Types of System calls:-

System calls are can grouped into six major categories:

- Process control
- File management
- Device management
- Information maintenance
- communication
- Protection

• Process control:-

A running program halt its execution either normally [end()] or abnormally [abort()]. A process or job executing one program may want to load() and execute() another program.

If we create a new job or process [create_process() or submit_job()], we should be able to control its execution. this control requires the ability to determine and reset the attributes of a job or process, including job priority, maximum allowable execution time, and so on [get_process attributes() and set_process_attributes()]. we may also want to terminate a, job or process that we created [terminate_process()].

some times we need to wait for specific event to occure [wait_event()]. the jobs(or) processes should then signal when that event occurred [signal_event()].

- end, abort
- load, execute
- create process, terminate process
- get process attributes, set process attributes
- wait for time
- wait event, signal event
- allocate and free memory.

2. File management system calls :-

We are able to create() and delete() files. Either system call requires the name of the file and file attributes. Once the file is created we need to open() it for use. We may also read(), write() or reposition(). Finally we need to close() the file.

Some of the operating systems provides many more system calls like move() and copy().

- create file, delete file
- open, close
- read, write, reposition
- get file attributes, set file attributes.

Device Management :-

A process may need several resources to execute - main memory, disk drivers, access to files. and so on. If the resource is available, they can

be granted, otherwise process have to wait until sufficient resources are available. A system with multiple users need to request() a device to use it, After finished with the device, we release() it.

once the device has been requested, we can read(), write() and reposition() the device. Like process, file the devices are also have some attri-butes [get_Device_attributes(), set_device_attributes()]

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

Information maintenance :-

Many system calls exist simply for the purpose of transferring information between the user program and operating system. For example most systems have a system calls to return the current time() and date(). other system calls may return information about system such

- get time or date, set time or date
- get system data, set system data
- get process, file or device attributes
- set process, file or device attributes.

communication :-

There are two common models for Inter-process-communication : message passing and shared memory. In message passing, the communicating processes exchanges messages with each one another to transfer messages. the messages can be exchanged either directly or indirectly.

Before communication can takes place a connection must be opened [create communication connection]. once the message transfer is finished the connection will be closed. ~~The message is~~

In shared memory model, processes use shared_memory_create() and shared_memory_attach() system calls to gain access to region shared by processes.

- create, delete communication connection
- Send, receive messages
- Transfer status information
- attach or detach remote devices.

protection :-

protection provides a mechanism for controlling access to the resources. Typically, system calls providing protection include set_permission() and get_permission(), which manipulate the permission setting of resources. The allow_user() and deny_user() system calls specify whether

particular users can or cannot be allowed access to certain resources

———— o ————

→ System Programs :-

The operating system is a collection of programs [system programs]. In a system structure the lowest level is hardware next is the operating system then ~~application~~ system programes and finally application programs.

The system programes are also called as system utilities, provide a convenient environment for program development and execution. The system programs or utilities are ~~bee~~ divided into following categories :-

- File managment :- These programs create, delete, copy, rename, print, dump, list and generally manipulate files and directories.

- Status information :- Some programs simply ask the system date, time amount of available memory, number of users.

- File modification :- Several text editors may be available to create and modify the content of files stored on disk or other storage devices. There may also be special commands to search contents of files or perform transformations of the text.

Programming-language support :- compilers, assemblers, debuggers, and interpreters for common programming languages are often provided with the operating system or available as a separate download.

program loading and execution :- once a program is assembled or compiled, it must be loaded into memory to be executed. the system may provide absolute loaders, relocatable loaders, linkage editors and overlay loader.

communication :- these programs provide the mechanism for creating virtual connection among processes, users and computer systems. these programs allow the users to send message to another's screens, to browse web pages, to send e-mail messages, to transfer files from one machine to another.

Background services :- All general-purpose systems have methods for launching certain system-program processes at boot time.

Some of these processes terminate after completing their tasks, while other continue to run until the system is halted. constantly running system-program processes are known as services, subsystems or daemons.

operating-system Design and Implementation :-

Design Goals :-

The first problem in designing a system is to define goal and specifications. At higher level, the design of the system is affected by the choice of hardware and type of system: batch, time sharing, single user, multiuser, distributed, real time, or general purpose.

Beyond this, the requirements (goals) are divided into two basic groups:

- user goals
- system goals.

User want certain obvious properties in a system. The system should be convenient to use, easy to learn and to use, reliable, safe and fast.

Similarly, the system should be easy to design, implement, and maintain; and it should be flexible, reliable, error free efficient.

Mechanisms and policies :-

Mechanism determines how to do something; policies determine what will be done

For example, the time construct, is a mechanism for ensuring CPU protection, but deciding how long we need to assign CPU to particular user

is a policy. we need to define mechanism and policy separately and clearly to acheive flexi-bility.

In worst case, each change in policy require a change in mechanism. policy decisions are impo-rtant for all resources allocation. whenever it is necessary to decide wheather or not to allocate a resources, a policy decision must be made:

Implementation :-

once, the operating system is designed, it must be implemented. Beacuse operating system is a collection of programs.

* old operating systems were written in assembly language, some of the modern operating system arealso written in assembly language,

* Actually, the operating system can be written in more than one language. the lowest levels of kernal might be assembly language.

* Higher-level routines might be in 'c', and system programs might be in c or c++, in interpreted scripting languages like PERL or python.

* the advantage of using high-level languages is; the code can be written faster, more compact and easier to understand, debug.

* The only possible disadvantages of implementing an operating system in a high-level language are reduced speed and increased storage requirement.

Operating-System Structure :-

- Simple Structure :-

- In this operating systems do not have well-defined structures.
- It will start as small, simple and with limited systems and grow towards the original scope.

Ex:- MS-DOS.

- MS-DOS was designed to provide more functionality in less space, so it was divided into modules. The following diagram shows the simple structure of MS-DOS

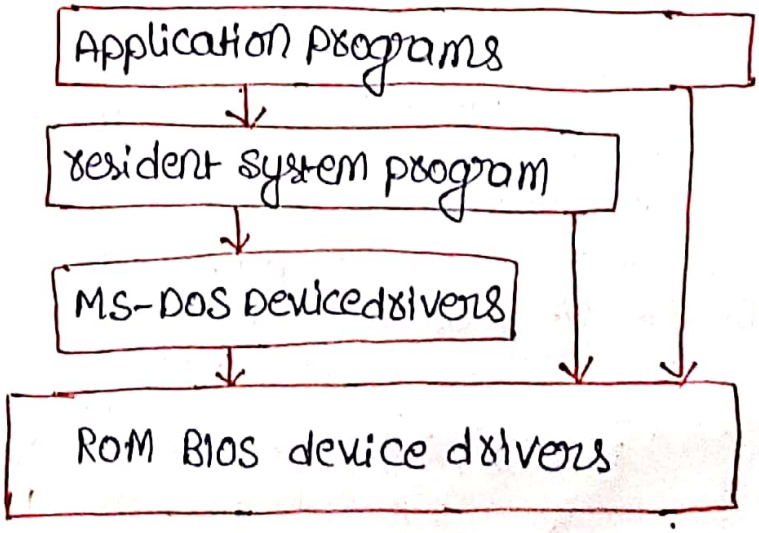

Fig:- MS-DOS layer structure

* In MS-DOS interfaces and system functionalities are not separated. Application programs are able to access the some system utilities [programs] without any restriction.

* some times, it will lead to entire system crash when user program was not properly executed.

* Another example of limited or simple structure is the original UNIX operating system.

* like MS-DOS UNIX also limited it's hardware functionally initially. It consists of two parts: the kernal and system program.

