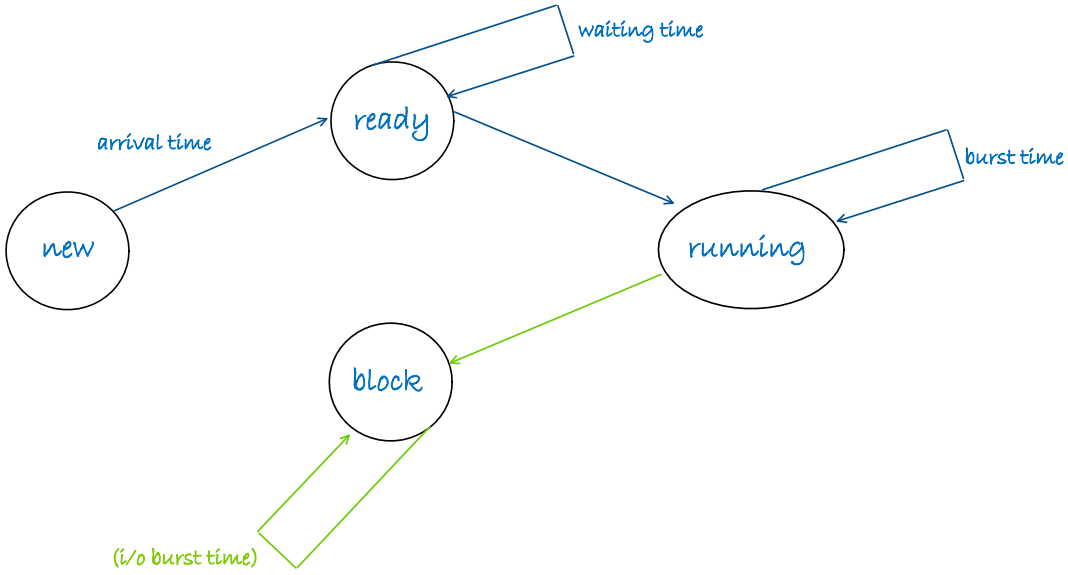
(ii) waiting time (W.T.) : process will wait in the ready queue for sometime to run on cpu. time spent by process for waiting to run on cpu.



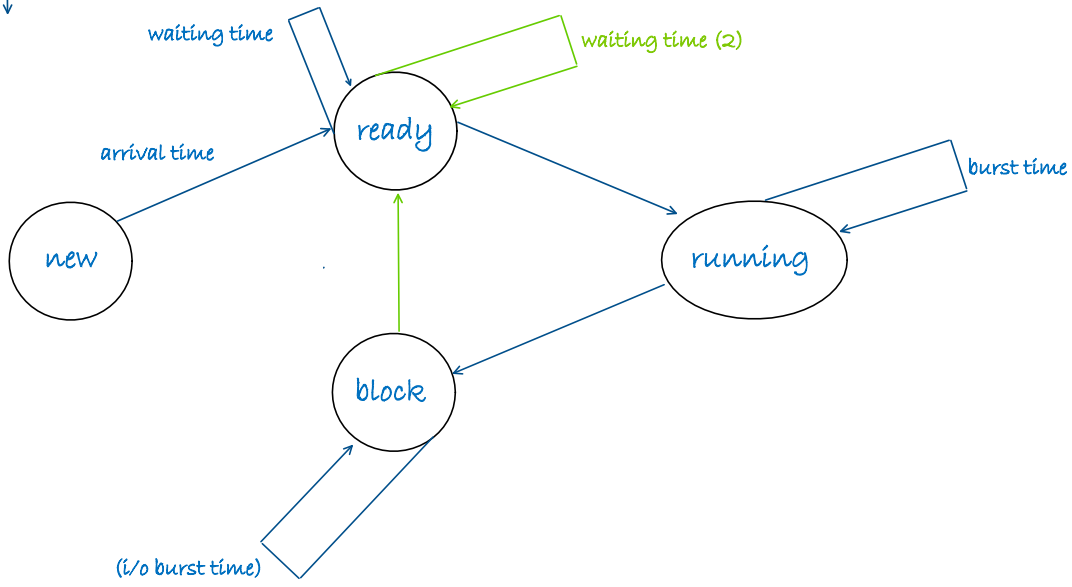
(iii) burst time (B.T.) : for whatever time it runs on cpu.



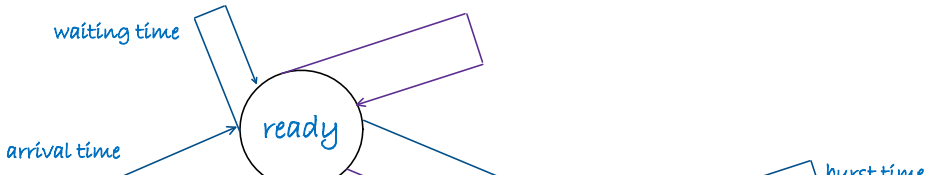
(iii) i/o burst time (I.O.B.T.) : the time spent by process to run on i/o.

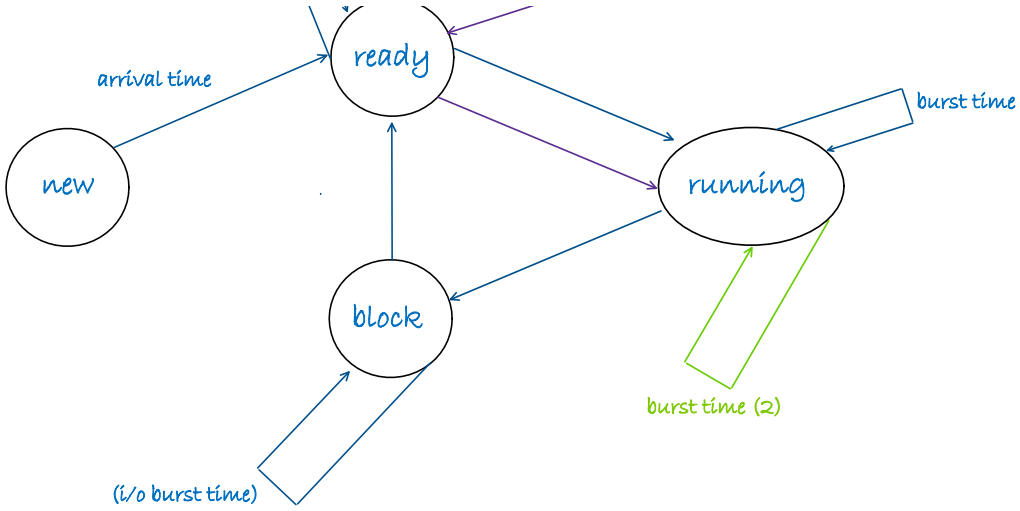


i/o completed and now process goes to ready queue again and waiting : waiting time (2)

