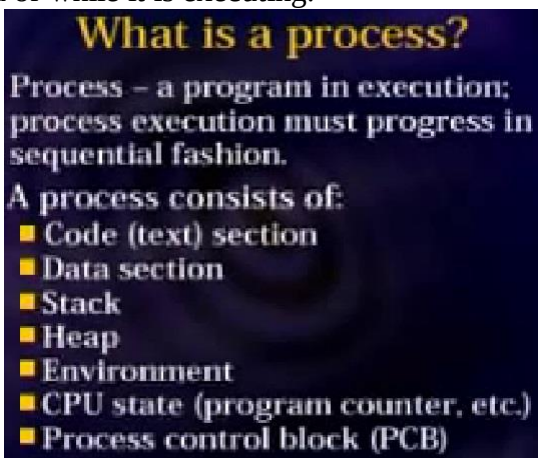
Operating System

Process and CPU management

Shahzad Ali Rana
Lecturer GCUF
03006641562
https://www.facebook.com/shahzad.rana.127

Process Concept

A process can be thought of as a program in execution. A process will need certain resources such as: CPU time, memory, files, and I/O devices – to accompany its task. These resources are allocated to the process either when it is created or while it is executing.



A process is the unit of work in most systems. Such a system consists of a collection of processes: operating system processes execute system code and user processes execute user code. All these processes may execute concurrently.

Although traditionally a process contained only a single thread of control as it ran, most modern operating systems now support processes that have multiple threads. A batch system executes jobs (background processes), whereas a time-shared system has user programs, or tasks. Even on a single user system, a user may be able to run several programs at one time: a word processor, web browser etc.

A process is more than program code, which is sometimes known as the text section. It also includes the current activity, as represented by the value of the program counter and the contents of the processor's register. In addition, a process generally includes the process stack, which contains temporary data (such as method parameters, the process stack, which contains temporary data), and a data section, which contains global variables.

A program by itself is not a process: a program is a passive entity, such as contents of a file stored on disk, whereas a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources. Although two processes may be associated with the same program, they are considered two separate sequences of execution. E.g. several users may be running different instances of the mail program, of which the text sections are equivalent but the data sections vary.

Processes may be of two types:

- **IO bound processes:** spend more time doing IO than computations, have many short CPU bursts. Word processors and text editors are good examples of such processes.
- **CPU bound processes:** spend more time doing computations, few very long CPU bursts.

Process States

As a process executes, it changes states. The state of a process is defined in part by the current activity of that process. Each process may be in either of the following states, as shown in Figure:

- **New:** The process is being created.
- **Running:** Instructions are being executed.
- **Waiting:** The process is waiting for some event to occur (such as an I/O completion or reception of a signal.
- **Ready:** The process is waiting to be assigned to a processor.
- **Terminated:** The process has finished execution.

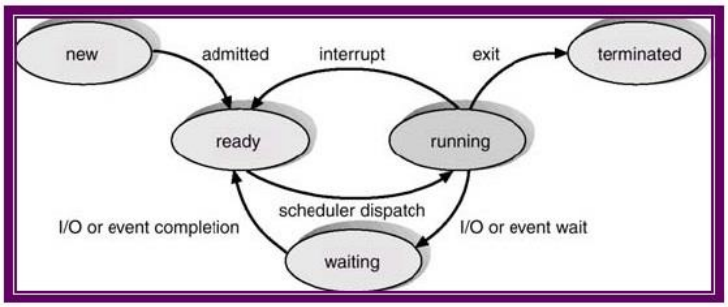
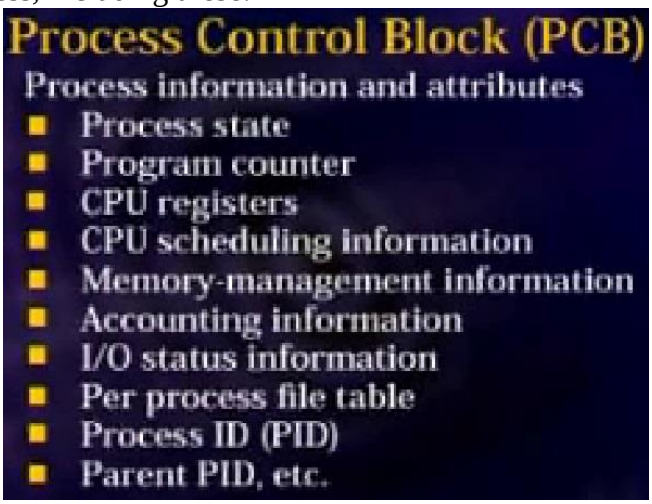


Figure Process state diagram

Process Control Block

Each process is represented in the operating system by a **process control block** (PCB) also called a task control block, as shown in Figure. A PCB contains many pieces of information associated with a specific process, including these:



- **Process state:** The state may be new, ready, running, and waiting, halted and so on.
- **Program counter:** The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers:** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers and general-purpose registers, plus any condition code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterwards.