*

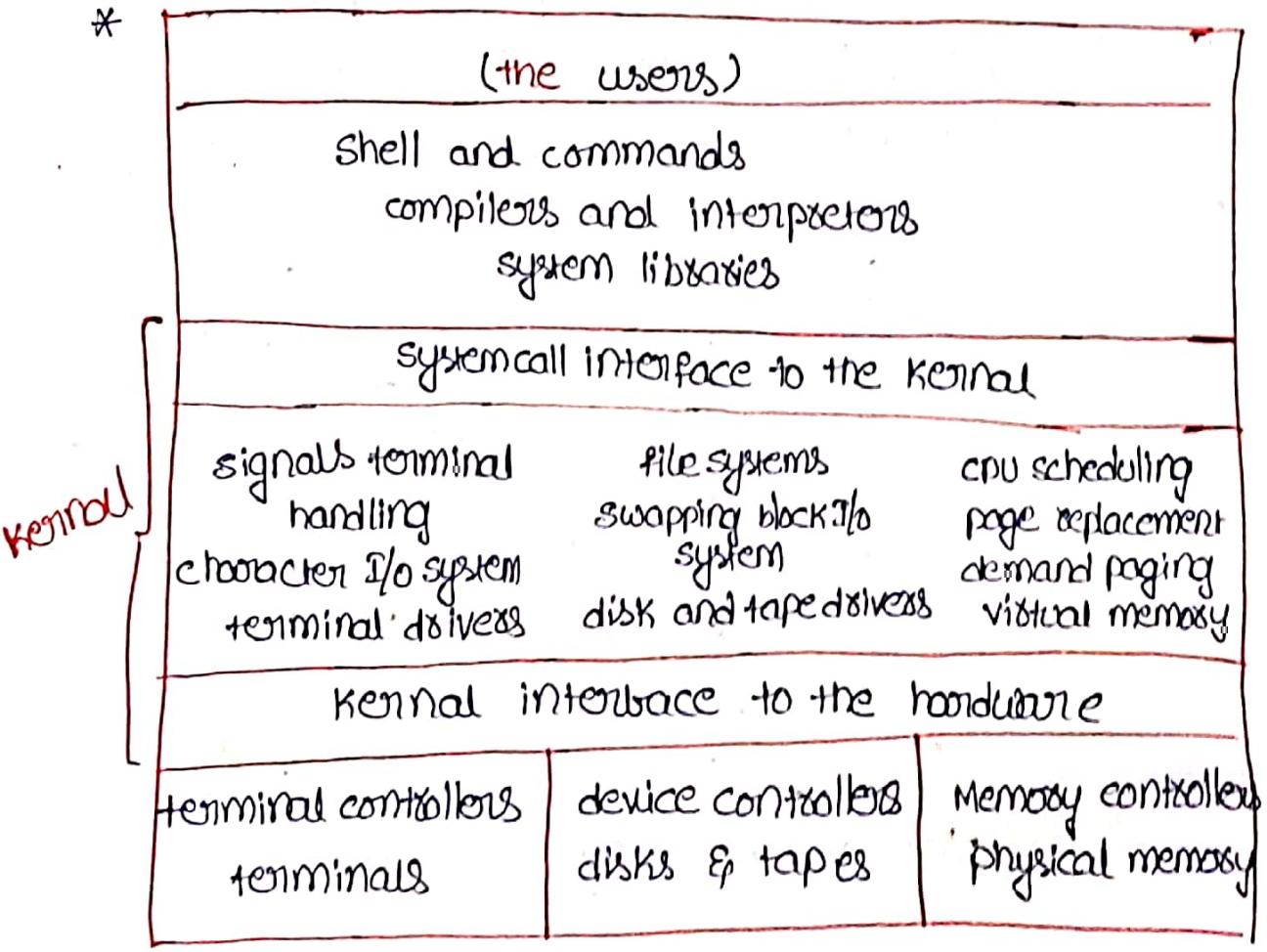

Fig:- Traditional UNIX system structure

* The kernal is further divided into a series of interfaces and device drivers, everything below the system-call interface and above physical hardware is the kernal.

* The kernal provides the file systems, CPU scheduling, memory managment and other operating system functions through system calls.

Layered Approach :-

For getting control over computer and over applications that make use of the computer, The operating system can be broken into pieces. With this we have more freedom in changing inner working of the system and for creating modular operating system.

The system can be made modular through layered approach.

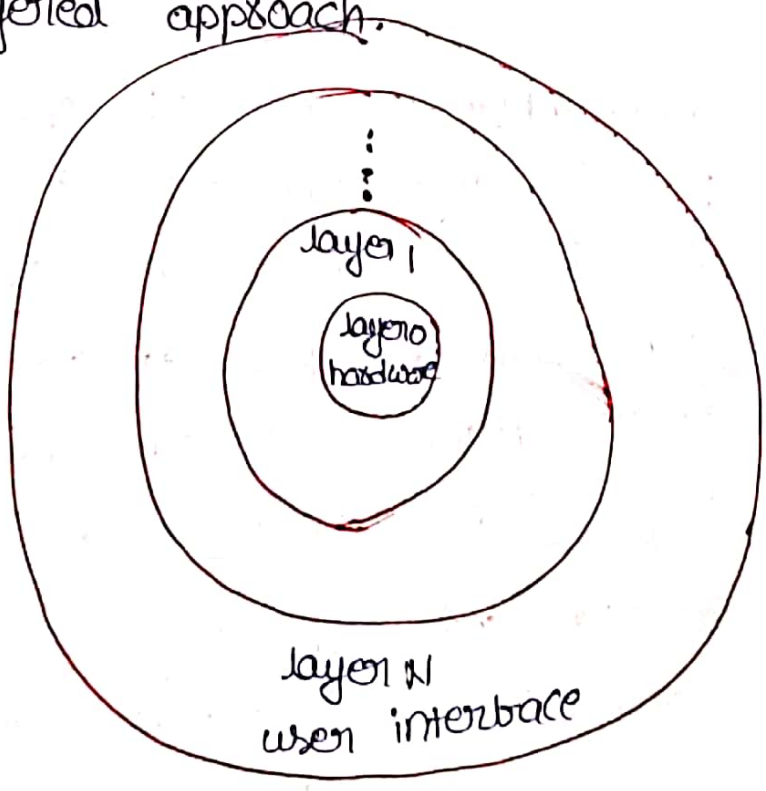

Fig :- A layered operating system.

* In layered approach operating system is broken into a number of layers [levels]. The bottom layer [layer 0] is the hardware; the highest [layer N] is the user interface.

* A typical operating system layer, consists of data structures and a set of routines that can be invoked by higher-level layers.

* Layer M, in turn, can invoke operations on lower-level layers.

* The main advantage of layered approach is simplicity of construction and debugging.

* The major difficulty with this approach involves appropriately defining the various layers, and this implementation is less efficient.

Microkernels :-

The major disadvantage in the UNIX operating system is, kernal part is large and difficult to manage. To overcome this disadvantage In mid-1980, a new operating system was developed at Carnegie Mellon university called "Mach" that can modularize the kernal by using microkernal approach.

This method structures the operating system by removing all non-essential components from the kernel and implementing them as system and user-level programs. The result is small kernal.

* the microkernals provide minimal process and memory management, in addition to a communication facility.

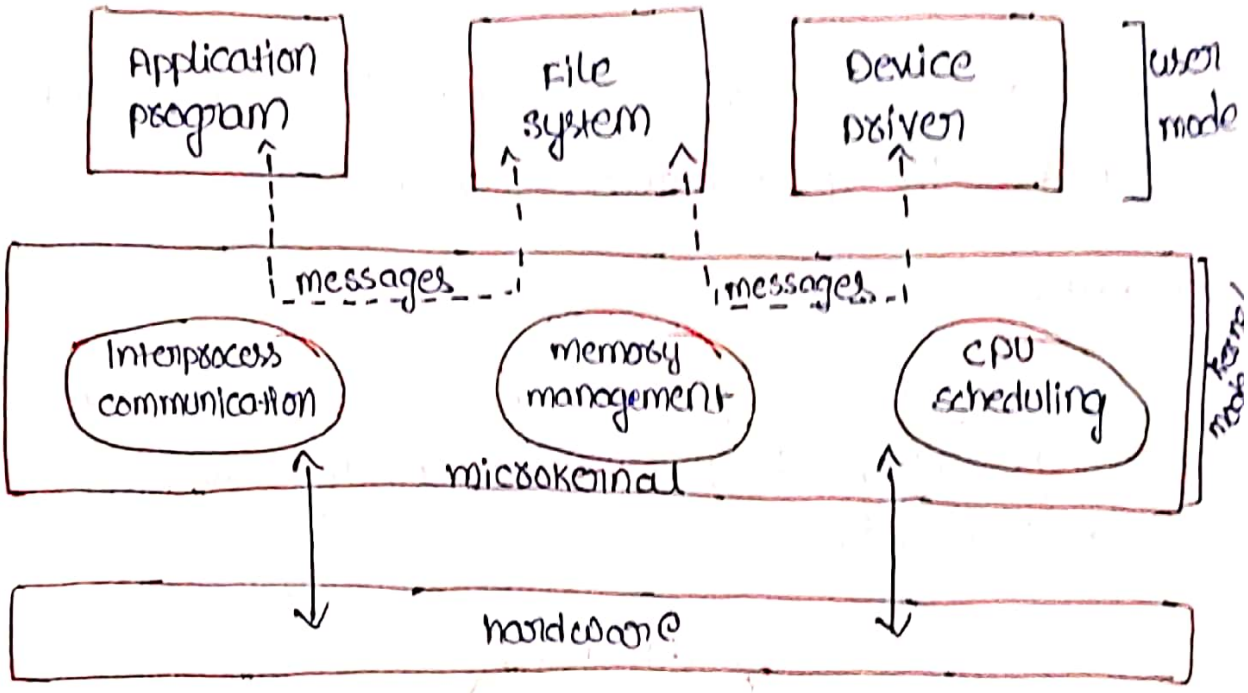

Fig :- Architecture of a typical microkernal.

