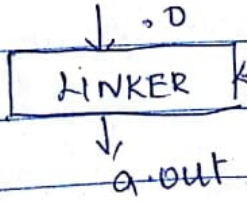
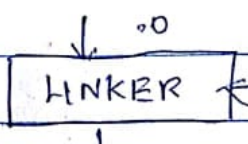
LINUX.

LIBRARY'S

* Library is a collection of predefined functions
* Library is collection of object files.
* There are 2 types of libraries with respect to OS.

Library

Static Dynamic.

Static	Dynamic.
* Helps in static linking	Helps in Dynamic linking 2 types of Dynamic linking * Load time linking * Runtime linking.
* also called as archeives	also called as Shared object (or) dll.
* extension for static library is .a Ex:- libc.a, libm.a	extension for dynamic library is .so (or) dll. Ex:- libc.so, libm.so.
	

⇒ If a.out is generated by linking to the static library, then function definitions are copied into executable file	If the linker is linking to the dynamic library, then the function definitions are not copied into executable file but they are used
⇒ a.out file size is more	a.out file size is less
⇒ a.out is independent of any other libraries during execution time	During execution time, a.out file is dependent on some libraries ./a.out + dependancy files also loaded.

File Command :-

File filename (object filename)

file command determines the type of the file

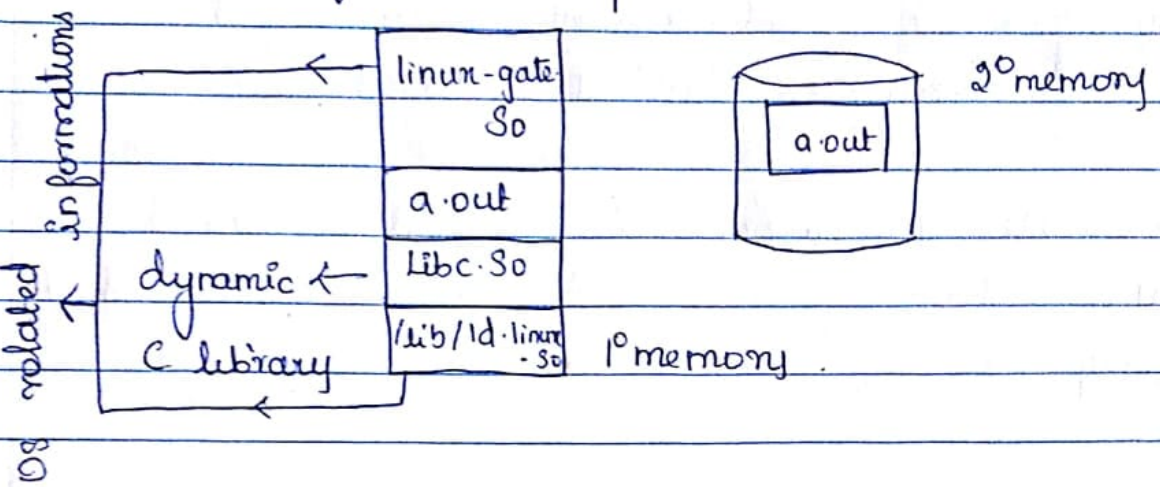
ELF ⇒ Executable linkable format 32 bit LSB (Linux Standard base) relocatable, Intel 80386 processor.

By default linker will link to dynamic libraries / hence function.

ldd command :-

ldd executable filename

This command displays the files on which the executable file is dependent.



* .o files & executables files are not portable, they are specific to certain environment (which is particular to that machine).
* whereas .c files are portable.

cc -static filename -o newname (exe).

The above command will make the linker to link to static library (i.e) we are explicitly saying to the linker to link to the static libraries.

ldd filename (exe).

o/p :- not a dynamic executable (becoz it is statically linked)

size filename (size command).

size command shows the size of the total executable file.

A function is said to be statically linked function, if it is defined inside the executable file.

A function is said to be dynamically linked function if it is not defined inside the executable but it is used.

An executable file (a.out) is statically linked executable file, if there exist no functions which are going to be linked in lateral stages.

An executable file (ELF) is a dynamically linked executable file, if there exist at least one function which is going to be linked in lateral stages.

Main (prog.c)

```
#include<stdio.h>
main()
{
printf("in main program\n");
printf("%d", Sum(10,10));
printf("mul = %d\n", mul(10,10));
printf("vector");
}
```

Sum.c

```
int Sum(int a, int b)
{
return(a+b);
}
```

mul.c

```
int mul(int a, int b)
{
return(a*b);
}
```

Print.c

```
void print(char* c)
{ puts(c);
}
```

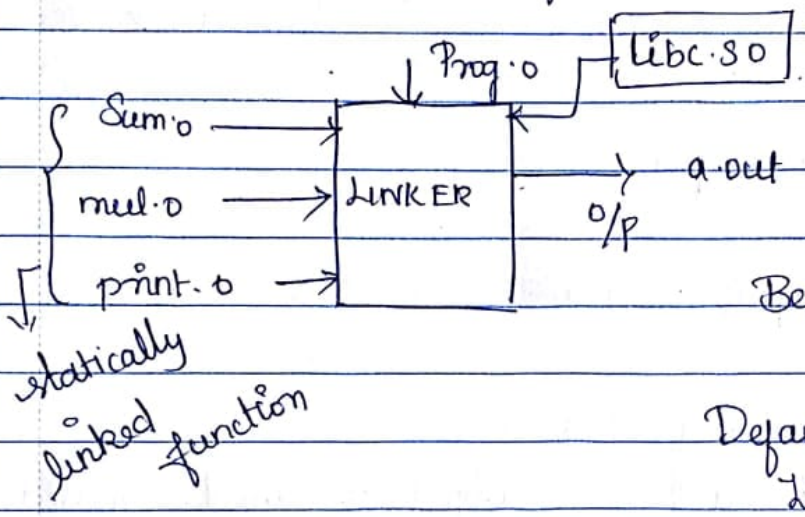
Create object file for all files.

nm prog.o (object file)

This Command will display the global Variables, static Variables and informations related to functions by providing the Object file in that ..,

T - means functions are defined

U - " functions not defined.



[Dynamically linked executable file]

Because printf & puts are not defined. By Default the linkers will link to dynamic libraries.

when we are giving static explicitly, then that executable file is statically linked executable file.

22/07/2017

POSITIONAL INDEPENDENT CODE :-

* The cc-fpic option specifies that the Compiler should generate position independent code
* This changes the way that the Compiler generates code for operation such as accessing global state & external

Variables, accessing string constants, and taking the addresses of functions.

* These changes allow the code to be located at any virtual address at runtime. This is necessary for shared libraries since there is no way of knowing at link time where the shared library code will be located in memory.

Object files Creation for Library

```
cc -c -fpic filename
```

Command for static library Creation :-

```
ar rcs libstatic.a objectfilesnamed (created using -fpic).
```

ar → archieves
rcs → r-insert, c-create, s-inden
libstatic.a → Name of our library.

Create static library and insert all the inden (files) provided.

Command for dynamic library Creation :-

```
cc -shared -o libdynamic.so objectfilename (created using fpic).
```

Note: Dynamic libraries will be generated as executable files whereas libraries are not.

```
#include<stdio.h>
main()
{
int a, b;
int opt, ret;
printf("Enter the value of a\n");
scanf("%d", &a);
printf("Enter the value of b\n");
scanf("%d", &b);
printf("Enter the option\n");
printf("1. Sum  2. Mul  3. Print\n");
scanf("%d", &opt);
if(opt==1)
{
ret = sum(a,b);
printf("ret = %d\n", ret);
}
else if(opt==2)
{
ret = mul(a,b);
printf("ret = %d\n", ret);
else if(opt==3)
{
print vector);
}
else
printf("Invalid operation");
}
```

Note: During Compilation, we should include the library that is created by us (i.e) as the function definitions are available in the library created by us, if we are not adding then it will show error, as the linker will link by default only to the C library.

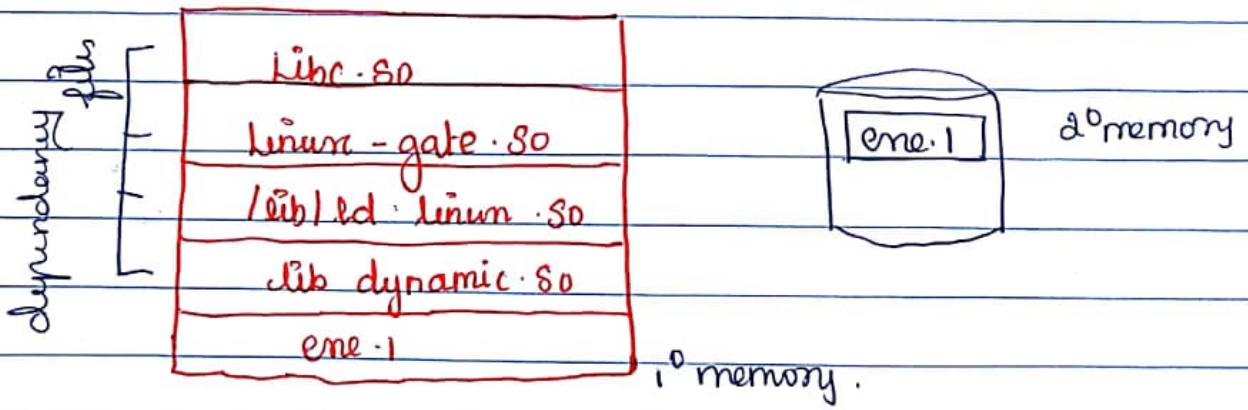
cc filename ./library name.

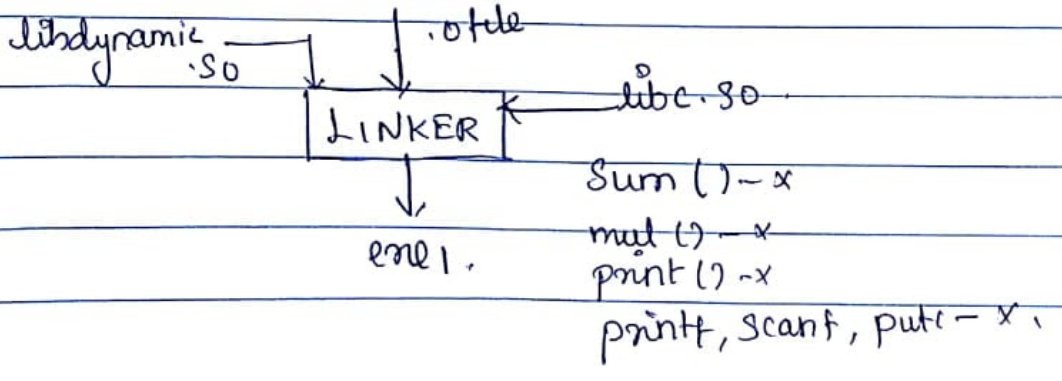
change the name of executable file

cc filename ./ library name -o exe1.
dynamic c

Hence exe1 is a dynamically linked executable file.

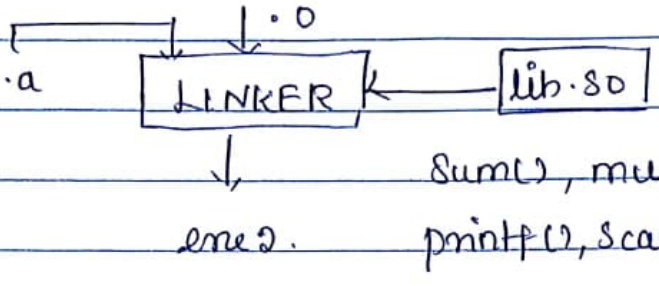
When using ldd exe1 command, it will show all the dependancy files of exe1, in that my library will also be included.





cc filename ./staticlibraryname -o exe2.

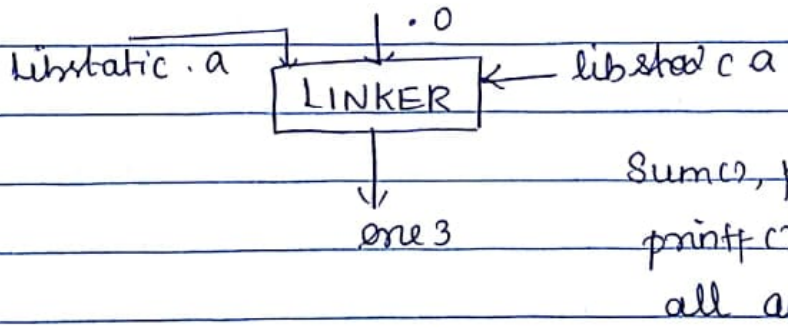
exe2 is a dynamically linked executable file



Sum, mul and print functions are statically linked, but printf, scanf and putc functions are not defined.

cc -static filename ./staticlibraryname -o exe3.

exe3 is a statically linked executable file.



if we use "ldd filename" command, then it will show "not a dynamic executable" because the function definitions are already copied into executable file and it is not dependent on any files after linker stage.

Copying libraries from pwd to lib path

cp ./libname /lib/ → must be a root user to copy.

Note: So by default linker will link only libc.s So to make it link with other library from the lib path (we should provide its linking command).

cc filename -ldynamic / to link my library also during
library file linker stage.

Dynamic Libraries are having the behaviour of executable file (i.e) during execution of program a.out is loaded into primary. linker by default link to dynamic libraries. So the dynamic library file are also loaded. Thus the dynamic library 2 executable file are in green color.

24/07/2017 RUNTIME LINKING:

To load the Certain libraries for the program according to the demand of the program during the runtime.

Programming interface to dynamic linking with loader:-

#include <dlfcn.h> => Header file.

These are the four functions to interface to dynamic linking loader

```
void *dlopen(const char *filename, int flag);

char *dlerror(void);
void *dlsym(void *handle, const char *symbol)
int dlclose(void *handle)
```

Link with -ldl.

dlerror(): The function dlerror() returns a human readable string describing the most recent error that occured from dlopen(), dlsym() or dlclose() since the last call to dlerror().

dlopen(): The function dlopen() loads the dynamic library filename by the null-terminated string filename & returns an opaque ("address") "handle" for the dynamic library

If dlopen() fails for any reason, it returns 0.

dlsym(): The function dlsym() takes an "handle" of a dynamic library returned by dlopen() and the null-terminated-symbol name, returning the address where that symbol is loaded into memory.

If the symbol is not found, in the specific library or any of the libraries that were automatically loaded by dlopen() when that library was loaded, dlsym() returns NULL.

dlclose(): The function dlclose() decrements the reference on the dynamic library handle handle.

If the reference count drops to zero & no of other loaded libraries use symbols in it, then the dynamic library is unloaded.

Ldd Command will display only the dependancy files loaded into the 1° primary memory along with the executable file, not the libraries loaded during the runtime.

```
#include <stdio.h>
#include <dlfcn.h>
int sum(int, int);
int mul(int,int);
void print(char *);
main()
{
int a,b;
int opt,ret;
void *handler;
int (*ptr)(int,int);
void (*p)(char *);
printf("enter a value\n");
Scanf("%d", &a);
printf("enter b value\n");
Scanf("%d", &b);
printf("1. Sum 2. mul 3. Print\n");
printf("Enter the option :\n");
scanf("%d", &opt);
if(opt==1)
{
handler = dlopen("/bdynamic.so", RTLD_LAZY);
if(handler==0)
{
printf("%s\n", dlerror());
return;
}
ptr = dlsym(handler, "sum");
if(ptr==0)
{ printf("%s\n", dlerror());
return;
}
ret = (*ptr)(a,b);
printf("ret = %d\n", ret);
}
```

```
else if (opt==2)
{
handler = dlopen ("libdynamic.so",
                  RTLD_LAZY);
if (handler==0)
{
printf("%s\n", dlerror());
return;
}
ptr = dlsym(handler, "mul");
if (ptr==0)
{
printf("%s\n", dlerror());
return;
}
ret = (*ptr)(a,b);
printf("ret = %d\n",ret);
}
```

```
else if (opt==3)
{ handler = dlopen ("libdynamic.
                    so", RTLD_NOW)
if (handler==0)
{
printf ("%s\n", dlerror());
return;
}
ptr = dlsym(handler, "print");
if (ptr==0)
{ printf ("%s\n", dlerror());
  return;
}
else printf ("in valid option\n"); }
```

RTLD LAZY & RTLD NOW :-

* Undefined function symbols in the library should be resolved only as the code is executed. If a piece of code requiring a particular symbol is not executed, the symbol never resolved.

* In this case first library is loaded into Virtual memory when a symbol is resolved then loaded into physical memory.

RTLD NOW :-

All undefined symbols in the library should be immediately resolved before dlopen()

Complete regardless of whether they will ever be required.

* In this case, library is loaded into physical memory directly.

Virtual Memory → Part of the Secondary memory, act as primary memory.

Physical Memory - Usually RAM.

26/07/2011

STATIC VS DYNAMIC

When a program is built by linking against a static library, the resulting executable file includes copies of all the object files that were linked into the program.

Thus, when several different executable files use the same object module, each executable file has its own copy of the object module. This redundancy of code has several disadvantages.

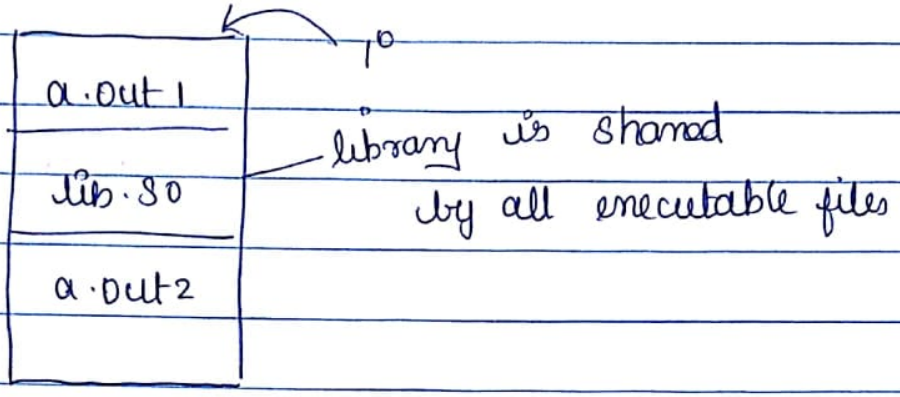
Disk space is wasted storing multiple copies of the same object modules. Such wastage can be considerable.

If several different programs are using the same modules, are running at the same time, then each holds separate copies of the object modules in memory demands on the s/m.

If a change is required to an object module in a static library, then all executables using that modules must be relinked in order to incorporate

the change. File relocation must be performed at runtime.

Because of this relocation process a program using a shared library may takes a little more time to execute than its statically linked equivalent



In static library, need recompilation when modification is done in library.
In dynamic library, does not need recompilation

OS CONCEPT WITH LINUX

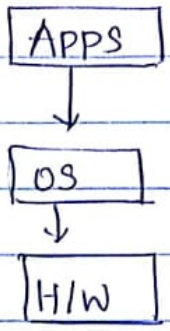
* Process Management
* Signal handling
* Resource Management
* File Management
* Interprocess Communication (IPC)
  - Pipes
  - named pipes
  - shared memory
  - message queues.
* Process Synchronisation using Semaphores
* Thread creation & thread Synchronisation
* Memory Management.

Os Booting Process:

An operation system is a system software that manages Computer H/w & S/w resources and provides servies for Computer program.

All computer programs, excluding firmware, require an Os to function.

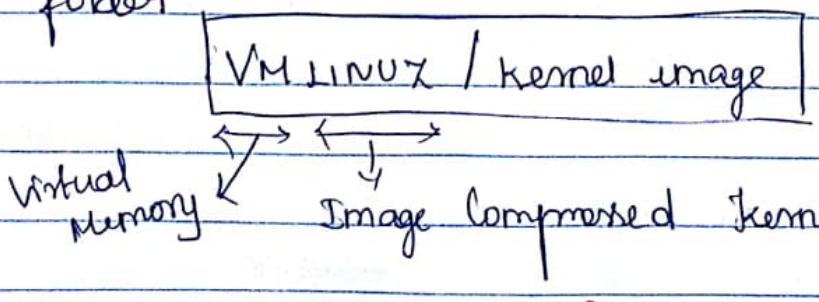
Os is called as Resource Manager.



Stages of booting process:-

1. BIOS
2. Master boot Record (MBR)
3. GRUB (Grand unified Boot Loader)
4. Init process
5. Runlevel program.

Boot folder



gcc -v → to know the Version of Compiler.

Components of OS:

* Applications (for every application Service is required).
* Services

Applications are optional but Services are mandatory.

Services are provided by OS.

OS is required when multi-tasking process.

↓ RTOS

↓ GPOS (General-purpose Operating System)

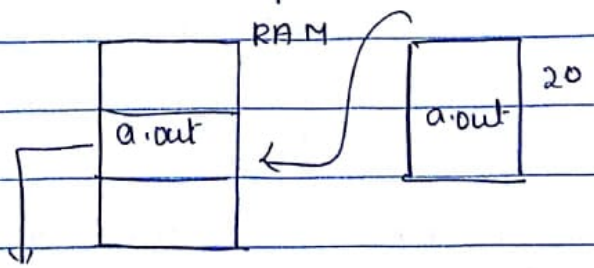
GPL - General Public License → Linux

Linux: Linux is Open-Source, it will support for many hardware architectures

Major Core of OS is called as Kernel

Linux is Complimented in C

Process Management: The program which is under execution is called as process.



This is the process.

Running more than one process, is possible under multi-processing.

Process Manager will take care about all the processes.

Command :- ./a.out & → background job
./a.out → foreground job

During background job, we can use any commands but in the case of foreground job, we can't use any commands.

Command :- Ps
will display all the process that is running under a particular terminal.

Command :- fg
To bring the background process to foreground process.

27/7/2017

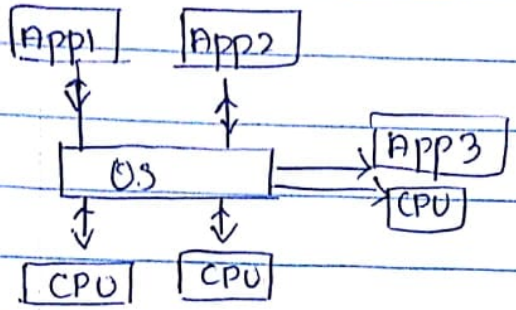
⇒ where there is need of handling the multiprocessing environment, there is a need of OS.

⇒ when 1 cpu is available, multiple processes ready to execute, process manager will take care.

⇒ multiprocessing basically 2 types
* H/W multiprocessing
* S/W multiprocessing

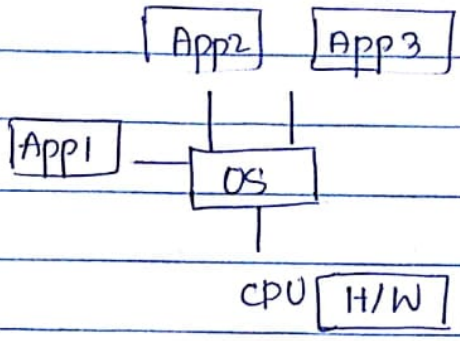
multi-processing

H/w multiprocessing



many H/w

S/w multiprocessing



many applications
2 / CPU.

* Cost is good	* cost is less
* No role of process manager	* Process manager is playing important role
* Performance is good	* Performance is slow

when a process is running in background it displays 2 data

[1] 1996

↓ ↓

Job Id Process Id.

when a process is running, process manager will allocate a unique number called process Id (PID).

TTY - pseudo terminal (number under what)

Process Id is the way by which Process manager identifies a particular process (Unique to each & every process).

The same job Id & process Id can't be allocated to the same process executing at little variance of time.

Ps command display the PID's and process running at the background.

Using ctrl+c : Termination of foreground process is possible dermination of background process is not possible.

Last process which enters into background process, will come first as a foreground process when we are using fg command.

To get a particular process out of a background process, then use fg command along with Job Id.

Syntax :- Fg [J].
Job_Id

To see all the jobs, running all the terminals / non - terminals

PS -e

In that '?' indicates that they are not running under any terminal.

To process which are running under background

DAEMON PROCESSES :- A daemon is a special-purpose process that is created & handled by the system in the same way as other processes, but is distinguished by the following characteristics.

* It is long-lived. A daemon process is often started at system boot and remains in existence until the system is shut down / majority of services.

* It runs in the background and has no controlling terminal (?) from which it can read input or to which it can write output.

Certain Commands can display their PID's but " " Can't " " PID's.

To know the Range of maximum PID, the process manager allocates can be known from

Vi /proc /sys/kernel/ Pid_max.

There are some PID's fixed for certain background process / services.

After reaching the maximum, the PID's will start from 300, but not from 0. on some of the PID's are already fixed for some background process. majority of the lower number are reserved for Some Services.

28/07/2017. TO PRINT @ THE PID of a PROCESS:-

man 3 getpid

Header file =># include <unistd.h>

Getpid → get the process ID

Pid_t getpid (void);

Pid_t is a typedefined unsigned int. The getpid() function shall return the process ID of the calling process.

getppid() → get the parent process ID.

Pid_t getppid(void);

Description:

* The getppid() function shall return the parent process ID of the calling process.
* Every process will have a parent process (to collect the status of child)
* For every process running by default under a particular terminal, bash will act as parent for every process
* Bash is like a shell.

```
#include <stdio.h>
main
{
printf ("in main..\n");
printf ("%d\n", getpid());
printf ("%d\n", getppid());
}
```

To kill the process running under background

Command :- kill -9 pid

↳ signal number (SIGKILL)

If any process is receiving signal -9, then that process will be killed / terminated

NOTE :

* Using ctrl+c a process is terminated if the process is running as foreground

* Using kill possible to terminate a process, if it running as a foreground & background.

MORE ABOUT PROCESS :-

STARVATION : In case of multiprocessing environment, when one process is under execution, remaining all jobs are starving for CPU called as starvation

TURN-AROUND TIME : Time gap b/w process submission to process completion (Total time of a process).

RESPONSE - TIME :- When process is submitted, the 1st response from the CPU (first instruction execution)

THROUGHPUT :- No of jobs completed per unit of time

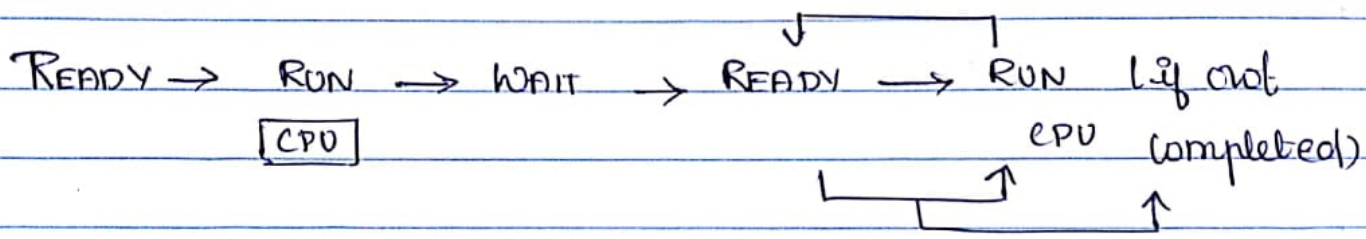
STATE OF PROCESS : During the life of the process, a process may not get CPU time always, if the process is not getting the CPU time at that time, the process may enter into different states

(i) Ready - state
(ii) Suspended state
(iii) Wait State
(iv) Delayed State.

Ready - state: If a process is in readystate, it needs only CPU, nothing else

* In multi-processing environment when one process is under execution, other process has to wait for the execution.

WAIT - State: when a process is executing, it may enter into wait state (waiting for I/O events)



Example for Wait State:

```
#include <stdio.h>
main()
{
int x;
printf("in main..\n");
printf("Pid = %d\n", getpid());
Scanf("%d\n", &x);
printf("x = %d\n", x);
while(1);
}
```

Command : ps -elgrep · pts/2

↓ pipe

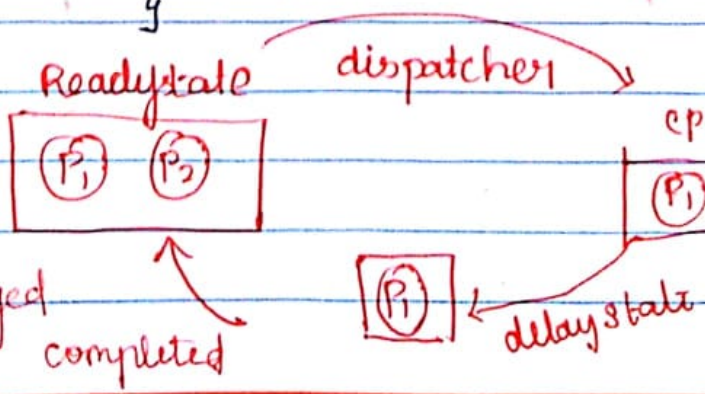
To know the status of a foreground process, running under a terminal using another terminal.

The o/p of ps -e is given as i/p to grep and checks with the given pattern 2 displays all the process running with that id which is given as pattern

DELAYED - State : During the life of the process, a process may delay.

Example :-

```
#include<stdio.h>
main()
{
int n;
printf("in main..\n");
printf("PID = %d\n", getpid());
printf("process entered in delayed state\n");
Sleep(20);
printf("delayed Completed\n");
while(1);
}
```



Ctrl+c → Process terminated

Ctrl+z → Process suspended.

Suspended process can be resumed, but a terminated process can't be resumed.

SUSPENDED State:-

* A process in its life time possible to suspend by receiving some signals.
* If the process is running as foreground possible to suspend by using (Ctrl+z), background not possible.

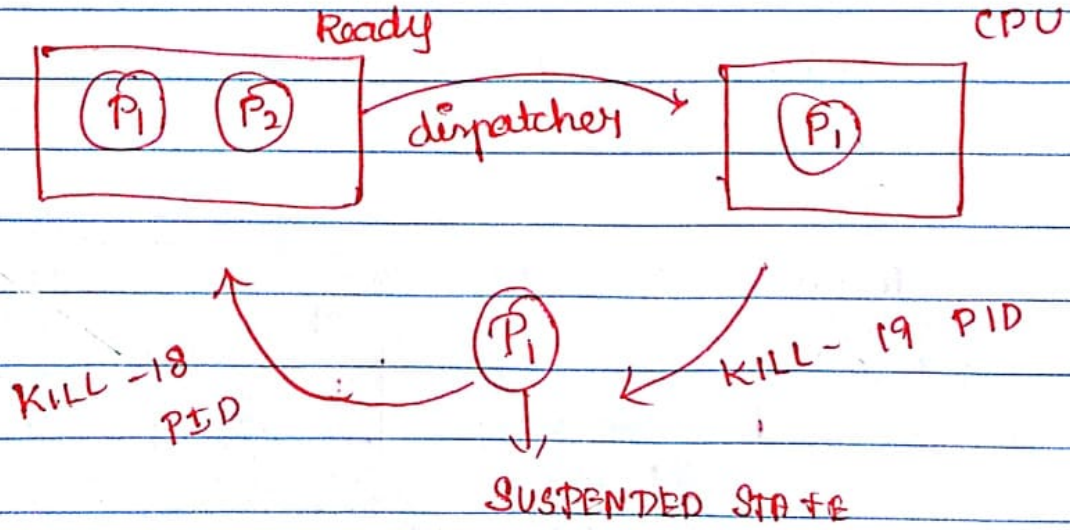
Command :- KILL -19 PID.

-19 → for Suspending

-9 → for termination

-18 → pid (resume).

Using kill possible to suspend both the background & foreground process.



29/7/17. CPU BOUND Vs I/O BOUND :-

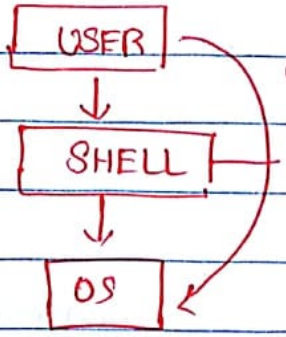
* If a process in its lifetime most of the cases demands the CPU means then it is called as CPU bound job.

* If a process, in its lifetime most of the case waiting for I/O events is called as I/O bound jobs.

* With respect to process manager, I/O jobs are best than the CPU bound jobs.

I/O bound jobs → Bash (bound again shell)

Shell is a command interpreter. It is an interface b/w user & OS.



Shell acts as a interface b/w user & OS.

Terminal is a logical window under which shell app is running.

Bash, ksh, sh - Shell.

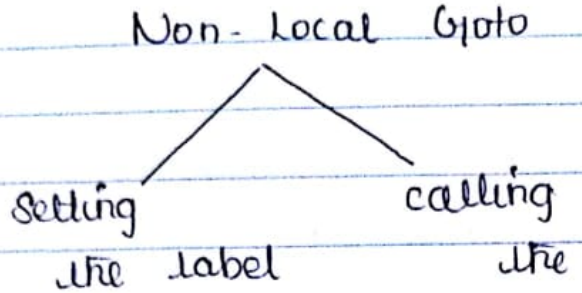
Command :- PS -el → (shows the long list of all the events).