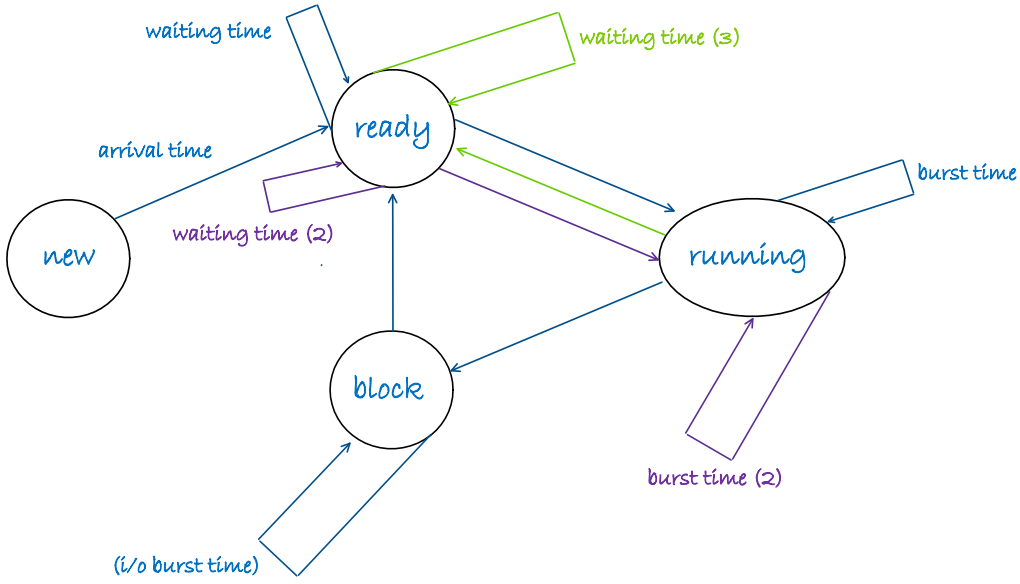


again, it got the chance to run on cpu, for whatever time it runs on cpu again : burst time (2)

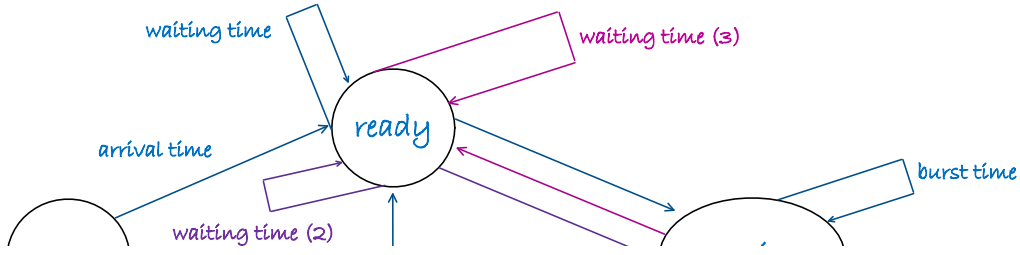


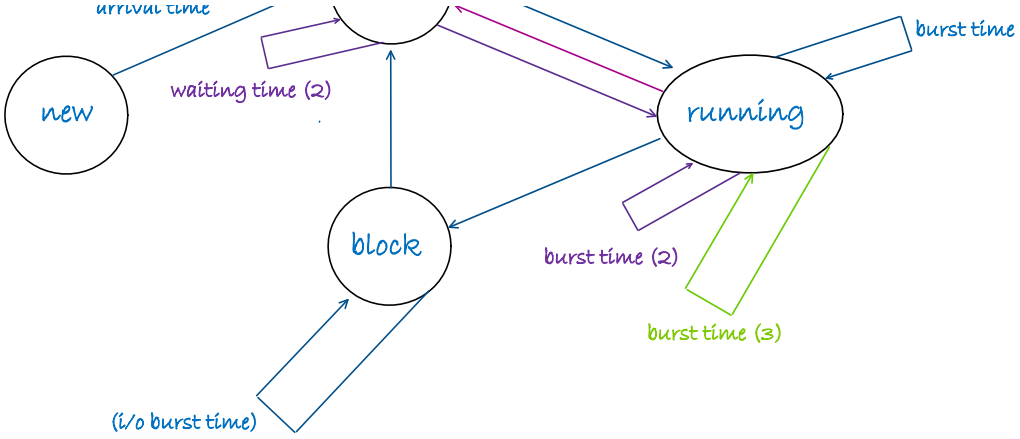


process may get pre-empt, again goes to ready state :
waiting time (3)