* the main function of the microkernal is to provide communication between the client program and various services that are also running in the user space

* The communication is provided through the message passing, for example, if client program wishes to access a file, it must interact with the file server.

* the client program and services won't interact directly, they communicate indirectly by exchanging messages with the microkernal.

* one of the benefit of the microkernal approach is that it makes extending the operating system easier. microkernals also provides more security and reliability, since most of the services are running in user mode.

* But, due to ~~increased~~ increased system-function overhead, the performance is low.

Modules :-

To increse the performance of a system, the operating system design involves using loadable kernal modules.

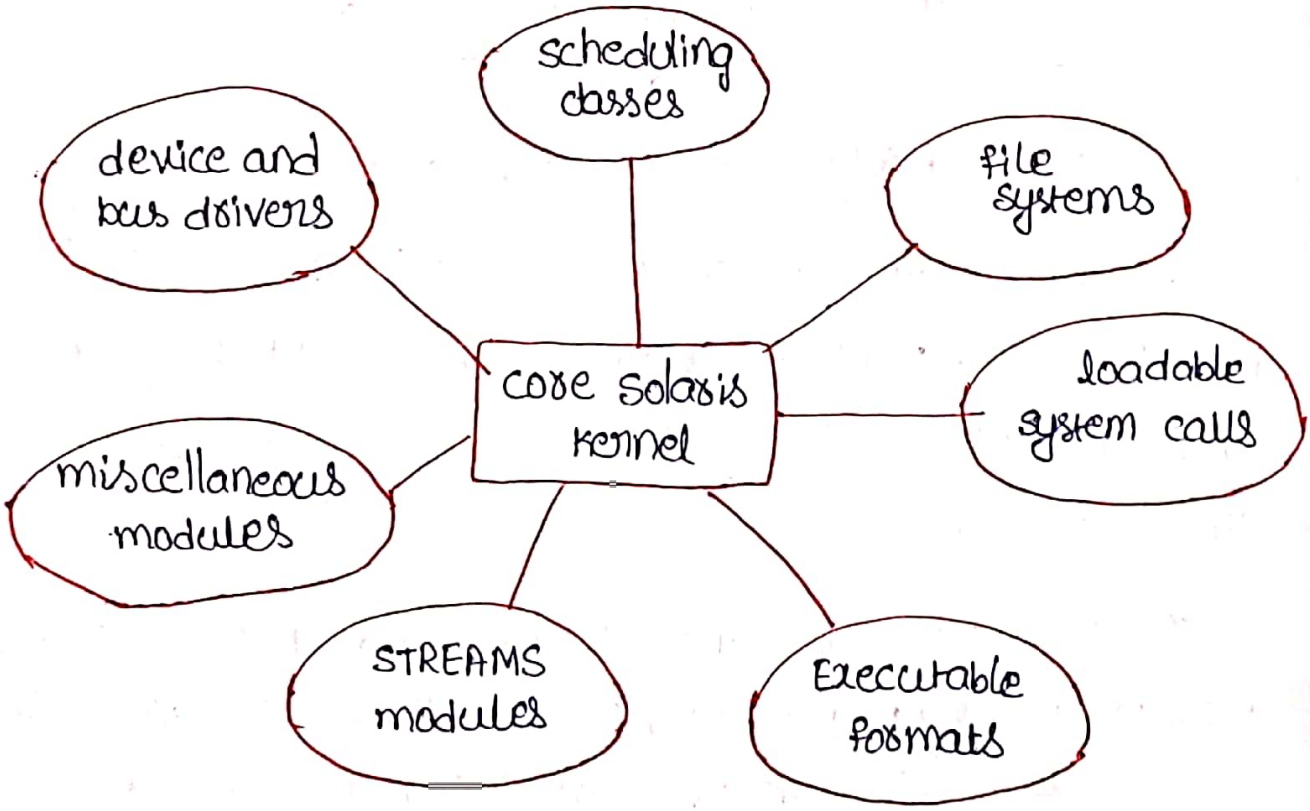

Fig:- Solaris loadable modules

* In this kernal has core components and links in additional services via modules, either at boot time or during run time.

* this type of implementation is common in modern UNIX, such as Solaris, Linux, and MAC OS X, as well as window.

* The idea of design is kernal provides core services and remaining services are implemented dynamically.

* the figure shows the Solaris operating system structure. The core kernal is connected with seven types of loadable kernal modules.

1. Scheduling classes
2. File systems
3. Loadable system calls
4. Executable Formats
5. STREAMS modules
6. Miscellaneous
7. Device and bus drivers.

* Linux also uses loadable kernal modules, primari-ly for supporting device drivers and file systems

Hybrid Systems :-

Very few operating systems adopt a single, strictly defined structure, but addressing performance, security and usability issues.

* For example, both Linux and Solaris are mono-lithic, because they are having single address space provides very efficient performance.

* They are also modular, a new functionality can be dynamically added to the kernel. Windows system also provide support for dynamically loadable kernal modules.

* The three hybrid system: the Apple Mac OS X and two most prominent mobile operating systems - iOS and Android.

- Mac OS X:

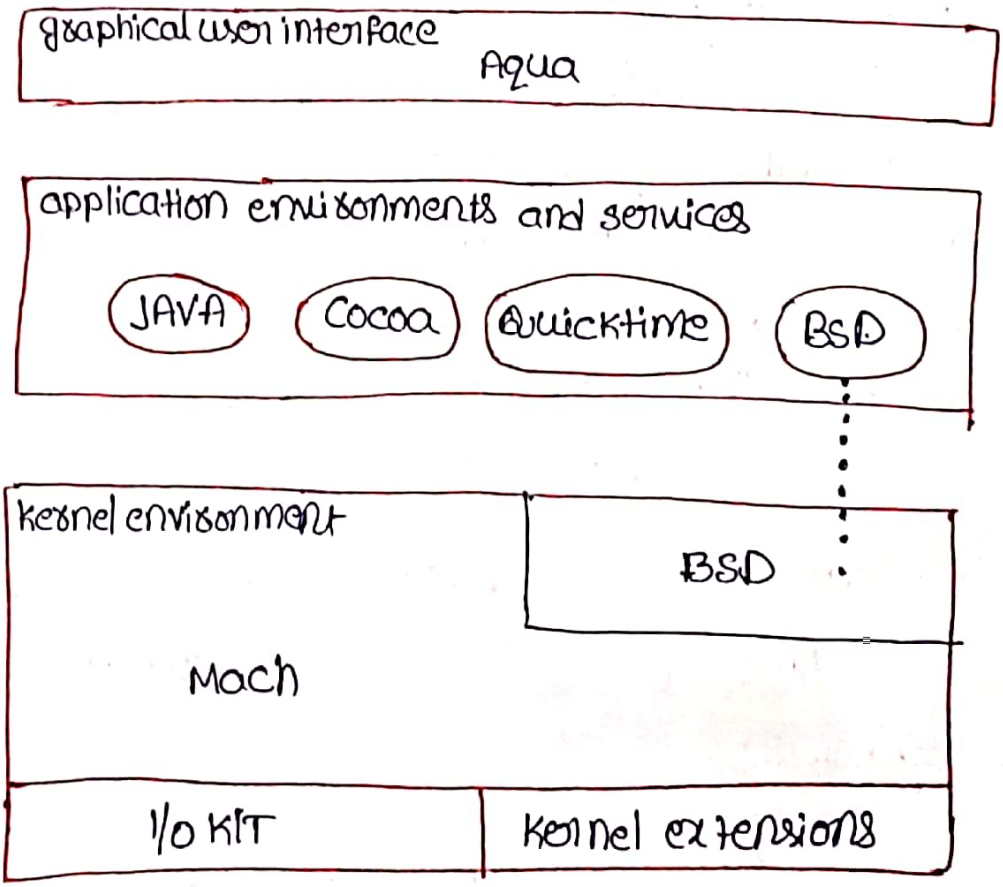

Fig:- the Mac OS X structure

* The Apple Mac OS X operating system uses a hybrid structure. The top layer includes the Aqua user interface, and set of application environments and services.