Gnome - terminal is the parent of bash / bash will run under the gnome terminal.

PID of Scheduler is '0'

PID of Init process is '1'.

NON-LOCAL GOTO SETJMP & LONGJUMP




```
int Setjmp (jmp_buf env)

void longjmp (jmp_buf env, int val)
```

Longjmp() and setjmp(3) are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

```
#include <stdio.h>
#include <setjump.h>
main ()
{
int r;
printf ("hello\n");
r= setjmp(var);
printf ("r = %d\n", r);
printf ("after setjmp...\n");
printf ("long jmp going to be called\n");
 Sleep (2);
longjmp (var, 1);
printf ("after long
        jump...\n");
}
```

*Setjmp & Longjmp are connected to each other

* when setjmp is executed first time it returns zero.

* If setjmp is executed because of long jmp, hence set jmp returns long jmp's 2nd argument

```
#include<stdio.h>
#include<setjmp.h>
 jmp_buf var;
 void abc();
main()
 {
 int r;
printf("hello..\n");
 r= setjmp(var);
 printf("r=%d\n",r);
 printf(" after Setjmp..\n");
 printf(" abc fun going to be called..\n");
 abc();
 printf("after abc..\n");
}
void abc()
{
printf("in abc func..\n");
 longjmp(var,1);
 printf("after long jump\n");
}
```

jmp_buf var
typedef struct variable.

* The functions described here are used for performing "non-local gotos" transfering execution from one function to a pre-determined location in another function

* The setjmp() function dynamically establishes the target to which control, will later be transferred and longjmp() performs the transfer of execution.

* The setjmp() function saves various information about the calling environment (typically the stack pointer, the instruction pointer possibly the values of other registers & the signal mask) in the buffer env for later use by longjmp(). In this case setjmp return 0.

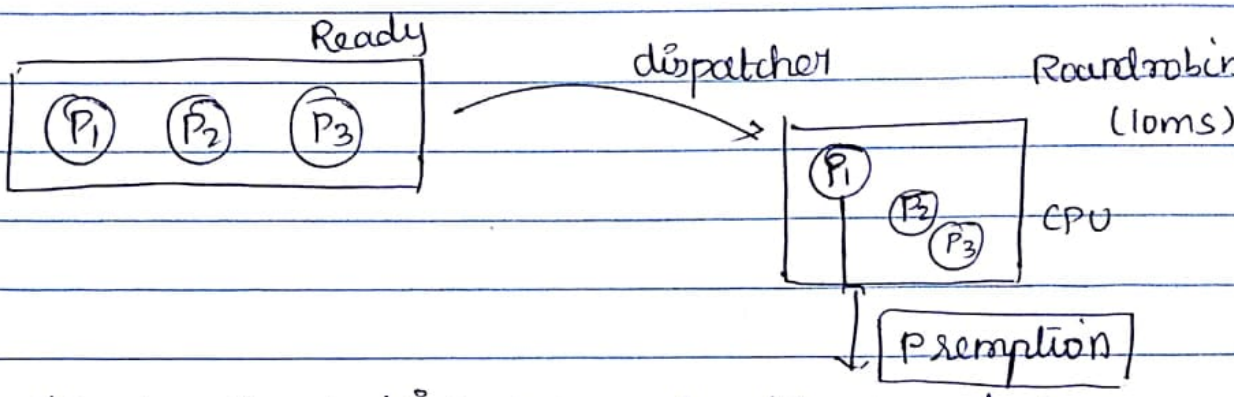
* The longjmp() function uses the information saved in env to transfer back to the point where setjmp() was called to restore (rewind) the stack to its state at the time of setjmp() call.

30/7/2017

```
#include<stdio.h>
#include<setjmp.h>
int sum(int, int);
int sub(int, int);
int mul(int, int);
int div(int, int);
jmp_buf v;
main()
{ int n1, n2, op, res, r;
r= setjmp(v);
if(r==3)
printf("error in mul function\n");
if(r==4)
printf("error in div function\n");
printf("enter n1 & n2 values\n");
scanf("%d %d", &n1, &n2);
printf("1. sum 2. sub 3. mul
                4. div\n");
printf("enter option\n");
scanf("%d", &op);
if(op==1)
{ res = sum(n1,n2);
printf("res= %d\n", res); }
else if(op==2)
{ res = sub(n1,n2);
printf("res= %d\n", res); }
else if(op==3)
{ res = mul(n1,n2);
printf("res= %d\n", res); }
else if(op==4)
{ res= div(n1,n2);
printf("res= %d\n", res); }
else
printf("invalid option\n");
}
int sum(int a, int b)
{ return(a+b); }
int sub(int a, int b)
{ return(a-b); }
int mul(int a, int b)
{ if(b==0 || a==0)
longjmp(v,3);
return(a*b); }
int div(int a, int b)
{ if(b==0)
longjmp(v,4);
return(a/b); }
```

Content switch :- Store the content of the running process into PCB (process control block) and load the content of the next process which is going to be executed into CPU.

* Switching from one process to another process is called content switch
* Mandatory in multiprocessing environment -



after (10ms) of time, each process forcably moved from CPU to ready state.
so that other processes get some time

PCB is created once, the process started executing, when the process gets completed, PCB is destroyed (i.e) life of the PCB is the life of the process.

PCB Content

PC :
SP :
CPU Reg :
Memory limit

"When switching from 1 process to another process, Content will also switch, Always CPU Contains running process content."

Premption: Even the process don't want to leave the CPU, forcably it is removed, so to allocate the CPU time to other process.

Non-Premption: Once a process is scheduled, until it is leaving CPU by its willingness and other process is around we made to wait.
Round Robin policy follows premption technique.

"Content switching Overheads the CPU, But without Content switching, there is no way in multiprocessing environment"

Content switch time can be possible to minimize, bt it is impossible to make it zero.
Using threads, Content switching can be minimized

SYSTEM FUNCTION:

System → execute a shell command
Header file → #include <stdlib.h>
int system (const char * command)
System() executes a command specified in command by calling /bin/sh -c command and returns after the command has been completed.

System() internally connects to the sh shell [/bin/sh-c] using system() possible to execute any command by calling internally sh shell.

```
#include <stdio.h>
main()
{
printf ("hello \n");
System(" ls"); // System("./a.out")/System("./P1");
System ("pwd");
printf ("hi \n");
}
```

System function usually loads the new process it can do also used for executing another process (like (program or any process)

```
# include <stdio.h>
main()
{
print (" hello \n");
System (" ./p.ou ");
System ("pwd");
printf ("hi \n");
}
```

```
./p
#include<stdio.h>
main()
{
printf (" in P1 process");
while (1);
}
```

o/p : hello
in p1 process

Main program

```
#include<stdio.h>
main()
{
printf("hello..\n");
printf("PID=%d, PPID=%d\n",
                getpid(), getppid());
System("./p1");
printf("hi..\n");
}
```

O/p:- PID = 1b PPID = Bash

P1 - process

```
#include<stdio.h>
main()
{
printf("in p1 process\n");
printf("PID=%d, PPID=
    %d\n", getpid(),
        getppid());
while(1);
}
```

PPID = a.out ; PID = 1D

Bash → a.out → sh → P1 process

System function follows the Sequential execution

In this above example, a.out process is not executing because P1 process is in while(1) hence P1 process keep on executing.

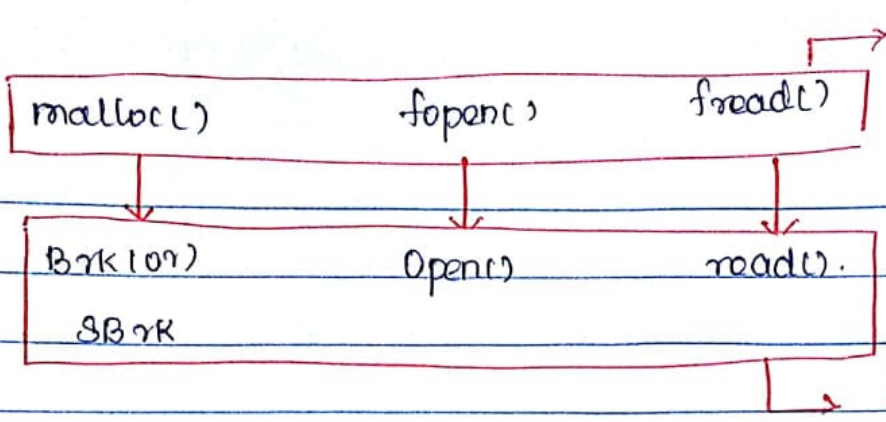
2/08/17.

LIBRARY FUNCTIONS Vs SYSTEM CALLS.

Library	System.
⇒ These are functions	⇒ System calls are also functions
⇒ Supported by library (or) Compiler supported fun	⇒ System calls are supported OS (Kernel).
⇒ Library functions are portable	⇒ System calls are not portable. (every OS having different interface).
⇒ Applications Implemented by with library functions are called as API (Application programming interface)	⇒ Called as SCI (system call Interface).
⇒ It supports buffered I/O concept	⇒ Systems call are not Supporting buffered - I/O

Programming

- Application programming
  - * API
- System programming
  - * Device Drive prog
  - * Kernel prog



Space

User Space

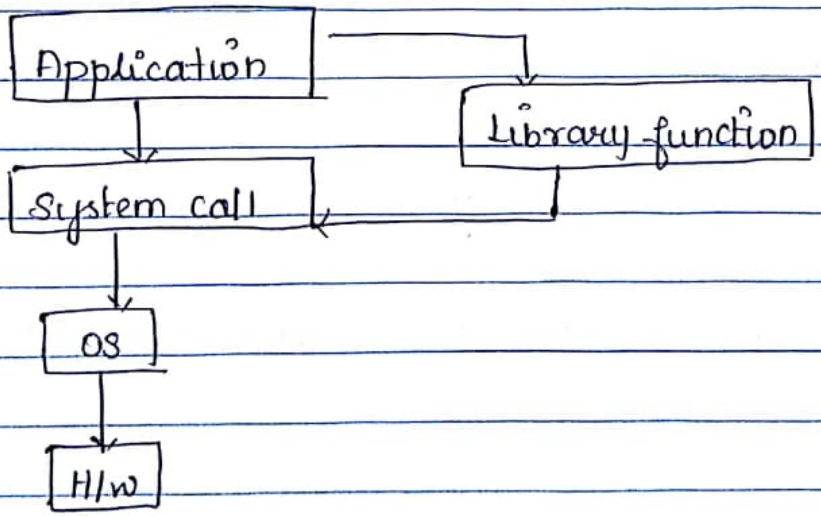
↓

User calls are always executed in User Space.

Kernel Space

↓

Systemcalls are always executed in Kernel space Systemcalls are interface b/w user space & kernel space.



System calls needs to work each and everytime when it called, but library functions works as whole and only once the System Call happens System call puts burden of CPU more than library functions.

"Sockets" are used for end systems Communication.

fopen() can open only text & binary file but Open system call can open any kind of file. To create socket files we should use only System calls (like Socket system call).

Note:- In Application, if we are wanting to use Os kernel space directly, System calls are mandatory to use. (Sockets :- like Sockets Creation, as that works only on Kernel Space).

Manual Pages & what is Contains

1. Executable programs or shell Commands
2. System Calls (functions provided by Kernel)
3. Library function calls (" " " Library).

Fork [System Call]

fork → Create a child process.

#include <unistd.h> pid_t fork(void);

↓

type defined unsigned int

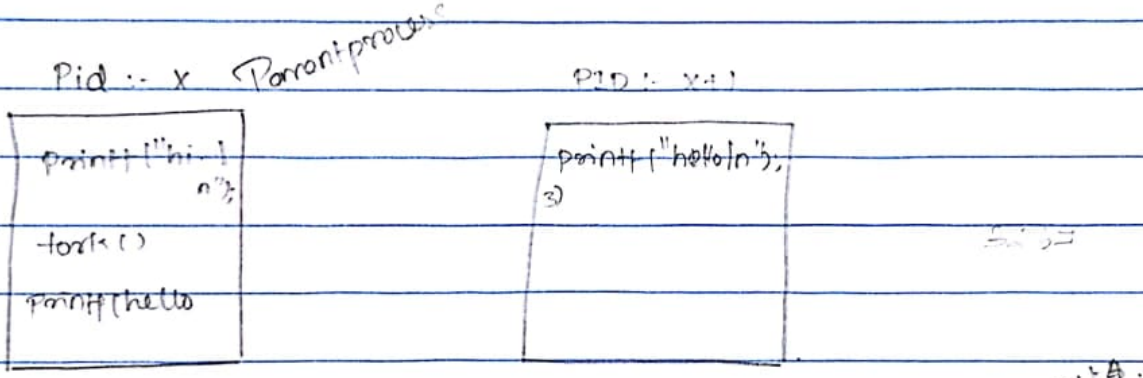
fork() Creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process, referred to as the parent.

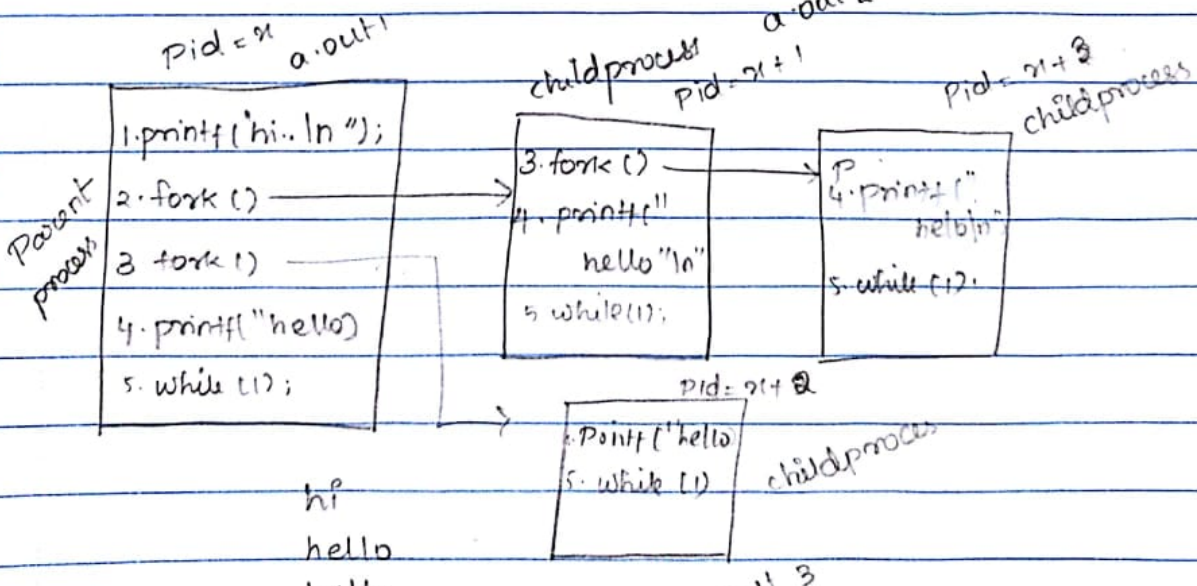
```
#include<stdio.h>
main()
{
printf("hi..\n");
fork();
printf("hello..\n");
}
```

O/P:

hi..
hello...
hello.

* Using fork when a process (child) is created it is exactly the duplicate of parent
* Duplicating all section of the parent process (code, data, heap, stack).





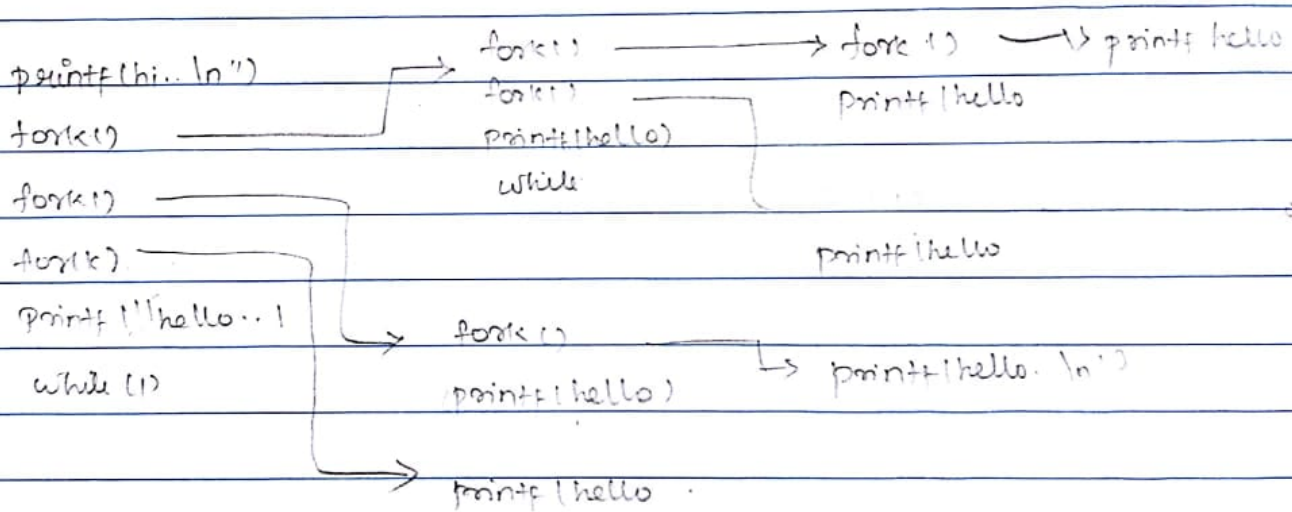
hi
hello
hello
hello
hello

```
# include <stdio.h>
main()
{
printf("hi..\n");
fork();
fork();
printf("hello..\n");
while(1);
}
```

O/p:-

hi
hello
hello
hello
hello .

Note:- New process created, the new process will run Concurrently with the parent process. Hence time updated for all the processes.



hi
hello
hello
hello
hello
hello
hello
hello
hello.

8 jobs e 1 time hi
8 time hello.

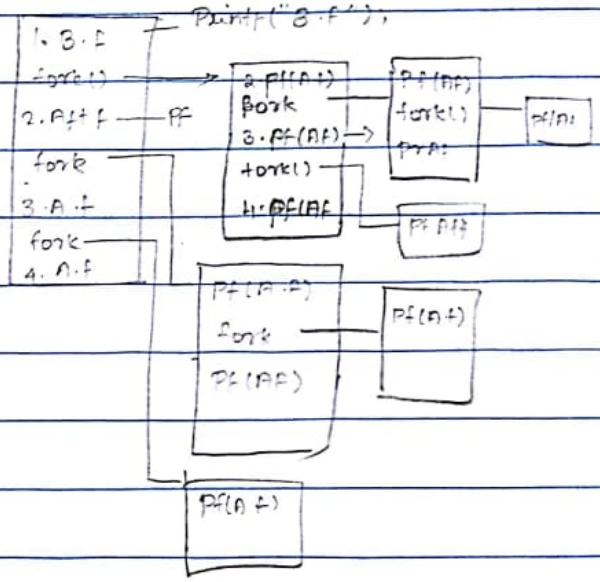
```
#include<stdio.h>
main()
{
printf("hi..\n");
fork();
fork();
fork();
printf("hello..\n");
while();
}
```

O/p :-

hi
hello
hello
hello
hello
hello
hello
hello
hello

3.8.17

```
main()
{
Printf("1. Before fork\n");
fork();
printf("2. After fork\n");
fork();
printf("3. After fork..\n");
fork();
Printf("4. After fork..\n");
}
```



O/p :-

Before fork
After fork
After fork
After fork
After fork
After fork
After fork
After fork

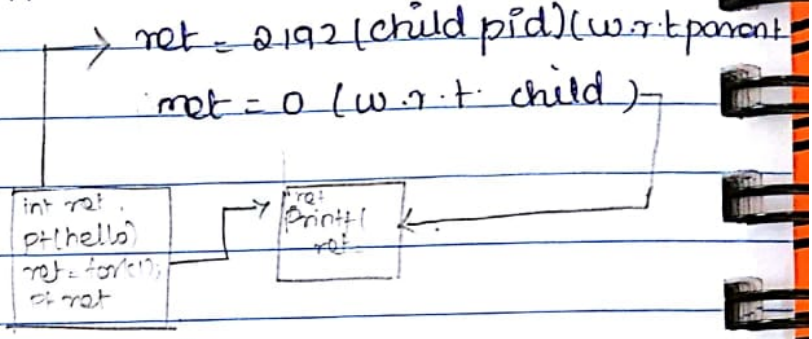
After fork
After fork
After fork
After fork
After fork
After fork
After fork

8 process

RETURN VALUE OF FORK:- On success, the PID of the child process is returned in the parent & 0 is returned in the parent child. On failure -1 is returned in the parent, no child process is created.

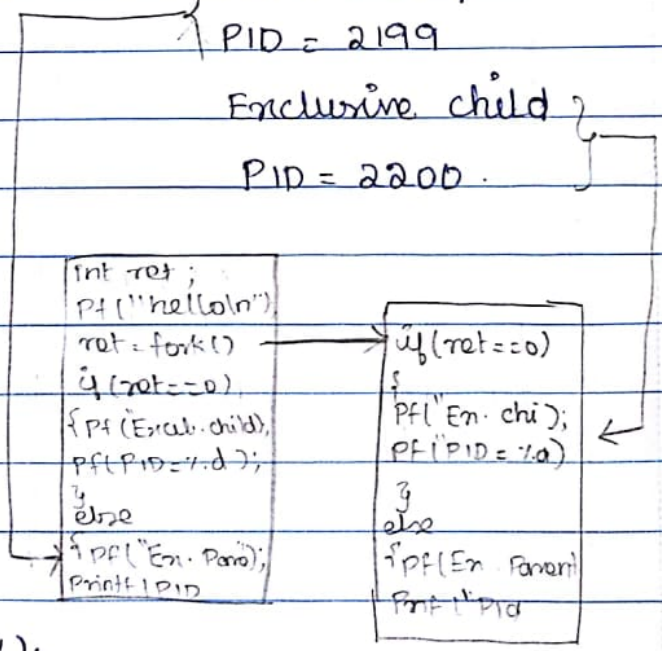
```
main()
{
int ret;
printf("hello\n");
ret=fork();
printf("ret=%d", ret);
}
```

o/p:- hello




```
main()
{
int ret;
printf("hello\n");
ret=fork();
if(ret==0)
{
printf("Exclusive child\n");
printf("PID=%d", getpid());
}
else
{
printf("Exclusive parent\n");
printf("PID=%d getpid()");
}
}
```

o/p:- hello



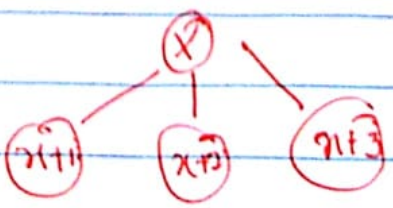
Note:- Using Return Value, we can divide the task to parent & child. "If" part is exclusively executed by child and "else" part is exclusively executed by parent.

```
main()
{
printf("hello\n");
if (fork()==0)
{
printf("exclusive child\n");
printf("child Pid=%d", getpid());
printf("child ppid=%d\n", getppid());
while(1);
}
else
{
printf("exclusive parent\n");
printf("Parent pid=%d\n", getpid());
printf("parent pid=%d\n", getpid());
while(1);
}
}
```

O/p:-

```
hello
exclusive parent
Parent Pid= 2209
Parent PPid= 1816 (bash)
exclusive child
Child Pid= 2210
Child PPid= 2209.
```

Wap to Create 3 child process from the same parent.



```
main()
{
if (fork==0)
{
printf("in child1..\n");
printf("Pid = %d\n", getpid());
printf("PPid = %d\n", getppid());
}
else
{
 if (fork()==0)
 {
 Printf("in child2 PID = %d PPID = %d\n", getpid(), getppid());
 }
 else
 {
 if (fork()==0)
 {
 printf("in child 3 PID = %d PPID = %d\n", getpid(), getppid());
 }
 else
 {
 while(1);
 }
 }
}
```

O/P:-

in child 1 Pid = 2229 PPid = 2228

in child 2 Pid = 2230 PPid = 2228

in child 3 Pid = 2231 PPid = 2228

What is orphan process?

An orphan process is a Computer process whose parent process has finished or terminated, though it remains running itself

What is Zombie process?

On Unix and Unix-like Computer operating systems, a zombie process or defunct process is a process that has completed but still has an entry in the process table. This entry is still needed to allow the pro-parent process to read its child's exit statles.

What is Cow?

Copy-on-write (cow), sometimes referred to as implicit sharing or shadowing, is a resource management technique used in Computer programing to efficiently implement a "duplicate" or "copy" operation on modifiable resources.

What is wait system call?

A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent continues its execution after wait system call instruction

4/09/17

Analysis Of \n :-

```
#include<stdio.h>
main()
{
printf("test msg.\n")
while(1);
}
```

o/p:-

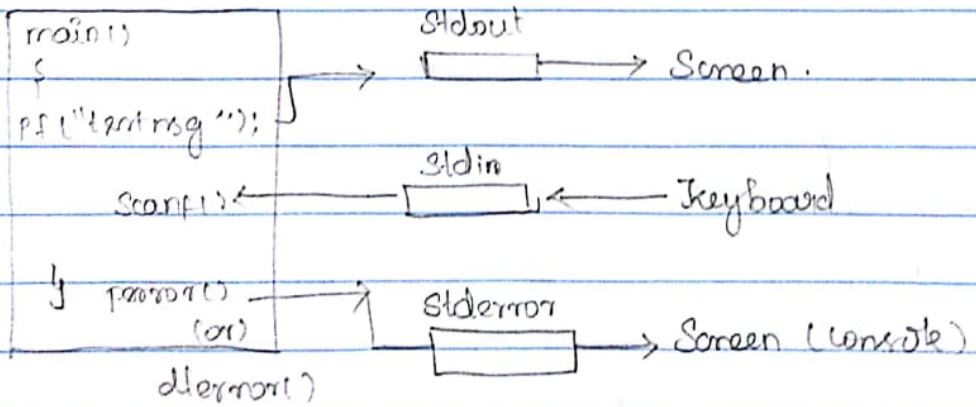
test msg...

```
#include<stdio.h>
main()
{
printf("test msg...");
while(1);
}
```

o/p:-

no o/p.

* When a process is created by default three buffers are attached to each process
* They are stdin, stdout, stderror.
  stdin connected to standard i/p (keyboard)
  stdout connected to standard o/p (screen)



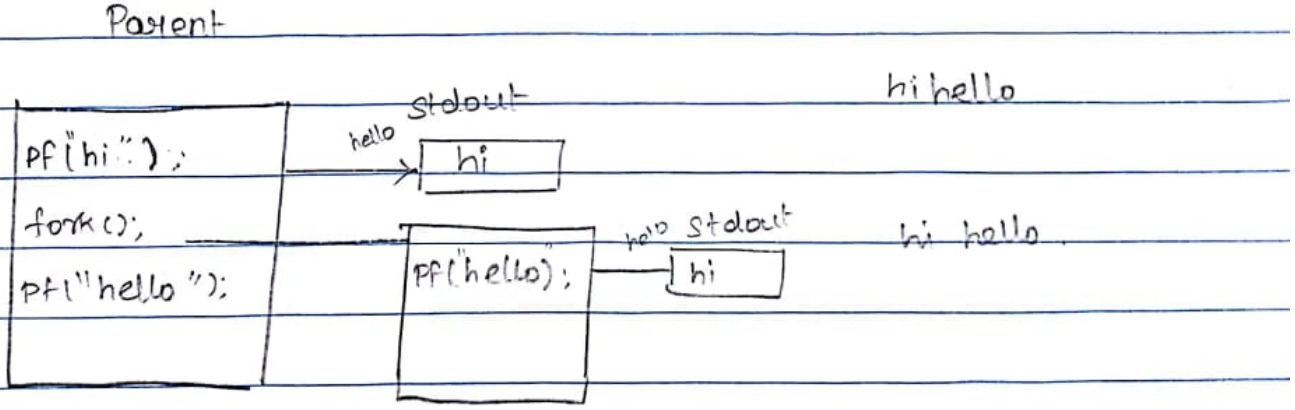
WHEN STDOUT BUFFER IS FLUSHED OUT ?.

① when process is terminated
② when there is a newline
③ when the buffer is full.

BUFFER SIZE :- 1KB.

```
#include <stdio.h>
main()
{
printf("hi... ");
fork();
printf("hello... ");
}
```

hi hello
command prompt
hi hello



Note:- when child process is created it will duplicate the parent buffer's content also

```
#include<stdio.h>
main()
{   char a[20], ch;
  printf(" Enter string... \n");
  scanf("%s%*c", a);
  printf(" Enter the character..\n");
  scanf("%c%*c", &ch);
  printf("a = %s", a);
  printf("ch = %c", ch);
}
```

O/p:-

a = abcd ch =

Command prompt.

"Enter" after the string is taken as newline So only ch = taken newline

```
main()
{
  char a[20], ch;
  printf("Enter the string..\n");
  scanf("%[^*]", a);
  printf(" Enter the character ...\n");
  scanf("%c", &ch);
  printf("a = %s", a);
  printf(" ch = %c", ch);
```

O/p:-

a = abcd ch = * Com.p. but "here" after the string there is '*' in the stdin buffer that is why ch taking * and is on the same line.

Note:-

If we give "_" in scanf, then the stdin buffer will be cleared.

```
#include<stdio.h>
#include<stdio_ext.h>
main
{
  char a[20], ch;
  printf(" Enter the String..\n");
```

```
scanf("%s", a);
__fpurge(stdin);
printf("Enter the character\n");
scanf("%c", &ch);
printf("a=%s", a);
printf("ch=%c\n", ch);
}
```

o/p :-
a=abcd ch=a

COPY-ON-WRITE :

A Variable which is common to both parent & child process is available in a location which is known by both the processes accessing that common variable only for reading purposes there is no problem, if any (parent/child) process modifying that common Variable data then a Separate page (copy) is created for that process called as copy-on-write

```
#include<stdio.h>
int i=10
main()
{
  if (fork() == 0)
  { printf("in child i=%d\n", i);
    i=200;
    printf("in child i=%d\n", i);
    sleep(1)
    printf("child exit...\n");
}
```

```
else
{
 sleep(5);
printf(" in parent .. \n");
printf (" i = %d\n", i);
}
```

O/p :-

in child = 10
in child i = 200
child exit
in parent ...
i = 10

Note :- "There is no Copy-on-write mechanism on code section, becoz code section is read only"

ORPHAN PROCESS :-

In parent and child relation, if parentprocess is completed first child becomes orphan (process not having parent).

Note :- For all Orphan proces "init" process will act as parent (PID of "init" is 1).

```
#include <stdio.h>
 main()
 {
    if (fork() == 0)
    {
    printf (" in child pid = %d  ppid = %d\n",
                        getpid(), getppid());
    printf (" child exit .. \n");
    }
```

```
else
{
  sleep(1);
printf("in parent pid=%d  ppid=%d", getpid(), getppid());
printf("parent exit..\n");

}
}
```

o/p :-

in child pid = x+1, ppid = x.
child exit
in parent pid = x, ppid = bash.
parent exit

```
#include<stdio.h>
main()
{
  if(fork()==0)
  {
   sleep(1); (for orphan process)
printf("in child pid=%d ppid=%d\n", getpid(), getppid());
printf("child exist..\n");
  }
else
{  sleep(10) → for zombie process (refer next page).
printf("in parent pid=%d ppid=%d\n", getpid(), getppid());
printf("parent exit\n");

}
}
```

o/p :-

in parent pid = x ; ppid = bash
parent exit
in child pid = x+1 ; ppid = 1
Child exit.

ZOMBIEE PROCESS:

In parent and child relation, if child process is completed first & parent is not collecting the status of child, then child becomes Zombiee.

Zombiee process also called as "Dead process" process dead but resources are not released.

<defunct> → means zombie, dead process.

```
#include<stdio.h>
main()
{
  if (fork()==0)
  {
   sleep(2);
   printf("in child pid=%d ppid=%d\n", getpid(), getppid());
   printf("child exit..\n");
  }
  else
  {
   sleep(10);
   printf("in parent pid=%d ppid=%d\n", getpid(), getppid());
   printf("parent exit..\n");
   while(1);
  }
```

o/p:-

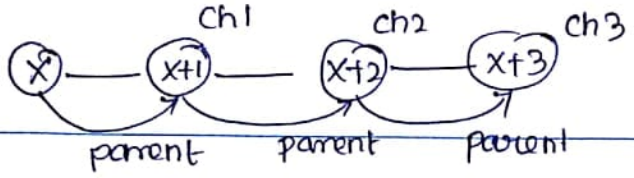
in child pid = x+1 ; ppid = x

Child exit

in parent pid = x ; ppid = bash

parent exit.