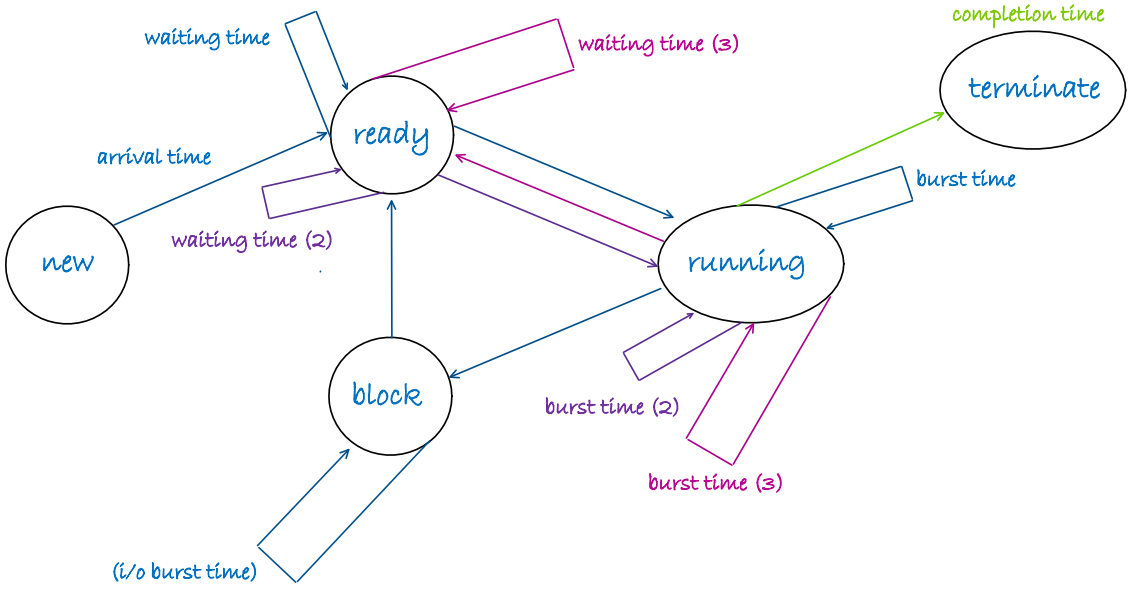


again, it got the chance to run on cpu, for whatever
time it runs on cpu again : burst time (3)





(iv) competion time (C.T.) : process may get completed.

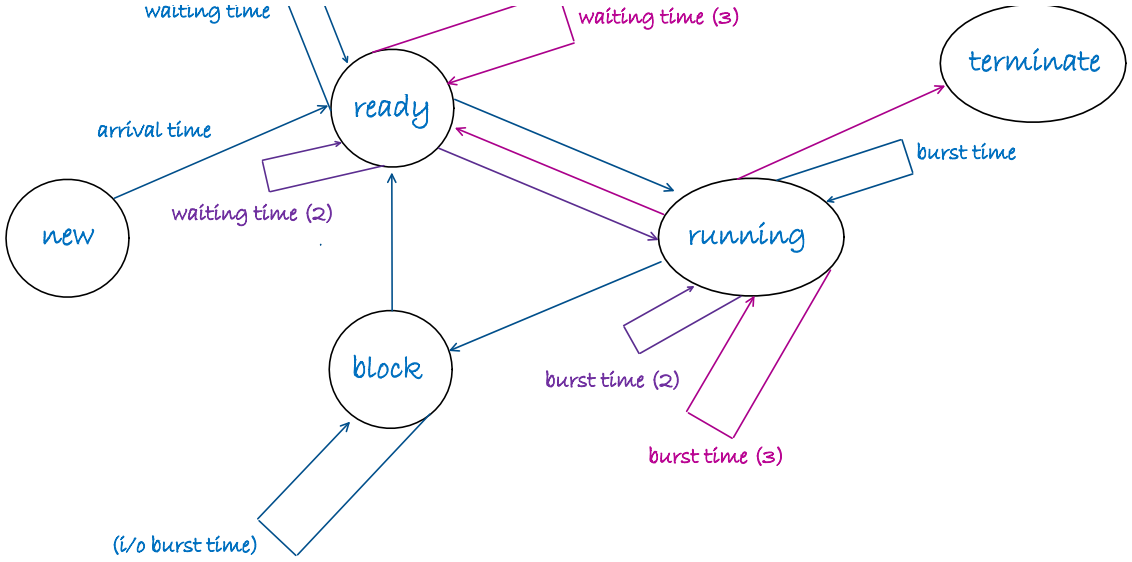


TAT (turn around time)

pri (lifecycle of process time)							
	$W.T_1$ (waiting time)	$B.T_1$ (burst time)	I.O.B.T (i/o burst time)	$W.T_2$ (waiting time)	$B.T_2$ (burst time)	$W.T_3$ (waiting time)	$B.T_3$ (burst time)
	R.Q. (ready queue)	cpu	I.O. (i/o device)	R.Q. (ready queue)	cpu	R.Q. (ready queue)	cpu

A.T. (arrival time)

C.T. (completion time)



turnaround time (TAT) : arrival se completion tak jo time spend karega that is turnaround time.
- formula : C.T. - A.T. (completion time - arrival time)

waiting time : waiting time of process spent in ready queue.
- formula : TAT - (BT + IOBT) [turn around time - (burst time + i/o burst time)
- if in case iobt : 0
then W.T.= TAT - BT (turn around time - burst time)
- TAT=W.T.+BT (waiting time + burst time)

concept : response time

(i) response time of process : time of arrival to the time when it runs on the cpu for first time.

(no interaction with cpu, no computation)

formula :
response time = first waiting time ($W.T_1$)

(ii) response time of request of a process : a process may have different requests, during its lifetime a process maybe associated with different several requests.

- time of admitting the request to the time which runs on the cpu for computation and generate the result : response time of process request.

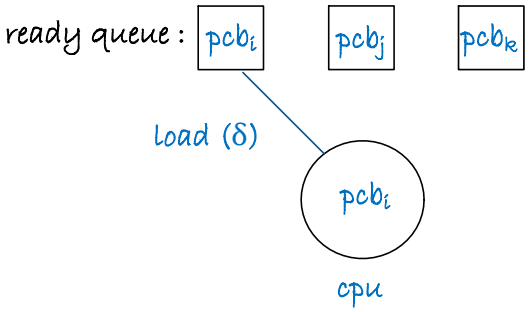
- computation time lagega, compute and generate the result fir jo first result aayega is response time of request of process.

concept : context switch overhead (dispatcher)

cpu scheduling overhead (δ)

time needed for the process to schedule and dispatch the process.

load (δ): (time needed for sts to decide which process to run and the dispatcher to delete the pcb from ready q and load it into the cpu.



concept : schedule length / length of schedule

total time taken to complete all processes as per schedule.

formula :

$l = \max(c_i) - \min(a_i)$ ——— arrival time of first process

completion time of last process

- how many such schedules are possible : n! (non-pre-emptive)
- if there are infinite schedules : pre-emptive

number of schedules

P_1	P_2	P_3	<1,2,3>
P_2	P_3	P_1	<2,3,1>
P_3	P_2	P_1	<3,2,1>

(ii) problem solving - fcfs

Friday, October 11, 2024 3:28 AM

framework for problem solving

- n : number of process $(p_1 \ldots p_n)$
- A_i : arrival time of p_i
- X_i : burst time of p_i
- Y_i : i/o burst time of p_i
- C_i : competion time of p_i