* The next level is kernel environment, which contains Mach microkernel and BSD UNIX kernel.

* Mach provides memory managment, support for Remote Procedure calls [RPC's] and Inter process communication [IPC] including message passing and thread scheduling.

* The BSD component provides a BSD command line interface, supports for networking and file systems and implementation of POSIX APIs including Pthreads.

* In addition to Mach and BSD I/O kit for development of device drivers and dynamically loadable modules.

- iOS :-

iOS is a mobile operating system designed by Apple to run its smartphone, the iPhone as well as its tablet computer, the iPad. The structure of iOS is same as Mac OS X, with added functionality pertinent to mobile devices but does not directly run Mac OS X applications. The structure is shown in figure

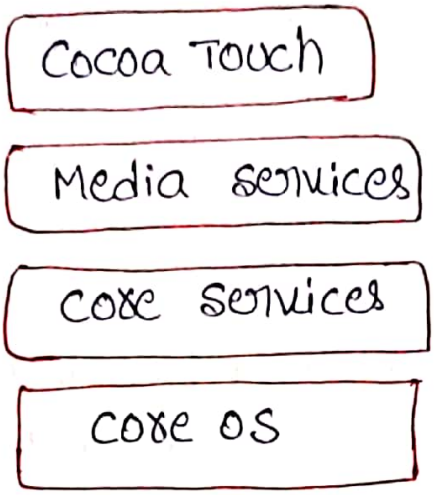

Fig:- Architecture of Apple's iOS

- Android :

Android is a open - sourced software. It follows layered approach. At bottom Linux kernel,

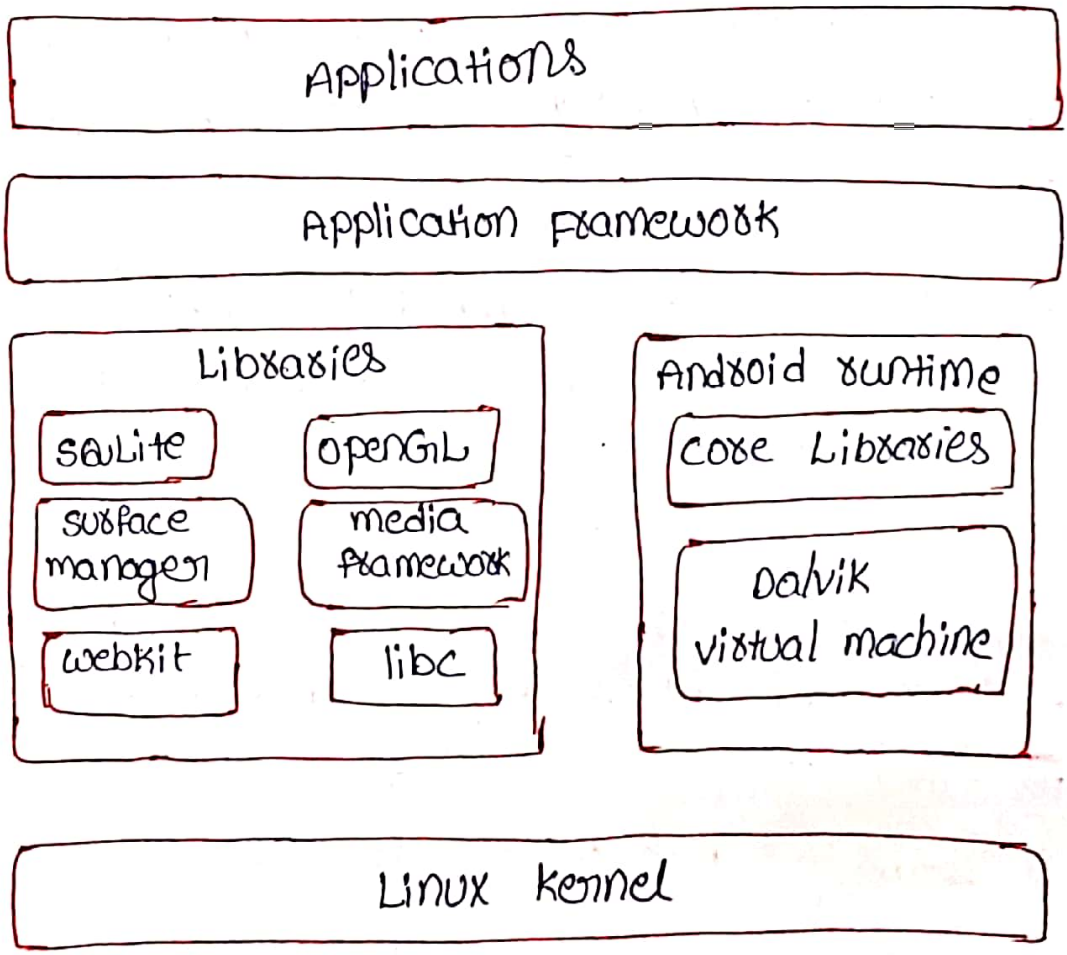

Fig:- Architecture of Google's Android

* Android is similar to ios in that, it is a layered stack of software that provides a rich set of framework for developing mobil applications.

* Linux is used primarily for process, memory and device-driver support for hardware and has been expanded to include power managment.

* the Android runtime

Process-concept:-

Informally, a process is a program in execution A process is more than the program code, which is sometimes known as text section. It is used to represent the current activity [represented by the program counter & contents of processor's registers]

* A process generally includes process stack which contains temporary data [such as function parameters, return address and local variables] and data section, which contains global variables.

* A process may also includes a heap, which is memory that is dynamically allocated during process runtime.

* The structure of process in memory is shown below:

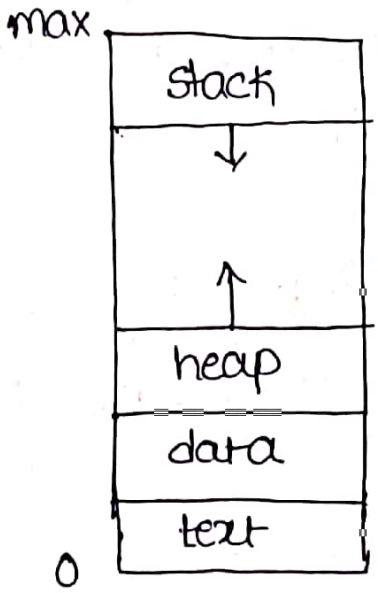

Fig:- Process in memory

* The program is a passive entity, such as a file contains list of instructions stored on disk, where as process is a active entity, which is executed by program counter and a set of associated resources.

* A program becomes process when an executable file is loaded into memory.

* for single process program there may be more than one process associated. For example, several users may running different copies of mail program. (or) same user may invoke many copies of the web browser programs.

Process States :-

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

- New :- The process is being created
- Running :- Instructions are being executed
- waiting : The process is waiting for some event to occur [such as I/O]
- Ready : the process is waiting to be assigned to processor
- Terminated :- The process has finished execution.