3/8/17 Wap to implement following.




```
main()
{
 if (fork()==0)
{
printf("Pid=%d  PPid=%d\n", getpid() ,getppid());

 if (fork()==0)
{
printf(" Pid=%d  Ppid=%d\n", getpid(), getppid());

 if (fork()==0)
{
printf(" Pid=%d  PPid=%d\n", getpid(), getppid());

else
while(1);
}
else
while(1);
}
else
while(1);
}
}
```

O/p:

Pid = 2240 PPid = 2239

Pid = 2241 PPid = 2240

Pid = 2242 PPid = 2241

SYSTEM VS FORK() :-

With Respect to System calls

```
main()
{
System (". /P1 ");
system ("Pwd");
}
```

P1.c

```
main()
{
Printf("in P1 process\n");
while(1);
}
```

Unable to execute because this follows Sequential execution.

With Respect to fork()

```
main()
{
if (fork==0)
System (". /P1 ");
else
System ("pwd");
}
```

fork() is used to achieve Concurrent process.

In parent & child relation, Parent jobs are running as fg & child jobs are running as bg

o/p:- home / class / V16CE9 / Linux
(w.r.t. S.C) command prompt : in P1 process

o/p:- home / class / V16CE9 / Linux
(w.r.t. F()). in P1 process

```
main()
{
if(fork==0)
system("pwd");
else
system("./p1");
}
```

o/p:- home/class/VI6CE9/linux
in P1 process.

```
main()
{
if(fork()==0)
{
printf("in child");
while(1);
}
else
{
printf("in process\n");
sleep(10);
printf("parent exit\n");
}
}
```

o/p:- in parent
in child
parent exit
command prompt

```
main
{ if(fork()==0)
 {
  printf("in child\n");
  sleep(10);
 printf("child exit\n");
 }
else
{
printf("in parent\n");
while(1);
}
```

o/p:- in parent
in child
child exit
no command prompt
(became parent job is fg).

Slide 17.

Generally PCB's are stored in kernel stack memory.

Command :- ulimit -u (to know number of process which can be created by user).

ulimit -a (to know all the limitations)

When fork() fails?

when a process is created PCB is created (stored in kernel stack). If there is no space to create a PCB than not possible to create the process.

Every OS is having certain limitations to handle maximum number of processes, once if it reaches to that limitation not possible to create the jobs.

RANDOM NUMBER GENERATION :-

Header file <stdlib.h>.

Prototype :- int rand (void);

void srand (unsigned int seed)

The rand() function returns a pseudo-random integer in the range 0 to RAND_MAX inclusive.

RAND_MAX :- 2147483647.

The srand() function sets its argument as the seed for a new sequence of pseudo-random integers to be returned by rand(). These sequences are repeatable by calling srand() with the same seed value.

If no seed value is provided, the srand() function is automatically seeded with a value of 1.

```
#include<stdio.h>
main()
{
srand(100);
int r;
r= rand();
printf("r= %d\n", r);
}
```

O/P:-

677741348

```
main()
{                    (getpid())
srand(100);
int r[5],i;
for(i=0; i<5; i++)
r[i]= rand(); // r[i]= rand()%10+1;
```

```
for(i=0; i<5; i++)
printf("r= %d\n",r);
}
```

O/p:-

5 random numbers

Note:- Same Sequence of numbers will be generated until we change the Seed value.

To change the Seed value for every execution we can use getpid(). As the pid value changes for every execution of a process.

```
#include <stdio.h>
main()
{
  if (fork==0)
  {
    int t1;
    Srand(getpid());
    t1 = rand()%10+1;
    printf("childprocess goes to delay of %d
                    Sec...\n", t1);
    Sleep(t1);
    printf("child exit..\n");
  }
  else
  {
    int t2;
    Srand(getpid());
    t2 = rand()%10+1;
    printf("Parent process goes to delay of %d
                    se..\n", t1);
    Sleep(t1);
    printf("Parent exit...\n");
  }
}
```

O/p:- Here the childprocess may be orphan or may be zombiee. If the parent gets completed 1st, child becomes "Orphan". If child gets Completed 1st, child becomes "Zombiee"

Note:- Child becoming orphan is no problem, but becoming zombiee creates a problem with Resources (Dead process not releasing the resources).

WAIT():-

Wait, waitpid - Wait for process to change state.

Header file:- <sys/types.h>
<sys/wait.h>

pid_t wait (int *status);
pid_t waitpid (pid_t pid, int *status, int options);

* All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed.

* A state change considered to be:- The child terminated; the child was stopped

by a signal ; or the child was resumed by a signal.

* In the case of terminated child, performing a wait allows the system to release the resources associated with the child ; if a wait is not performed, then the terminated child remains in a 'zombie" state.

wait & waitpid():-

The wait() system call suspends execution of the calling process until one of its children terminates, The call wait(&status) is equivalent to

waitpid(-1, &status, 0); ⇔ wait(&stat)

How to avoid child as zombie?:-

```
#include <stdio.h>
main()
{
  if (fork()==0)
  {
    printf("in child..\n");
     sleep(5);
    printf("child exit...\n");
     exit(0);
  }
  else
  {
    int stat;
```

wait(&stat):- Parent will block until child gets terminated and also collects the exit value.

wait(0):- Parent will block until child gets terminated (to collect the status) but the parent is not interested to collect the exit value.

wait (&stat); — till this both child & Parent will execute (concurrent)
printf ("stat = %d\n", stat); } only parent will execute.
while (1);
}

Note:- Writing wait in parent possible to collect the status of the child by parent (so child will not become as "Zombiee").

Limitation of Wait:-

* Writing wait in parent causes parent process gets blocked, because of this job con-currency is not possible to achieve.
* The above point is one of the limitation of wait system call.

Difference between Return & exit

```
#include <stdio.h>
main()
{
printf("hi..\n");
abc();
printf("hello..\n");
}
void abc(void)
{
printf("In abc\n");
return; // exit(0);
printf("After return\n");
}
```

O/p:-

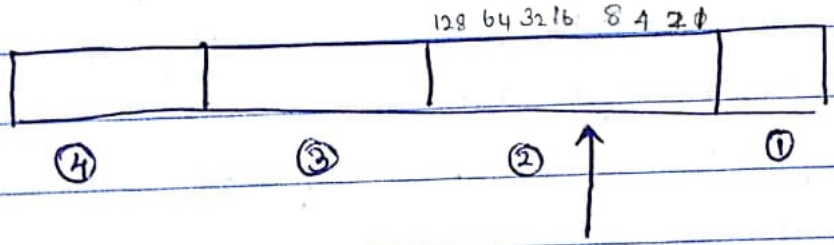
```
hi
in abc
hello
```

hi
in abc...

Note: When return is used, it will return to the function who ever called it, but the use of exit will terminate the process completely.

Exit value range (0 to 255)

" Exit value is stored in 8nd byte of 4 byte integer & it is stored in higher order byte of 2 byte integer "



Qs:- Difference between exit(0), exit(1) & exit(2)?

" " exit2 and _exit()

What wait System call returns?

7/08/17

Process termination
- Normal
  - Success
  - failure
- Abnormal
  - Because of Receiving signals.

```
if (fork == 0)
{
  if (workdone)
    exit(0)        } normal success
  else
    exit(1)        } normal failure.
}
wait (&status)
stat = stat >> 8
printf ("stat = %d\n", stat)
```

} To get the exact value of exit as (1).

Wap to create 3 child process from the same parent, use rand() functions to generate the delay between 1 to 10 Secs. Design a parent process in such a way that it should wait for all child process termination and display which process is completed in the order.

int a[3];

```
#include<stdio.h>
main()
{
    a[0]=
  if (fork()==0)
  {
  printf("in child process 1\n");
   Srand( getpid())
    t1 = rand() %10+1;
  printf("child1 goes to delay of %d sec", t1);
   Sleep(t1);
   exit(0); exit(0);
 }
else
{
    a[1]=
   if (fork()==0)
   {
    printf("in childprocess 2\n");
     Srand( getpid())
      t2 = rand() %10+1;
     printf("child2 goes to delay of %d sec", t2);
      Sleep(t2);
      exit(2); exit(0);
   }
   else
    {
        a[2]
      if (fork()==0)
      {
       printf(" in child process 3\n");
```

```
Srand (getpid())
t3 = rand() % 10+1;
printf (" child 3 goes to delay of %d sec", t3);
Sleep(t3);
exit(3); /exit(0);
}
else
{
int wstat, ret;
while ((ret = wait(&stat)) != -1) //(ret = wait(0)) != -1
{
    stat >>= 8; //
if (stat == 1)    if(ret == a[0])
printf ("child1 terminated");
if (stat == 2)    if (ret == a[1])
printf ("child2 terminated");
if (stat == 3)    if(ret == a[2])
printf ("child3 terminated");
}
}
}
}
```

RETURN VALUE OF WAIT:

wait(): on success, returns the process ID of the terminated child, on error -1 is returned.

On error, wait() returns -1, One possible error is that the calling process has no children, which is indicated by the errno value ECHILD.

wait() → exclusively in parent

wait exit() " " Child

What is the wait & waitpid()

What is the exec family of functions:

execl()

execlp()

What is the difference between fork & vfork().

$10/7/17

DIFFERENCE BETWEEN EXIT() & _EXIT().

exit()	_exit()
1. It is a library function	It is a System call.
2. Normal process termination	Here it is normal immediate process termination.
3. All the buffers / streams which are opened are closed properly/flushed out before termination of the process	All opened buffers/ streams are not flushed out, but process is terminated immediately.

```
#include <stdio.h>
main()
{
printf("hello..");
exit(0);
}
```

O/p:-

hello.

```
#include <stdio.h>
main()
{
printf("hello..");
_exit(0);
}
```

O/p:-

no o/p.

Note: exit() library function will clear the buffer (by flushing all the content) before communicating to the _exit() System call.

exit()	_exit()
all functions which are registered with atexit() and onexit() are called in the reverse order of their registration	Functions which are registered with at exit (or) on_exit() are not called if _exit is used.

Atexit(): Register a function to be called at normal process termination.

Header file #include <stdlib.h>

int atexit (void (*function) (void));

Functions so registered are called in the reverse order of their registration, no arguments are passed.

The same function may be registered multiple times: it is called as once for each registration.

```
#include<stdio.h>
#include<stdlib.h>
void fun1();
void fun2();
main() {
printf ("before function\n");
atexit (fun1);
printf (" After fun1 \n");
atexit (fun2); printf(" After fun2 \n");
printf (" Main function exit ..\n");
exit(0); / _ exit(0);
}
void fun1()
{
```

O/p:- _exit(0)

```
before function
After fun1
After fun2
main function
exit
```

```
printf("in fun1 body...\n");
}
void fun2()
{
printf("in fun2 body...\n");
}
```

o/p:-

before main function
after fun1
after fun2
Main function exit
in fun2 body
in fun1 body

wait pid().

wait()	waitpid()
wait will take 1 argument	waitpid() will take 3 argument.
not possible to wait for a specific child	It is possible to wait for a specific child status collection.
By default wait will collect only terminated child status	waitpid() also by default will collect the terminated child status, but it is possible to change using waitpid 3rd arg (options).

Waitpid() 3rd Argument :-

The Value of options is an OR of zero or more than the following constants.

* WNOHANG - Return immediately if no exal child has exited

* WUNTRACED - Also return if a child has stopped

* WICONTINUED - Since (linux 2.6.10) also return if a stopped child has been resumed by delivery of SIGCONT.

```
#include <stdio.h>
#include <sys/wait.h>
main()
{
  if (fork() == 0)
  {
  printf ("in child...\n");
   sleep(5);
   printf ("child exit...\n");
   exit(0);
  }
  else {
  int stat;
  waitpid(-1, &stat, WNOHANG);
  printf("stat= %d\n", stat);
  } }
```

Note: Parent procus is not blocked, so job Concurrency is achieved (parent & child running parallely).

* when parent is executing waitpid() at that time if child terminate, parent will collect status, if not terminated at that time, parent will not wait, so child may be orphan (or) Zombiee.

```
#include <stdio.h>
#include <sys/wait.h>
main()
{
if(fork()==0)
{
printf("in child | pid = %d \n", getpid());
sleep(10);
printf(" child exit.. \n");
exit(0);
}
else
{
int stat;
waitpid (-1, &stat, WUNTRACED); // WCONTINUED.
printf(" Stat = %d \n", stat);
}}
```

O/p:- in child pid = x
Stat = 4991./
Suspend it by using another terminal
kill -19 pid

(or) O/p:- in child pid = x
Stat = 65535 /
Another terminal.
Suspend it by using
kill -19 pid
Resume it kill -18 pid

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
main()
{
if (fork()==0)
{
printf("in child PID= %d\n", getpid());
sleep(20);
printf(" Child exit \n");
exit(0);
}
else
{
int stat;
wait(-1, &stat, WCONTINUED)
printf(" in status= %d\n", stat);
}
```

O/p :-

in child Pid = x.

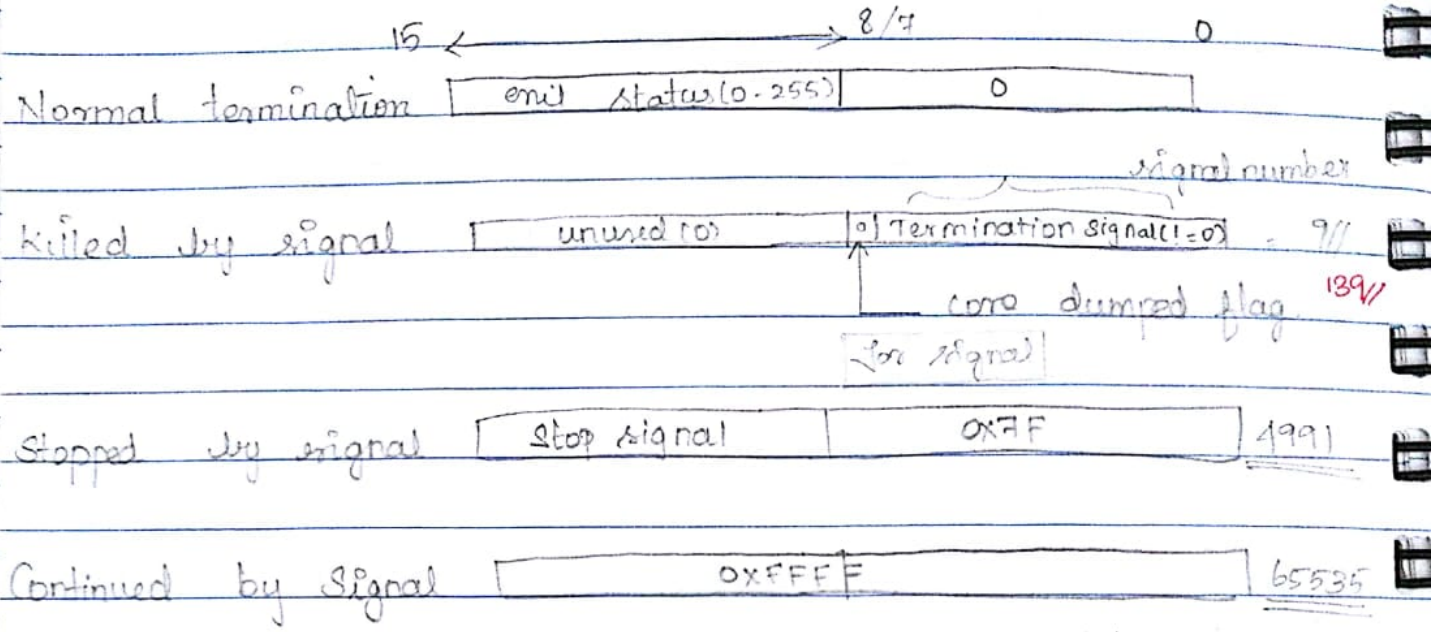
Status = 9.

terminate the process using kill -9 Pid.

fork()	Vfork()
child process and parent process has separate address space.	Child process & parent process shares the same address space
Parent and child process execute simultaneously.	Parent process remains suspended till the child completes its execution
If the childprocess alters any page in the address space, it is invisible to the parent process as the address space are Separate	If child process alters any page in the address space, it is invisible to the parent process as they share the same address space.
fork() uses Copy-on-write as an alternative where the parent & the child share Same pages until any one of them modifies the shared page	Vfork() does not use Copy-on-write.

9/08/17.

VALUE RETURNED IN THE STATUS ARGUMENT OF WAIT() AND WAITPID():



Normal termination → becoz of exit

Abnormal termination → becoz of receiving some signal

The signal number '9' is not having the capability to generate "Core dumped" file. that is why its value is '0'.

Signal no. '11' → Segmentation fault (core dumped)

SIGSEGV

EXEC FAMILY OF FUNCTIONS :-

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg,
           ..., char * const envp[]);
int execv(const char *path, char *const argv[])
int execvp(const char *file, char *const argv[]);

int execvpe(const char *file, char *const argv[],
            char * const envp[]);
```

All these exec family of functions are communicating with a system call execve

The execl family of functions replaces the current process image with a new process image.

The initial argument for these functions is the name of a file that is to be executed

ls command path ⇒ /bin/ls

long list of the data in pwd } ⇒ ls -l ; ls -li (number something).

ps -el ⇒ (all jobs under the terminal with long list)

```
#include <stdio.h>
main
{
printf("hi..\n");
System("ls");
printf("hello..\n");
}
```

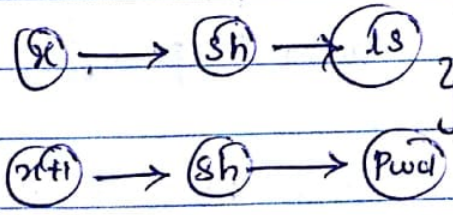
o/p:- hi
ls command gets executed
hello.

```
#include <stdio.h>
main
{
printf("hi..\n");
execl ("/bin/ls","ls",NULL);
printf("hello..\n");
}
```

o/p:- hi
ls command gets executed.

Note:- The current process "a.out" process image is replaced by "ls" process image hence 'hello' is not printed.

```
#include<stdio.h>
main()
{
if (fork()==0)
{
System("ls");
while(1);
}
else
{
System("pwd");
while(1);
}
```



```
#include <stdio.h>
main()
{
if (fork() == 0)
{
execl ("/bin/ls", "ls", NULL);
while (1);
}
else
{
execl ("/bin/pwd", "pwd", NULL);
while (1);
}
```

(2) pwd
(1) ls
} 2 jobs.

Note: exec family of functions are useful for minimizing the number of process creation

```
#include <stdio.h>
main ()
{
printf ("PID = %d  PPID = %d \n", getpid(); getppid());
execl ("./p1", "p1", NULL);
printf ("after exec... \n");
}
```

o/p

PID = x ; PPID = bash

PID = x ; PPID = bash

p1

```
#include <stdio.h>
main ()
{ printf ("PID = %d PPID = %d \n", getpid(), getppid());
}
```

```
#include <stdio.h>
main()
{
printf(" hi .. \n");
execl (" /bin/ cal ", " cal ", NULL);
perror( execl );
printf(" hello .. \n");
}
```

O/p :-

hi ...

bash: No such file or directory

hello ...

NOTE: In case of execl we should compulsorily provide the path where the file is exactly located, if not the call gets failure Hence no replacement of process image takes place.

```
#include <stdio.h>
main()
{
printf ("hi .. \n");
execlp (" cal ", "cal", NULL);
perror ("execlp");
printf ("hello .. \n");
}
```

O/p :-

hi

cal command gets executed

Note: In case of "execl", that function will search only in the provided path whereas "execlp" will search in all the path for that command.

perror - print a system error message.

Header file #include <stdio.h>, #include<errno.h>.

void perror(const char *s);

The routine perror() produces a message on the standard error o/p, describing the last error encountered during a call to a system or library function.

10/8/17

Command :- #echo $ path.
↳ shell variable

all shell variables are in capital letters, path is one of the shell variable.

cal command :- /usr/bin/cal.

In shell scripting language echo is like printf and path is like variable.

Command :- env will display all the shell Variables or Environmental Variables under our working terminal.

* Shell designers are providing some variables
* All shell variables are in Capital letters.

Main function third argument can be used to know under which environment, and access the shell variables.

To print all the shell variables.

```
#include <stdio.h>
main( int argc, char * argv[], char *env[])
 {  int i;
   for(i=0; env[i]; i++)
    printf("%s\n", env[i]);
    sleep(1);
 }
```

```
# include<stdio.h>
main(int argc, char *argv[], char* env[])
{
 int i;
 for(i=0; argv[i] ;i++)
   printf("%s\n", env[i]);
   sleep(1);
}
```

Note:- argv[argc] = '\0'.

Every process inherit the properties of the shell under which it is working. & every process is aware of the shell / environment under which it is working / running.

```
#include<stdio.h>
main()
{
char * ap[] = {"ls", "-l", NULL};
printf ("Hello...\n");
execv ("/bin/ls", ap);
printf ("hi..\n");
}
```

o/p:- hello
ls command gets executed

```
#include <stdio.h>
main()
{
char * ap[] = {"ls", "-l", NULL}
printf ("Hello.. \n");
execvp ("ls", ap);
printf ("hi..\n");
}
```

o/p:- Same o/p.

To know whether the current process & the new replaced process is running under the same environment or not:-

```
#include <stdio.h>
main(int argc, char *argv[], char * env[]);
{ int i;
  for
  printf(" Before Image Replacement \n");
  for( i=0; env[i]; i++)
  { printf("%s\n", env[i]);
    Sleep(1);
  }
printf(" After Image Replacement \n");
 execlp("p1" "p1" NULL);
}
```

O/p:-

Before Image Replacement
display all shell Variables
After Image Replacement
display all shell Variables (same)

P1 process

```
#include <stdio.h>
main(int argc, char * argv[], char * env[]);
{
  int i;
  for(i=0; env[i]; i++)
  {
    printf("%s\n", env[i]);
    sleep(1);
  }
}
```

Note:

* When a process is executing it knows that under which environment it is executing and by default all shell variables are inherited into that process.
* If the running process image is replacing with a new process image, that new process image also will run under same environment.

```
#include <stdio.h>
main( int argc, char * argv[], char * env[])
{
 int i ;
 char * en[] = {"USER = xxxx", "HOME = /home/yyyy", NULL};
 printf (" Before Replacement \n");
 for(i=0; env[i] ; i++)
 {
  printf(" %s\n", env[i]);
  sleep(1);
 }
 printf (" After Replacement \n");
 execle(" ./p1", "p1", NULL, en);
}
```

P1 - process (same)

o/p:-

Before Image Replacement display all Shell variables

After Image Replacement changed Shell variable

USER = xxxx

HOME = /home/yyyy.

Vfork() - Create a child process & block parent.

Header file :- <sys/types.h>
<unistd.h>

prototype :- pid_t vfork(void);

Vfork() behaviour is undefined in the following cases :-

i) Modifies any data other than a variable of type Pid_t.

ii) Returns from the function in which vfork was called

(iii) If there is no exit / _exit.

Limitations for Vfork() :-

* No job currency as parent gets blocked
* No concept of copy-on-write
* Modification of child will affect parent and vice-versa.

```
#include <stdio.h>
main()
{
printf("hello..\n");
fork();
printf("hi...\n");
}
```

o/p:-
hello..
hi..
hi..

```
#include <stdio.h>
main()
{
printf("hello..\n");
fork();
printf("hi...\n");
exit(0)
}
```

o/p:-
hello..
hi...
hi...
a.out (core dumped)
(o/p undefined if there is no exit()/_exit)

```
#include <stdio.h>
main()
{
if(vfork()==0)
{
printf("in child..\n");
sleep(10);
printf("child exit...\n");
exit(0);
}
else
printf("in parent..\n");
}
```

o/p:-
hello..
hi..
hi..

o/p:-
in child
child exit
in parent.
(refer limitations of vfork()).

```
#include <stdio.h> int i = 1;
main()
{
printf (" hello.. \n");
fork();
i++;
printf ("i = %d\n", i);
printf ("pid = %d\n", getpid());
}
```

O/P:- i = 2 PID = 2376
i = 2 PID = 2377.

```
#include <stdio.h>
int i = 1;
main()
{
printf ("hello..\n");
vfork();
i++;
printf ("i = %d  pid = %d\n", i, getpid());
}
```

O/P:-
i = 2 PID = 2391
i = 3 PID = 2390.

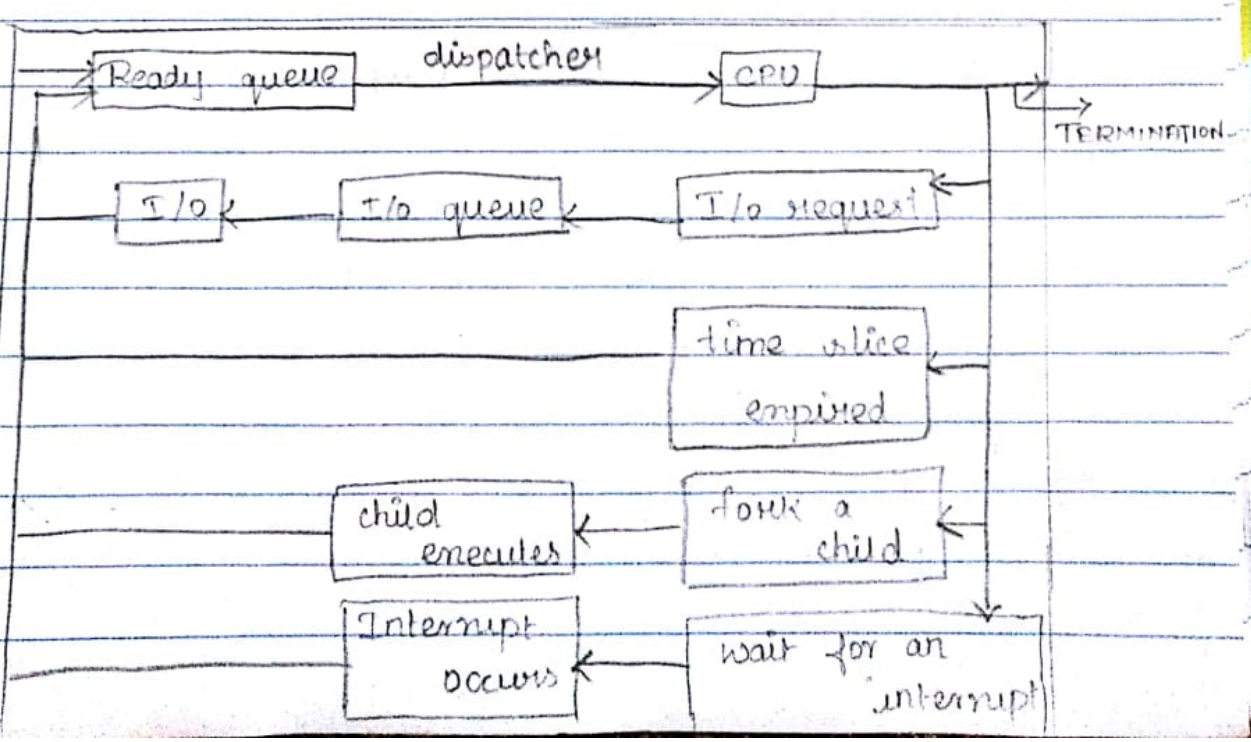
11/08/17

Vfork() is expressly designed to be used in programs where the child forms an immediate exec() call.

Two features distinguish the vfork() system call from fork() and make it more efficient.

* No duplication of virtual memory pages or page tables is done for the child process. Instead, the child shares the parent's memory until it either performs a successful exec() or calls _exit() to terminate.
* Execution of the parent process is suspended until the child has performed an exec() or _exit().

PROCESS & CPU SCHEDULING:



Heart of the OS: Scheduler

Signals are called as software interrupts.

CONTEXT SWITCHING

* Save the state of the old process.
* Load the saved state for the new process
* Content - Switch time is Overhead, the system does no useful work which switching

CPU Scheduler:-

A CPU scheduler, running in the dispatcher, is responsible for selecting of the next running process.
. Based on the particular strategy.

When does CPU Scheduling happen?

- A process switches from the running state to waiting state (e.x I/O request)
- A process switches from the running state to the ready state
- A process switches from waiting state to ready state (completion of an I/O operation)
- A process terminates.

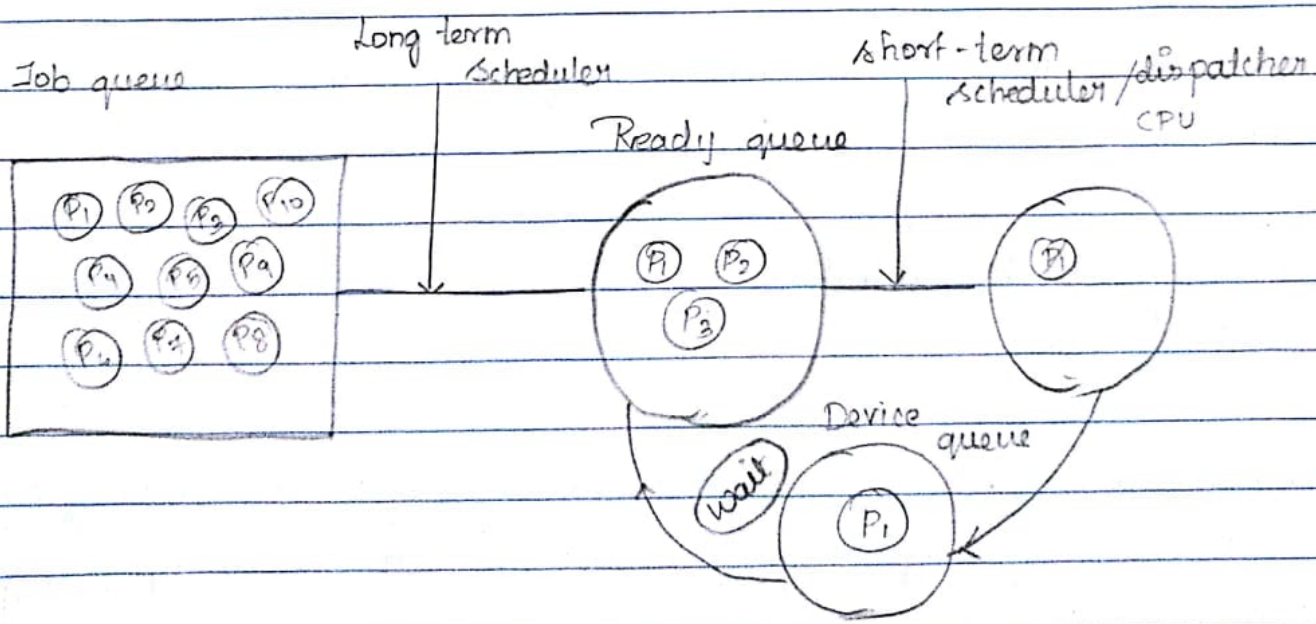
Scheduling queues.

CPU Schedulers use various queues in the scheduling process.

* Job queues : Consist of all processes
  . All jobs (processes), once submitted, are in the Job queue.

Ready queue :-

* All processes that are ready and waiting for execution are in the ready queue.
* Usually, a long-term scheduler / job scheduler selects processes from the job queue to the ready queue.
* CPU Scheduler / short term Scheduler selects a process from the ready queue for execution.



Device Queue :-

* When a process is blocked in an I/O operation, It is usually put in device queue (waiting for the device)
* When the I/O operation is Completed, the process is moved from the device queue to the ready queue.

Note :- If the time slice got completed, those processes are shifted from CPU to ready queue. Whereas suspended process usually enters in the device queue after getting resumed only enters into the ready queue.

Performance metrices for CPU Scheduling :-

CPU utilization :- percentage of the time that CPU is busy.

Throughput :- The number of processes Completed per unit time.

Turn around time :- The interval from the time of submission of a process to the time of Completion.

Wait time :- The sum of the periods spent waiting in the ready queue.

Response time :- The time of submission to the time the first response is produced.

In microcontroller :- CPU waiting is called polling (continuously monitoring the pins).

Other performance metrics :-

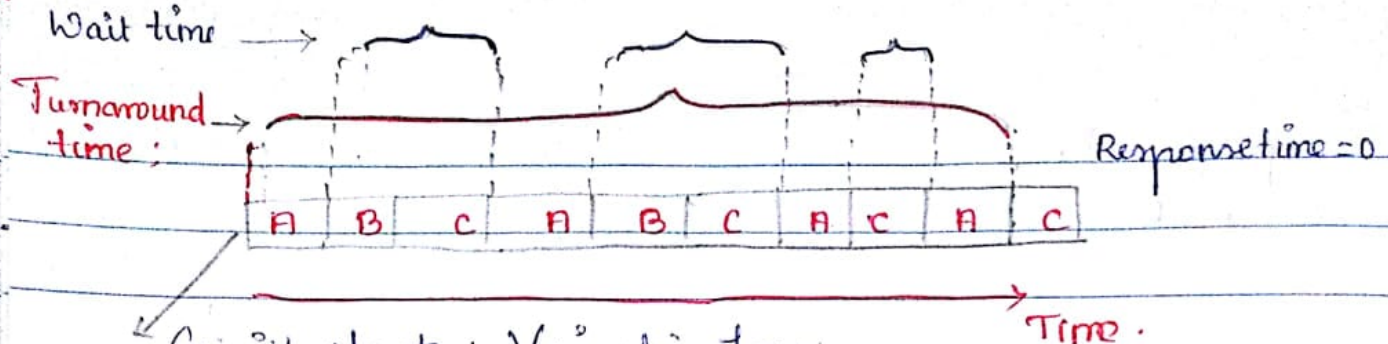
* Fairness :- It is important, but harder to define quantitatively.