(i) turn around time

$$TAT(p_i) = C_i - A_i$$

$$\text{Avg. } TAT(p_i) = 1/n \sum_{i=1}^{n} (C_i - A_i)$$

(ii) waiting time

$$W.T(p_i) = (C_i - A_i) - (X_i + Y_i)$$

$$\text{Avg. } W.T = 1/n \sum_{i=1}^{n} (C_i - A_i) - (X_i + Y_i)$$

(iii) schedule length / length of the schedule

$$l = \max(c_i) - \min(a_i)$$

concept - first come first served (fcfs)

(i) tie/conflict : when multiple processes run at same time.
(lower p_{id} is the metric to follow generally)

questions :

(i)

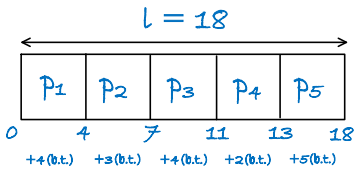
process no.	a.t.	b.t.	c.t.	tat	w.t.
1	0	4			
2	0	3			
3	0	4			
4	0	2			
5	0	5			

assume :
- iobt = 0 (cpu bound process)
- initally $\delta = 0$

mode of operation : multiprogramming

(i) gantt chart

multiple processes arrival time is same hence conflict/tie : start from lower p_{id}



process no.	a.t.	b.t.	c.t.	tat	w.t.
1	0	4	4		
2	0	3	7		
3	0	4	11		
4	0	2	13		
5	0	5	18		

(ii) turnaround time and average turnaround time

turnaround time = $(c_i - a_i)$

4-0=4
7-0=7
11-0=11
13-0=13
18-0=18

average turnaround time = $\text{Avg. TAT } (p_i) = 1/n \sum_{i=1}^{n} (C_i - A_i)$

$= (4+7+11+13+18)/5 = 10.6$

process no.	a.t.	b.t.	c.t.	tat	w.t.
1	0	4	4	4	
2	0	3	7	7	
3	0	4	11	11	
4	0	2	13	13	
5	0	5	18	18	

(iii) waiting time and average waiting time

waiting time = $(c_i - a_i) - (x_i + y_i)$

4-4=0
7-3=4
11-4=7
13-2=11
18-5=13

average waiting time = $\text{Avg. W.T} = 1/n \sum_{i=1}^{n} (C_i - A_i) - (X_i + Y_i)$

$(0+4+7+11+13)/5 = 7$

process no.	a.t.	b.t.	c.t.	tat	w.t.
1	0	4	4	4	0
2	0	3	7	7	4
3	0	4	11	11	7
4	0	2	13	13	11
5	0	5	18	18	13

(iv) response time

response time = first waiting time ($W.T_1$)

response time = 7

point to remember -
for non-pre-emptive scheduling with i/o burst time are 0 then average waiting time = response time.

(v) length of the schedule

- length of the schedule = $\max(c_i) - \min(a_i)$

length of the schedule = 18 - 0 = 18

(v) throughput

$\eta = n/l$

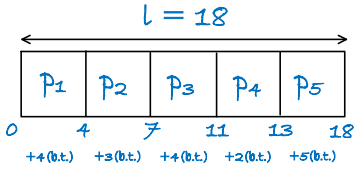
$\eta = 5/18 = 0.2778$

n : number of processes
l : length of the schedule

mode of operation : uniprogramming

(i) gantt chart

multiple processes arrival time is same hence conflict/tie : start from lower p_{id}



their arrival time will nullify because in uniprogramming another process can only run on cpu when already running process is finished so just for reference the arrival time timeline may look like this.

process no.	a.t.	b.t.	c.t.	tat	w.t.
1	0	4	4		
2	4	3	7		
3	7	4	11		
4	11	2	13		
5	13	5	18		

- there is no ready queue, they are in job queue.

(ii) turnaround time and average turnaround time

turnaround time = $(c_i - a_i)/(b.t.)$

4-0=4
7-4=3
11-7=4
13-11=2
18-13=5

process no.	a.t.	b.t.	c.t.	tat	w.t.
1	0	4	4	4	
2	4	3	7	3	
3	7	4	11	4	
4	11	2	13	2	
5	13	5	18	5	

(iii) waiting time and average waiting time

waiting time = 0

their waiting time is 0 because process only arrives and runs on cpu once the execution of already running process is completed.

process no.	a.t.	b.t.	c.t.	tat	w.t.
1	0	4	4	4	0
2	0	3	7	7	0
3	0	4	11	11	0
4	0	2	13	13	0
5	0	5	18	18	0

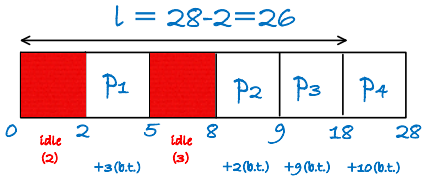
concept - first come first served with idleness

process no.	a.t.	b.t.	c.t.	tat	w.t.
1	2	3			
2	8	1			
3	9	9			
4	15	10			

assume :
- iobt = 0 (cpu bound process)
- initally δ=0

mode of operation : multiprogramming

(i) gantt chart



process no.	a.t.	b.t.	c.t.	tat	w.t.
1	2	3	5		
2	8	1	9		
3	9	9	18		
4	15	10	28		

(ii) turnaround time and average turnaround time

turnaround time = $(c_i - a_i)$

5-2=3
9-8=1
18-9=9
28-15=13

average turnaround time = Avg. TAT $(p_i) = 1/n \sum_{i=1}^{n} (C_i - A_i)$

$= (3+1+9+13)/4 = 6.5$

process no.	a.t.	b.t.	c.t.	tat	w.t.
1	2	3	5	3	
2	8	1	9	1	
3	9	9	18	9	
4	15	10	28	13	