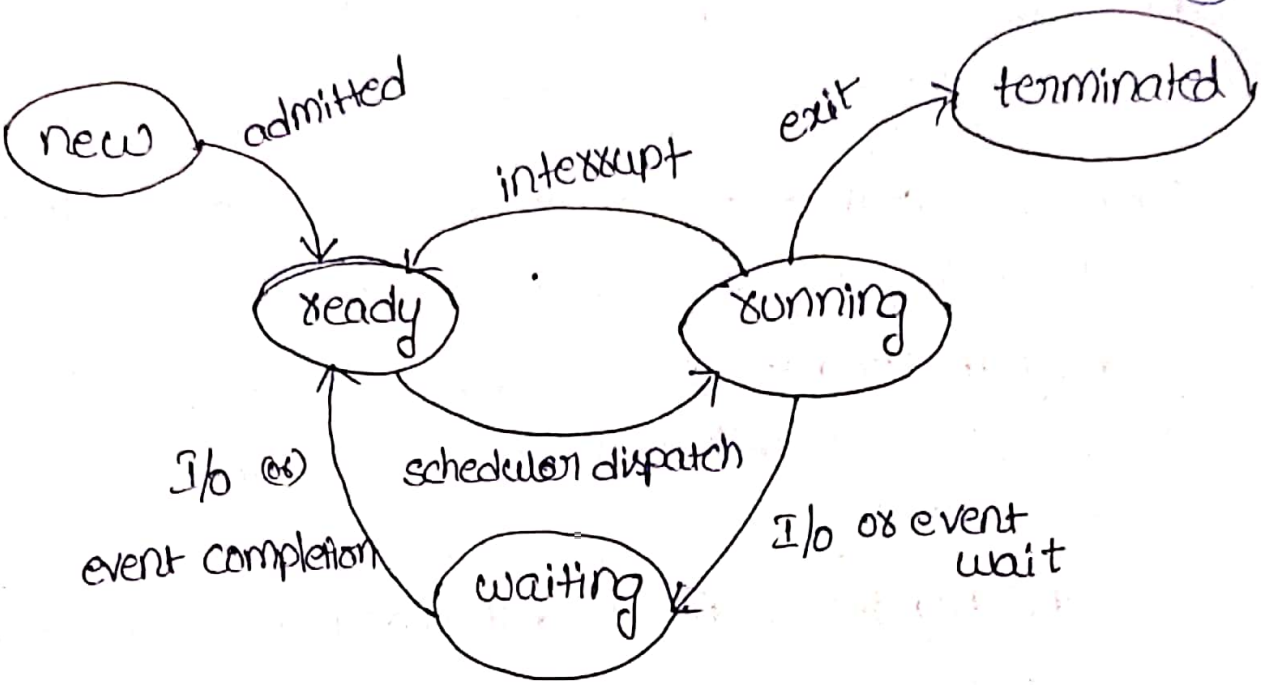

Fig:- Diagram for Process state

The names may vary across operating system. It is important to realize that only one process can be running on any processor at any instant.

Process control Block:-

Each process is represented in operating system by a process control block [PCB] also called as task control block. A PCB is shown in Figure:

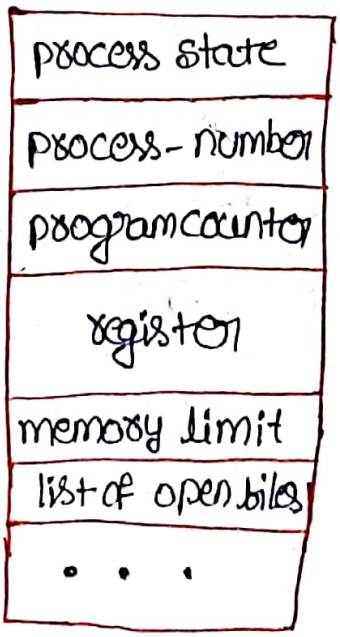

Fig:- process control Block (PCB)

PCB contains many piece pieces of information associated with a specific process,

- Process state: The state may be new, ready, running, waiting, halted. and so on.
- Program counter: The program counter indicates the address of next instruction to be executed for this process.
- CPU Registers: Along with program counter, CPU registers like accumulator, index registers, stack pointers, and general purpose registers to save the state information of process.
- CPU - scheduling information: It includes process priority, pointers to scheduling queue and other scheduling parameters.
- Memory management information: This information includes page tables, segment tables, value of base and limit registers, depending up on OS.
- Accounting information: This includes the amount of CPU time used, time limits, accounts numbers, job or process number ....
- I/O status information: This information includes the list of I/O devices allocated to the process, list of open files, and so on.

If interrupt is occurred in the middle of execution the PCB is saved before control transfer. [shown in figure]

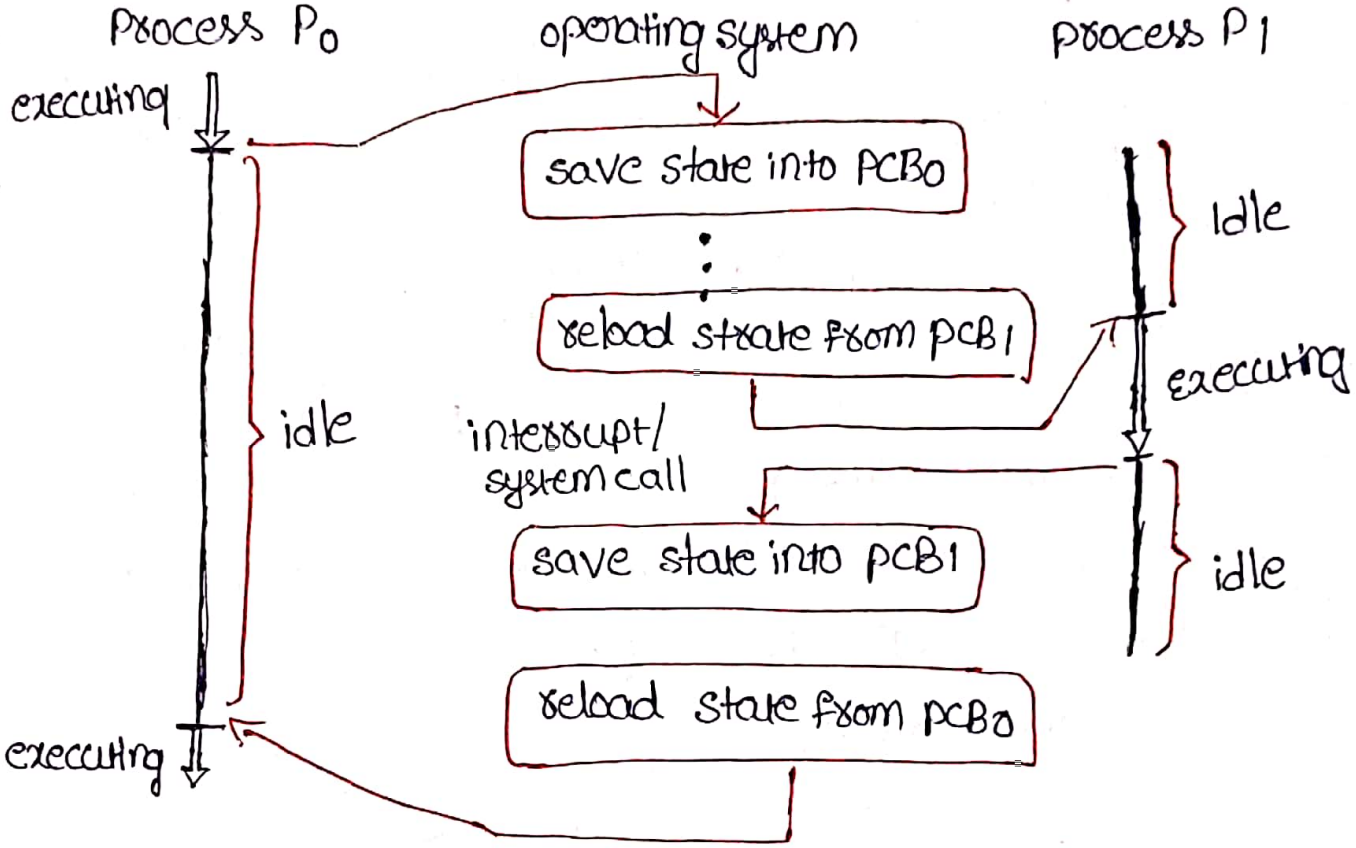

Fig :- Diagram showing CPU switch from process to process.

Threads :-

A process is a program that performs a single thread of execution. For example, an word process program, a single thread of instructions is being executed. i.e the user cannot simultaneously type in character and run spell checker with the same process.

Process Scheduling :-

The objective of multiprogramming is to maximize CPU utilization. The objective of time-sharing is to switch the cpu among processes so frequently that users can interact with each program while it is running.

The process [cpu scheduler] scheduler will select the available [from set of available processes] for executing on the CPU. In single processor system, at a time only one process is going to execute, remaining has to wait untill CPU is free.

- **Scheduling Queue :-**

The processes that are residing in main memory and are ready and waiting for execution are kept on a list called the "Ready Queue". this Queue is generally stored as linked list.