Goal :

* Maximum CPU utilization, Throughput & Fairness
* Minimize turnaround time, waiting time & Response time

DETERMINISTIC MODELING EXAMPLE :

* Suppose we have processes A, B and C submitted at time 0.
* We want to know the response time, waiting time, and turn around time of process B.



Ganitt chart : Visualise how processes execute.

For B.

R.T = 1 units

Wait time = 3 units

T.A.T = 5 units

For C

R.T = 2 units

W.T = 6 units

T.A.T = 10 units.

For A.

R.T = 0

wait time = 5 units

T.A.T = 9 units.

Preemptive Versus Non-preemptive Scheduling :-

Preemptive : Schedule a new process even when the current process does not intend to give up the CPU.

Non-Preemptive :- Only Schedule a new process when the current one does not want CPU any more.

Scheduling policies:-

* FIFO (first in first out) (First come, first Served)
* Round Robin
* SJF (Shortest job first)
* Priority scheduling
* Multilevel feedback queues (RoundRobin + priority).
* Many more ...

Generally lowest no is having the priority in linux (0 to 99)

12/08/17 FIFO:-

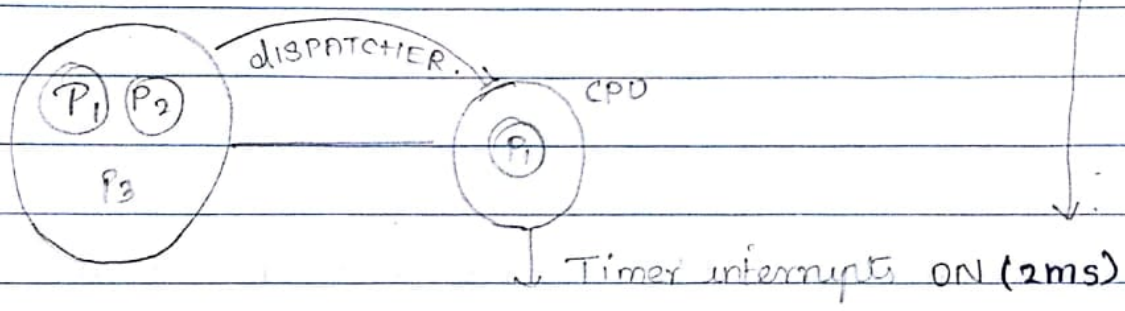
FIFO: assigns the CPU based on the orders of requests

* Non-Preemptive: A process keeps running on a cpu until it is blocked or terminated.
* Also known as FCFS (First come, first Serve)
* Simple
  - Short jobs can get stuck behind long jobs
  - Turnaround time is not ideal.

ROUND ROBIN :-

Round Robin (RR) periodically releases the CPU from long-running jobs.

* Based on the timer interrupts so short jobs can get a fair share of CPU time.
* Pre-emptive :- a process can be forced to leave its running state and replaced by another running process.
* Time slice :- Interval between timer interrupts.

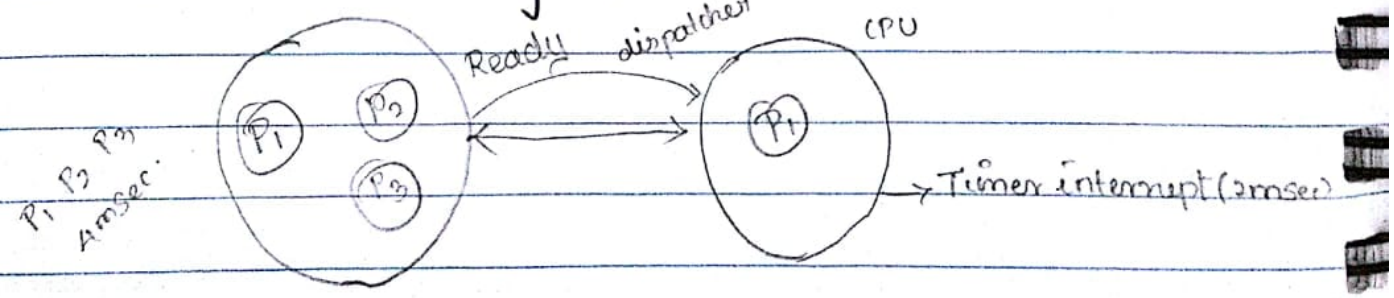


Majority of systems following Round Robin policy sets minimum time as 2ms.

More on Round Robin.

* If time slice is too long
  Scheduling degrades to FIFO.
* If time slice is too short
  Throughput Suffers
  Content switching cost dominates.

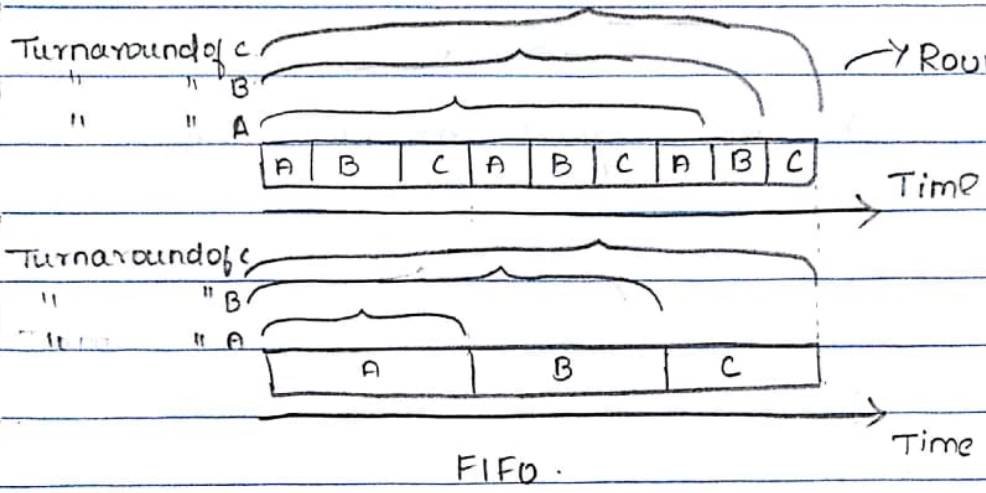
P1 P2 P3
2m 2m 2m
① - context ① - context ① - context



Note:- CPU is fair to all jobs in case of RoundRobin whereas FIFO is not.

FIFO VS RoundRobin

Suppose we have three jobs of equal length



FIFO.

Round Robin :-

* Shorter response time
* Fair Sharing of CPU.

More precisely, not good in terms of turnaround time.

SHORTEST JOB FIRST (SJF):-

* SJF runs whatever job puts the least demand on the CPU, also known as Shortest time to Completion first

* Great for short jobs
* Small degradation of long jobs

Drawbacks of Shortest job first :-

Starvation : Constant arrivals of short jobs can keep long ones from running.

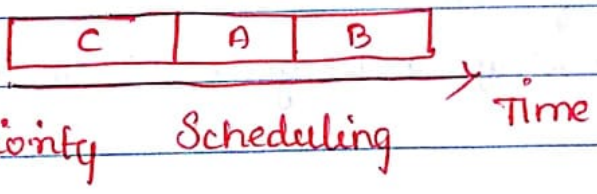
* There is no way to know the completion time of jobs (most of the time).

PRIORITY SCHEDULING :-

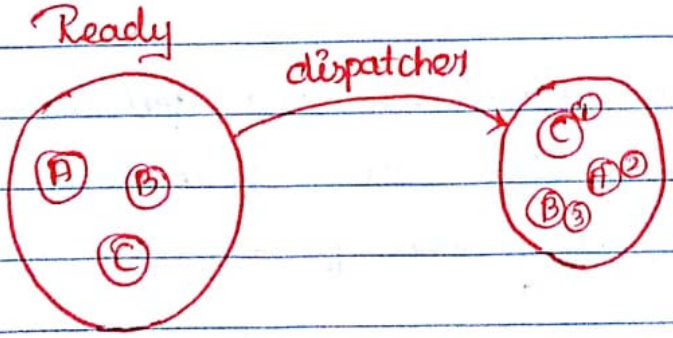
* The process with the highest priority runs first

Priority 0 :- C
Priority 1 :- B A
Priority 2 :- D

* Assume that low numbers represent high priority



priority Scheduling



Priority Scheduling follows the pre-emption technique.

Note: The process with lowest priority number is executing, at that time when a highest priority number (job) is ready, then that process is pre-empted and the highest priority job is executed but vice versa does not takes place.

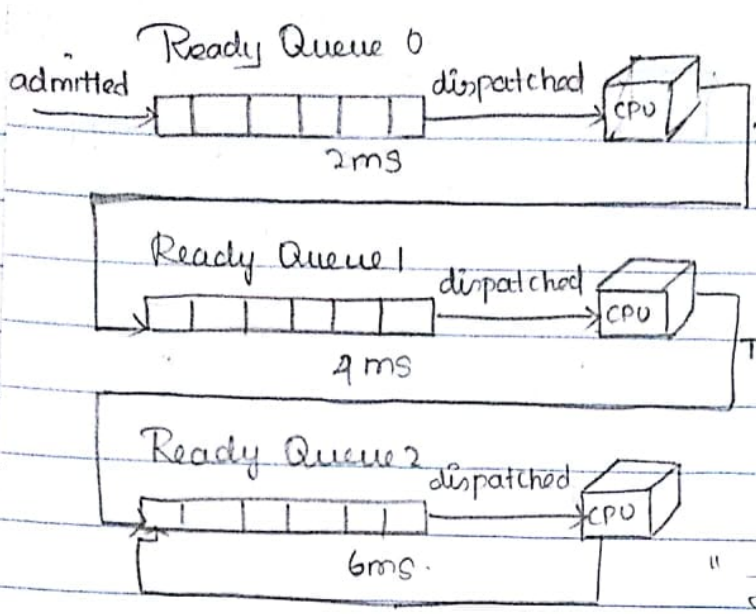
MULTILEVEL FEEDBACK QUEUES:

* Multi-level feedback queues:- Use multiple queues with different priorities

* Round Robin at each priority level
* Run highest priority jobs first
* Once those finish, run next highest priority, etc.
* Jobs start in the highest priority queue.
* If time slice expires, drop the job by one level.
* If time slice does not expire, push the job up by one level.

Commonly used priority policy is Multi level feedback Queues.

MULTILEVEL FEEDBACK QUEUE SCHEDULING



Process in RQ1 are Scheduled only when no Process exists in RQ0

Process is RQ2 are Scheduled only when no process exists in RQ1

"Longer Processes Gradually drift downward"

14/08/2017

NICE COMMAND & RENICE COMMAND.

NICE: We can change the priority of the process when it is started & not the running process.

RENICE: We can change the priority of running process.

Priority No is from -99
Default priority no assigned is 80.

PRI - Priority of Process
NI - Niceness Value by default 0.

Note: In GPOS the priority levels are 0-99 by default once a process is ready default priority no is 80

Run a program with modified Scheduling priority

Niceness range from -20 to +19

-20 (most favourable Scheduling) $\begin{array}{r} 80 \\ -20 \\ \hline 60 \end{array}$

low priority value, Hence given high priority

+19 (least favourable) $\begin{array}{r} 80 \\ +19 \\ \hline 99 \end{array}$.

High value, Hence given less priority

Command to change the priority value.

Nice -(5) ./a.out 2 (85 PRI)

↓
the PS

Nice -[-5] ./a out 2 (75 PRI).

↓
-ve

RENICE: Alter the priority of Running Process

Command: renice 5 pid renice -5 pid

↓ ↓
niceness value +ve niceness value -ve

SIGNAL HANDLING

Signal:

Signal is a notification to a process that an event has occured.

→ Signals are called as Software interrupts.

→ When a process will receive a signal is unknown.

→ A process may receive a signal in its life time, by other process (or) from Hardware

```
#include <stdio.h>
main()
{
int *p = 0
printf ("%d\n", *p);
}
```

Process is compiled Compiler will only check the Syntax & not logic

O/p:- Segmentation fault

Process is terminated because of accessing un-authorized memory. It is terminated by the signal [SIGSEGV => 11]

```
#include <stdio.h>
main()
{
while(1);
printf("hello\n");
}
```

O/p:-

hello..

hello.

:

:

Ctrl+c → Process terminated

Ctrl+c :- The process will receive the signal SIGINT and process is terminated

SIGINT - 2.

Command :- kill -l

All 64 signals will be displayed. User can use maximum of 32 signals.

```
#include <stdio.h>
main()
{
printf("%d\n", 10/0);
}
```

O/P :-
Floating point exception
[process is terminated by SIGNAL]
SIGFPE → 8.

A process will receive the signal from H/w (or) possible to receive from another process by using kill (command).

SIGCONT ⇒ 18 (Resume the Suspended process)
SIGSTOP ⇒ 19 (Suspend the process)
SIGKILL ⇒ 9 (Kill the process)

Write a program to use kill command.

Kill - send signal to process to implement kill command, we can use kill function.

int kill (Pid_t pid, int sig);

kill on success 0 returns
" " Error -1 returns

```
#include <stdio.h>
main(int argc, char * argv[])
{
if (argc != 3)
{
printf (" ./a.out Signal no PID \n");
return;
}
if (kill (atoi(argv[2]), atoi(argv[1])) < 0)
Perror ("kill");
printf (" no %d signal send to process %d \n",
        atoi(argv[1]), atoi(argv[2]));
}
```

O/p :- ./a.out Signal no PID

which signal want to send to which process.

Only 2 types of signals can be generated by keyboard (i) SIGINT (ctrl+C) → (2)
(ii) SIGQUIT (ctrl+\) → (3)

Raise :- Send to a signal to a caller. 16/08/17

int raise (int sig);

↳ Signal No.

The raise function sends a signal to the calling process or threads.

```
#include <stdio.h>
main()
{
printf("hello...\n");
raise(11);
printf("hi...\n");
while(1);
}
```

O/P :-
hello...
Segmentation fault.

kill(-11 Pid_t pid ⇔ raise(11)

kill(Pid_t pid 11)

Pause :- Wait for signal.

int pause(void);

Pause() causes the calling process to sleep until a signal is delivered that either terminates the process (or) causes the invocation of a signal catching function.

Return Value:- Pause() returns when a signal was caught and the signal catching function returned. In this case, pause() returns -1.

```
#include <stdio.h>
main()
{
printf("hello..\n");
pause();
printf("bye...\n");
printf("hello..\n");
}
```

o/p :-

```
hello
ctrl+c
process terminated.
```

SIGNAL : ANSI C handling signal (Not for generating signal, used for changing action of signal)

```
typedef void (*Sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler)
                                  (function pointer)
```

Note: Signals are asynchronous events
Signals are not generated
Used for changing the action of signal

signal() sets the disposition of the signal signum to handler, which is either SIG_IGN, SIG_DFL, or the address of a programmer-defined function (a "signal handler").

Case (i):-

```
#include <stdio.h>
void isr (int n)
{
printf ("in isr ..\n");
}
main()
{
printf ("hello ..\n");
signal(2, isr);
printf ("after signal..\n");
while(1);
}
```

o/p:-

hello
after signal
When pressing ctrl+c
we get
in isr ..
in isr ..
Process is not terminated because signal no: 2 is modified to isr.

* when a process is created PCB created
* Similarly when process created a table created called as signal table

Sig No	Action
1	SIG_DFL
2	SIG_DFL
3	SIG_DFL

SIG_DFL - Default Action
SIG_IGN - Ignore the signal
isr - user defined isr.

Case (ii):

```
#include <stdio.h>
#include <signal.h>
void isr (int n)
{
printf ("in isr ..\n");
printf ("n=%d\n",n);
signal (n, SIG_DFL);
printf ("isr exit\n");
main()
{
printf ("hello..\n");
signal (2,isr);
printf ("after signal...\n");
while (1);
}
```

O/p:-

```
hello
after signal
```

Signal 2 is sending by (ctrl+c)

```
In isr
n=2
isr exit
```

On second time sending the signal 2, its default action (termination) happens

Case (iii):-

```
# include <stdio.h>
# include <signal.h>
void isr (int n)
{
printf ("in isr ..\n");
```

```
printf ("n= %d\n", n);
signal (n, SIG_IGN);   signal is Ignored
printf (" isr exit..\n");
}
main ()
{
printf ("hello..\n");
Signal (2, isr);
printf (" after signal..\n");
while (1);
}
```

O/P:-

```
hello...
After Signal.
ctrl+c.
in isr...
n=2
isr exit..
no response for
    ctrl+c..
```

Case (iv):-

```
#include <stdio.h>
# include <signal.h
void isr (int n)
{
printf (" in isr...\n");
printf (" n= %d\n", n);
signal (n, SIG_DFL);
printf (" isr exit..\n");
}
main ()
{
printf (" hello..\n");
Signal (2, isr);
```

```
Signal (S, isr);
printf ("after signal...\n");
while(1);
}
```

o/p:-

```
hello
After signal
```

ctrl+\ in isr

```
n=3
```

ctrl+c in isr

```
n=2.
```

/(Each signal will execute isr only once not more than 1)

Case (v):-

```
#include <stdio.h>
#include <signal.h>
void isr (int n)
{
printf ("in isr..\n");
}
main()
{
int ret;
signal(2, isr);
printf("hello..\n");
ret = pause();
printf("ret = %d\n", ret);
printf ("after pause...\n");
```

printf("bye..\n");

O/p:-

hello...

ctrl+c

in in

ret = -1

after pause

bye.

ALARM :- SIGALRM (+14)

Alarm - set an alarm clock for delivery of a signal

unsigned int alarm(unsigned int seconds);

Alarm() arranges for a SIGALRM signal to be delivered to the calling process in seconds.

* If seconds is zero, no new alarm() is scheduled
* In any event any previously set alarm() is cancelled.

```
#include <stdio.h>
main()
{
printf ("hello..\n");
printf ("hi..\n");
alarm(10);
printf ("bye..\n");
while(1);
}
```

o/p:-

hello
hi
bye
process terminated after 10 seconds.

MULTIPLE ALARMS: (WILL NOT BLOCK THE PROCESS)

```
#include <stdio.h>
main()
{
printf("hello..\n");
alarm(10);
alarm(5);
alarm(1);
printf("hi..\n");
while(1);
}
```

o/p:-

hello
hi
Alarm clock (1) sec.

Note: The last alarm is replaced with the first alarm value. For one process it is not possible to set multiple alarms for a process. Even if we are writing more than once, the last one only will be considered.

Alarm(5) belongs to periodic interrupt.

```
#include <stdio.h>
#include <signal.h>
void isr(int n)
{
printf("in isr..\n");
}
main
{
printf("hello..\n");
```

o/p:-

hello
hi
in isr
:

```
alarm(5);
Signal ( SIGNAL RM, isr);
printf ("hi \n");
 while(1);
 }
```

```
#include <stdio.h>
# include <signal.h>
void isr (int n)
{
printf ("in isr..\n");
alarm(2);
}
main
{
printf("hello..\n");
alarm(5);
Signal(SIGALRM, isr);
printf ("hi..\n");
while(1);
}
```

o/p:-

```
hello
hi
in isr
in isr
in isr
:
:
```

(hello, hi — after 5 se; in isr, in isr, in isr — for every 2 sec)

Return Value of alarm :-

alarm() returns the number of seconds remaining until any previously scheduled alarm was due to be delivered, or zero if there was no previously scheduled alarm.

```
#include <stdio.h>
main()
{
 int ret;
 printf("test1...\n");
 ret=alarm(10);
 printf("1) ret = %d\n",ret);
 ret = alarm(5);
 printf("2) ret = %d\n", ret);
 ret = alarm(2);
 printf("3) ret = %d\n", ret);
}
```

O/p:-

test1...
1) ret = 0
2) ret = 10
3) ret = 5.

```
#include <stdio.h>
 main()
 {
 int ret;
 printf("test1...\n");
 ret = alarm (10);
printf("1) ret = %d\n", ret);
   sleep(3);
 ret= alarm(5);
printf("2) ret = %d\n", ret);
   sleep(1);
 ret = alarm(2);
printf("3) ret = %d\n", ret);
}
```

O/p:-

test1...
1) ret = 0
2) ret = 7
3) ret = 4.

Prog to disable (ctrl+c)(2) and (ctrl+\)(3) upto 10 Seconds:

```
#include <stdio.h>
#include <signal.h>
void isr( int n)
{
alarm(10);
printf("in isr..\n");
Signal( 2, SIG_DFL);
Signal(3, SIG_DFL);
}
main()
{
Signal(SIGINT, SIG_IGN);
Signal(SIGQUIT, SIG_IGN);
Signal(SIGALRM, isr);
alarm(10);
while(1);
}
```

Disable signal (sigint) and sigquit in your process restore sigint to sigdfl, when it occurs 4th time and also restore sigquit to sigdfl when it occurs 2nd time.

```
#include<stdio.h>
#include<signal.h>
int c, d;
void isr(int n)
{
  if(n==2)
  {
   c++;
   if(c==4)
   signal (2, SIG-DFL);
  }
 else if(n==3)
  {
   d++
   if(d==2)
   signal(3, SIG-DFL);
  }
main()
{
signal(SIG.    , isr);
                INT
signal(SIG      , isr);
                QUIT

while(1);
}
```

Observe that when terminal gets terminates, jobs which are executing under that particular terminal will receive sighup signal (SigHup).

Note:- Whenever a terminal is closed, all the process under that terminal is terminated by receiving SIGHUP (1) Signal.

```
#include <stdio.h>
#include <signal.h>
main()
{
printf("pid=%d\n ppid=%d\n", getpid(), getppid());
Signal(1, SIG_IGN); // open another terminal send signal
while(1);                IGN.          no:- 9 to the ppid
}                                      #kill -9 ppid(Bash)
```

Observe that when child process is terminated, parent receiving SIGCHLD (17) signal.

```
#include <stdio.h>
# include <signal.h>
 void isr (int n)
  { if (n==17)
  printf("Received SIGCHLD signal);
  }
```

```
main()
{
 if(fork()==0)
 {
  printf(" in child..\n");
  sleep(10);
  printf(" child exit..\n");
 }
 else
 {
  signal(14, isr);
  while(1);
 }
}
```

Note:- Default Action for SIGCHLD signal is Ignore. Parent process will ignore the signal.

Majority (90 %) of signals action is termination, 10% are having different actions.

18/08/17

ACHIEVING JOB CONCURRENCY :

```
#include <stdio.h>
#include <signal.h>
void isr (int n)
{
printf (" in isr..\n");
wait(0);
}
main()
{
if (fork()==0)
{
printf ("in child = %d\n", getpid());
usleep(10);
printf (" Child exit..\n");
}
else
{
signal (17, isr);
while (1);
}
}
```

O/p:

```
in child = Pid
child exit..
in isr...
```

Note:- Here the job currency is achieved by using wait in ISR(Interrupt Service routine) So the parent is not blocked.

Wap to deliver SIGALRM to long process after 10(n) seconds and do the same repeatedly with -n 10 - isr , 9 - isr . when n reaches to zero terminate the process [using cla].

```
#include<stdio.h>
#include<signal.h>
int i;
void isr(int n)
{
 isr printf(" in isr..\n");
   i=i-1
  if(i==0)
   raise(9);
  alarm(i);
  printf(" alarm is set for %d secs\n", i);
}
main(int argc, char * argv[])
{
  i = atoi(argv[1]);
  Signal(SIGALRM, isr);
  sleep(i);
 } alarm(i);
  while(1);
}
```

O/p:-

```
./a.out 10
in isr..
alarm is set for 9sec..
:
in isr..
alarm is set for 1 sec..
killed
```

Note: The signals SIGKILL (9) & SIGSTOP (19) cannot be caught or ignored

These 2 are non-blockable signals. It will do its default actions. It cannot be either ignored or modified do ISR.

```
#include <stdio.h>
#include <signal.h>
void isr (int n)
{
printf ("in isr ..\n");
}
main ()
{
printf ("PID = %d\n", getpid());
signal (9, isr);
while (1);
}
```

O/P:-

Pid = x.

kill -9 x (in another terminal)

killed.

Prog to find the current action of a signal

Note:- Return Value of Signal: The signal() returns the previous value of the signal handler (old action value is returned when a new action is set).

2. DFL

```
#include <stdio.h>
#include <Signal.h>
void isr()
{
}
main()
{
void (*ptr)(int);
int n;
Signal(2, SIG_IGN);
printf("enter signal number..\n");
Scanf("%d", &n);
ptr = Signal(n, isr);
      Signal(n, ptr);
if (ptr == SIG_DFL)
printf("Default..\n");
else if (ptr == SIG_IGN)
printf("Ignored..\n");
 else
 printf("isr...\n");
}
```

Note: When one signal ISR is executing same signal is blocked but other signals are allowed.

```
#include <stdio.h>
#include <signal.h>
void isr (int n)
{ unsigned long int i;
  printf ("in isr ..\n");
  for (i=0; i<4000000 ;i++);
   "      "      "
   "      "      "
   "      "      "
   "      "      "
   :      :      :
  printf ("exit of isr..\n");
}
main()
{
  Signal(2, isr);
  while(1);
}
```

o/p:- ctrl+c
in isr ...
again ctrl+c is not accepted during executing but other signals are allowed.

SIG_ACTION :-

Sigaction - examine & change a signal action

int sigaction (int signum, const struct sigaction *act, struct sigaction *oldact);

The sigaction structure is defined as something like :

```
Struct sigaction {
  ✓void     (* sa_handler)(int);
  ✗void     (* sa_sigaction)(int, siginfo_t *, void *);
  ✓sigset_t  sa_mask;
  ✓int       sa_flags;
  ✗void     (* sa_restorer)(void);
  };
```

19/08/17

In case of ubuntu when we are using sa_handler there is no need of using sa_sigaction.

The sa_restorer element is obsolete and should not be used.

i) Sigaction (num, &v, 0) — not interested to collect old action
   - we are interested to set the new action of a signal but not interested to collect the old action.
   (&v : structure variable address)

(ii) Sigaction (num, 0, &v) - we are not interested to set the new action, but interested to collect the old action.

(iii) Sigaction (num, &v1, &v2) -> Interested to Set the new action & collect the old action.

```
#include <stdio.h>
#include <signal.h>
 void isr (int n)
 {
 printf (" in isr .. \n");
  }
 main ()
 {
 printf ("in main . \n");
 Signal (2, isr);
 printf (" main exit . \n");
 while (1);
 }
```

To change the action of signal to isr using Sigaction.

```
#include <stdio.h>
# include <Signal.h>
 Void isr (int n)
  {
  printf (" in isr .. \n");
  }
  main ()
   {
```

```
Struct sigaction v;

v.sa_handler = isr;
sigemptyset (&v.sa_mask);
v.sa_flags = 0;
sigaction (2, &v, 0);
while (1);
}
```

o/p:-

ctrl+c in isr
ctrl+c in isr
⋮

⟺ signal (2, isr).

* sa_handler in the structure members contains the current action.

To find the current action of a signal using sigaction.

```
#include <stdio.h>
#include <signal.h>
void isr(int n)
{
in isr...!
printf(" in isr ..\n");
}
main()
{
Struct sigaction v;
Signal(2, isr);
sigaction (2, 0, &v);
```

o/p:-

ctrl+c in isr...
isr...

```
if (v.sa_handler == SIG_DFL)
  printf (" Default ... \n");
else if (v.sa_handler == SIG_IGN)
  printf ("Ignore ... \n");
else
  printf ("     int ... \n");
while(1);
}
```

Note:- Using signal function to find the current action, it is mandatory to set the dummy action. In case of sigaction it is no needed.

FLAGS

SA_RESETHAND:-

Restore the signal action to the default state once the signal handler has been called.

```
#include<stdio.h>
#include<signal.h>
void int (int n).
{
  printf(" in int .. \n");
}
main()
```

```
{
Struct sigaction V;
V.sa_handler = isr;
Sigemptyset (&V.sa_mask);
V.sa_flags = SA_RESETHAND;
Sigaction (2, &V, 0);
while (1);
}
```

O/p:-
ctrl+c in isr
ctrl+c process terminate.

SA_NOCLDSTOP:-

If signum is SIGICHLD, do not receive notification when child processes stop or resume. This flag is only meaningful when establishing a handler for SIGICHLD. SIGTTIN SIGTTOU.

Note: The parent process will receive SIGICHLD signal not only during child process getting terminated by also any state change ((i.e) running process termination, running process suspension, suspended process resumed).

To make the parent receive SIGICHLD for only child process termination (not for any other state change).

```
#include<stdio.h>
#include<signal.h>
void isr (int n)
{
```

```
printf(" in isr..\n");
wait(0);
}
main()
{
if(fork()==0)
{
printf(" in child prent pid = %d\n", getpid());
sleep(10);
printf(" child exit..\n");
exit(0);
}
else
{
struct sigaction V;

V.sa_handler = isr;
sigemptyset(&V.sa_mask);
V.sa_flags = SA_NOCLDSTOP;
sigaction(17, &V, 0);
while(1);
}
}
```

SA_NOCLDWAIT:

If signum is SIGCHLD, do not transform children into zombies when they terminate.

To collect the status of the child without writing wait in parent or ISR.

```
#include <stdio.h>
#include <Signal.h>
void isr (int n)
{
printf("in isr..\n");
}
main()
{
Struct Sigaction V
   if(fork() == 0)
   {
  printf ("in child PID=%d\n", getppid());
   sleep(10);
   printf ("child exit..\n");
    exit(0);
   }
   else
   {
     V.sa_handler = isr
      Sigemptyset (V.sa_mask);
```

```
V.sa_flags = SA_NOCLDSTOP | SA_NOCLDWAIT;
sigaction (17, &v, 0);
while(1);
}
```

SA_NODEFER :

Do not prevent the signal from being received from within its own signal handler.

To respond to the same signal when the isr of that Signal executing.

```
# include <stdio.h>
# include <signal.h>
void isr (int n)
{ unsigned long int i;
printf ("in isr .. \n");
for (i=0; i<40000000 ; i++);
for (i=0; i<40000000 ; i++);
                "
                "
                "
                :
printf ("isr exit .. \n");
}
main ()
{
  struct sigaction v;
     v.sa_handler = isr;
     sigemptyset (&v.sa_mask);
```

```
V.Sa_flags = SA_NODEFER;
Sigaction (2, &v, 0);
while(1);
}
```

SA_MASK :-

```
int sigemptyset (sigset_t *set);
int sigfillset (sigset_t *set);
int sigaddset (sigset_t *set, int signum)
```
(to block extra signals) along with sigemptyset

```
int sigdelset (sigset_t *set, int signum).
```
(to exclude some extra signals) with sigfillset.

Sigemptyset() → allow all other signals mask only one signal (ISR)

Sigfillset() → mask all other signals and allow that signal whose ISR is designed (ISR)

Description :-

Sigemptyset() initializes the signal set given by set to empty, with all signals excluded from the set

Sigfillset() initialises set to full, including all signals.

Sigaddset() and sigdelset() add and delete respectively signal signum from set.

Sigemptyset(&V.sa_mask) } block 3 also along with ISR
Sigaddset(&V.sa_mask, 3) } signal.

Sigfillset(&V.sa_mask) } allow 3 alone
Sigdelset(&V.sa_mask, 3) }

21/Oct/17

RESOURCE MANAGEMENT :

Process Resource Limits:

Each process has a set of resource limits that can be used to restrict the amounts of various system returns resources that the process may consume.

Limits
- SoftLimit
- Hardlimit.

SoftLimit :- The limit provided by the OS. Softlimit value can be changed upto hardware limit.

Hardlimit :- The maximum point of the softlimit.

The getrlimit() and setrlimit() system calls allow a process to fetch and modify its resource limits.

Headerfile <sys/resource.h>.

int getrlimit (int resource, struct rlimit *rlim);
int setrlimit (int resource, const struct rlimit *rlim).

Both returns 0 on success or -1 on error.

* Getrlimit is used to get the resource information
* Setrlimit is used to set the resource according to our purpose.

```
Struct rlimit {
    rlim_t rlim_cur ; // Soft limit (actual process limit)
    rlim_t rlim_max ; // hard limit (ceiling for rlim_cur)
};
```

These fields correspond to the 2 associated limits for a resource, the Soft (rlim_cur) and hard (rlim_max) limits. (The rlim_t data type is an integer type). The soft limit governs the amount of the resource that may be consumed by the process. A process can adjust the Softlimit to any value from 0 upto the hard limit

For most resources, the sole purpose of the hard limit is to provide this ceiling for the Soft limit.