(iii) waiting time and average waiting time

waiting time = $(c_i - a_i) - (x_i + y_i)$ average waiting time = Avg. WT $= 1/n \sum^{n} (C_i - A_i) - (X_i + Y_i)$

(iii) waiting time and average waiting time

waiting time = $(c_i - a_i) - (x_i + y_i)$

$3-3=0$

$1-1=0$

$9-9=0$

$13-10=3$

average waiting time = $Avg.\ W.T = 1/n \sum_{i=1}^{n} (C_i - A_i) - (X_i + Y_i)$

$(0+0+0+3)/4 = 0.75$

process no.	a.t.	b.t.	c.t.	tat	w.t.
1	2	3	5	3	0
2	8	1	9	1	0
3	9	9	18	9	0
4	15	10	28	13	3

(iv) response time

response time = first waiting time ($W.T_1$)

response time = 2

(v) length of the schedule

- length of the schedule = $\max(c_i) - \min(a_i)$

length of the schedule = $28 - 2 = 26$

(vi) throughput

$\eta = n/l$

$\eta = 4/26 = 0.1538$

n : number of processes

l : length of the schedule

(vii) % cpu idleness during schedule length

= units when cpu was idle/schedule length

= $3/26 = 0.1154 = 11.5\%$

(vii) % cpu efficiency during schedule length

$100\% - 11.5\% = 88.5\%$

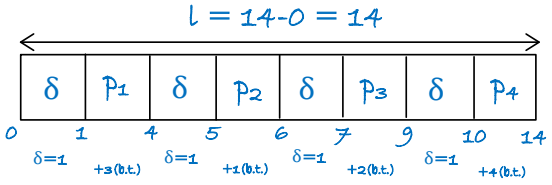
(iii) first come first serve with scheduling overhead (δ=1)

process no.	a.t.	b.t.	c.t.	tat	w.t.
1	0	3			
2	2	1			
3	5	2			
4	8	4			

assume :
- iobt = 0 (cpu bound process)
- initally δ=1

mode of operation : multiprogramming

(i) gantt chart



process no.	a.t.	b.t.	c.t.	tat	w.t.
1	0	3	4		
2	2	1	6		
3	5	2	9		
4	8	4	14		

(ii) turnaround time and average turnaround time

turnaround time = $(c_i - a_i)$

4-0=4
6-2=4
9-5=4
14-8=6

average turnaround time = Avg. TAT $(p_i) = 1/n \sum_{i=1}^{n} (C_i - A_i)$

$= (4+4+4+6)/4 = 4.5$

process no.	a.t.	b.t.	c.t.	tat	w.t.
1	0	3	4	4	
2	2	1	6	4	
3	5	2	9	4	
4	8	4	14	6	

(iii) waiting time and average waiting time

waiting time = tat - (b.t. + n.s)

0 + [4 (process scheduled at 4) -2 (process arrived at 2)] + (6-5) + (9-8)

0,2,1,1

process no.	a.t.	b.t.	c.t.	tat	w.t.
1	0	3	4	4	0
2	2	1	6	4	2
3	5	2	9	4	1
4	8	4	14	6	1

point to remember :
if just in case questions says (δ)is considered as waiting time then only include it

average waiting time = $\text{Avg. W.T} = 1/n \sum_{i=1}^{n} (C_i - A_i) - (X_i + Y_i)$

(0+2+1+1)/4=1

(iv) response time

response time = first waiting time ($W.T_1$)

response time = 1

(v) length of the schedule

- length of the schedule = $\max(c_i) - \min(a_i)$

length of the schedule = 14 - 0 = 14

(vi) throughput

$\eta = n/l$

$\eta = 4/14 = 0.2857$

n : number of processes
l : length of the schedule

(vii) % cpu efficiency during schedule length

10/14 = 0.7143 = 71.43%

(viii) % cpu overhead during schedule length

4/14 = 0.2857 = 28.57%

(iv) first come first serve with IOBT

bt + iobt = service time

concurrent io : there are multiple io services available (without any dependency on each other)

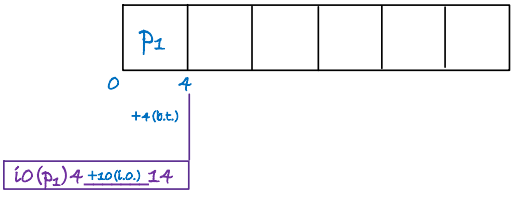
process no.	a.t.	b.t.	iobt	b.t.	c.t.	tat	w.t.
1	0	4	10	3			
2	3	2	3	4			

assume :
- iobt = 1
- initally δ=0

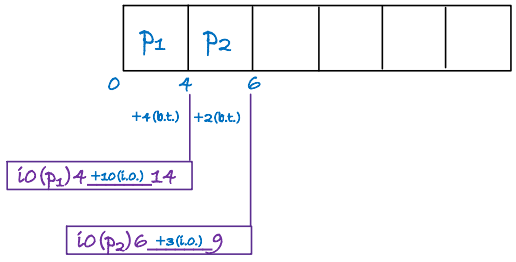
mode of operation : multiprogramming

(i) gantt chart

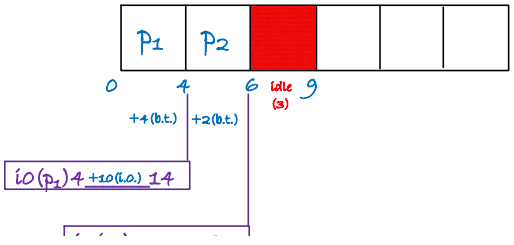
(a) p_1 arrives at 0, burst time completes at 4 and it goes for io for another 10 units.

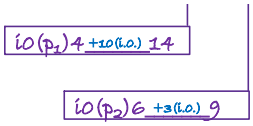


(b)p_2 arrived at 3, as soon as p_1 goes for io, p_2 starts execution and completes at 6 and then it goes for io for another 3 units.

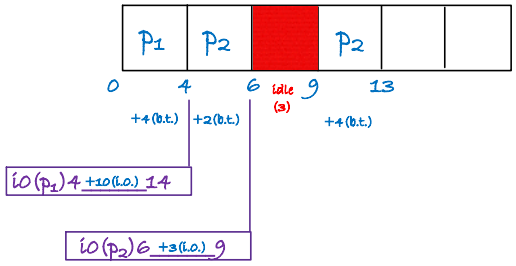


(c) now the cpu is idle for next 3 units because p_1 completes its i/o on 14 and p_2 completes its i/o on 9.