- **CPU Scheduling information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information:** This information may include such information such as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.
- **Accounting information:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- **I/O status information:** The information includes the list of I/O devices allocated to the process, a list of open files, and so on.

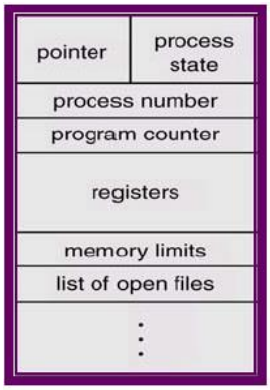


Figure Process control block (PCB)

Process Scheduling

The objective of multiprogramming is to have some process running all the time so as to maximize CPU utilization. The objective of time-sharing is to switch the CPU among processors so frequently that users can interact with each program while it is running. A uniprocessor system can have only one running process at a given time. If more processes exist, the rest must wait until the CPU is free and can be rescheduled. Switching the CPU from one process to another requires saving of the context of the current process and loading the state of the new process, as shown in Figure 5.4. This is called **context switching**.

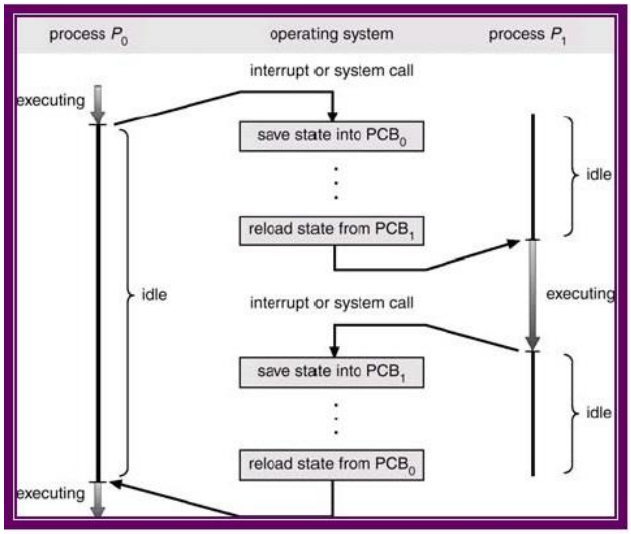
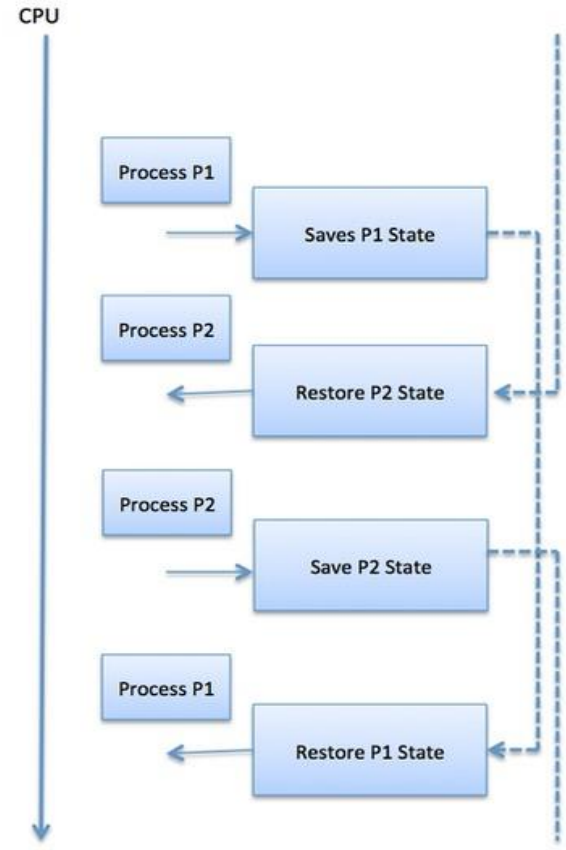


Figure Context switching

Context Switch

A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.



Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers. When the process is switched, the following information is stored for later use.