RLIMIT_STACK :- The maximum size of the process stack, in bytes. Upon reaching this limit, a SIGSEGV signal is generated.

```
#include <stdio.h>
#include <sys/resource.h>
main()
{
 struct rlimit r;

 getrlimit (RLIMIT_STACK, &r);
 printf (" Soft = %d\n", r.rlim_cur);
 printf (" Hard = %d\n", r.rlim_max);

 r.rlim_cur = 1000;

 setrlimit (RLIMIT_STACK, &r);
 getrlimit (RLIMIT_STACK, &r);
 printf ("Soft = %d\n", r.rlim_cur);

}
```

O/p :-

default { Soft = 8388608
Hard = 4294967295

Soft = 1000.
changed

The changed limits will be affected only in that process not in other processes all.

RLIMIT_CPU :- CPU time limit in seconds, when the process reaches the soft limit, it is sent a SIGXCPU signal. The default action for this signal

is to terminate the process.

```
#include <stdio.h>
#include <sys/resource.h>
main()
{
struct rlimit v;
getrlimit (RLIMIT_CPU, &v);
printf (" Soft = %d\n", v.rlim_cur);
printf (" Hard = %d\n", v.rlim_max);

v.rlim_cur = 100 ;

setrlimit (RLIMIT_CPU , &v);
getrlimit (RLIMIT_CPU, &v);
printf ("Soft = %d\n", v.rlim_cur);
while(1);
}
```

O/p :-

Soft = 4294967295

Hard = 4294967295

Soft = 10.

Time - get time in Seconds.

Headerfile :- #include <time.h>.

time_t time (time_t *t);

time() returns the time as the number of seconds since the Epoch time (1970-01-01)

```
#include <stdio.h>
#include <time.h>
{
time_t t;
while(1)
{
time(&t);
printf("%u\n",t);
printf("%s\n", ctime(&t);
sleep(1);
system("clear");
}
```

o/p :- 4

2017 August 21-08-11-15-30

epochtime

Note:- Epoch time is Converted to normal time (i.e) current year, month & time Can be got using ctime() function

RLIMIT_CORE :- Maximum Size of core file. when 0 no core dump files are created, when non-zero, larger dumps are truncated to this Size.

Core - Core dump file

The default action of certain signals is to Cause a process to terminate and produce a core dump file, a disk file Containing an image of the process's memory at the time of termination.

This image can be used in a debugger (e.g., gdb(1) [(i.e) tool to analyze the process step by step). to inspect the state of the program at the time that it terminated.

There signals that can generate core dumped on files are created.

```
#include <stdio.h>
#include <sys/resource.h>
main ()
{
struct rlimit v;

getrlimit (RLIMIT_CORE, &V);
printf ("Soft = %d\n", V.rlim_cur);
printf ("Hard = %d\n", V.rlim_max);

V.rlim_cur = 1000;

Setrlimit (RLIMIT_CORE, &V); getrlimit (RLIMIT_CORE, &V);
printf ("Soft = %d\n", V.rlim_cur);

}
```

O/p :-

Soft = 0
Hard = 4294967295

Soft = 1000

To get core dumped files we need to change the Soft limit value.

SIGISEGV - 11
SIGILL - 4
SIGABRT - 6
SIGFPE - 8
SIGQUIT - 3

} Signals that have the capability to generate core-dumped files.

RTO's signals

SIGIBUS- 10
SIGSYS - 12
SIGTRAP - 5
SIGXFSZ

22/08/17

RLIMIT_FSIZE :- The maximum size of files that the process may create. Attempts to extend a file beyond this limit result in delivery of a SIGXFSZ signal. By default, this signal terminates a process.

```
#include <stdio.h>
#include <sys/resource.h>
main()
{
  struct rlimit v;
  getrlimit(RLIMIT_FSIZE, &v);
```

O/p :-

SOFT : 4294967295
Hard :- 4294967295.

```
printf ("Soft = %d\n", v.rlim_cur);
printf ("Hard = %d\n", v.rlim_max);
}
```

```
#include<stdio.h>
#include<sys/resource.h>
main()
{
struct rlimit v;
getrlimit(RLIMIT_FSIZE, &v);
printf("Soft = %d\n", v.rlim_cur);
printf("Hard = %d\n", v.rlim_max);
v.rlim_cur = 10;

setrlimit(RLIMIT_FSIZE, &v);
getrlimit(RLIMIT_FSIZE, &v);
printf("Soft = %d\n", v.rlim_cur);
printf("Hard = %d\n", v.rlim_max);

FILE *fp;
char a[20] = "abcdefghijklmno"

fp=fopen("temp","w");

fprintf(fp,"%s", a);

}
```

O/p:-

```
Soft = 4294967295
Hard = 4294967295
Soft = 10
Hard= 4294967295
File size exceeded
(core dumped).
```

DATA IN TEMP

abcdefghij, only 10 bytes.

as we set the Soft limit to only 10

FILE MANAGEMENT:

How files are stored in the secondary memory.

* A file system is an Organised collection of regular files and directories.
* One of the strengths of Linux is that it Supports a wide Variety of file Systems, including the follows.,

* The traditional ext2, ext3, ext4 file System (LINUX)
* Microsoft FAT, FAT32, NTFS file System (WINDOWS).

The filesystem types (/proc/filesystems file) currently known by the kernel can be viewed in the Linux-Specific.

System related all functions are stored in proc.

Command /proc/filesystems

File-System Structure:

The basic unit for allocating space in a file System is a logical block, which is some multiple of contiguous physical blocks on the disk device on which the file System resides.

The logical block size on ext2 is 1024, 2048 or 4096 bytes

Memory is used served as block by block, (i.e) Logical block. Generally in Linux (4 blocks of memory is reserved, each block 1024 bytes).

1 block = 1024 bytes

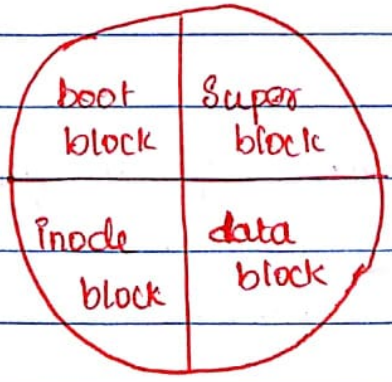
4 x 1 block = 4096 bytes.

To know how much blocks reserved

command :- ls -lg . filename

When the memory exceeds 4096 bytes then again 4 blocks of memory is reserved.

Interacting often with the Secondary memory is time consuming process (thus 4 blocks are reserved as a whole when it exceeds 4096, like 4097 then again 4 blocks are reserved).



Once OS is installed, file system is divided into

4 major partitions

Boot block: This is the first block, in filesystem boot block is not having any information related to filesystem rather it contains information related to OS booting process.

Super block: Immediately after boot block is Superblock. It contains information related to filesystem * Size of the inode table.
* Size of the logical block

Inode block: Each file & directory having an entry in inode table (everything in Linux is treated as a file) Every file/directory has a unique Inode
Entry in Inode table stores the information like type of file, file permissions, Size of file.

Data block: Majority space in filesystem is reserved for datablock (where files & directories are stored)

To know the inode number of the file

Command: ls -li filename.

Note: No 2 files have the same inode number.

Command: ls -i filename.

Just filename & inode number.

Command : ls -l filename

-rw-rw-r-- 1 vector vector 2 Aug 22 11:17 1.c

↓ type of file.

Linux supports, the different types of file.

* Regular file (-)
* Directory file (d)
* char device file (c) } → Perice deriver files
* Block device file (b) }
* link file (l)
* pipe file (p) or FIFO → IPC (Interprocess Communication)
* Socket file (s)

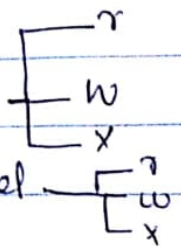
-rw-rw-r-- 1 Vector Vector 2 Aug 22 11:17 1.c

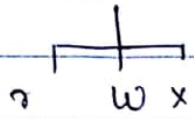
- : type of file
rw-rw-r-- : file permissions
1 : No. of links
Vector : user name
Vector : group name
2 : Size of file
Aug 22 11:17 : Time stamp (last modified time).
1.c : filename

23/08/17

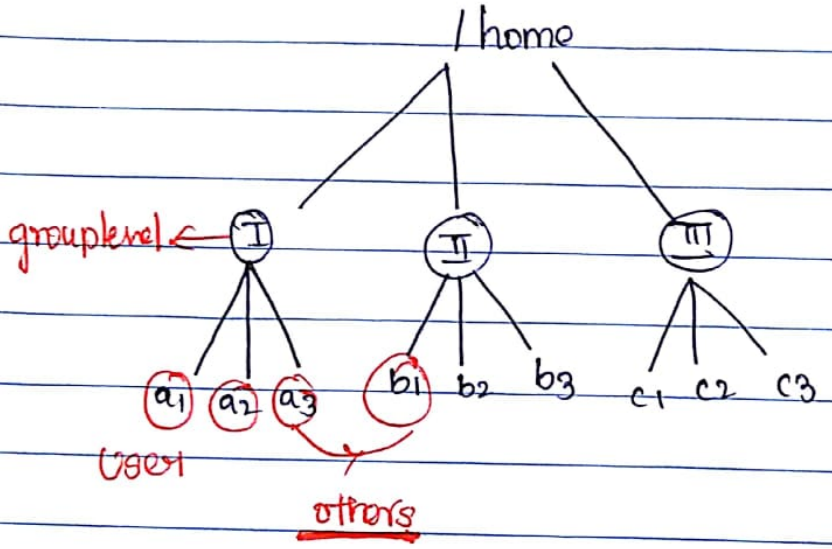
3 levels of permission given, when a file is created.

* U - User level
* g - Group level
* o - others





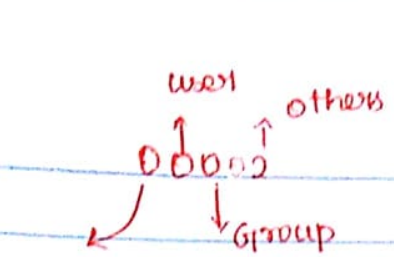
r - read
w - write
x - execute.



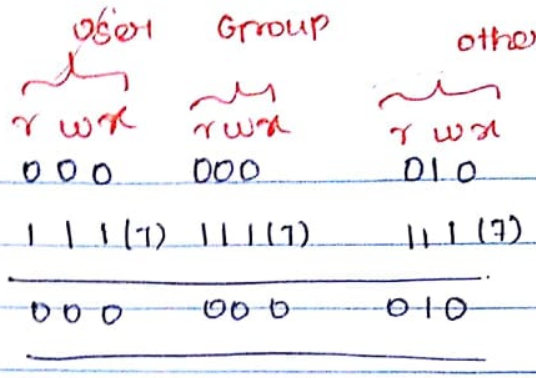
Files we are created using Vi and cat command are not executable files.

File permission are provided based on U-mask value

Command to know umask value :- umask.



Octal number system.



umask value is bitwise 2 with umask value

0- permission enabled

1 - permission disabled.

Generally umask value is 002, when need we can change the umask value.

If the terminal is closed or system is restarted, umask value set's to default.

Using umask all ready created, existing files permission not possible to change.

To change the file permission of already existing file chmod command can be used.

chmod → all permission possible to change.

→ Individual permissions possible to change.

write

Chmod o+ w filename (individual)

others enable filename (whose permission we want to change).

other

chmod o - w filename (individual)

disable

disable

All permission :- chmod ugo - r filename

read

user group other

Copy Command :- If the destination file is not there, it will be created & copy the data from source to destination. After this, the relation between 2 files are terminated. There is no link between them

Command :- ln Source destination.

Same like copy command. But there is link between Source & destination. So modification of one file will affect the another file. These two files are called link files

LINK

Hard link (ln)

Soft link (ln - s)

also called as Symbolic link file.

Hard link

* Created using ln command
* ln file1 file2
  alias names are Created.
* No two files are created (like file1 2 files2) an alias name is created for file1 ; and both the names are pointing to Same content.
* Link Count is updated.

```
-rw-rw---- 2 ravindra ravindra 21 Aug 23 10:39 file2
```

link count.

* Inode numbers of hard link files are Same (as both the names are pointing to Same Content).
* Original file and link file, both are having the Same Size.
* Deleting a file, content not deleted only link Count deleted.
* If the link count (1) is One. And if we are deleting the file then the content also deleted
* Deleting the Original file possible to get the data from hard link files.
* Hard link files are of regular type.

Soft link :-

* Soft link files are created using ln -s
* Link counts are not updated / no alias name created
* Soft link files are acts as shortcut files
* Inode numbers are not same.
* Original file size & link file size are not same.
* Soft link file size depends on Orginal file name number of characters.
* Soft link files are not regular files, they are Link files.
* Deleting the Original file, not possible to get the data from shortcut file.

Command :- ln -s Source destination.

Long list view:- sl → tl / Size of sl is 2 as tl has 2 characters in the name.

Wap to find the size of the file, without opening the file.

Stat → fstat, lstat - get file status

Headerfile :- #include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat (const char *path, struct stat *buf);
int fstat (int fd, struct stat *buf)
int lstat (const char *path, struct stat *buf);

31-

Stat() stats the file pointed to by path and fills in buf

lstat() is identical to stat(), except that if path is a symbolic link, then the link itself is stat-ed, not the file that it refers to.

fd file descriptor

fstat() is identical to stat(), except that the file to be stat-ed is specified by the file descriptor fd.

st_size a member of struct to get the size of the file, st_ino → prints the inode number.

29/08/17 Return Value of stat, fstat, lstat :-

On success, 0 is returned
" failure, -1 " "

```
#include <stdio.h>
#include <sys/stat.h>
main(int argc, char * argv[])
{
 struct stat v;
  if(argc!=2)
  {
  printf(". /a.out filename ..\n");
   return;
   }
 if(stat(argv[1],&v)<0)
  {
  perror("stat");
  return;
   }
 printf("%d\n",v.st_size);
 printf("%d\n", v.st_ino);
}
```

Note: Using stat() not possible to get the information of soft file (link file).

Stat function not having the capability to get the link file information, even if we use it we will get information of the original file.

lstat() will get the link file information.

```
#include <stdio.h>
#include <sys/stat.h>
main(int argc, char * argv[])
{
 struct Stat v;
 if(argc!=2)
 {
 printf(". /a.out  filename..\n");
 return;
 }
if( lstat( argv[1], &v)<0)
 {
 perror(" lstat ");
 return;
 }
printf("%d\n", v.st_size);
printf("%d\n", v.st_ino);
}
```

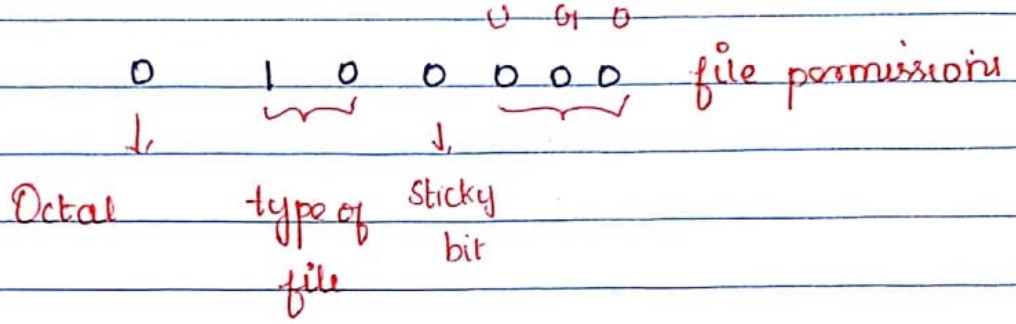
Wap to find out that given files are link files are not, if they are link files what type of link it is.

```
#include <stdio.h>
#include <sys/stat.h>
main( int argc, char * argv[])
{
 struct stat V1 , V2 ;
 if (argc != 3)
 {
 printf(" ./a.out filename1 filename2 \n");
 return)
 }
Stat( argv[1], &V1);
Stat( argv[2], &V2);

if ( V1.st_ino == V2.st_ino)

{
   lstat( argv[1], &V1);
   lstat( argv[2], &V2);
     if ( V1.st_ino == V2.st_ino)
        printf(" Hard link file..\n")

      else
        print (" Soft link file..\n");
 }
 else
    print ("no link ..\n");
 }
```

Prog to find the type of a file using CLA:

st_mode - member of the stat structure.

for regular file st_mode value is




```
#include <stdio.h>
#include <sys/stat.h>
main( int argc, char * argv[])
{
 Struct st v1;
  if (argc != 2)
  {
   printf (". /a.out filename. \n");
   }
Stat ( argv[1], &v);

if (S_ISREG (v.st_mode))
    printf (" Regular file.. \n");
else if (S_ISDIR (v.st_mode))
    printf (" It is a Directory file.. \n");
```

```
else if (S_ISLNK(V.st_mode))
printf(" It is a link file..\n");
else
printf(" It is a pipe file..\n");
}
```

```
time_t   st_atime  /* time of last access */
         st_mtime  /* time of last modification */
         st_ctime  /* time of last status change */
```

st_atime : Last time, file is opened for reading purpose.

st_mtime : Last time, file is updated, like when file contents are updated.

st_ctime : Last time, file permission change.

Note:- when m_time is updated c_time is also updated but when ctime is updated m_time not updated.

```
#include <stdio.h>
#include <sys/fcntl.h>
main (int argc, char * argv[])
{
  struct st v;
  if (argc != 2)
  {
  printf (" ./a.out filename ..\n");
  return;
  }
 stat (argv[1], &v);
 printf ("atime = %ulp", v.st_atime);
 printf ("mtime = %uln", v.st_mtime);
 printf ("ctime = %uln", v.st_ctime);
}
```

→ display , epoch time

↓

to change to Now.

ctime(&v.st_mtime)

30.08.2017

DIRECTORY FILES :-

Opendir - Open a directory

Header file :- #include <sys/types.h>
#include <dirent.h>

DIR *opendir (const char * name);

The opendir() function opens a directory stream corresponding to the directory name and returns a pointer to the directory stream.

The opendir() function return a pointer to the directory stream, on error it returns null.

readdir - read a directory

Headerfile :- #include <dirent.h>

Struct dirent *readdir(DIR *dirp);

The readdir() function returns a pointer to a dirent structure representing the next directory entry in the directory stream pointed to by dirp.

It returns NULL on reaching the end of the directory stream or if an error occured.

Command :- ls -a. (print hidden files).

Note:- Once a directory is created, by default 2 files available in each directory. They are . and .., and they are known as Hidden files.

. → present working directory

.. → parent "

ls command usually does not have the capability to display hidden files. Hence ls -a is the command used to display open hidden files.

```
#include<stdio.h>
#include <dirent.h>
main (int argc, char * argv[])
{
 DIR * dp ;
 Struct dirent *p ;

 dp = opendir ( argv[1] ) ;

 if (dp == NULL)
 {
 printf (" opendir ") ;
 return ;
 }
 while ( p= readdir( dp))
 printf ("%s\n", p->d_name) ;
 }
```

O/p :- ./a.out temp
1.c
.
..
2.c

Note: filename size not more than 256 bytes.

Wap to find out that in a given directory, specific file is found are not using command line arguments.

./a.out directory filename.

```
#include<stdio.h>
#include <dirent.h>
main(int argc, char * argv[])
{
 DIR *dp;
 Struct dirent *p;
 dp= opendir ( argv[1])
 if ( dp==NULL)
 {
 perror (" opendir ");
 return;
 }

 while (p=readdir (dp))
 {

  if (!strcmp (p->d_name, argv[2]);
  { printf ("file found..\n");
    return;
  }
```

```
if(argc!=3)
{
printf (" ./a.out directory filename\n");
return;
}
```

```
}
printf("Not found..\n");
}
```

can use the path, absolute path, "." to open the directory, parent directory means ".."

Wap to implement ls -l command

st_nlink → (No. of hard links)

st_uid → (user id)

st_gid → (group id).

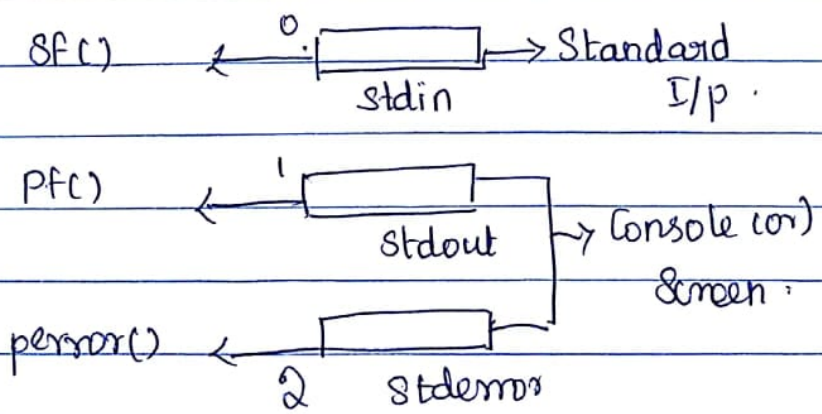
FILE HANDLING USING SYSTEM CALLS :-

31/08/17

* When a process is created by default PCB created
* Similarly, when a process is created 3 files are attached to each process called as

stdin ⇔ 0
stdout ⇔ 1
stderror ⇔ 2
} magic numbers

* These files (buffers) are represented with a small non-negative integer numbers 0, 1, 2. called file descriptors.



Systemcalls for file handling in Linux are

Open()	read()	close()	dup()	lseek()
Create()	write()	fcntl()	dup2()	

In case of library functions we usually use file pointers whereas in case of system calls we usually use file descriptors.

In fopen case after opening the file it returns the address to file pointer, bt open system call returns file descriptor to the user.

Using library function we can open only regular files (text file or Binary file) bt we cannot open socket, pipe & any other files.

```
#include <stdio.h>
main()
{
  int i=0;
  while(1)
  {
  printf("hello..\n");
    i++;
  if (i==3)
    close(1); /close(0).
  }
}
```

O/p:

hello
hello
hello
hello (wait for ch)
hello (wait for ch)
hello (wait for ch)
:
:
:

close - close a file descriptor.

Header file : #include <unistd.h>.

int close (int fd)

close() closes the file descriptor, so that it no longer refers to any file and may be reused.

Open, Create - open and possibly create a file or device.

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h> (flags are defined)

int open (const char *pathname, int flags);
int open (const char *pathname, int flags, mode_t mode);
int creat (const char *pathname, mode_t mode);

flags

Mandatory flags	Optional flags	
O_RDONLY	O_CREAT	O_EXCL
O_WRONLY	O_APPEND	
O_RDWR	O_TRUNC	

Open() and Creat() return the new file descriptor, or -1 if an error occured.

fopen() function opened in write mode will create the file if it is not there but Open System call has certain parameters

```
fopen("temp", "w")
        ||
Open("temp" O_WRONLY|O_CREAT|O_TRUNC)
```

Command :-

```
cd proc/pid/fd
ls -l.
```

ls -l /proc/pid/fd

to know where the file descriptor opened or not

```
# include <stdio.h>
# include <fcntl.h>
main()
{
int fd;
fd=Open("temp", O_WRONLY|O_CREAT|O_TRUNC);
if (fd<0)
{
printf("fd=%d\n", fd);
```

```
    perror("Open");
}
printf("fd = %d\n", fd);
}
```

fp = fopen("temp", "r")

⇕

fd = Open("temp", O_RDONLY)

fp = fopen("temp", "a")

⇕

fd = Open("temp", O_WRONLY | O_CREAT | O_APPEND)

fp = fopen("temp", "a+")

⇕

fd = Open("temp", O_RDWR | O_CREAT | O_APPEND)

fd = open("temp", O_RDWR | O_CREAT) ✓ possible

⇕ ✗

fp = fopen("temp" (✗)) → There is no equivalent fopen (Not possible)

Note:- Whatever fopen() does Open() can do, bt whatever open() does fopen() can't.

Given a pathname for a file, open() returns a file descriptor, a small, non-negative integer for use in subsequent system calls (read(2), write(2), lseek(2), fcntl(2), etc.).

Third option of the open() system refers to the file permissions (0664) if not provided the default file permissions will be provided based on umask value.

fd = open ("temp" O_WRONLY | O_CREAT | O_TRUNC, 0664).

Create() is equivalent to open() with flags equal to O_CREAT | O_WRONLY | O_TRUNC.

fd = creat ("temp", 0664)

⇕

fd = Open ("temp", O_CREAT | O_WRONLY | O_TRUNC ...)

11/09/17

Read - Read from a file descriptor.

Header file :- <unistd.h>.

ssize_t read (int fd, void *buf,
size_t count);
unsigned int typedef.

Read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.

If the count is zero, read() returns zero and has no other results, If count is greater than SSIZE_MAX / the result is unspecified

read (fd, a), 5) → count 5 variables (characters)

read (fd, &ch, 1) → 1 character

SSIZE_MAX : 2147483647

On success, number of bytes read is returned (zero indicates end of the file) & the file position is advanced by this number.

On error -1 is returned.

```
#include <stdio.h>
#include <fcntl.h>
main()
{
 int fd,
 char ch;
 fd = fopen("temp", "O_RDONLY");
 if (fd < 0)
 {
 printf("f = %d\n", f);
 perror("open");
 return;
 }
while (read (fd, &ch, 1) > 0)
 printf ("ch = %c\n", ch);
}
```

O/P:- temp

```
abcdefghijkl
ch=a
ch=b
ch=c
ch=d
ch=e
ch=f  ch=k
ch=g  ch=l
ch=h
ch=i
ch=j
```

Note:- Read system call is not responsible for adding '\0' at the end of the array. in case of string, so while printing the string some garbage data will be displayed.

(1)
```
#include <stdio.h>
#include <fcntl.h>
main()
{
 int fd, ret;
```

```
Char a[20];
fd= Open ("temp", O_RDONLY);
if (fd < 0)
{
printf ("fd=%.d\n", fd);
perror ("Open");
return;
}
ret = read (fd, a, 5)
a[ret] = '\0';
printf("%.s\n", a);
}
```

O/p :- Temp

abcdefghijk

console :-

abcde.

IInd Solution to add NULL Character :-

bzero - Write zero-valued bytes.
(byte zero)

#include <Strings.h> ⇒ Headerfile.

void bzero (void *s, size_t n);

The bzero() function sets the first n bytes of the area starting at s to zero (bytes containing '\0').

```
#include <stdio.h>
#include <fcntl.h>
main()
{
 int fd
 char a[128]
 bzero (a,128); /memset (a, 0, 128)
 fd = open( "temp" O_RDONLY);
 if (fd<0)
 {
 printf ("fd= %d\n", fd);
  perror("Open");
  return;
 }
 read ( fd, a , 5)
 printf ("%s\n", a);
```

III solution :-

memset - fill memory with a Constant byte

Headerfile :- #include <strings.h>

void *memset (void *s, int c, size_t n);

The memset() function fills the first n bytes of the memory area pointed to by s with the constant byte c.

Note: In case of memset() we have the option (i.e) which with what value we want to fill the array whereas in bzero by default it will fill with zero.

Write - Write do va file descriptor.

Header file : # include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);

Write() writes up to count bytes from the buffer pointed buf to the file referred to by the file descriptor fd

On success, the number of bytes written is returned (zero indicates nothing was written). on error -1 is returned.

```
# include <stdio.h>
# include <fcntl.h>
main()
{
  char a[20];
  int fd;
  fd = Open ("temp", O_WRONLY|O_CREAT|O_TRUNC);
      if (fd<0)
```

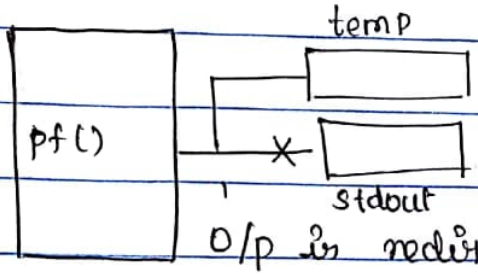
```
{
printf ("fd = %d\n", fd);
perror(" open");
return;
}
printf ("enter the data to be written into file..\n");
write (fd, a, strlen(a));   scanf ("%s", a);
}
```

```
#include <stdio.h>
#include <fcntl.h>
main()
{
int fd;
close(1);
fd = open ("temp", O_WRONLY | O_CREAT | O_TRUNC);
if (fd < 0)
{
perror ("Open");
return;
}
printf ("Value of fd = %d\n", fd);
}
```

O/p:-

On screen, nothing will print

bt temp contains the data



printf() is written into file.

O/p is redirected from Stdout to file

Andy this case is called as Output-redirection.

```
#include<stdio.h>
#include<fcntl.h>
main()
{
 int fd, i;
 int a[5] = {10,20,30,40,50}; close(1);
 fd = Open("temp", O_WRONLY | O_CREAT | O_TRUNC);
 if(fd<0)
 {
 printf("fd = %d\n", fd);
 perror("Open");
 return;
 }
for(i=0; i<5; i++)
printf("%d ", a[i]);

}
```

o/p :-

On console o/p

in temp

10 20 30 40 50

From file to the array using fd.

```
#include<stdio.h>
#include<fcntl.h>
main()
{
 int fd, i;
 int b[5];
 close(0);
 fd = Open("temp", O_RDONLY);
   if(fd<0)
 {
```

```
printf("fd=%d\n", fd);
perror("Open");
return;
}
for (i=0; i<5; i++)
Scanf("%d", &b[i]);
for (i=0; i<5; i++)
printf("%d ", b[i]);
}
```

Note:- I/p is redirected from stdin to file, and this is called as I/p redirection

lseek() is for file descriptor, fseek() for file pointer.

lseek() - reposition read/write file offset.

```
#include <sys/types.h>
#include <unistd.h>
```

Off_t lseek(int fd, Off_t offset, int whence);

The lseek() function repositions the offset of the open file associated with the file descriptor fd to the argument offset according to the directive whence.

Upon successful Completion, lseek() returns the resulting off-set location as measured in bytes from the beginning of the file. On error the value (off_t) -1 is returned.

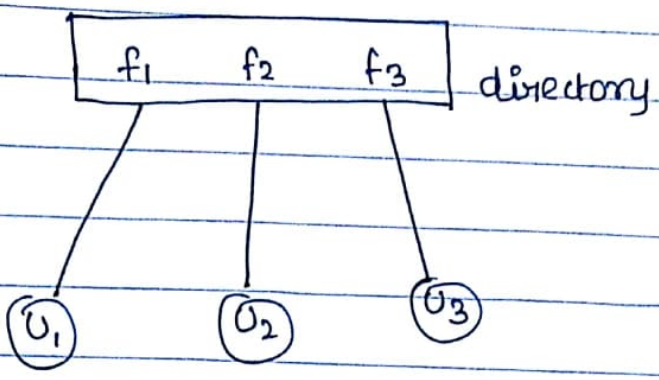
2/09/17

STICKY_BIT :-

On older Unix implementation, the sticky bit was provided as a way of making commonly used programs run faster. If the sticky bit was set on a program file, then the first time, the program was executed, a copy of the program text was saved in the swap area - thus it sticks in swap, and loads faster on subsequent executions. Modern UNIX implementations have more sophisticated memory-management systems, which have rendered this use of the Sticky permission bit obsolete.

In modern implementations (including Linux), the sticky permission bit serves another, quite different purpose. For directories, the sticky bit acts as the restricted deletion flag. This makes it possible to create a directory that is shared by many users who can each create and delete their own files in the directory but can't delete files owned by other users, can't rename the

files.



To set the sticky bit in "others" execute bit can also be used for sticky bit

rwx rwx rwx
u g o

command :- chmod o+t directory file name

rwx rwx rwt
↓
can act both as stickybit and the other users has execute permission

Sticky bit on directory files means this modern implementation. On Files means store that frequent use file in swap area., (i.e) older unix implementation.

Dup - Duplicate a file descriptor.

Headerfile: #include <unistd.h>

int dup (int oldfd)
int dup2 (int oldfd, int newfd).

* These system calls create a copy of the file descriptor oldfd

* dup() uses the lowest-numbered unused descriptor for the new descriptor.

* dup2() makes newfd be the copy of oldfd, closing newfd first if necessary

* If oldfd is not a valid file descriptor, then the call fails and the newfd is not closed

* If oldfd is a valid file descriptor, and newfd has the same value as oldfd, then dup2() does nothing, and returns newfd.

After a successful return from one of these system calls, the old and new file descriptors may be used interchangeably.