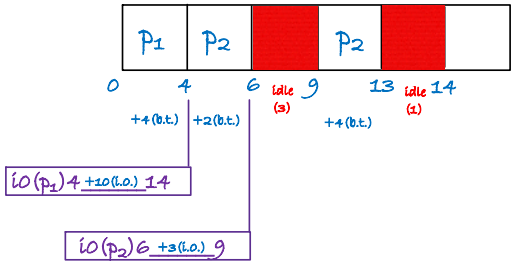




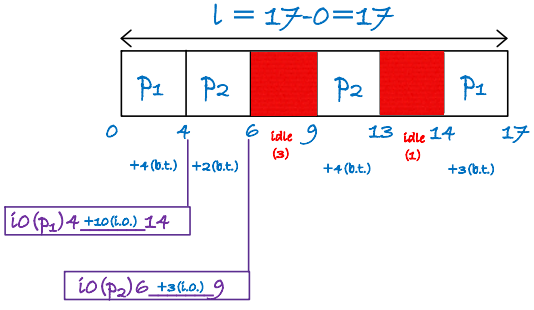
(d) p_2 completes it i/o and runs for next 4 units and completes its execution on 13



(e) p_2 is completed on 13 but the cpu will be idle for 1 more unit because p_1 is still performing its io operation.



(f) p_1 runs again on cpu and executesfor another 3 units and completes its execution.



process no.	a.t.	b.t.	iobt	b.t.	c.t.	tat	w.t.
1	0	4	10	3	17		
2	3	2	3	4	13		

(ii) turnaround time and average turnaround time

turnaround time = $(c_i - a_i)$

17-0=17

13-3=10

average turnaround time = Avg. TAT $(p_i) = 1/n \sum_{i=1}^{n} (C_i - A_i)$

= (17+10)/2=13.5

process no.	a.t.	b.t.	iobt	b.t.	c.t.	tat	w.t.
1	0	4	10	3	17	17	
2	3	2	3	4	13	10	

(iii) waiting time and average waiting time

waiting time = tat - (b.t + iob.t.)

(17-17) = 0

(10-9) = 1

average waiting time = Avg. W.T = $1/n \sum_{i=1}^{n} (C_i - A_i) - (X_i + Y_i)$

(0+1)/2=0.5

process no.	a.t.	b.t.	iobt	b.t.	c.t.	tat	w.t.
1	0	4	10	3	17	17	0
2	3	2	3	4	13	10	1

(iv) response time

response time = first waiting time ($W.T_1$)

response time = 0

(v) length of the schedule

- length of the schedule = max(c_i) - min(a_i)

length of the schedule = 17 - 0 = 17

(vi) throughput

$\eta = n/l$

$\eta = 2/17 = 0.1176$

n : number of processes

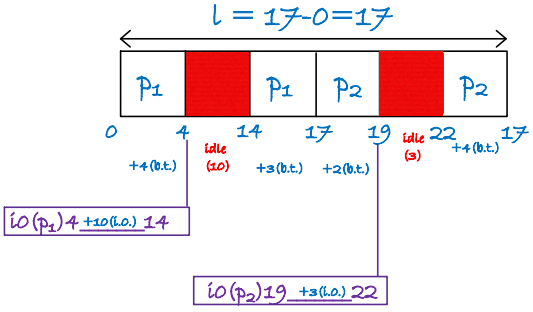
l : length of the schedule

(vii) % cpu idleness during schedule length

4/17 = 0.2353 =23.53%

mode of operation : uniprogramming environment

first p_1 completes it execution and then p_2



concept : shortest job first (SJF)/shortest process next(SPN)

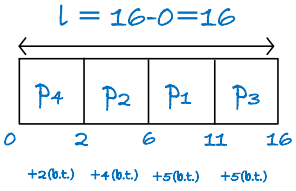
selection critieria : burst time, lowest burst time runs on cpu

tie breaking rule : if burst time is same then check arrival time and even if arrival time is same then select lower p_{id}.

mode of operation : non-pre-emptive

process no.	a.t.	b.t.	c.t.	tat	w.t.
1	0	5			
2	0	4			
3	0	5			
4	0	2			

(i) gantt chart



process no.	a.t.	b.t.	c.t.	tat	w.t.
1	0	5	11		
2	0	4	6		
3	0	5	16		
4	0	2	2		

(ii) turnaround time and average turnaround time

turnaround time = $(c_i - a_i)$

$11-0=11$

$6-0=6$

average turnaround time = $\text{Avg. TAT}(p_i) = 1/n \sum_{i=1}^{n} (C_i - A_i)$

$= (11+6+16+2)/4 = 8.75$

16-0=16

2-0=2

process no.	a.t.	b.t.	c.t.	tat	w.t.
1	0	5	11	11	
2	0	4	6	6	
3	0	5	16	16	
4	0	2	2	2	

(iii) waiting time and average waiting time

waiting time = tat - (b.t + iob.t.)

11-5=6

6-4=2

16-5=11

2-2=0

average waiting time = $\text{Avg. W.T} = 1/n \sum_{i=1}^{n} (C_i - A_i) - (X_i + Y_i)$

(6+2+11+0)/4 = 4.75

process no.	a.t.	b.t.	c.t.	tat	w.t.
1	0	5	11	11	6
2	0	4	6	6	2
3	0	5	16	16	11
4	0	2	2	2	0

(iv) response time

response time = first waiting time ($W.T_1$)

response time = 0

concept : shortest remaining time first (SRTF)

pre-emptive s.j.f.

mode of operation : pre-emptive

we will start with the process with least burst time.