A ready queue header contains pointers to first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready Queue. like CPU other devices have their own device queue, since the more than one process may request same I/o device,

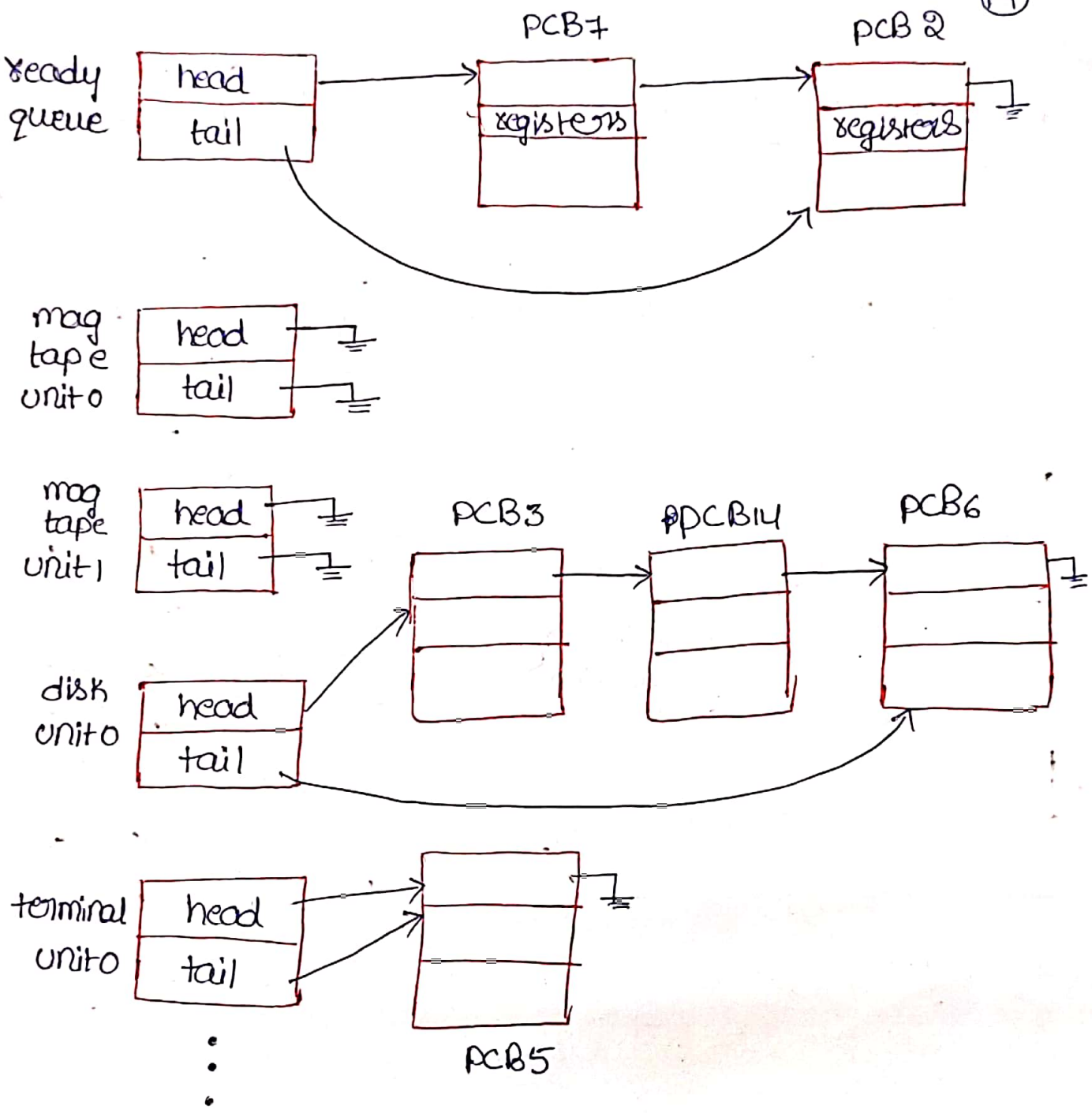

Fig :- The ready queue and various I/O device queues.

The common representation for process scheduling is queueing diagram. shown in below figure. two types of queues are there : ready queue and set of device queues. the circle represents the resources, arrows indicates the flow.

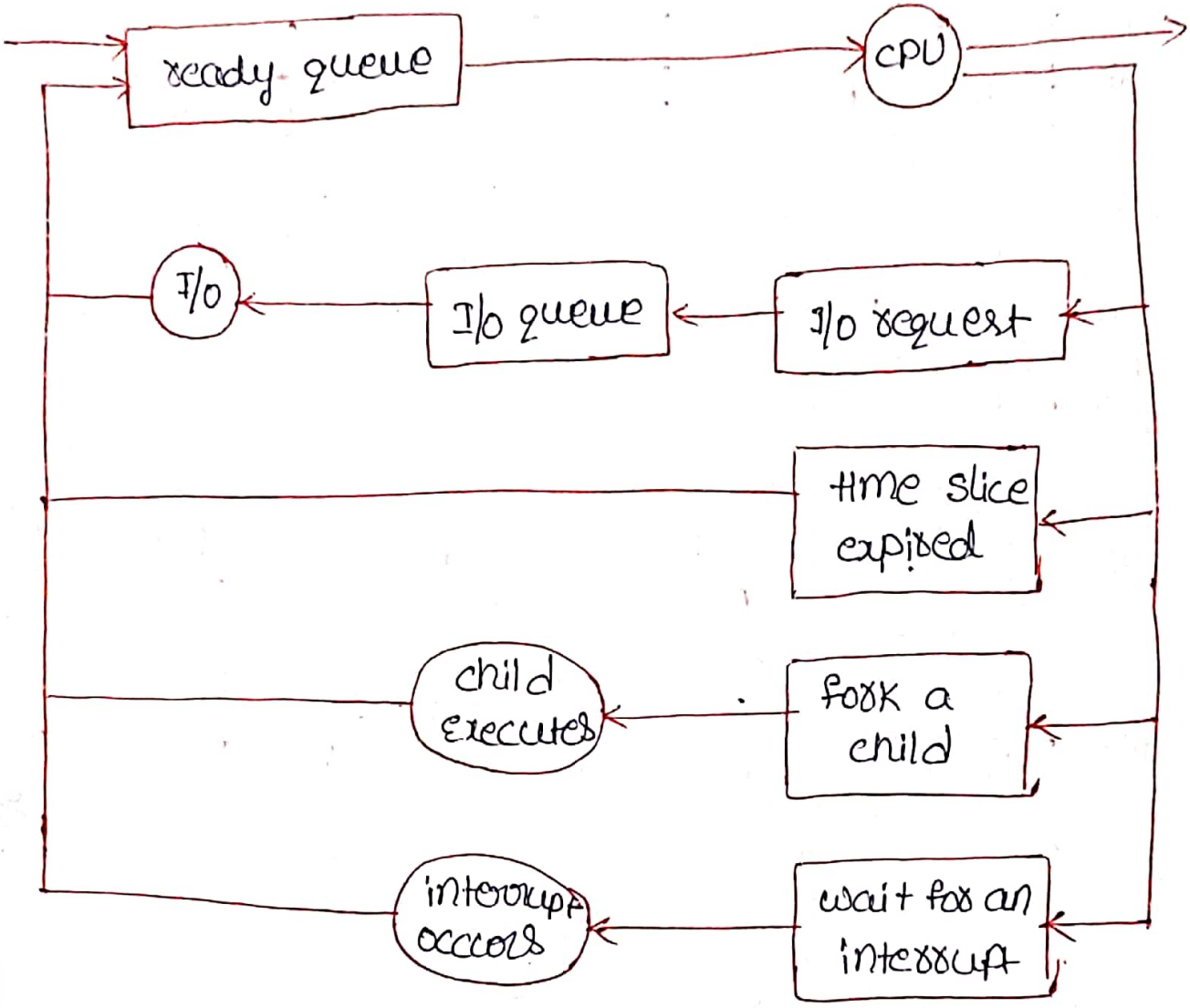

Fig:- Queueing diagram representation of process scheduling.

A new process is intially put in the ready queue. The processes [which are ready queue] waits untill it is selected for execution (or) dispatched. once the process is allocated the CPU and is executing, one of several events could occurs:

- The process could issue I/O request and then placed in I/O queue.
- The process could create a new child

process and wait for the child termination.

- the process could be removed forcibly from the CPU, as a result of an interrupt and be put back in ready queue.

In first two cases, the process eventually switches from waiting state to the ready state and put back in ready queue.

- Schedulers :-

A process migrates among the various scheduling queues throughout the execution. The operating system will select the processes from the queue based on some fashion. This selection process is carried out by scheduler.

- The long-term scheduler or job scheduler, select the process from the pool and loads into memory for execution.

- The short term or CPU scheduler, selects among the processes that are ready to execute and allocates the CPU to one of them.

- The medium-term schedulers, remove a process from memory & reduce the degree of multi-programming. Later, the process can be re-introduced into and its execution can be continued when it left off.

- long-term scheduler executes less frequently. The long-term schedulers control the degree of multiprogramming [no. of processes in memory]

- The long-term schedulers are invoked only when process leaves the system. long-term scheduler takes more time to decide which process should select for execution.

~~X Short-term schedulers selects a new process for the CPU frequently. X~~

- The long-term scheduler makes a careful selection. In general the processes are either I/O bound or CPU bound processes.