On success, these systemcall return the new descriptor, On error -1 is returned.

```
#include <stdio.h>
#include <fcntl.h>
main()
{
  int fd1, fd2;
printf("  pid = %d\n", getpid());
fd1 = Open("temp", O_WRONLY|O_CREATE|O_TRUNC);
 if (fd1<0)
 {
 perror("Open");
 return;
 }
fd2 = Opei dup(fd1);
printf("fd1 = %d\n", fd1);
printf("fd2 = %d\n", fd2);

write (fd1, "Hello", 5);
write (fd2, "India", 5);
}
```

O/p:- In temp.
Hello India
" means both fds are pointing to Same directory file

```
#include<stdio.h>
#include<fcntl.h>
main()
{
 int fd1;
 fd1 = Open("temp", O_WRONLY|O_CREAT|O_TRUNC);
 if(fd1<0)
 {
 perror("open");
  return;
  }
 Close(1);
 dup(fd1);
  execlp("date","date", NULL)
  printf("after execlp\n");
 }
```

O/p:-

no o/p on terminal

temp:

Date command executed

dup2() can be used, with what number (fd) the duplication should happen (given explicitly).

```
#include<stdio.h>
#include<fcntl.h>
main()
 {
  int fd1, fd2;
 printf("PID = %d\n", getpid());
```

```
fd1 = Open("temp", O_WRONLY|O_CREAT|O_TRUNC);
if (fd1<0)
{
perror("Open");
return;
}
dup2(fd1, 5); // 3 and 5 refering to same directory.
while(1);
}
```

04/09/17

INTER PROCESS COMMUNICATION (IPC).

* Communication b/w two processes which are running under same system possible with the help of IPC mechanism.

* Two processes which are running in different systems Communication possible with the help of Sockets. Sockets :- end system communication.

* IPC (or) mechanism is possible to implement by using following mechanism

(i) Pipes
(ii) Named pipes (FIFO)
(iii) Message Queues
(iv) Shared Memory

} process Communication

(v) Semaphores - process Synchronisation

PIPES:

Operations involved in Pipes are

(i) Create the pipe
(ii) Write data into pipe
(iii) Read data from pipe.

How to Create pipe:

pipe, pipe2 – Create pipe.

Header: #include <unistd.h>

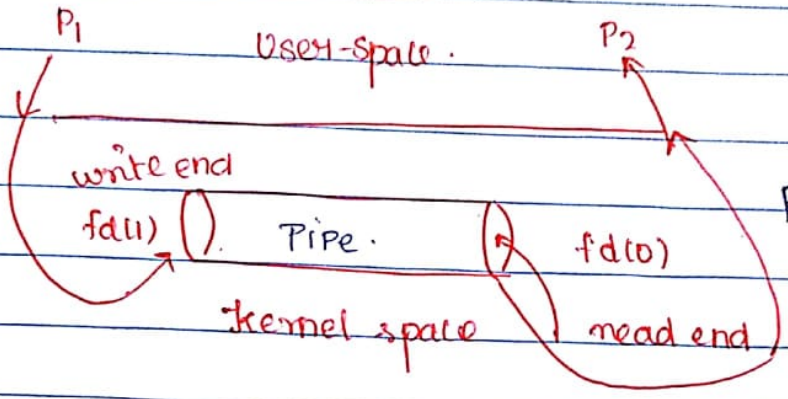
int pipe (int pipefd[2])

int pipe2 (int pipefd[2], int flags)

1] Pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication
2] A pipe has read end and write end, the data written to the write end of a pipe can be read from the read end of the pipe
3] A pipe is created using pipe(), which creates a new pipe and returns two file descriptors, one referring to the read end of the pipe and other referring to the write end of the pipe.
3] Pipes can used to create a communication channel between the related processes.

On success, zero is returned, on error -1 is returned.

* Always Fd[0] is for read of the pipe
" fd[1] is " write " " "
* Pipe will be created in kernel space.



Pipe ⇒ temporary buffer in kernel space.

* The process in the user space communicates with the pipe created in kernel space and communicates to the other process through pipe, (i.e) through write end and read end of the pipe.
* Using pipes non-related process can't be communicated (limitation of pipe).
* Related process (i.e parent & child processes).

* Reading from empty pipe process got blocked, writing into full pipe process got blocked.

```
#include <stdio.h>
main()
{
 int fd[2];
   if ( pipe(fd) < 0)
   {
   perror("pipe");
    return;
   }
printf( fd[0] = %d   Fd[1] = %d\n", fd[0], fd[1]);

if (fork() == 0)
{
 char a[20];
 scanf("%s", a);
 write (Fd[1], a, strlen(a)+1);
}
else
{
 char b[20]
 read (Fd[0], b, sizeof(b));
 printf (" After reading from pipe = %s\n", b);
}
}
```

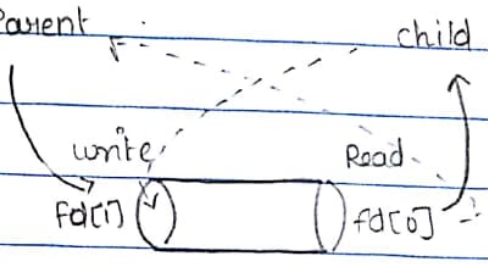
o/p:-

abcdefg
after reading from
pipe : abcdefg

Note:- Data can't be stored temporar temporarily permanently in the pipe.

* pipe created means, a temporary buffer in kernel space is reserved, that acts a channel for communication b/w related process [i.e) parent and child process]

Prap to implement the following 'task'. Parent process will write the data into the pipe, The child process will read the data from the pipe, after reading change the case (data) and write back to the pipe, parent will read and display.




```
#include <stdio.h>
char * casechange(char *);
main()
{ int fd(2);
  if ( pipe(fd) <0)
  {
  perror("pipe");
  return;
  }
if (fork() == 0)
{
  Char a[20];
  read ( Fd[0], a,      sizeof(a);
  printf ("After read from parent = %s", a);
  Casechange(a);
```

```
write ( Fd[1], a, strlen(a)+1);

else
{
  char b[20]
  scanf("%s", b)
  Write ( Fd[1], b, strlen(b)+1 );
  Sleep(1);
  read ( Fd[0], b, sizeof(b));
  printf("After read = %s\n", b);
}
}

void Uppercase( char * S)
{
   for (i=0; S[i]  ; i++)
       S[i] = S[i] - 32;
       return;
}
```

Write a program to find the size of the pipe.

pipe Size : 65536

```
#include <stdio.h>
main()
{
  int count = 0.
```

```
char ch = 'a';
int Fd[2];
pipe (fd);
while(1)
{
if(write (Fd[1], &ch, 1)>0)
  count++;
printf("pipe Size = %d\n", count);
}
}
```

O/P:-

pipe Size = 1

pipe Size = 0

:

:

pipe Size = 65536.

05/09/2017

To find the Size of the pipe using pipe2().

```
#include <stdio.h>
#include <fcntl.h>
main()
{
int Count = 0
char ch = 'a';
int fd[2];
pipe2 (fd, O_NONBLOCK)
while(1)
{
if (write (Fd[1], &ch, 1) != -1)
  Count++;
else
```

```
    break;                                    O/P :-
}                                             Pipe Size :- 65536.
printf ("pipe Size = %d\n", Count);
}
```

Observations :-

1) observe that by closing the write end of file descriptor and Try to read from pipe (what read returns)
2) Observe that by closing the read end of the pipe (file descriptor) and try to write into pipe.

LIMITATIONS OF PIPES :-

* Using pipes only related processes can communicate (i.e child and parent process) and not non-related process.
* There is no name in pipe system

(NAMED PIPES) FIFO :-

FIFO :- First in First out Special file, named pipe.

* A FIFO special file (a named pipe) is similar to a pipe, except that it is accessed as part of the file system.
* It can be opened by multiple processes for reading or writing.
* When processes are exchanging data via the FIFO, the kernel passes all data internally without writing it to the file system.
* Thus, the FIFO special file has no contents on the file system.
* The file system entry merely serves as a reference point so that processes can access the pipe using a name in the file system.
* The FIFO must be opened on both ends (reading & writing) before data can be passed. Normally, Opening the FIFO blocks until the other end is opened also.

FIFO pipe can be created:-

mk fifo - Command.

mk fifo()

named pipes can't store the data.

The size of the named pipe at any point shows zero bytes

Named pipe creation :-

mkfifo - make a FIFO special file (a named pipe)

Header :- #include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);

mkfifo() makes a FIFO Special file with name pathname. Mode Specifies the FIFO's permissions.

On success mkfifo() returns 0, On error returns -1.

1 process.

```
#include <stdio.h>
#include <fcntl.h>
main()
{
 int fd;
    mkfifo("f1", 0664);
    perror("mkfifo");
    printf("hi..\n");
    fd = open("f1", "O_WRONLY");
    if(fd<0)
    {
     perror("Open")
      return;
    }
```

O/p :-

```
mkdir : ---
hi....
```

hello.. (printed only after another process is opened the pipe for reading).

printf ("hello..\n");
}

2 process

```
#include <stdio.h>
#include <fcntl.h>
main()
{
int fd;
mkfifo("fi", 0664);
perror("mkfifo");
printf("hi...\n");
fd= fopen("fi", "O_RDONLY");
if(fd<0)
{
perror("open");
return;
}
printf("hello..\n");
}
```

O/p:-

```
mkdir ---
hi ...
hello...
```

```
#include<stdio.h>
#include<fcntl.h>
main()
{
int fd;
char a[20];
mkfifo("fi", 0664);
perror("mkfifo");
fd = fopen("fi", O_WRONLY);
if (fd<0)
{
perror("Open");
return;
}
while(1)
{
scanf("%s", a);
write(fd, a, strlen(a)+1);
}
}
```

```
#include<Stdio.h>
#include<fcntl.h>
main()
{
int fd;
char a[20];
mkfifo("fi", 0664);
perror("mkfifo");
fd = fopen("fi", O_RDONLY);
if (fd<0)
{
perror("Open");
return;
}
While(1)
{
Read(fd, a, sizeof(a));
printf("read: %s\n", a);
}
}
```

O/P:- mkfifo ---

aaaa
bbbb
cccc
} Through Keyboard.

O/P:- mkfifo ---

read: aaaa
read: bbbb
read: cccc

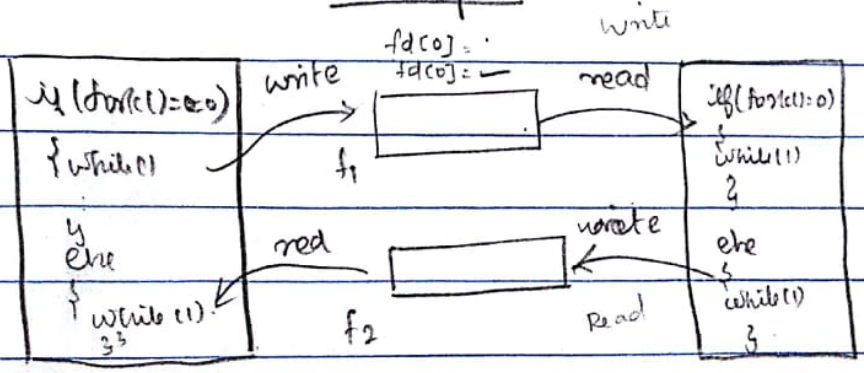
```
#include <stdio.h>
#include <fcntl.h>
main()
{
int fd[2];
char a[20];
mkfifo("f1", 0664);
mkfifo("f2", 0664);
fd[0] = open("f1", O_WRONLY);
fd[1] = Open("f2", O_RDONLY);
while(1)
{ scanf("%s", a);
write(Fd[0], a, strlen(a)+1);
Read(Fd[1], a, sizeof(a));
}while(1)
}
```

```
#include <stdio.h>
#include <fcntl.h>
main()
{
int fd[2];
char a[20];
mkfifo("f1", 0664);
mkfifo("f2", 0664);
fd[0] = open("f1", O_RDONLY);
fd[1] = open("f2", O_WRONLY);
while(1)
{

read(fd[0], a, sizeof(a));
Scanf("%s", a);
write(Fd[1], a, strlen(a)+1);
}
}
```

06.09.17

Implementation of full duplex.

Full duplex



Process 1

```
#include<stdio.h>
#include<fcntl.h>
main()
{
int Fd[2];
mkfifo("f1", 0664);
mkfifo("f2", 0664);
Fd[0] = Open("f1", O_RDWR);
Fd[1] = Open("f2", O_RDWR);

if (fork() == 0)
{ while(1)
{ char a[20];
  Scanf("%s", a);
  write(fd[0], a, strlen(a)+1);
}
else
{ char b[20];
while (1)
{ read(Fd[0], b, Sizeof(b));
  Printf("%s", b);
}
}
```

OSI -> Open Source Interconnection

Process 2

```
#include<stdio.h>
#include<fcntl.h>
main()
{
int fd[2];
mkfifo("f1", 0664);
mkfifo("f2", 0664);
fd[0] = Open("f1", O_RDWR);
fd[1] = Open("f2", O_RDWR);
if (fork() == 0)
{
 While(1)
{ char a[20]; Scanf("%s", a)
write
 read(fd[0], sizeof(a)); strlen(a)+1
 printf("%s", a);
}
}
else
{ char b[20]
Scanf read(fd[0], b, sizeof(b));
Print("%s", a);
```

Limitation of Named -pipe:-

* There is no specific addressing mechanism, when P_1 is writing into pipe, which ever the process is scheduled that process will read data from named pipe.
* Both the process should be alive, when one process is opened in write mode, other process should open in read mode.

always the size of named pipe = 0.

read(fd

fcntl - manipulate the file descriptor (useful record locking, file locking)

Header: #include <unistd.h>
#include <fcntl.h>

int fcntl (int fd, int cmd, ... /*arg*/);

File descriptor flags:-

The following commands manipulate the flags associated with a file descriptor. Currently, only one such flag is defined: FD_CLOEXEC, the close-on-exec flag.

Default value of FD_CLOEXEC is Zero.

fcntl() can be used to set the value of close-on-exec from default of any other value like (1).

```
#include <stdio.h>
#include <fcntl.h>
main()
{
 int fd
 fd= open("temp","O_WRONLY|O_CREAT|O_TRUNC);
 if(fd<0)
 {
  perror("open");
  return;
 }
 write(fd,"hello",5);
 fcntl(fd,F_SETFD,1);  //
 execl("./P1","P1",NULL);
 printf("after execl...\n");
}
```

o/p:-
Set the o/p of P1
cc ./P1.c -o P1.
o/p:- temp
hellochennai
if fcntl is included
o/p:- In temp
hello only

Other process:- (P1)

```
#include <stdio.h>
main()
{ int ret;
ret=write(3,"chennai",7); printf("ret=%d\n",ret);
}
```

pipe2 (fd, O_NONBLOCK).

fd[0], & fd[1] both the ends are non-block.

fcntl (Fd[1], F_SETFL, O_NONBLOCK)

Only write end of the pipe is non-block.

```
#include <stdio.h>
#include <fcntl.h>
main()
{
 int count = 0;
 char ch = 'a';
 int Fd[2];
 pipe(fd);
 fcntl (Fd[1], F_SETFL, O_NONBLOCK);
 while( write (Fd[1], &ch, 1) != -1)
    count++;
 printf("pipe size= %d\n", count);
}
```

07:09:2017 :

Duplicating a file descriptor :

F_DUPFD :- (long)

Find the lowest numbered available file descriptor greater than or equal to arg and make it to be a copy of fd. This is different from dup(2), which uses exactly the descriptor

Record lock (or) FILE LOCKING :

* Writing more than one process into same file, chances of data corruption.
* More than once process can read data from a file but writing into file leads to data corruption.

Lock1.c (Process) Lock2.c

```
#include <stdio.h>
#include <fcntl.h>
main ()
{
char ch[20] = "abcdefghijk";
int fd, i=0;
fd = Open ("temp" O_WRONLY|O_CREAT|O_APPEND);
if (fd<0)
 {
  perror ("Open");
  return;
 }
```

```
printf("writeing into the file");
while(1)
{
write(fd, a+i, 1);
    i++;
 Sleep(2);
3. printf("Writing Completed..\n");
}
```

lock2.c (Process)

```
#include<stdio.h>
#include<fcntl.h>
main()
{
char ch[20]="1234567890";
int fd, i=0;
fd=Open("temp" O_WRONLY|O_CREAT|O_APPEND);
if(fd<0)
{
perror("Open");
return; }
printf("Writing into the file");
while(1)
{
write(fd, a+i, 1);
  i++;
Sleep(2);
} printf("Writing Completed..\n");
}
```

o/p:-

```
temp file
a1b2c3d4e5f6g7h8i9
jok.
```

ADVISORY LOCKING:

* F_GETLK, F_SETLK and F_SETLKW are used to acquire, release, and test for the existence of record locks
* The third argument, lock, is a pointer to a structure that has at least the following fields

```
Struct flock{
    ...
    short l_type ; /Type of lock : F_RDLOCK, F_WRLCK, F_UNLCK/
    short l_whence; / How to interprot l_start ; SEEK_SET,
                                    SEEK_CUR , SEEK_END /
    off_t l_start / Starting offset for lock /
    off_t l_len / Number of bytes to lock /
    pid_t l_pid / PID of process blocking our
                    lock ( F_GETLK Only) /.
    ...
};
```

F_SETLK (struct flock *)

* Acquire a lock (when l_type is (F_RDLCK or F_WRLCK)) or release a lock (when l_type is F_UNLCK) on the bytes specified by the l_whence, l_start and l_len fields of lock
* If a conflicting lock is held by another process, this call returns -1.

F_SETLKW :-

Do for F_SETLK, but if a conflicting lock is held on the file, then wait for that lock to be released.

v.l_len -> 0 means Complete file.
v.l_start -> 0 from the whence
v.l_whence -> SEEK_SET (from beginning)

With Synchronisation :

```
#include<stdio.h>
#include<fcntl.h>
main()
{
struct flock v;
char a[20] = "abcdefghijk";
int fd, i=0
fd = open("temp", O_WRONLY|O_CREAT|O_TRUNC);
if (fd<0)
{ perror("open");
return;
}
v.l_type = F_WRLCK;
v.l_whence = SEE_SET;
v.l_start = 0;
v.l_len = 0;
```

```
printf(" before fcntl.\n");
fcntl( fd, F_SETLKW, &v); (contents in this structure to lock the file where fd referring)
printf (" after fcntl..\n");
fcntl
printf(" Writing into the file..\n");
while (a[i])
{
 write( fd, a+i, 1);
    i++;
  sleep(2);
}
printf ("\nWriting Completed..\n");
v.l_type = F_UNLCK;
fcntl( fd, F_SETLK, &v);
printf (" Lock released..\n");
}
```

Another process with the same above code but content is char ch = "1234567890".

Hence the content in temp file :-

abcdefghijk1234567890.

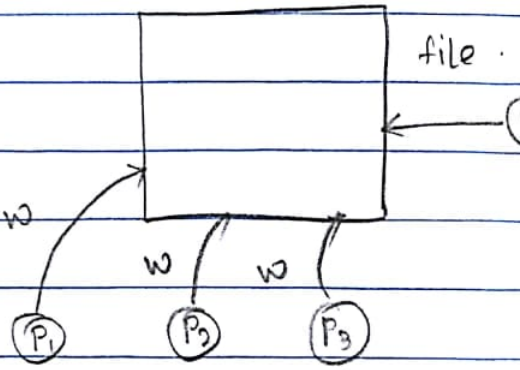
one process writes, other process waits when one completed, the lock is released and the other process accesses the lock of that file.

08-09-2017

When lock is released ?

* when a process is terminated all the locks made by that process are released automatically
* close(Fd) also released.
* Using fcntl function possible

```
V.l_type = F_UNLCK
fcntl(fd, F_SETLK, &V);
```



(This process want to know that which process having lock on file) (using pid).

```
struct flock V1, V2;
int ret;
V.l_type = F_WRLCK
V.l_whence = 0
V.l_start = 0
V.l_len = 0
ret = fcntl(Fd, F_GETLK, &V2)
if(ret == -1)
{
  Fcntl(Fd, F_GETLK, &V2);
  printf("%d\n", V2.l_pid);
}
```

fcntl is one of the Synchronization technique only when the resource is file.

MESSAGE QUEUES:

(i) msgget()
(ii) msgsnd()
(iii) msgrcv()
(iv) msgctl()

Command:- ipcs -q
(interprocess communication) (message Queues)

This command will show any message Queues created or not.

msgget - get a message Queue identifier.

Header file
```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

```
int msgget(key_t key, int msgflg);
```

If successful, the return value will be the message queue identifier (a non-negative integer), otherwise -1.

Key number should not be zero, but it can be greater than zero.

How to create the message Queue?

```
#include <stdio.h>
#include <Sys/msg.h>
main()
{
 int id
 id = msgget(5, IPC_CREAT|0664);
  if(id<0)
  {
  perror("msgget");
   return;
   }
  printf("msg q created).
}
```

O/p :-

msgq created

to see

ipcs -q.

msgrcv, msgsnd – message operations.

int msgsnd (int msqid, Const void * msgp, size_t msgsz, int msgflg);

ssize_t msgrcv (int msqid, void *msgp, size_t msgz, long msgtyp, int msgflg);

The msgsnd() and msgrcv() system calls are used, respectively to send messages to, and receive messages from, a message Queue.

The calling process must have write permission on the message queue, in order to send a message, and read permission to receive a message.

The msgp (messagepointer) argument is a pointer to caller-defined structure of the following general form.

```
Struct msgbuf {                    (In RCV)
        long mtype ;  /* message type must be >0 */
        char mtext[1]; /* message data */
};
```

otherwise it behaviour is undefined.

Every message written in message Queue has a type number. Sending side, message type must be > Zero, bt receiving side the message type may be 0, >0, <0

Both msgsnd() and msgrcv() returns -1 on failure otherwise msgsnd() returns 0 and msgrcv() returns the number of bytes actually copied in to the mtext array.

message queues has the capability to store the message.

```
#include <stdio.h>
#include <sys/msg.h>
#include <string.h>
main (int argc, char *argv[])
{
int id, ret;
struct msgbuf V;
if (argc != 3)
{
printf (". /a.out type, message\n");
return;
}
Pd = msgget (5, IPC_CREAT | 0664);
if (id < 0)
{
perror ("msgget");
return;
}
V.mtype = atoi (argv[1]);
```

```
struct msgbuf()
{ int mtype;
  char mtext[20];
};
```

O/p. Write in M.Q

./a.out 1 Hello

./a.out 2 chennai

./a.out 3 India

```
strcpy(v.mtext, argv[2]);
ret = msgsnd(id, &v, strlen(v.mtext)+1, 0);
printf("ret = %d\n", ret); // for cross verification
}
```

Reading from empty message Queues, the process gets blocked until message is available in message Queue.

0 - In the receiving process type number-0 specifies like FIFO.

```
#include <stdio.h>
#include <sys/msg.h>
#include <string.h>
struct msgbuf{
  int mtype;
  char mtext[20];
  };
main(int argc, char *argv[])
{
  int id, ret;
  struct msgbuf v;
      if(argc!=2)
        {
        printf("./a.out type..\n");
        return;
        }
```

```
id = msgget(5, IPC_CREAT|0664);
if(id<0)
{
perror("msgget-");
return;
}
msgrcv(id, &v, sizeof(v.mtext), atoi(argv[1]), 0);
printf("read = %s\n", v.mtext);
}
```

Reads from message Que

O/P :- ./a.out 0
read : hello
./a.out 0
read : chennai
./a.out 0
read : India

NOTE:-

(i) when a process is sending message into message Queue, msg type must be greater than zero (otherwise behaviour is undefined)

(ii) when a process is receiving a message from Queue type number may be zero (or) greater than zero (or) less than zero.

09/09/17

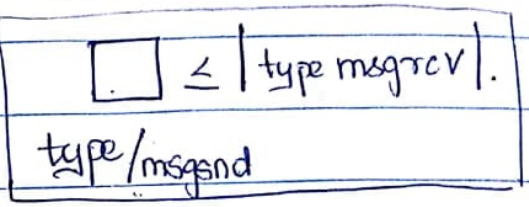
The argument type specifies the type of message requested as follows. (Receiving Side)

* If msgtype is 0, then the first message in the queue is read.

* If msgtype is >0, then the first message in the queue of type msgtype is read, unless MSG_EXCEPT was specified in msgflag, in which case the

first message in the Queue of type not equal to msgtyp will be read.

* The msgtype is < 0, then the first message in the Queue with the lowest type less than or equal to the absolute value of msgtyp will be read.



Sending side types must be > 0.

FLAGS:

IPC_NOWAIT :- Return immediately if no message of the requested type is in the Queue.

IPC-NOWAIT FLAG is similar to the O_NONBLOCK flag in the piper.

MSG_EXCEPT :- Used with msgtyp greater than 0 to read the first message in the queue with message type that differs from msgtyp.

MSG_NOERROR:-

To truncate the message text if longer than msgsz bytes.

Note: Reading the partial message is not possible, if we want read we need to read completely. (Partially reading the message not possible).

Using MSG_NOERROR flag we can partially read, bt when the message is read partially from the message queue apart from the partial connect remaining all other content also lost in message Queue.

Reading partial messages from Queue is not advisable, chances of data

Wap to implement the full duplex Communication using message Queues (4ss)

To delete the message Queue. through Command

(i) ipcrm -q O/Id.

(ii) ipcrm -Q Keynumber.

Msgctl - message Control Operations.

int msgctl (int msgid , int cmd , Struct msqid_ds *buf).

msqueue id

datastructure

Valid Values for cmd are

IPC_STAT :-

Copy information from the kernel data structure associated with msqid into msqid_ds structure pointed to by buf.

struct msqid_ds V;

msgctl (id , IPC_STAT, &V);

Once a message Queue is created, Kernel will maintain a separate data structure for each Queue.

IPC_SET :-

Write the Values of some members of the msqid_ds structure pointed to by buf to the kernel data-Structure associated with this Queue.

exactly opposite to IPC_STAT.

Changing the data in the kernel, by first writing into the st

IPC_RMID :- Immediately remove the message Queue.

msgctl (id, IPC_RMID 0).

remove the message Q

to delete the message Queue using IPC_RMID

LIMITATIONS OF MESSAGE QUEUES :-

* Message Queues are connectionless, there is no way to know that how many processes are attached to the particular message Queue (kernel not maintaining this information.

* Message Queues are slow in the way of communication, more burden on kernel because kernel should create a unparate datastructure for each message Queue and when a process is requesting a message with a specific type, kernel will check type of message available in Queue or not if available, it will send the msg and clears in Queue (for every message type checking).

ONE OF THE SLOWEST IPC COMMUNICATION.

11/09/17

SHARED MEMORIES.

* Among all shared memory it is the fastest IPC Communication.

* Shared Memory is asynchronous IPC mechanism

* There is no concept of empty shared memory and full shared memory.

* Shared memory implemented with the help of memory management (virtual add).

* Calls used in shared memory
    * shmget()
    * shmat()
    * shmdt()
    * shmctl()

callshmget() to create a new shared memory segment or certain the identifier of an existing segment (i.e) one created by another process. This call returns a shared memory identifier for use in later calls.

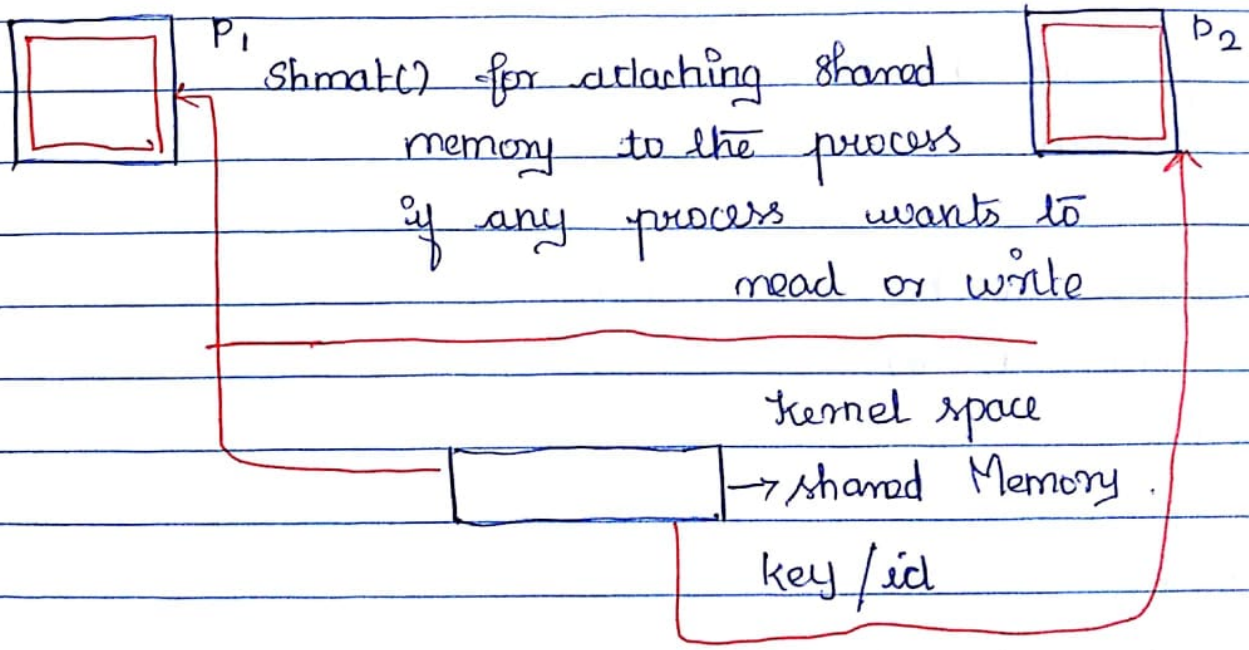
shmget (key_t key, size_t size, int shmflag);

(returns shared memory segment identifier on success on -1 on error).

Use shmat to attach the shared memory segment, that is make the segment part of the virtual memory of the calling process.

void *shmat (int shmid, const void * shmaddr,
int shmflg);

*Return addr at which shared memory is attached on success (or) (void *) -1 on error.



The process (P1) and (P2) are communicated by using shared memory

ret address of shmget

id = shmget (11, 20, IPC_CREAT | 0644);

shmat (id, 0, 0)

Command :- ipcs -m → displays the shared memory

```
#include<stdio.h>
#include<sys/shm.h>
main()
{
 int id;
 int *p,i;
 id = shmget (11, 20, IPC_CREAT|0664);
 if (id<0)
 {
  perror("shmget");
  return;
 }
printf ("Segment Created...\n");
P=shmat(id,0,0);
while (1)
{
 P=*p;
 printf ("%d\n",i);
  Sleep(1);
 }
}
```

```
#include<stdio.h>
#include<sys/shm.h>
main()
{
 int id;
 int *p, i=0;
 id = shmget(11,20, IPC_CREAT|0664);
 if(id<0)
 {
 perror("shmget");
 return;
 }
 printf("Segment Created...\n");
 p= shmat(id,0,0);
 while(1)
 {
 *p=i;
 i++;
 Sleep(1);
 }
}
```

At this point, the shared memory segment can be treated just like any other memory available to the program. In order to refer to the shared memory, the program uses the address value returned by the shmat() call, which is a pointer to the start of the shared memory segment

in the processes Virtual address space.

Call shmdt() to detach the shared memory segment. After this call, the processes can no longer refer to the shared memory. This step is optional, and happens automatically on process termination

Call shmctl() to delete the shared memory segment the segment will be destroyed by only after all currently attached process have detached it

Ipcrm -m id } shortcut key for
Ipcrm -M key } deleting shared memory

Shmdt :-

int shmdt (const void * shm addr);

shmctl → shared memory ctrl.

int shmctl (int shmid, int cmd, struct shmid_ds *buf);

shmctl (id, IPC_RMID, 0)

Shmat 2nd & 3rd argument :-

Specifying a non-null value of shmaddr (ie) either the second or third option

list above) is not recommended for the following reasons

* It reduces the portability of an application. An address valid on one UNIX implementation may be invalid (or) another

* An attempt to attach a shared memory segment at a particular address will fail if that address is already in use

```
p = shmat (id, 0, 0)
```

* To attach a shared memory segment for read only access, we specify the flag SHM_RD only in shmflag
* Attempts to update the contents of a read-only segment result in a segmentation fault.

* If Shm_RD only is not specified, the memory can be both read and modified.

```
p = shmat (id, 0, 0)
```

12/09/17

Command :- man pagesize.

getpagesize - get memory page size.

```
#include <unistd.h>

int getpagesize(void);
```

The function getpagesize() returns the number of bytes in a memory page, where "page" is a fixed-length block.

20 bytes of memory means not exactly 20 bytes, it is rounded to the pagesize.

SEMAPHORES :-

* Semaphore is an object, known by the kernel, used by the processes for synchronisation
* Semaphore is not for process communication it is for process synchronisation.
* Semaphore is working based on the principle of Test and Set condition.

Semget()

Semctl()

Semop()

Command :- ipcs -s

to see whether any Semaphore is there are not.

ipcs -a

will display all the limits of shared memory, Semaphores etc..,

Semget - get a Semaphore set identifier.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

Headerfile.