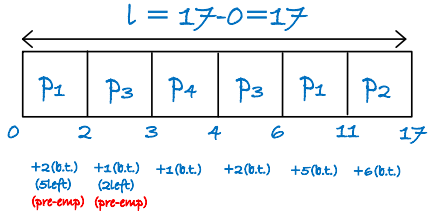
continue to run the process until the new process arrives, once new process arrives we compare the remaining time of the running process with burst time of new process

if burst time of new process is strictly less than the remaining time of running process then running process will be pre-empted and get back to ready queue and the new process will get the the chance to run on cpu

cycle will repeat till the completion of all processes.

process no.	a.t.	b.t.	c.t.	tat	w.t.
1	0	7			
2	1	6			
3	2	3			
4	3	1			

(i) gantt chart



process no.	a.t.	b.t.	c.t.	tat	w.t.
1	0	7	11		
2	1	6	17		
3	2	3	6		
4	3	1	4		

(ii) turnaround time and average turnaround time

turnaround time = $(c_i - a_i)$

11-0=11
17-1=16
6-2=4
4-3=1

average turnaround time = $\text{Avg. TAT } (p_i) = 1/n \sum_{i=1}^{n} (C_i - A_i)$

= (11+16+4+1)/4=8

process no.	a.t.	b.t.	c.t.	tat	w.t.
1	0	7	11	11	
2	1	6	17	16	
3	2	3	6	4	
4	3	1	4	1	

(iii) waiting time and average waiting time

waiting time = tat - (b.t + iob.t.)

11-7=4
16-6=10

average waiting time = $\text{Avg. W.T} = 1/n \sum_{i=1}^{n} (C_i - A_i) - (X_i + Y_i)$

(4+10+1+0)/4=3.75

4-3=1

1-1=0

process no.	a.t.	b.t.	c.t.	tat	w.t.
1	0	7	11	11	4
2	1	6	17	16	10
3	2	3	6	4	1
4	3	1	4	1	0

point to remember :

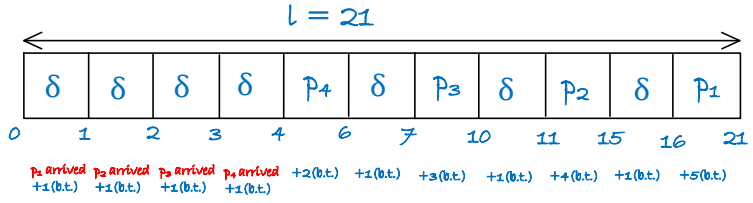
- once new processes ends, apply sjf.

concept : shortest remaining time first (SRTF) with cpu scheduling overhead (δ)

process no.	a.t.	b.t.	c.t.	tat	w.t.
1	0	5			
2	1	4			
3	2	3			
4	3	2			

(δ) = 1

(i) gantt chart



process no.	a.t.	b.t.	c.t.	tat	w.t.
1	0	5	21		
2	1	4	15		
3	2	3	10		
4	3	2	6		

(ii) turnaround time and average turnaround time

turnaround time = $(c_i - a_i)$

21-0=21

15-1=14

average turnaround time = Avg. TAT $(p_i) = 1/n \sum_{i=1}^{n} (C_i - A_i)$

= (21+14+8+2)/4 = 11.25

15-1=14
10-2=8
6-3=2

process no.	a.t.	b.t.	c.t.	tat	w.t.
1	0	5	21	21	
2	1	4	15	14	
3	2	3	10	8	
4	3	2	6	2	

(iii) waiting time and average waiting time

waiting time = tat - (b.t + iob.t.)

21-5=16
14-4=10
8-3=5
2-2=0

average waiting time = $Avg.\ W.T = 1/n \sum_{i=1}^{n} (C_i - A_i) - (X_i + Y_i)$

(16+10+5+0)/4=7.75

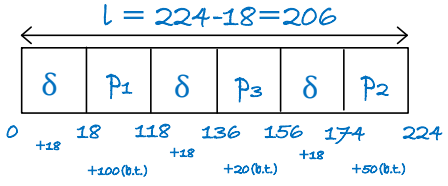
challenge : for what range of values of δ, 0<δ<ub s.t, SRTF continues to perform better than SJF with respect to average turn around time?

process no.	a.t.	b.t.
1	0	100
2	25	50
3	50	20

for δ = 18

mode of operation : non-pre-emptive (sjf)

l = 224-18=206

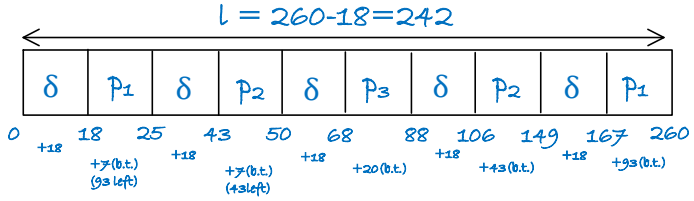


avg tat = (118+(224-25)+(156-50))/3
= 141

for δ = 18

mode of operation : pre-emptive (srjf)

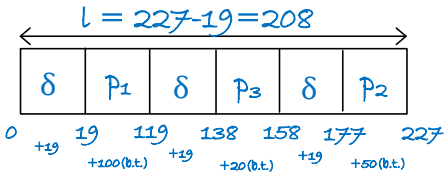
l = 260-18=242



avg tat = (260+124+38)/3 = 140.6667

for δ = 19

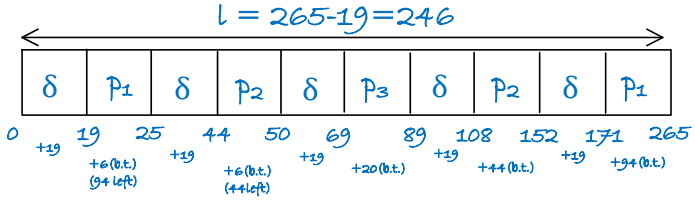
mode of operation : non-pre-emptive (sjf)



avg tat = (119 + (227-25) + (158-50))/3
= 143

for δ = 19

mode of operation : pre-emptive (srjf)



avg tat = (265+127+39)/3 = 143.6667

range of values of δ, 0<δ<18 s.t, SRTF continues to perform better than SJF with respect to average turn around time

result of previous gantt chart i calculated :

for δ =	avg.tat (s.j.f)	avg.tat (s.r.t.f)
2	109	92.66
4	106.33	98.66
6	117	104.66
13	127	122.33
12	129	122.66
25	156.66	161.66
24	153	158.66
22	150.66	152.66
21	147	149.66
19	143	143.66
18	141	140.66

concept : perfomance of sjf

sjf keep on favouring short process

advantages :

(i) maximum throughput
(ii) min average tat and minimum average w.t.

disdvantages :

(i) burst time of processes are not known (non-implementable), starvation to longer processes.