- I/O bound processes is one that spends more time on I/O operations, during execution. The CPU bound process is one that spends more time on computations.

- the long-term schedulers selects a good process mix of I/O bound and CPU bound to ~~make~~ improve perbalmance.

- whereas, short-term scheduler selects a new process for the CPU frequently.

- Some operating systems, such as tim-sharing systems, having medium term schedulers shown in figure.

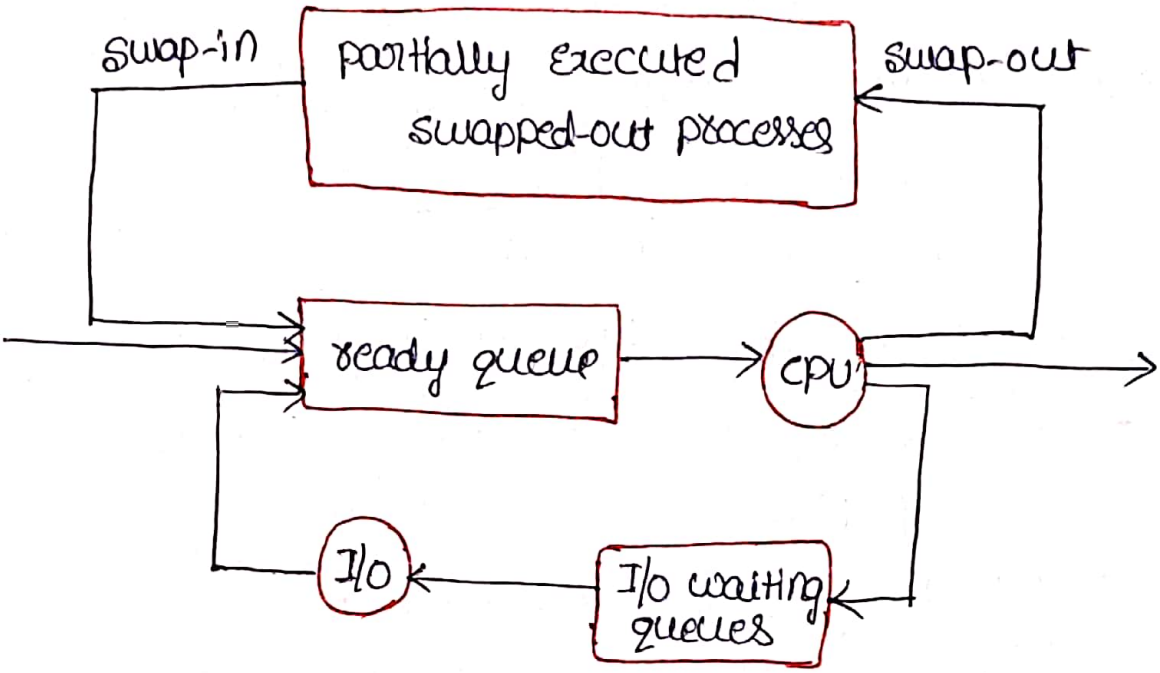

Fig:- Addition of medium-term scheduling to the queueing diagram.

- The medium-term scheduler will take out a process from memory and after sometime reintroduced into memory and execute it where it left off previously. This is called "swapping"

- Swapping may be necessary to improve the process mix (or) because a change in memory requirments.

- context switch :-

when the interrupt occurs, the operating system changes from its current task, to run a interrupt ['in kernal]. Before Going to execute the interrupt service routine, the system need to save the current contex [status] of the process running on the CPU, so that it can restore it after interrupt processing finished.

- The context is represented in the form of PCB for every process.
- The PCB includes the value of the CPU registers, process state, and memory managment, CPU scheduling information.
- Generally, we perform a state save of the current process and restore state when it is resumed.
- Switching the CPU from current running process to another process saving the current ~~state~~ process state and reload with different process is known as context switch.
- context switch time is pure overhead, during context switch won't do any useful work while switching.

Operations on Processes :-

In most of the systems, all processes are executes concurrently, and they may be created and deleted dynamically. Every system much process provide mechanism how the process is created and terminated. The following mechanisms are defined in windows and UNIX systems.

- **Process creation :-**

For single program, multiple processes are possible, i.e During execution, a process may create several processes. The creating process is called parent process, and new process is called as child process. Some times, each of these new process may create other process, forms a tree of processes.

Most of operating systems identify the processes according to a unique process identifier [or pid], which is an integer. By using pid, system can access various attributes of process with in kernal. The following tree diagram illustrates the typical process tree in Linux operating system.

- the init process [which is always has a pid=1] serves as a root parent process for all user processes.

- once the system has booted, the init

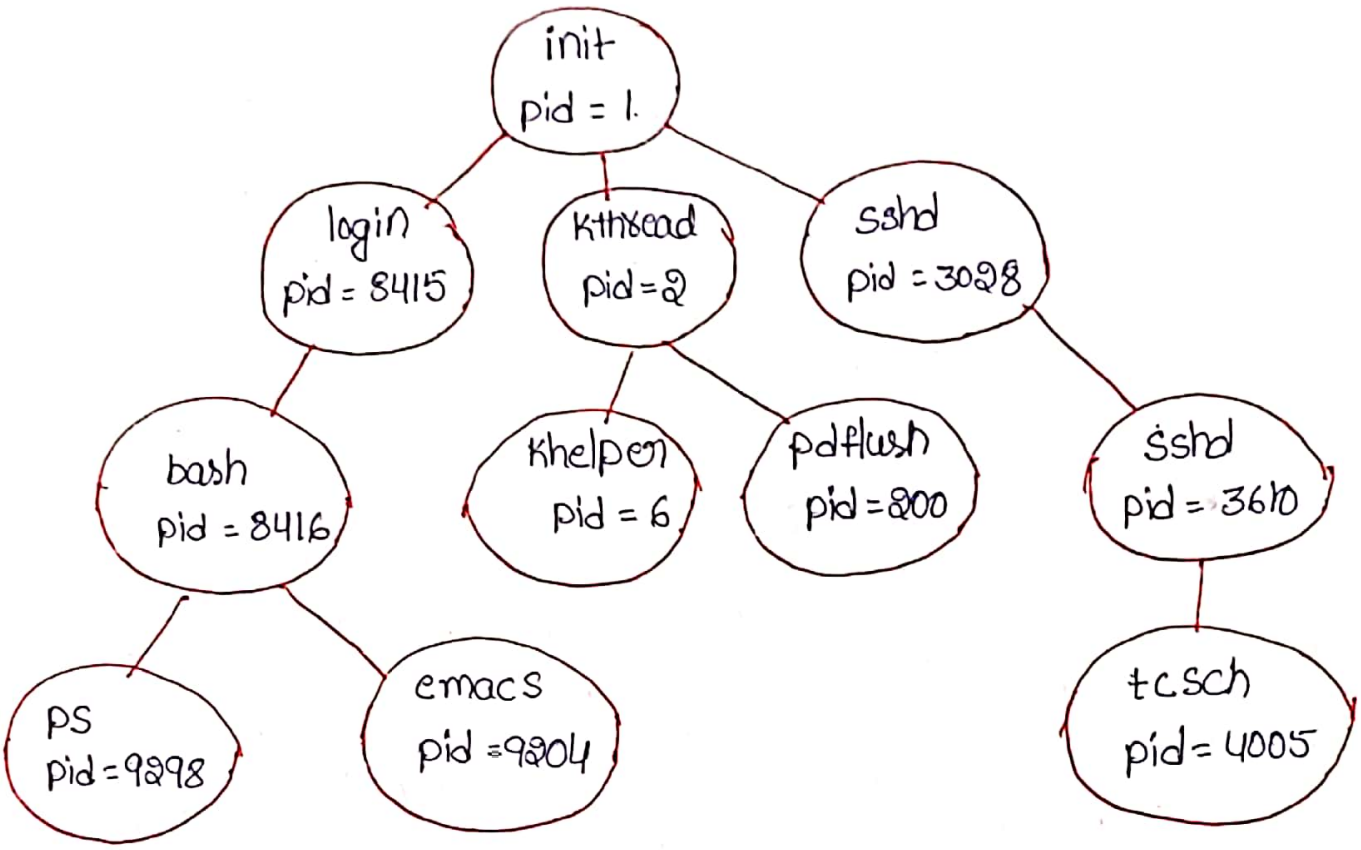

Fig:- A tree of processes on a typical Linux system.

process can also create various user processes, such as web or print servers as ssh and other functionalities.

- The kthread process is responsible for creating additional processes that perforims tasks on behalf of the kernal.