- Program Counter
- Scheduling information
- Base and limit register value
- Currently used register
- Changed State
- I/O State information
- Accounting information

CPU MANAGEMENT

Scheduling Queues

As shown in Figure 5.5, a contemporary computer system maintains many scheduling queues. Here is a brief description of some of these queues:

- **Job Queue:** As processes enter the system, they are put into a job queue. This queue consists of all processes in the system.
- **Ready Queue:** The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the ready queue. This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB is extended to include a pointer field that points to the next PCB in the ready queue.
- **Device Queue:** When a process is allocated the CPU, it executes for a while, and eventually quits, is interrupted or waits for a particular event, such as completion of an I/O request. In the case of an I/O request, the device may be busy with the I/O request of some other process, hence the list of processes waiting for a particular I/O device is called a device queue. Each device has its own device queue.

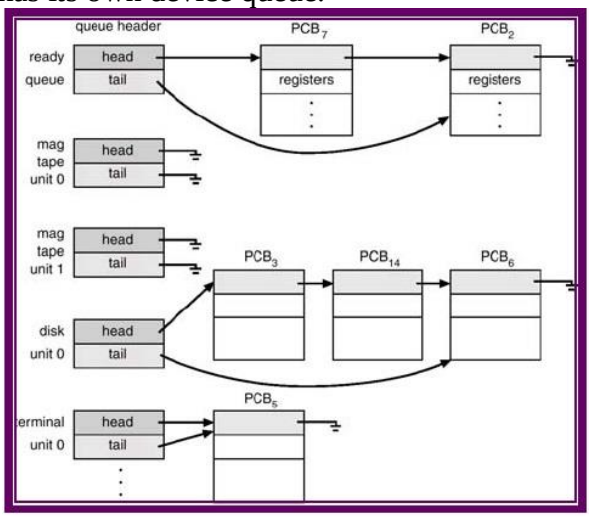


Figure Scheduling queue

In the queuing diagram shown in Figure below, each rectangle box represents a queue, and two such queues are present, the ready queue and an I/O queue. A new process is initially put in the ready queue, until it is dispatched. Once the process is executing, one of the several events could occur:

- The process could issue an I/O request, and then be placed in an I/O queue.
- The process could create a new sub process and wait for its termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

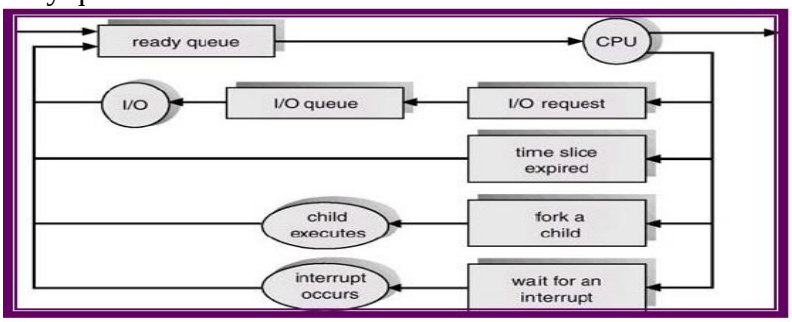


Figure Queuing diagram of a computer system

Process Scheduling Techniques

Schedulers

A process migrates between the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The appropriate scheduler carries out this selection process.

Schedulers are special system software which handles process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types

- Long term scheduler
- Medium term scheduler
- Short term scheduler

Long Term Scheduler

It is also called a job scheduler. A long-term scheduler determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling.

The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

On some systems, the long-term scheduler may not be available or minimal. Time-sharing operating systems have no long term scheduler. When a process changes the state from new to ready, then there is use of long-term scheduler.

Short Term Scheduler

It is also called as CPU scheduler. Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them.

Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers.

Medium Term Scheduler

Medium-term scheduling is a part of swapping. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium-term scheduler is in-charge of handling the swapped out-processes.

A running process may become suspended if it makes an I/O request. A suspended process cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other processes, the suspended process is moved to the secondary storage. This process is called swapping, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

Comparison among Scheduler

S.N.	Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
3	It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
4	It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.
5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

Swapping:

Swapping is a general term for exchanging blocks of program code or data between main and secondary memory. An executing program encounters swapping in three ways:

• **Swapping in:** Moving the entire program into main memory at the start of execution.

• **Swapping out:** Moving the entire program out of main memory at the completion of execution.

•**Page swapping:** Moving individual pages of the program in or out of main memory during execution of a program.