- sjf is non-implementable because burst time is unknown and if we run and complete the processes to know it then why we will schedule it back onto the cpu?

then why do we still use sjf?

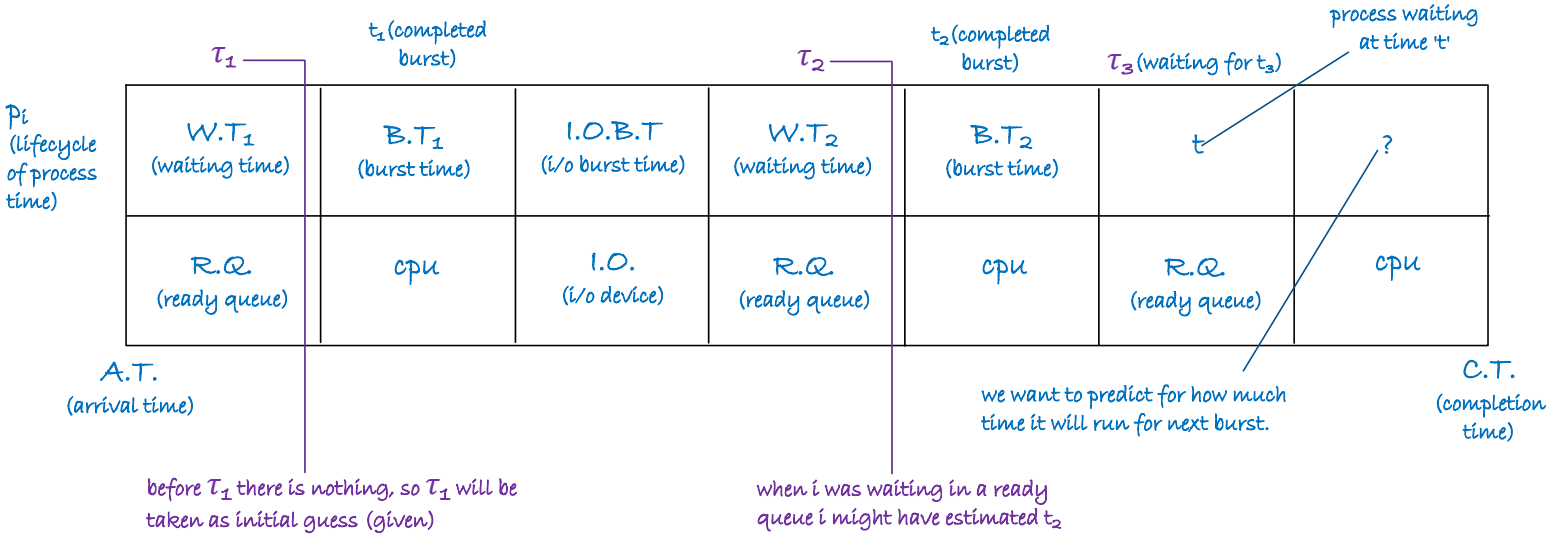
(i) benchmark to measure the perfomance of other algorithms with respect to s.j.f.

(ii) we cannot implement it with actual burst times however we can implement it with predicted/approximated burst times.

concept : exponential averaging technique (aging algo)

to predict next cpu burst of a process.

- let p_i be process;
- t_i be completed burst of a process;
- τ_i be predicted burst of a process;
- $\tau_{(n+1)}$ be next cpu burst of a process;



formula : $\tau_{(n+1)} = \alpha t_n + (1-\alpha)\tau_n$

t_n = actual length of n^{th} cpu burst

τ_{n+1} = predicted value of the next cpu burst

$\alpha, 0 \leq \alpha \leq 1$

question :

consider a system using exponential averaging technique to predict the next cpu burst of the process; the previous runs of the process has generated the burst times of 6, 12, 8, 10.

for $\alpha=1/2$ and $\tau_1=8$ what is the next predicted cpu burst time?

$\tau_{(n+1)} = \alpha t_n + (1-\alpha)\tau_n$

$\tau_{(5)} = 1/2(t_4+\tau_4) = 1/2(10+\tau_4)$ (τ_4 is unknown)

$\tau_{(4)} = 1/2(t_3+\tau_3) = 1/2(8+\tau_3)$ (τ_3 is unknown)

$\tau_{(3)} = 1/2(t_2+\tau_2) = 1/2(12+\tau_2)$ (τ_2 is unknown)

$\tau_{(2)} = 1/2(t_1+\tau_1) = 1/2(6+8)$ ($\tau_1=8$) $= 7$

$\tau_{(3)} = 1/2(t_2+\tau_2) = 1/2(12+\tau_2)$ ($\tau_2=7$) $= 9.5$

$\tau_{(4)} = 1/2(t_3+\tau_3) = 1/2(8+9.5)$ ($\tau_3=9.5$) $= 8.5$

$\tau_{(5)} = 1/2(t_4+\tau_4) = 1/2(10+8.5)$ ($\tau_4=8.5$) $= 9.375$

Consider the following four processes with arrival times (in milliseconds) and their length of CPU bursts (in milliseconds) as shown below:

Process	P_1	P_2	P_3	P_4
Arrival time	0	1	3	4
CPU burst time	3	1	3	Z

These processes are run on a single processor using preemptive Shortest Remaining Time First (SRTF) Scheduling Algorithm. If the average waiting time of the processes is 1 millisecond, then the value of Z is

if z<3

cpu : | P_1 | P_2 | P_1 | P_4 | P_3 |

0 1 2 4 4+z 7+z

avg. w.t. = (1+0+(z+1)+0)/4

1 = (z+2)/4

z=2

if z>3

cpu : | P_1 | P_2 | P_1 | P_3 | P_4 |

0 1 2 4 7 7+z

avg. w.t. = (1+0+1+3)/4

= 5/4 = 1.25

chapter - 3 (memory management)

Saturday, October 5, 2024 12:57 AM

chapter - 4 (file system and disk management)

Saturday, October 5, 2024 12:58 AM