```
int Semget (key_t key, int nsems, int semflg);
```

If successful, the return value will be the Semaphore set identifier (a non-negative) otherwise -1 is returned

Semaphore number (int nsems) depends on the number of Resources.

Semaphore number starts from 0, 1, 2 like array index hence it is called as Semaphore Array.

Default value of semaphore is 'Zero'.

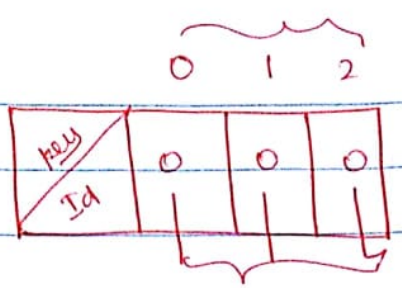


fig. 1.

```
#include <stdio.h>
#include <sys/sem.h>
main()
{
int id;
id = Semget (11, 3, IPC_CREAT|0664);
if (id<0)
 {
 perror("Semget");
 return;
 }
printf("Semaphore array Created..\n");
}
```

O/P :- refer fig. 1.

Semctl - Semaphore Control Operations

int Semctl (int Semid, int Semnum, int cmd, ...);

on failure => -1
on success => depends on the command given

Wap to find the present value of semaphore

```
#include <stdio.h>
#include <sys/sem.h>
main( int argc, char * argv[])
{
  int index, ret, id
    if( argc != 2)
    {
    printf(" ./a.out Semnum .\n");
    return;
    }
id = Semget( 11, 3, IPC_CREAT | 0664);
 if ( id < 0)
 {
 perror( "Semget ");
  return;
 }
index = atoi [argv[1]);
ret = Semctl ( id, Index, GETVAL);
printf [ret = %d\n", ret);
}
```

O/P:-

```
./a.out 0
ret=0
./a.out 1
ret=0
./a.out= 2
ret=0
./a.out = 3
ret = -1.
```

Wap to set the value of semaphore.

```
#include<stdio.h>
#include<sys/sem.h>
main(int argc, char *argv[])
{
 int inden, id, val;
  if(argc!=3)
  {
  printf("./a.out semnum Semval);
  return;
  }
 id = semget(11, 3, IPC_CREAT|0664);
  if(id<0)
  {
  printf("Semget");
  return;
  }
inden = atoi(argv[1]);
val = atoi(argv[2]);

semctl(id, inden, SETVAL, Val);
}
```

O/p:-

./a.out 0 1
./a.out 1 2
./a.out 2 3

Using the previous program we can cross verify the values of Semaphore

Semop - Semaphore Operations

int Semop(int semid, struct sembuf *Sops,
unsigned nsops);

Semop will either allow a process or block the process.

```
Struct Sembuf
{
unsigned short sem_num; /* Semaphore number */
short sem_op ; /* Semaphore Operation */
Short Sem_flg /* Operation flags */
```

Semop() performs operations on selected Semaphores in the set indicated by Semid. Each of the nsops elements in the array pointed to by Sops specifies an operation to be performed on a single semaphore.

1 semaphore is enough for a single resource.

Sem_op value is linked with Semaphore Value

Sem_op	Sem value
0	[0] → resource is free, Process Allo
0	not zero → " is busy, Process blocked

```
#include <stdio.h>
#include <sys/sem.h>
main()
{
  int id; struct sembuf V;
  id = semget(11, 3, IPC_CREAT|0664);
   if (id<0)
   {
   perror("Semget");
    return;
   }
  V.sem_num = 0;
  V.sem_op = 0;
   V.sem_flg = 0;
  printf("before..\n");
   semop(id, &V, 1);
  printf("After..\n");
}
```

O/P

if Semaphore Value is 0
before...
After..
[process is allowed]
if Semaphore Value is !=0
before...
[process is blocked]

Rightnow those many Semphore Using

Seq-num = 0 → 0/1/2 Semaphore number

Seq-op = 0 → | 0 | Semaphore Value

13/09/17

If Sem_op is zero, the process must have read permission on the Semaphore set. This is a "wait-for-zero" Operation. If Semval is zero, the operation can immediately proceed.

Record locking Using Semaphores:

```
#include <stdio.h>
#include <sys/sem.h>
#include <fcntl.h>
main()
{
 int id, fd, i =0;
 char a[20] = "abcdefghijk";
 Struct Sembuf V;
 fd = open("temp" O_WRONLY|O_CREATE|O_APPEND);
  if (fd<0)
  {
   perror("open");
   return;
  }
 id = Semget (11, 3, IPC_CREAT|0664);
  if (id<0)
  {
   Perror ("Semget");
   return;
  }
```

```
V.Sem_num = 0;
V.Sem_op = 0;
V.Sem_flg = 0;
printf("Before Resource access..\n");
  Semop(id, &V, 1);
  Semctl( id, 0, SETVAL, 1);
printf("Entered into the critical Section of code..\n");
while (a[i])
{
  write(fd, a+i, 1);
    i++; Sleep(2);
}.
printf("Writing Completed ...\n");
printf("Release the Resource..\n");
  Semctl( id, 0, SETVAL, 0);
printf("Resource Released..\n");
```

→ Critical Section of code.

Other process with same code, char c[20] is different data.

CRITICAL SECTION OF CODE:

It is a part of the process, where a common resource is accessed, when one process critical section of code is executing other process critical section code should note execute.

Note:: each process

Separate for ↑ ⇐ Sem_op can be +ve, -ve, 0 but Sem_value can't be -ve ⇒ Common for all process

```
#include<stdio.h>
#include<sys/sem.h>
#inlude<fcntl.h>
main()
{
 int id, fd, i=0;
 Struct sembuf v;
 char a[20]= "abcdefghijk";
 fd= Open("temp", O_WRONLY|O_CREAT|O_APPEND);
 if(fd<0)
 {
  Perror("open");
  return;
 }
 id= Semget(11, 3, IPC_CREAT|0664);
  if(id<0)
  {
   perror("Semget");
  } return;
 v.Seg_num=0;
 V.Seg_op= 0;
 V.Sem_flg= IPC_NOWAIT);
 printf("before...\n");
 ret= Semop(id, &v, 1);
  if(ret==-1);
  {
  printf("Resource busy \n");
                                printf("Semop ret
                                    Value=%d\n",ret);
                                return;
                                }
```

If Sem_op is a +ve int :

* If Sem_op is a positive integer, the operation adds this value to the Semaphore value
* This operation can always proceed - it never forces a process to wait.

```
# include <stdio.h>
# include <sys/sem.h>
# include <
main()
{
int id
id = semget(11, 3, IPC_CREAT|0666); Sem_val is
if (id<0)
{
perror("Semget");
return;
}
v.sem_num = 1;
v.sem_op = 1;
v.sem_flg = 0;
printf(" before .. \n");
semop (id, &v, 1);
printf ("after ... \n");
}
```

O/p :-

for every process execution, the Sem_val is incremented on the basis of Sem_op value.

Note:- Sem_op = 1 acts as the master process and it will never end ever blocks the process. The Sem_op value is added to Semaphore value. This is useful for setting Sem_Val when required.

If Sem_op is a -ve int :-

* If Sem_op is less than zero, the process must have alter permission on the Semaphore set, If Sem_val is greater than or equal to the absolute value of Sem_op, the operation can proceed immediately, otherwise process got blocked.

```
if (Sem_val >= |Sem_op|)
    proceed ;
else
    block ;
```

* If sem_op is a -ve number, this operation subtracts this value from the semaphore value.

* If multiple number of resources are available, this Sem_op negative number makes the resource available for all the process (on the basis of Semval)

Semval = Allows that many number of processes.

Resource taken, semval is reduced.

(Ex:- Bus ticket booking)

```
#include <stdio.h>
#include <sys/sem.h>
main()
{
 int id, ret;
 struct sembuf v;
 id = Semget(11,3,IPC_CREAT|0664);
  if (id<0)
   {
    perror("Semget");
    return;
   }
 v.sem_num = 1
 v.sem_op = -1
 v.sem_flg = 0
 Printf("before...\n");
 Semop(id, &v, 1);
 printf("after...\n");
```

O/p:-

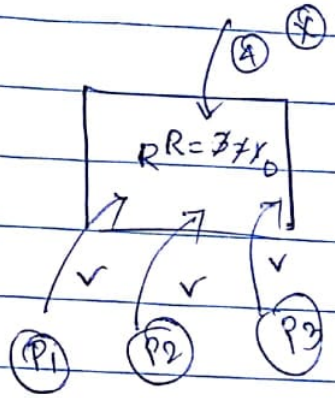
for every process execution, Sem_val is reduced on the basis of Sem_op value.

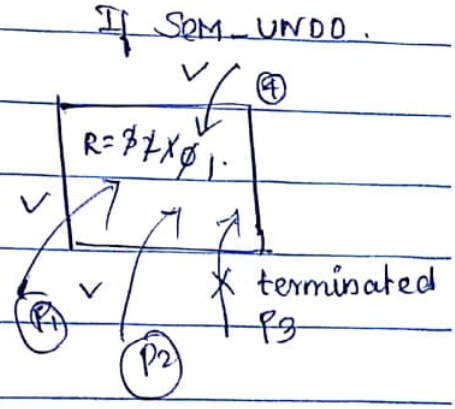
Difference b/w binary Semaphores Vs Counting Semaphores

15/09/17

SEM_UNDO:

```
#include<stdio.h>
#include<sys/sem.h>
main()
{
int id, ret;
struct sembuf v;
id = Semget (11, 3, IPC_CREAT|0664)
if(id<0)
{
perror("Semget");
return;
}
V.Sem_num = 1;
V.Sem_op = -1;
V.Sem_flg = SEM_UNDO;
printf("before...\n");
Semop(id, &v, 1);
printf("after...\n");
while(1);
}
```





Description of SEM_UNDO:-

If this flag is used during process execution Sem_op value is Subtracted from Sem_val, but one process

terminated, Sem.op value is added to the Semval.

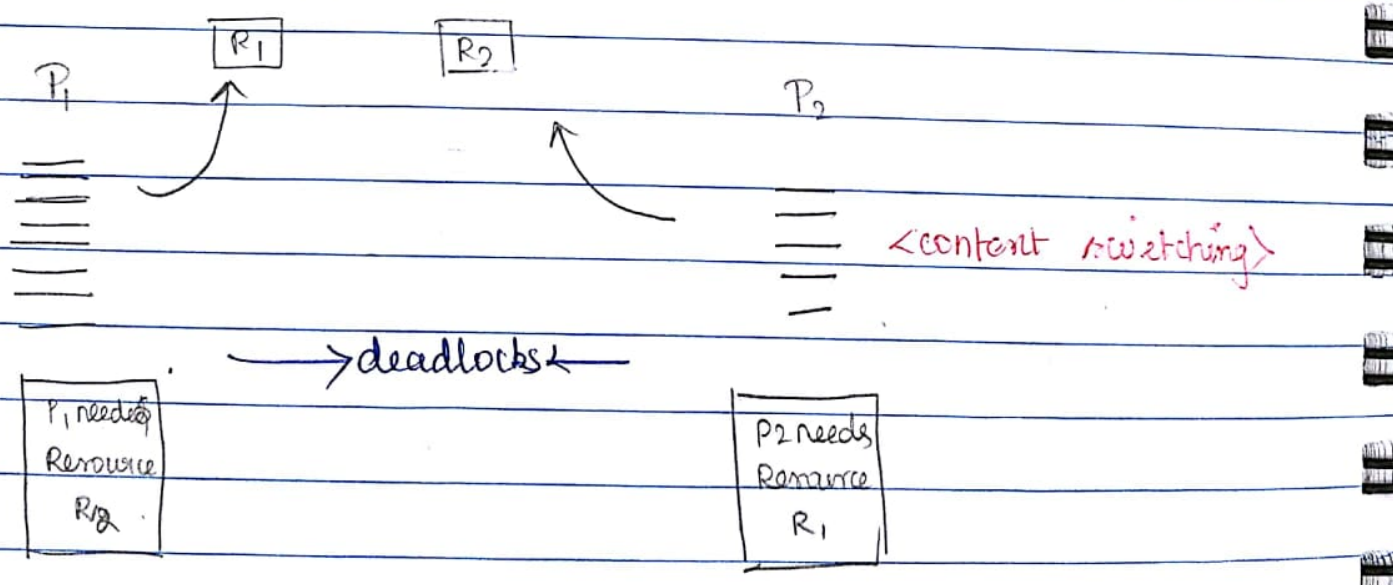
Binary VS Counting Semaphores:-

Binary Semaphores:-

If the Semaphore value is b/w 0 and 1 is called as binary Semaphores (wait for Zero Synchronisation).

Counting Semaphore:-

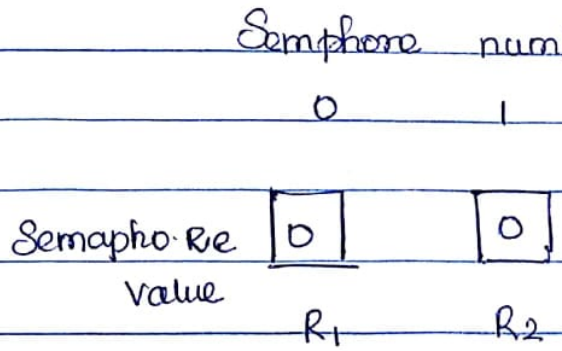
Initally set the Semaphore value to max, count value, represents the maximum number of processes allowed to access a resource.



Process waiting for the resources taken by another process.

Deadlock:

In an Operating system, a deadlock is a situation which occurs when a process or thread enters a waiting state because a resource requested is being held by another waiting process, which in turn is waiting for another resource held by another waiting process. If a process is unable to change its state indefinitely because the resources requested by it are being used by another waiting process, then the system is said to be in a deadlock.




```
Struct sembuf V[2]
{
V[0].Sem_num = 0 ;  |  V[1].Sem_num = 1 ;
V[0].Sem_opa = 0 ;  |  V[1].Sem_op = 0 ;
V[0].Sem_flg = 0 ;  |  V[1].Sem_flg = 0 ;
```

Semop(id, v, 2) → How many Semaphores need to check before entering into critical section of code.

After checking whether both Sem_values are '0', the process is allowed, after entering into Critical Section, using semctl function change the Sem_value of both the Semaphore as '1' so that, any other process can't access the resource till the first process gets completed.

What is Race Condition?

THREADS

* Threads is a Light weight process / part of a process
* One process may have multiple Threads, each thread is Scheduled independently.
* All the threads belongs to the same process will share many attributes are Common (code, data, heap).
* Stack memory separate for each thread.
* Content switch time b/w two Threads will take less time than the Content Switch time b/w two processes (because many attributes are common).
* Communication between two processes needs IPC mechanism.
* Communication between two threads no need of any IPC (Sharing of data b/w threads easy).

16/09/17

Content switch time minimized

Common Attributes for threads belong to same process:-

* PID, PPID
* Controlling terminal
* Open file descriptor
* CPU_time
* Resource limits
* Nice values.

Thread Specific attributes are:-
(In TCB)

* Thread Id
* Thread Specific data
* Stack
* errno Variables.

Like PCB there is TCB (Thread control Block).

THREADS CREATION:-

Pthread_create - Create a new thread.

#include <pthread.h> => Header file.

int pthread_create (pthread_t *thread, Const pthread_attr_t * attr, void * (* start_routine) (void *), void *arg);

pthread_t * thread → Thread Id.

Const pthread_attr_t * attr → Thread attributes

[void *(* Start_routine) (void *)] → Thread code start point

void * arg → optional data passing.

Compile and link with -pthread

The pthread_create() function starts a new thread in the calling process. The new thread starts execution by invoking Start_routine();

Command to display thread.

ps -eL | grep pts/0

will display like process id's only, but the real thread id is stored in pthread_t Variable

Threads creation :

```
#include <stdio.h>
#include <pthread.h>
void * thread1(void *p)
{
printf("Hello...\n");
while(1);
}
main()
{
pthread_t t1;
pthread_create(&t1, 0, thread1, 0);
printf("thread id = %u\n", t1);
while(1);
}
```

O/P :-

Thread id = 3071597104

hello...

if 4th argument is used.

In thread1.

Instead of '0'.

In thread1 (optional)

```
#include <stdio.h>
#include <pthread.h>
void * thread1(void *p)
{
printf("In thread1...\n"); printf("%s\n",(char *)p);
while(1);
}
```

```
void *thread2(void *p)
{
printf("in thread2....\n");
printf("%s\n", (char*)p);
while(1);
}
main()
{
pthread_t t1,t2;
pthread_create(&t1, 0, thread1, "Hello thread1");
pthread_create(&t2, 0, thread2, "Hello thread2");
printf("Thread1 id = %d\n", t1);
printf("Thread2 id = %d\n", t2);
while(1);
}
```

O/p:- thread1 id = x
thread2 id = y
In thread1...
Hello thread1...

In thread2...
Hello thread2...

Note: If the main process under which threads created is terminated, the threads also gets terminated. (No concept of parent & child).

Pthread_self() → to get the threadid, which thread we are calling that function

Pthread_self() - Obtain ID of the calling thread.

Headerfile -> #include <pthread.h>

pthread_t pthread_self(void);

The pthread_self() function returns the ID of the calling thread.

```
void * Thread1(void *p)
{
printf("in thread1 .\n");
printf("%u\n", pthread_self());
while(1);
}
```

o/p :-

In thread 1

xxxxxxxxx (Thread id)

pthread_join - Join with a terminated thread

int pthread_join(pthread_t thread, void **retval);

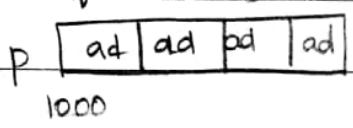
* The pthread_join() function waits for the thread specified by thread to terminate.
* If the thread has already terminated, the pthread_join() returns immediately.

```
#include <stdio.h>
#include <pthread.h>
void *Thread1 (void *p)
{
printf (" in thread1 ... \n");
Sleep(10);
printf (" Thread1 exit ... \n");
}
void * Thread2 ( void * p)
{
printf ("In thread 2 ... \n");
Sleep(5);
printf (" Thread2 exit ... \n"); pthread_exit (" thread 2 bye bye \n");
}
main ()
{
char *p;
pthread_t t1,t2;
pthread_create (&t1, 0, thread1, 0);
pthread_create (&t2, 0, thread2, 0);
pthread_join (t2, &p);
printf ("%s\n", p);
}
```

O/p :-

```
In thread2 ...
Thread2 exit
Thread 2 bye bye
```

Status collected on itself address stored in p, string const



pthread_exit : terminate calling thread.

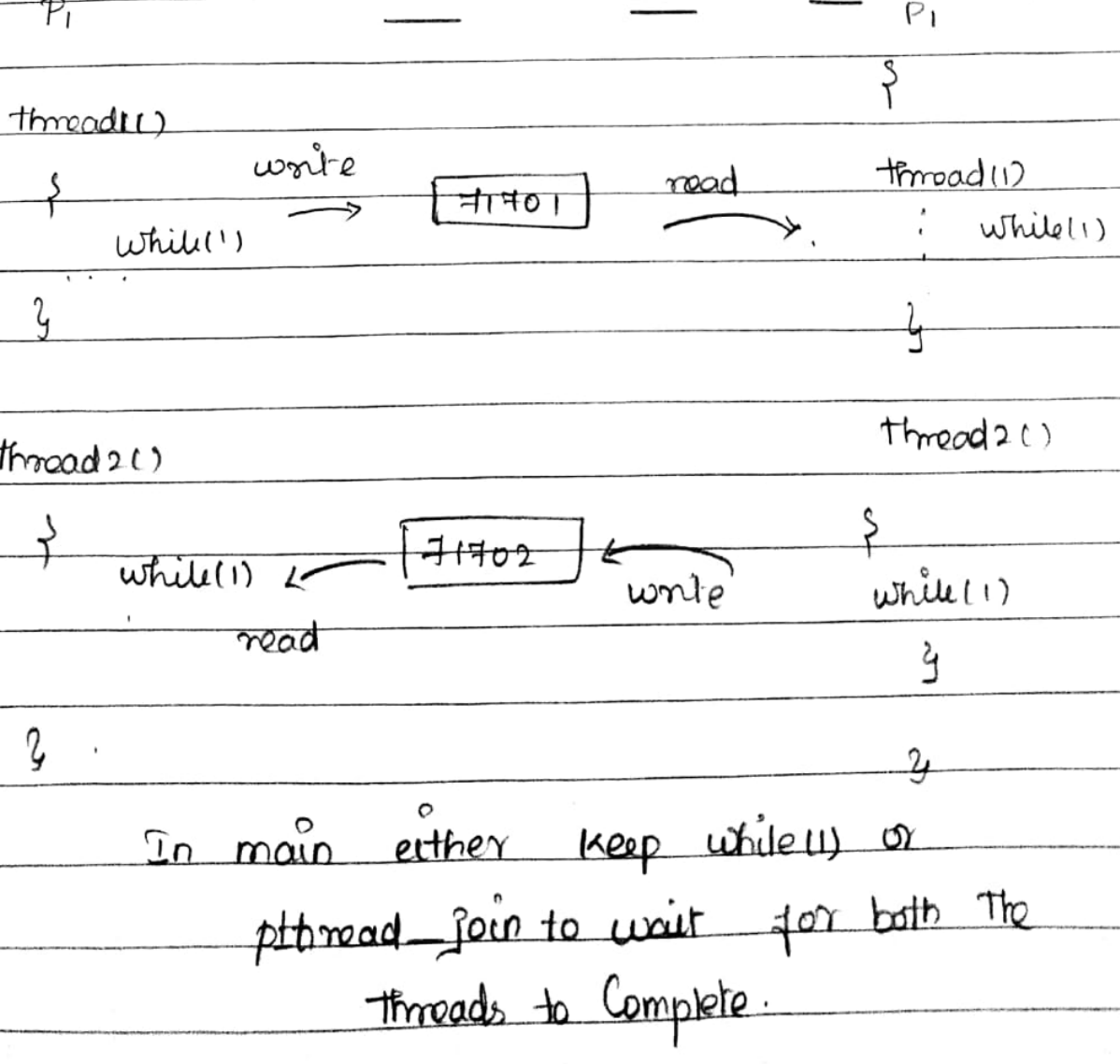
void pthread_exit (void * ret val);

The pthread_exit() function terminates the calling thread and returns a value via retval.

pthread_exit(0) → not interested to send any status

pthread_join (threadid , 0) → Not interested to collect the status of the terminated thread.

FULL DUPLEX COMMUNICATION USING THREADS



In main either keep while(1) or pthread_join to wait for both the threads to Complete.

Implementation of full duplex communication using threads.

```
#include <stdio.h>
#include <pthread.h>
#include <fcntl.h>
#include <string.h>
void *thread1 (void *p)
 {
   int fd;
   char a[20];
  fd= open("f1", O_RDWR);
  while(1)
  {
   scanf("%s ", a);
   write(fd, a, strlen(a)+1);
  }
 }
void *thread2 (void *p)
 {
  int fd;
  char a[20];
  fd = open("f2", O_RDWR);
  while(1)
  {
   read(fd, b, sizeof(b));
   printf("%s\n",b);
 }
```

```
main()
{  pthread_t t1, t2;
   mkfifo ("f1", 0664);
   mkfifo ("f2", 0664);
   pthread_create(&t1, 0, thread1, 0);
   pthread_create(&t2, 0, thread2, 0);
   while(1);
}
```

same program for other process but, the process should read inside write and vice-versa. (refer the diagram).

(Interchange "f1" and "f2" in 2 threads).

18/9/17.

int pthread_attr_init (pthread_attr_t *attr);

```
pthread_attr_t V;
pthread_attr_init(&V);
```
} Example.

It Initialises the thread attributes object pointed to by attr with default attributes values.

pthread_attr_setstacksize, pthread_attr_getstacksize - set / get stack size attribute in thread attributes object.

29.9 Process Vs Threads (629)

```
int pthread_attr_setstacksize (pthread_attr_t * attr,
                               size_t stacksize);
int pthread_attr_getstacksize (pthread_attr_t * attr,
                               size_t *stacksize);
```

```
int stack;
pthread_attr_t V;
pthread_attr_init (&V);  // upto this same for all attributes
pthread_attr_getstacksize (&V, &stack)
    stack = stack - 1000;
pthread_attr_setstacksize (&V, stack)
pthread_create (&t1, &V, thread1, 0);
```

for which Resource we need to change the default value, we should use different functions for a particular attribute.

↓

[Same in the final (To create the thread with modified thread specific attributes]

g [2] //10

Thread1 Resc [ ] Thread 2

```
{
printf("%d", g, 0) => 0
```

g+1

```
PF("%d", g) => 10
}
```

```
{
printf("%d", g) => 1
```

g=10

```
}
```

Protecting Accesses to Shared Variables: Mutexes.

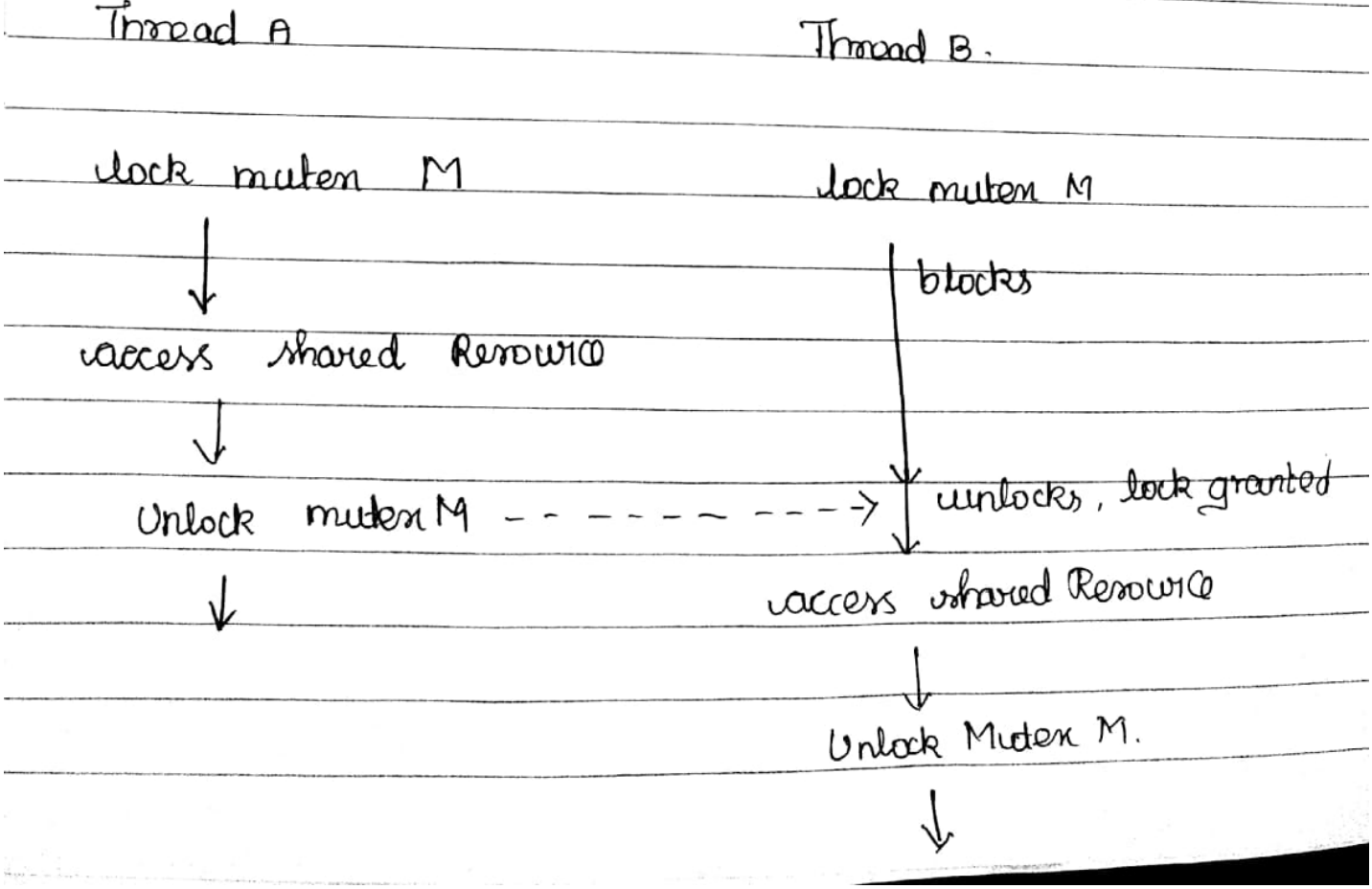
* One of the principal advantages of threads is that they can share information via global Variables.
* However this easy sharing comes at a cost: we must take care that multiple threads do not attempt to modify the same Variable at the same time, or that one thread doesn't try to read the value of a variable while another thread is modifying it.
* The term critical section is used to refer to a section of code that accesses a shared resource and whose execution should be atomic; that is, its execution should not be interrupted by another thread that simultaneously accesses the same shared resource.

Mutexes is one type of synchronisation technique for Threads.

* To avoid the problems that can occur when threads try to update a shared Variable, we must use a mutex (short for mutual exclusion) to ensure that only one thread at a time can access the Variable.

* A mutex has 2 states : locked & unlocked. At any moment, at most one thread may hold the lock on a mutex. Attempting to lock a mutex that is already locked either blocks or fails with an error.
* When a thread locks a mutex, it becomes the owner of that mutex. Only the mutex owner can unlock the mutex.
* Each thread employs the following protocol for accessing a resource.
  - lock the mutex for the shared resource
  - access the shared resource ; and
  - Unlock the mutex.

USING A MUTEX TO PROTECT A CRITICAL SECTION.



Accessing Common data without Synchronisation :

```
#include <stdio.h>
#include <pthread.h>
int g=0;
void * thread1 (void *p)
{ g++;
printf("In thread 1 g = %d\n",g);
 Sleep(2);
printf("In thread1 g = %d\n",g);
}
void * Thread2 (void *p)
{
printf("In thread 2 g = %d\n",g);
   g=10;
   Sleep(1);
    g++;
printf("In thread 2 g = %d\n",g);
}
main()
{
pthread_t t1,t2;

pthread_create(&t1, 0, thread1, 0);
pthread_create(&t2, 0, thread2, 0);
while(1);
}
```

o/p:-
In thread 1 g = 1
In thread 2 g = 1
In thread 2 g = 11
In thread 1 g = 11
but expected

In thread 1 g = 1 → Sleep
In thread 1 g = 1
In thread 2 g = 1 → Sleep.
In thread 2 g = 11
So proper Synchronization required

Synchronisation with Muten :-

```
#include<stdio.h>
#include<pthread.h>
    int g=0;
pthread_muten_t m1 = PTHREAD_MUTEX_INITIALIZER.
void * Thread1 (void *p)
 {
  pthread_muten_lock(&m1);
    g++;
  printf("In thread1 g= %d\n", g);
   Sleep(10);
  printf("In thread1 g= %d\n", g);
   pthread_muten_unlock(&m1);
 }
void * thread2 (void *p)
 {
  pthread_muten_lock(&m1);
   printf("In thread2 g= %d\n", g);
    g=10;
   sleep(1);
    g++;
   printf("In thread2 g= %d\n", g);
   pthread_muten_unlock(&m1);
 }
```

```
main
{
 pthread_t t1, t2
 pthread_create(&t1, 0, thread1, 0);
 pthread_create(&t2, 0, thread2, 0);
 while(1);
}
```

Write a program to implement ps -elgrep | wc