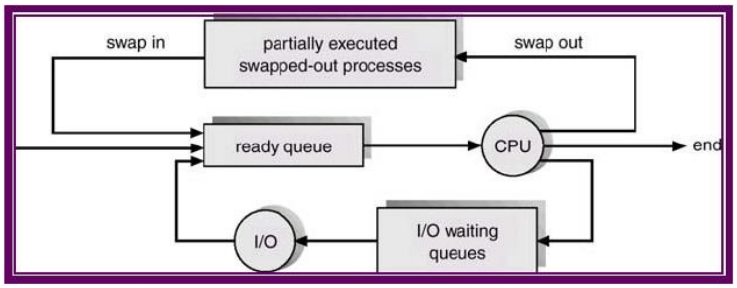


Figure Computer system queues, servers, and swapping

The time to complete a swap is typically 10,000 to 100,000 times the basic instruction time. Since swapping is so expensive compared with instruction execution, system designers have always sought ways to minimize and mask it.

The term originated in the time-sharing systems of the early 1960s. These systems had insufficient memory to allow more than one program to be loaded (swapped in) for execution at a time. At the end of a time slice or stop for input/output, their operating systems swapped out the executing program and then swapped in another program. A single program could be swapped many times during its execution. Swapping was a perfect description of the main work of time-sharing -- switching the CPU from one program to another.

In those early systems, swapping and CPU execution were disjoint activities. The operating system controlled the swapping overhead by setting time slices to be multiples of the average swapping time. MIT's CTSS was able to guarantee that the CPU would be executing programs about 50% of the time with this strategy (1).

To improve CPU efficiency to near 100%, operating systems of the late 1960s incorporated multiprogramming. Swapping was limited to swapping in and swapping out. The CPU could be switched among loaded programs without further swapping. Swaps were masked by performing them in parallel with CPU execution, without interrupting or slowing the CPU. Multiprogramming is often combined with virtual memory. In that case, the operating system may allocate fewer pages of memory to a program than the size of its address space. There will be page swapping during execution. The operating system maintains a complete copy of the address space in a swap file on the disk; each page fault swaps a page from that file with a page from main memory. The paging algorithm attempts to minimize page swapping.

Modern operating systems use swapping in all these forms. Windows XP and Vista, Mac OS 10, and Linux all combine multiprogramming and virtual memory. They allow users the option of turning off virtual memory; in which case, the operating system will swap in a program's full address space at the beginning and then execute without page swapping.

Operations on Processes

The processes in the system execute concurrently and they must be created and deleted dynamically thus the operating system must provide the mechanism for the creation and deletion of processes.

Process Creation

A process may create several new processes via a create-process system call during the course of its execution. The creating process is called a **parent process** while the new processes are called the **children** of that process. Each of these new processes may in turn create other processes, forming a tree of processes. Figure shows partially the process tree in a UNIX/Linux system.

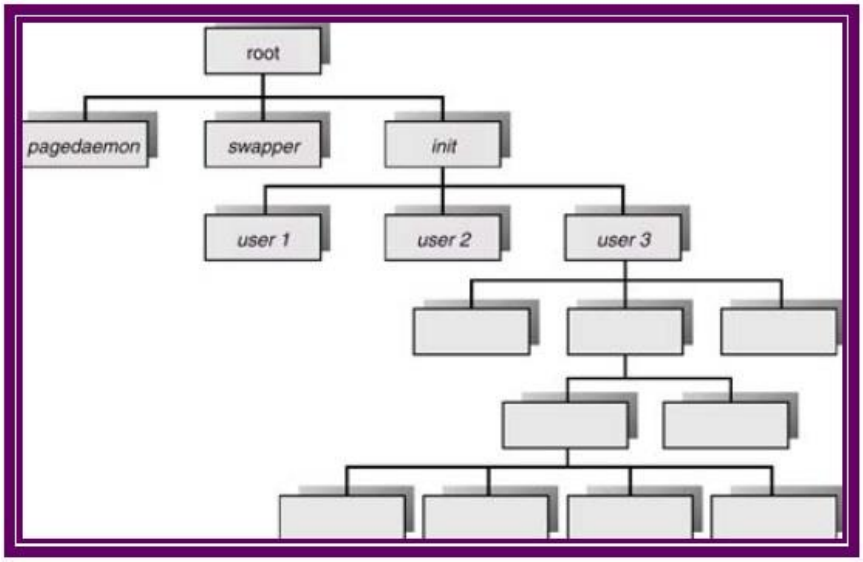


Figure Process tree in UNIX/Linux

In general, a process will need certain resources (such as CPU time, memory files, I/O devices) to accomplish its task. When a process creates a sub process, also known as a child, that sub process may be able to obtain its resources directly from the operating system or may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among several of its children. Restricting a process to a subset of the parent's resources prevents a process from overloading the system by creating too many sub processes.

When a process is created it obtains in addition to various physical and logical resources, initialization data that may be passed along from the parent process to the child process. When a process creates a new process, two possibilities exist in terms of execution:
1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

There are also two possibilities in terms of the address space of the new process:
1. The child process is a duplicate of the parent process.
2. The child process has a program loaded into it.