- The sshd process is responsible for managing clients the connect to the system by using ssh.

- login is responsible for managing clients the directly log into the system.

- In the above example [diagram], client has logged on and is using bash shell, which has been assigned pid 8416.

- on Linux and UNIX systems, we can obtain a listing of processes by using the ps command.

```
ps -el
```

- In general, when child process is created, it need certain resources to finish execution. The child process may request the resources directly from operating system or use the resources held by parent process.

- when a process creates a new process, two possibilities for execution exist:

  - The parent process continue to execute concurrently along with its children.
  - The parent waits until some or all of its children have terminated.

- whenever process is created, there are two address-space possibilities for new process:

  - the child process is a duplicate of the parent process
  - the child process has a new program loaded into it.

creating processes in UNIX :-

- In UNIX each process is uniquely identified by its process Id, which is a integer value.
- In UNIX process creation is done by using fork() system call. the child process consists of copy of address space of original [parent] process.
- This mechanism gives allows the parent process to communicate easily with its child process.
- Both child & parent processes executes concurrently, with only one difference: the return value for fork() is zero for new(child) process, whereas nonzero [pid of chid] for parent.
- After fork() system call, one of the two processes uses exec() system call to replace the memory space with new program.
- The parent can create more children; or it will execute issue wait() system call to move itself into ready queue until child process terminated.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
    pid_t  pid;
    /* fork a child process */
    pid = fork();
    if(pid<0)  /* fork error occurred*/
    {
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if(pid==0)
    {                    /* child create */
        execlp("/bin/ls", "ls", NULL);
    }
    else
    {  /* parent process */
     wait(NULL);  /* wait till child terminate*/
     printf("child complete");
    }
return 0;
}
```

Fig:- creating separate process using UNIX fork() system call

The child inherits using parent resources and address space with UNIX command /bin/ls. The parent wait till child to terminate by invoking wait() system call. once the child process completes either implecitly or explicity by invoking exit(). system call. At this point parent will resume its execution.

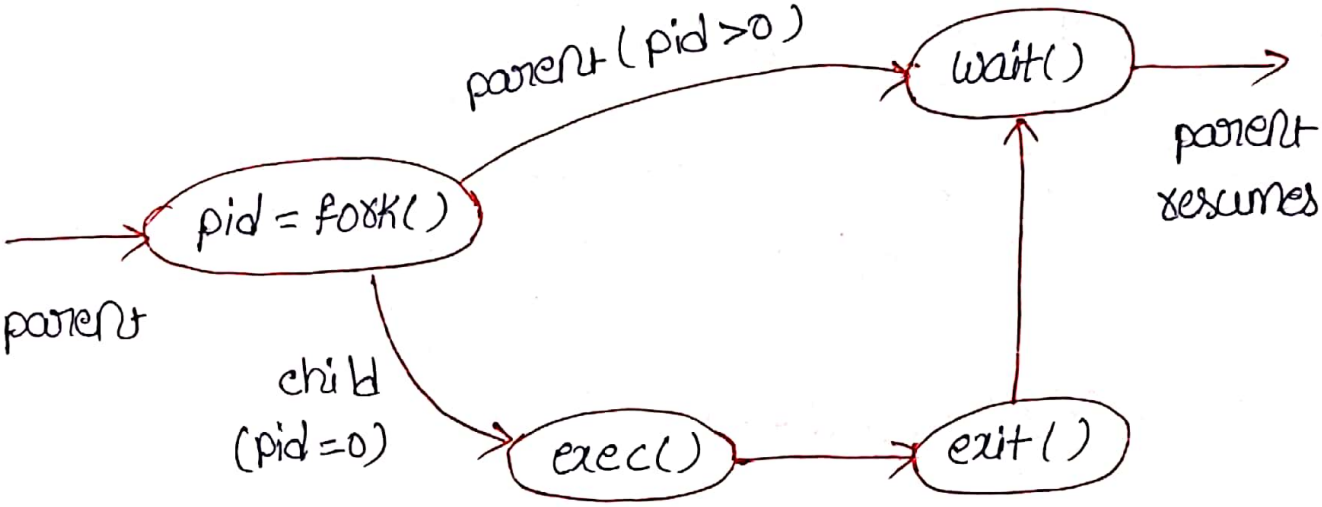

Fig:- process creation using fork() system call.

process creation in windows OS:-

Processes are created in the windows API using the CreateProcess() function which is similar to fork(). The only difference is address-space. In fork() the child duplicates [uses] the parent address space, but in CreateProcess() new address space is allocated to child process.

The other difference is fork() won't take any parameters, where as CreateProcess() will take parameters (not less than ten). The following c program illustrates the CreateProcess(), which creates a child process that loads an application mspaint.exe.

```
#include <stdio.h>
#include <windows.h>
int main(VOID)
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    /* allocate memory */
    ZeroMemory (&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory (&pi, sizeof(pi));
    /* create child process */
    if (!CreateProcess (NULL, "C:\\WINDOWS\\system32\\
        mspaint.exe", NULL, NULL, NULL, FALSE,
        0, NULL, NULL, &si, &pi))
    {
        fprintf (stderr, "create Process Failed");
        return -1;
    }
    WaitForSingleObject (pi.hProcess, INFINITE);
    printf ("child complete");
```

/* parent wait till child to complete */

```
/* close handles */
CloseHandle (pi.hProcess);
CloseHandle (pi.hThread);
}
```

Fig:- Creating child process using the windows API

- The CreateProcess() is defined with default values of ten parameters. The two parameters are instances of the STARTUPINFO and PROCESS-INFORMATION.
- STARTINFO specifies many properties such as windows size and appearance and standard input and output files.
- PROCESS-INFORMATION contains a handle and the identifiers to the newly created process and it's various threads.
- ZeroMemory() function allocates memory to each of these structures [process/threads].
- The first two parameters are application name and command-line parameters. If Application name is NULL [as above], the command line parameters specifies the application to load.

In the above example, we are loading the Microsoft windows mspaint.exe application.

- Along with these parameters, we use default parameters for inheriting process and thread handles and specifying that there will be no creation flags and parents existing environment and starting directory, along with two pointers which are created at the beginning.

- The parent process waits for the child to complete by invoking wait() system call, In this we waitForSingleObject() system call with parameter as child process handle.

Once child process is terminates, the control returns from waitForSingleObject() function in the parent process.

• Process Termination :-

A process terminates when it finishes execution and ask the operating system to delete the deallocate physical and virtual memory, open files and I/O buffers by using system calls.

Sometimes termination of one process can cause the termination of another process via appropriate system call. it can happened only by parent process.

The parent may terminate the execution of one of its childers for variety of reasons, such as:

- The child has exceeded its usage of some of the resources that it has been allocated.
- The task assigned to the child is no longer required.
- The parent is terminated; in this case the operating system does not allow a child to continue without parent.

The phenomenon, in which the child is not allow to exist if its parent has terminated is called as "Cascading termination".

For example, in Linux and UNIX system, we can terminate a process by using exit() system call with exit status para as parameter:

```
/* exit with status 1 */
exit(1);
```

if the parent process is waiting for child to terminate by using wait() system call. The the wait() system call defined with parameter as exit status of child. wait() system call return process_id, so that parent can identify which of it's children has terminated:

```
pid_t pid;
int status;
pid = wait(&status);
```

- when a process terminates, its resources are deallocated by the operating system. The operating system store the pro current processes information in the form of process table.

- If parent process is waiting for child to terminate by using wait() system call, even the child process entry exist after its termination. This is beac

- when a process terminates, all it's allocated resources are deallocated by operating system. The operating system maintains a process table which holds a information about the current processes.

- A process which has finished it's execution but it's entry exist in the process to report to its parent process is known as zombie process.

- A child process always first becomes a zombie process.

- whenever parent executes the wait() system call, it will get the exit status of child, then only the child process entry will be removed from process table.

- In case, parent process didn't invoke wait() the child processes considered as orphans. In UNIX and LINUX all orphan processes assigning to init process as the new parent.

Inter-process communication :- [IPC]

Inter process communication is the mechanism provided by the operating system that allows processes to communicate each other. the processes executing concurrently in the operating system may be either:

- independent processes
- cooperating processes.

- The processes are said to be independ it can not affect or affected by the other processes.

i.e Any process that does not share data with any other process is independent.

- A process is said to be cooperating if it affects or affected by other processes.

i.e Any process that share data with other process is cooperating.

Cooperating processes requires "Inter-process communication [IPC]", which will allow processes exchange data and information. there are two fundamental models of interprocess communi-cation:

1. shared memory
2. Message passing.