```
execlp("ps", "ps", "-e", NULL);
pipe(fd)
close(1)
dup2(1, fd[1])

close(0)
dup2(0, fd[0])
read(fd[0], b, sizeof(b))
strstr(
```

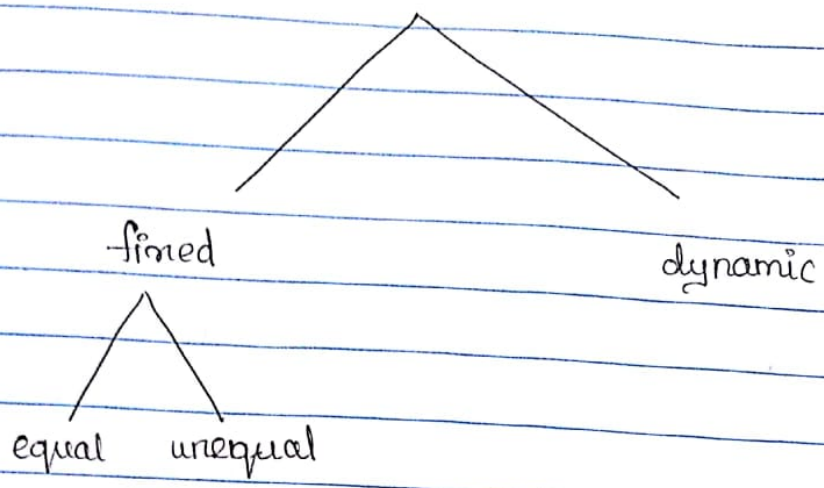
19/9/17. Threads Vs ~~to~~ Process

Semaphores	Mutexes.
* For process synchronisation, usually semaphores are used.	Mutexes are useful for threads synchronisation
* In case of semaphores when one process accessing the resource by making semaphore value is one, any process can make semaphore value zero.	In case of threads (Mutexes) if a thread makes a lock, it becomes the Owner of the mutex and only that thread can release the lock, no other threads can't release the lock.

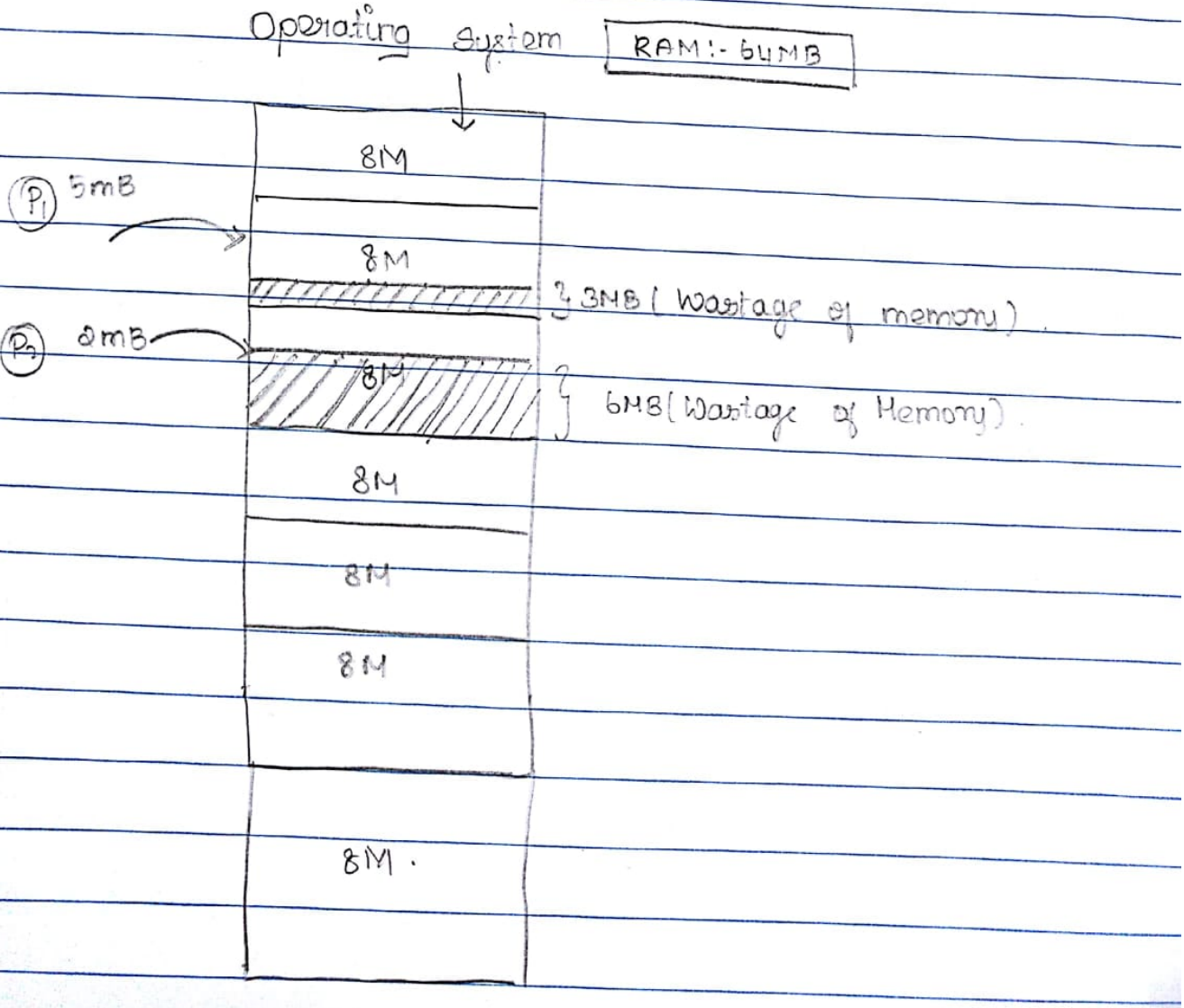
MEMORY MANAGEMENT:-

The principal operation of Memory management is to bring processes into main memory for execution by the processor. In almost all the modern multiprogramming systems, this involves a sophisticated scheme known as Virtual memory.

Memory Partitioning



Fixed equal Memory Partition :-



fixed:- No. of partitions are fixed (8)
Equal:- All partition size are equal (8MB) each.

One partition, one process,

Internal fragmentation

(RAM) Main memory utilization is extremely inefficient. Any program, no matter how small, occupies an entire partition.

In our process, there may be a program whose length is less than 2 Mbytes, yet it occupies an 8-Mbyte partition whenever it is swapped in.

The phenomenon, in which there is wasted space internal to a partition due to the fact that the block of data loaded is smaller than the partition, is referred to as Internal fragmentation.

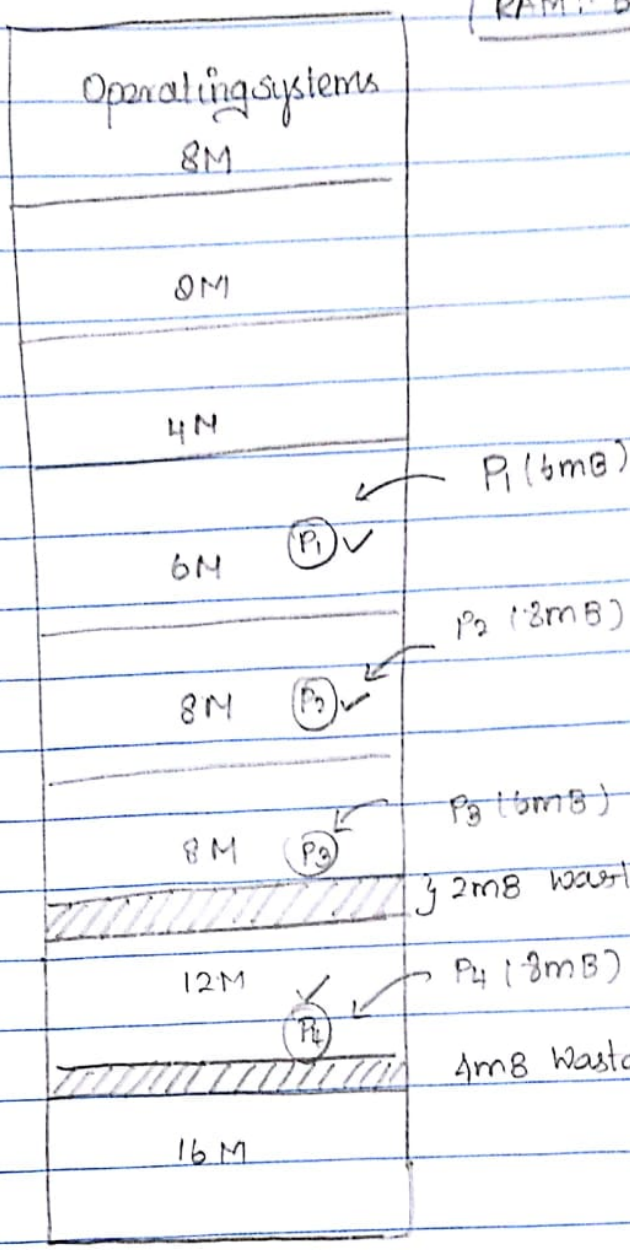
FIXED UNEQUAL MEMORY PARTITION:-

fixed:- No. of partitions are fixed (8)

Unequal:- All partition size are unequal.

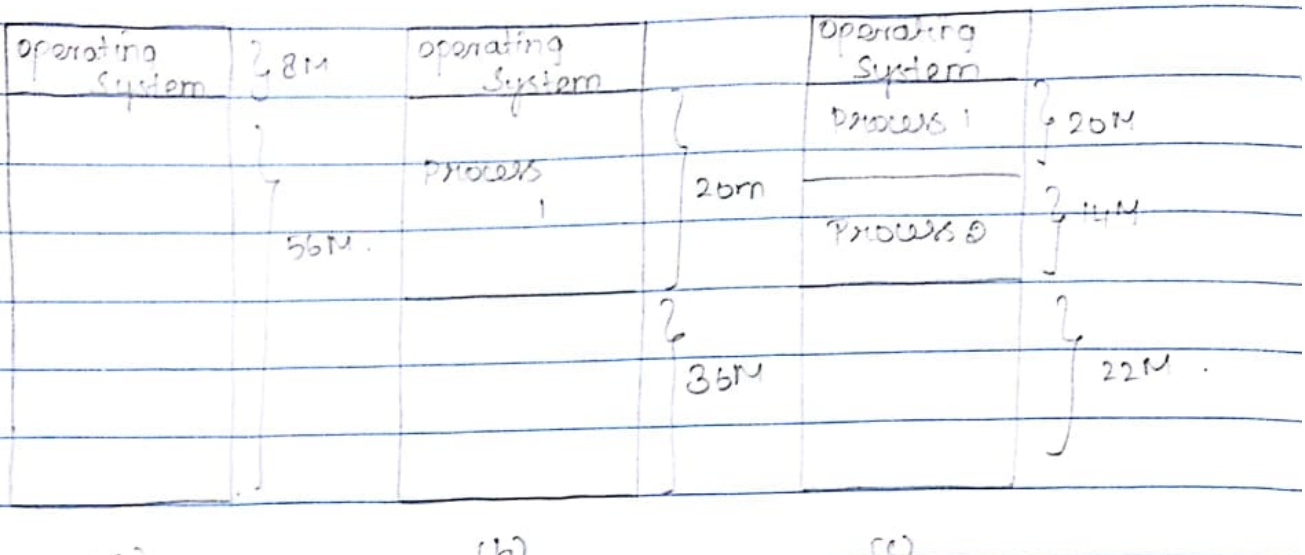
Fixed - Unequal Memory Partition

RAM: 64 MB

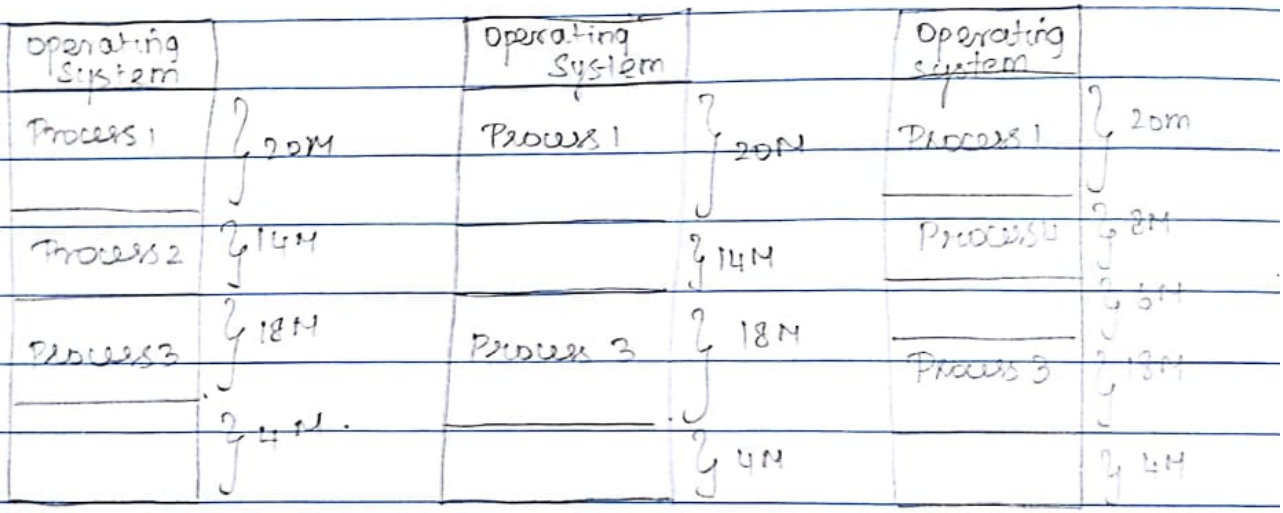


* The number of partitions specified at system generation time limits the number of active (not suspended) processes in the system.
* Because partition sizes are pre-set at system generation time, small jobs will not utilize partition space efficiently.

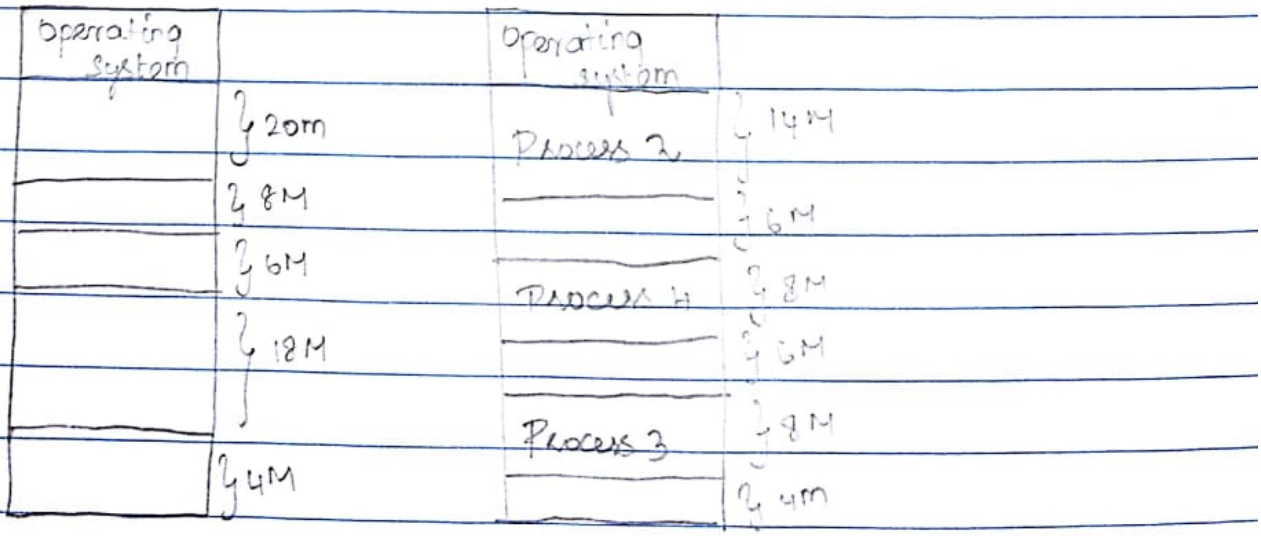
20/9/17 DYNAMIC PARTITIONING



(a) (b) (c)



(d) (e) (f)



(g) (h)

Dynamic partition is good for initial process. Dynamic partition are decided when the process is loaded.

* As this example shows, this method starts out well, but eventually it leads to a situation in which there are a lot of small holes in memory.
* As time goes on, memory becomes more and more fragmented, and memory utilisation declines. This phenomenon is referred to as external fragmentation.

Solution:

One technique for Overcoming external fragmentation is Compaction. From time to time, the OS shifts the processes so that they are contiguous and so that all of the free memory is together in one block.

Disadvantages

The difficulty with compaction is that it is a time-consuming procedure and wasteful of processor time.

PAGING:

⇒ All modern operation systems are following the Concept of Virtual Memory. (part of Secondary memory but acts as a primary memory)

⇒ When a process is loaded from Secondary memory, it is not directly loaded to primary memory (RAM), Instead it is loaded into Virtual Memory.

⇒ Virtual Memory is divided into Small units called as pages

⇒ Primary Memory (RAM) is divided into Small units called as frames.

⇒ Frame Size must be equal to the pageSize.

logical address (or) Virtual address = page.No + offset / addres (which instruction)

physical address = frame no + offset

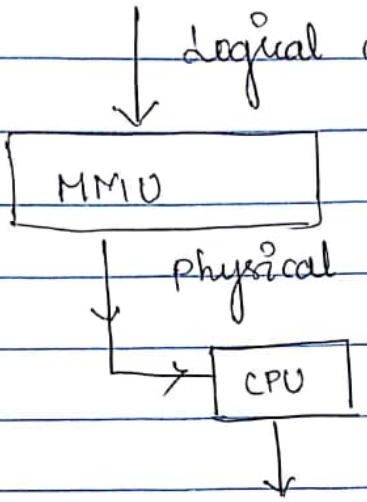
⇒ Program Counter Contains Virtual addresses of physical address.

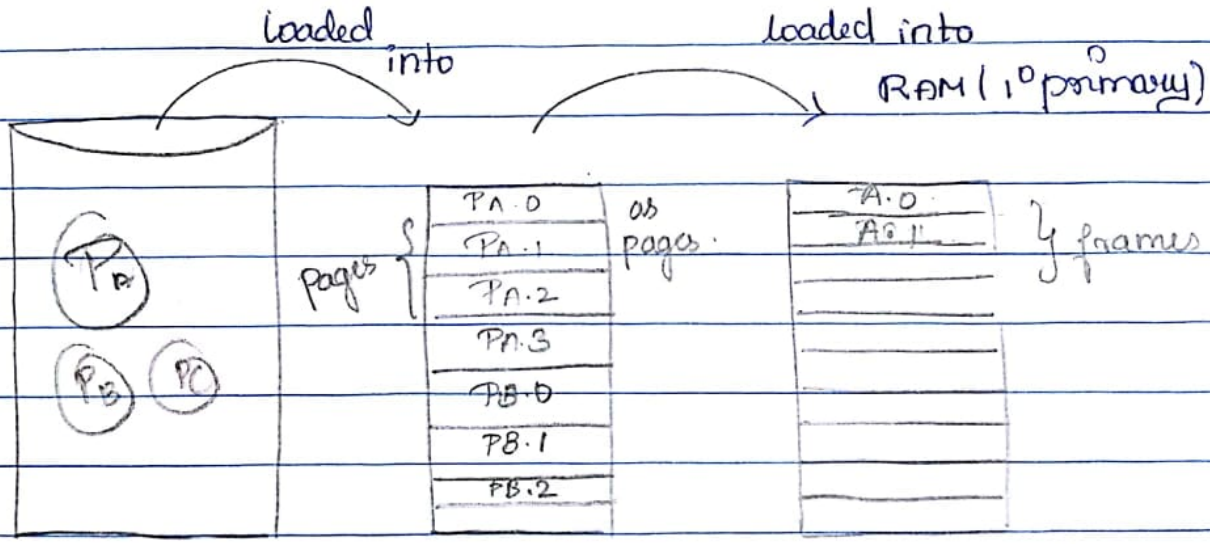
⇒ C program can deal with Virtual memory not physical memory.

(address we are getting are Virtual memory)

⇒ MMU (Memory Management Unit) will convert logical address into physical address.

⇒

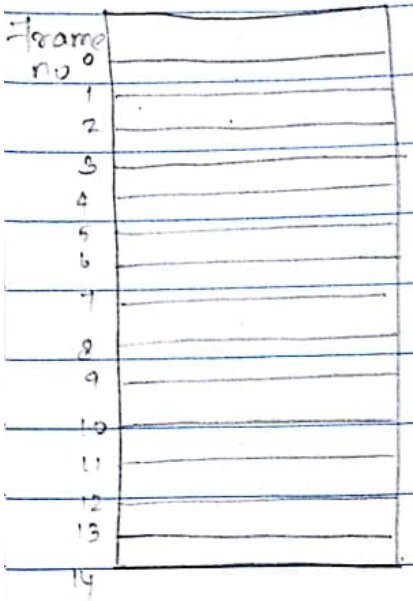




The program is executed page by page into the 1° memory which is divided as frames.

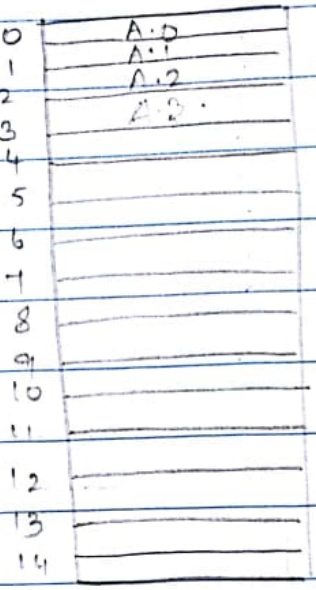
Frame Size = Page Size

RAM (MAIN MEMORY)



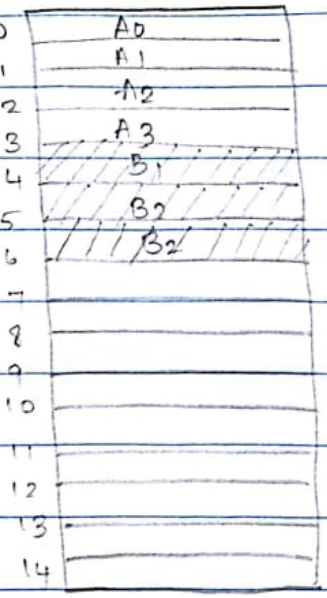
(a) Main memory frames

RAM

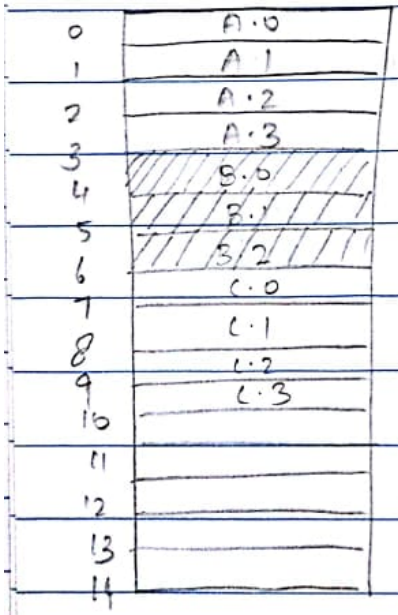


(b) Load Process A

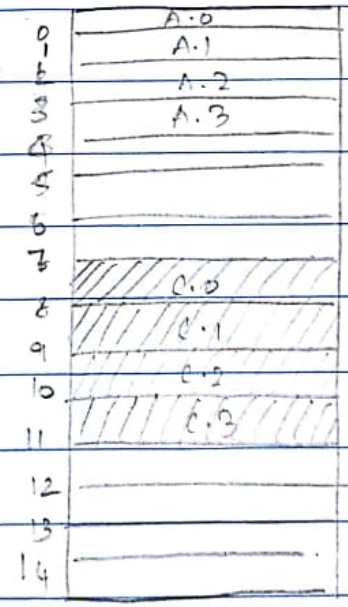
RAM.



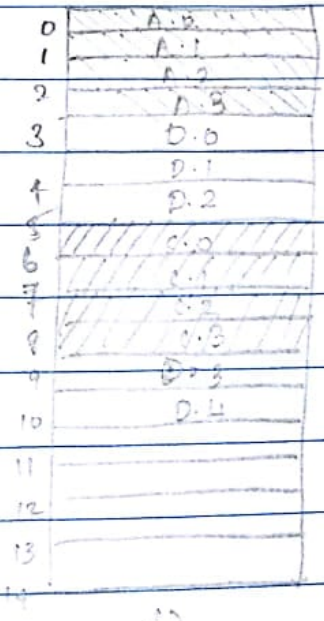
(c) Load Process B



(d) Load process C



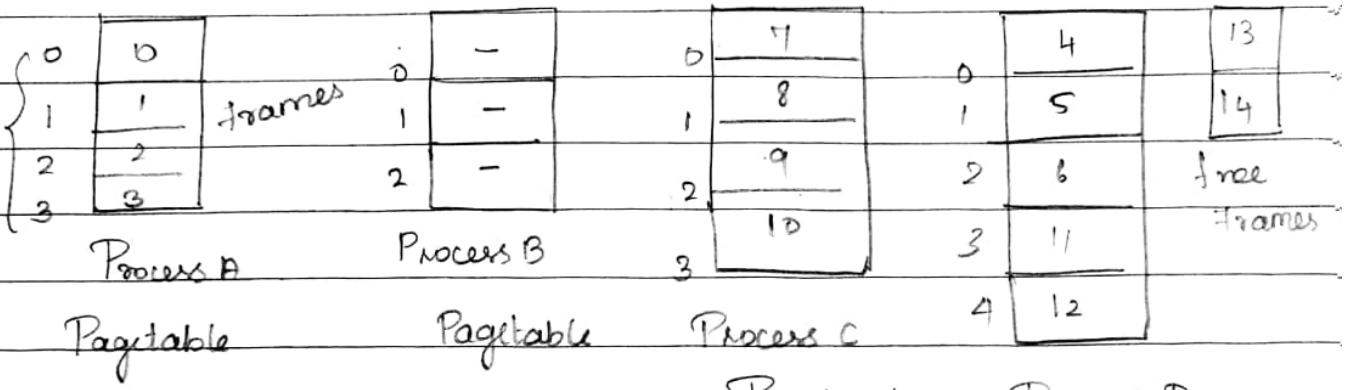
(e) Swap B



(f) Load process D

Every Pagetable Contains ~~every~~ each Pages where it is loaded into frames (which frame it is loaded).

Pagetable :- Page No , Frame No



16 bit address / 10 bits for offset, 6 bits for pages

Pc 1502

000001	0111011110.

↓

Page No (000101) Offset (1478).

↓

(MMU) convert into Physical address.

[000101] + Offset.

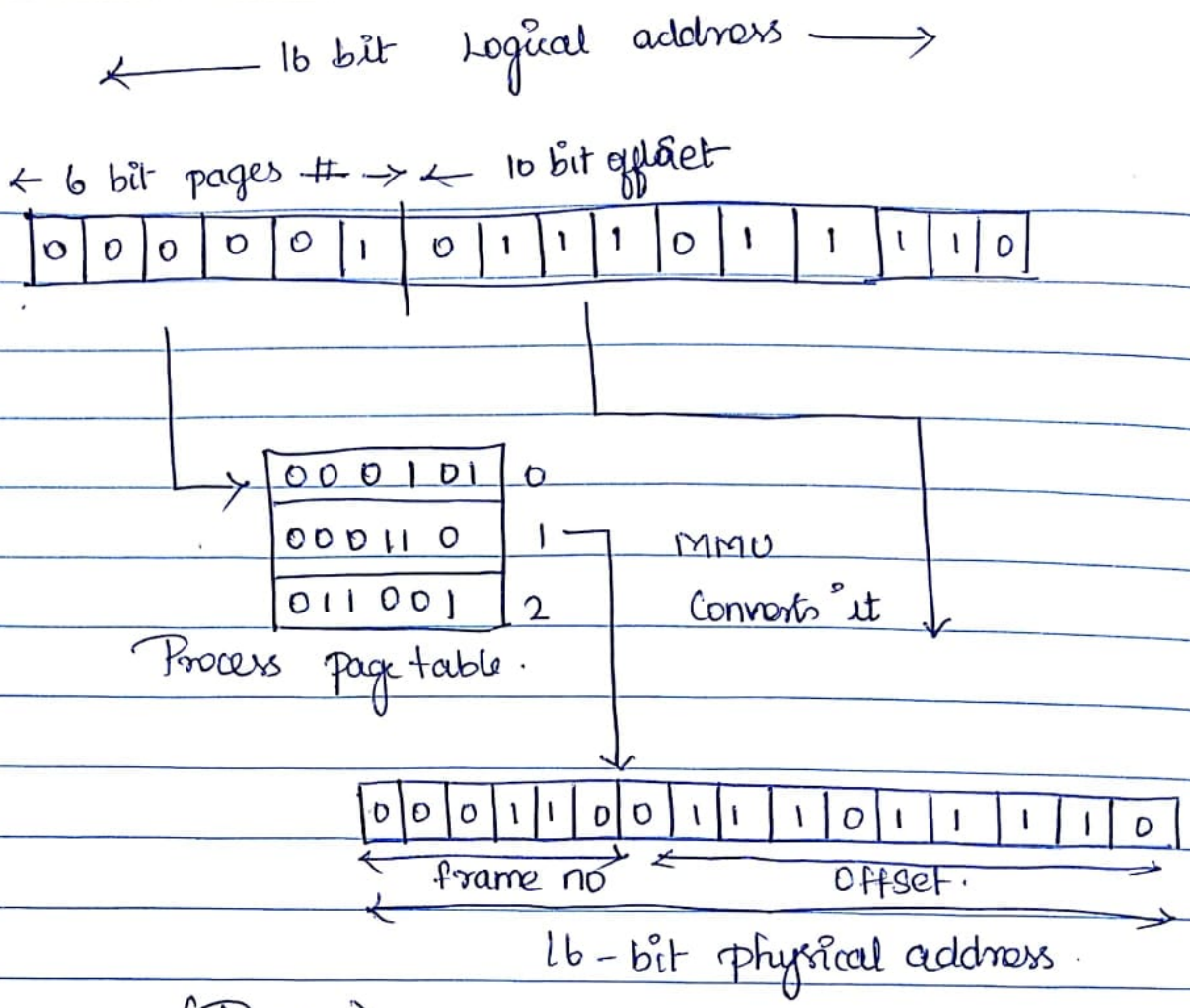
↓

Frame No

2^{16} = 64kB.

1KB = 1024 bytes.

1024
478
502



(Paging)

Program Counter → Page No + offset → Page table adds frame number

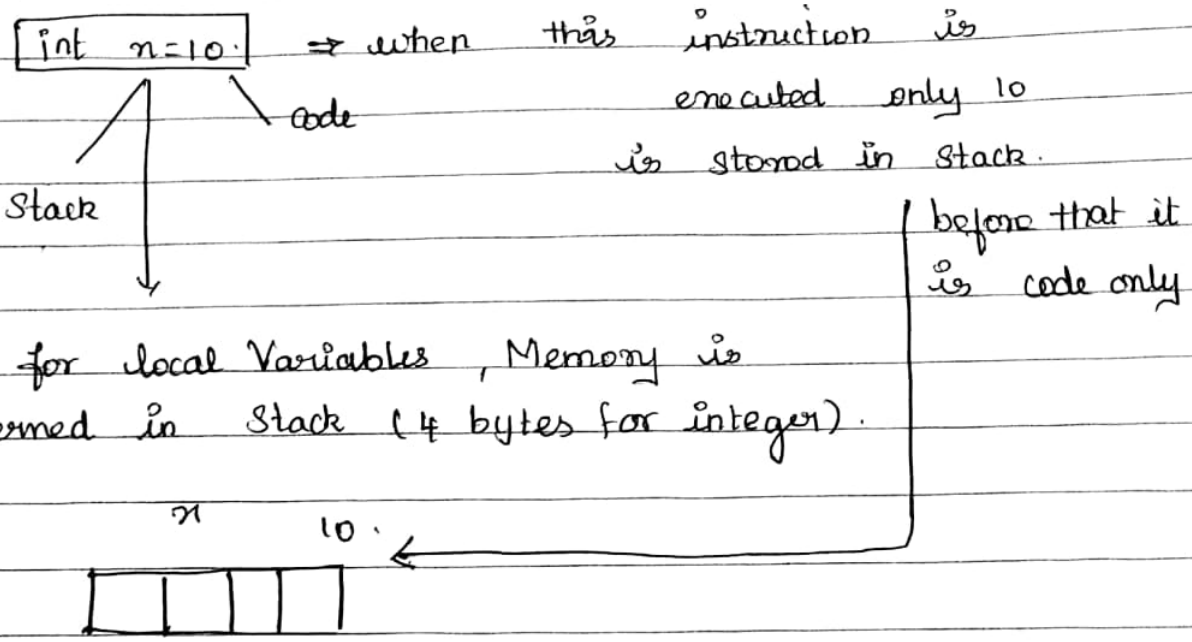
(Virtual address)

Physical address ← Frame no + offset ← MMU

→ Given to CPU for execution.

2/10/17.

stack, heap, data and code are all from Virtual memory.



⇒ when this instruction is executed only 10 is stored in stack. before that it is code only

Demand Page :-

In virtual memory system, demand paging is a type of swapping in which pages of data are not copied from disk to RAM until they are needed.

As there is much less physical memory than Virtual Memory the operating system must be careful that it does not use the physical memory inefficiently. One way to save physical memory is to load virtual pages that are currently being used by the executing Program.

Page Fault :-

An interrupt that occurs when a program requests data that is not currently in real memory. The interrupt triggers the operating system to fetch the data from a Virtual Memory and load it into RAM.

An Invalid page fault or page fault error occurs when the operating system cannot find the data in Virtual memory. This usually happens when the Virtual Memory area, or the table that maps Virtual addresses to real addresses, becomes Corrupt.

Dirty Pages :-

* If a process Needs to bring a Virtual page into physical memory & there are no free physical pages available, the Operating system must make room for this page by discarding another page from physical memory

* If the pages to be discarded from physical memory came from an image or data file and has not been written to then the page does not need to be Saved. Instead it can be discarded and if the process needs that page again it can be brought back into memory from the image or data file.

However, if the page has been modified, the Operating system must preserve the contents of that page so that it can be accessed at a later time. This type of page is known as a dirty page.