In order to consider these different implementations let us consider the UNIX operating system. In UNIX its process identifier identifies a process, which is a unique integer. A new process is created by the fork system call. The new process consists of a copy of the address space of the parent. This mechanism allows the parent process to communicate easily with the child process. Both processes continue execution at the instruction after the fork call, with one difference, the return code for the fork system call is zero for the child process, while the process identifier of the child is returned to the parent process.
Typically the execlp system call is used after a fork system call by one of the two processes to replace the process' memory space with a new program. The execlp system call loads a binary file in memory –destroying the memory image of the program containing the execlp system call.—and starts its execution. In this manner, the two processes are able to communicate and then go their separate ways. The parent can then create more children, or if it has nothing else to do while the child runs, it can issue a wait system call to move itself off the ready queue until the termination of the child.
The parent waits for the child process to terminate, and then it resumes from the call to wait where it completes using the exit system call.

Process termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by calling the exit system call. At that point, the process may return data to its parent process (via the wait system call). All the resources of the process including physical and virtual memory, open the files and I/O buffers – are de allocated by the operating system.
Termination occurs under additional circumstances. A process can cause the termination of another via an appropriate system call (such as abort). Usually only the parent of the process that is to be terminated can invoke this system call. Therefore parents need to know the identities of its children, and thus when one process creates another process, the identity of the newly created process is passed to the parent.
A parent may terminate the execution of one of its children for a variety of reasons, such as:

- The child has exceeded its usage of some of the resources that it has been allocated. This requires the parent to have a mechanism to inspect the state of its children.
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates. On such a system, if a process terminates either normally or abnormally, then all its children must also be terminated. This phenomenon referred to as cascading termination, is normally initiated by the operating system.

Considering an example from UNIX, we can terminate a process by using the exit system call, its parent process may wait for the termination of a child process by using the wait system call. The wait system call returns the process identifier of a terminated child, so that the parent can tell which

of its possibly many children has terminated. If the parent terminates however all its children have assigned as their new parent, the init process. Thus the children still have a parent to collect their status and execution statistics.

System Calls:

3.6 SYSTEM CALLS

The system calls are also known as *supervisor calls*. A system call is how a program requests a service from an operating system's kernel. System calls provide an interface between the running application and operating system. These are used to request the operating system to perform different operations in the computer system. For example, a process is created and deleted through system calls. Similarly, the process can change its state from one to another and many other I/O operations can be performed through system calls.

Suppose an application is developed to make a duplicate copy of data from one file to another. Two filenames must be given to the application. First filename as "input file" to read the data and second as "output file" to write the data. The application must open the input file and create the output file. Each of these I/O applications requires many I/O system calls, such as:

- ★ There may be error conditions for each operation. When the program tries to open the input file, it may find that there is no file of that name or that file is protected against access. So the program displays a message through system call. Similarly, program is terminated abnormally through another system call.
- ★ If the input file exists, then program must create new output file. If the output file already exists with the same name, then program should be terminated through system call or delete the existing file through system call and create new one using another system call etc.
- ★ If both input and output files are opened (or set up), the program reads data from input file through system call and writes data to output file through another system call. In data reading process, one of the following events may occur.
  - ➢ Program may find that end-of-file has been reached.
  - ➢ Hardware failure may occur during reading process.
  - ➢ Incorrect data format may be found.

  So a message is displayed through system call and the program may terminate through another system call.

- ★ Similarly, different errors may occur during write process. So a message is displayed through system call and the program may terminate through another system call.
- ★ If there is no error in I/O operation and the duplication of file is created successfully, the program closes both files through system call and displays a message through another system call. Finally, the program is terminated normally through system call.

It is to be noted that in a simple application several system calls may have been used. A computer system may execute thousands of system calls per second. Most programmers never see this level of detail. The system calls are generally available as routines within C and C++ etc.

3.6.1 Types of System Calls

The main types of system calls are as follows:-

1- Process Control

These types of system calls are used to control the processes such as;

- ★ End (terminates the process normally).
- ★ Abort (terminates the process abnormally).
- ★ Load and execute process.
- ★ Create process and terminate process.
- ★ Get or set process attributes etc.

In Unix operating system, the most important system calls used to control the processes are as follows;

fork() This system call is used to create new process. After executing fork() system call, the parent process continues running, in parallel with the child. The parent may create many children. Each child may also create its own children through fork() system call. So a tree like structure of processes is created as shown in figure 3.8.

wait() The parent may issue this system call to move itself into waiting state, while the child runs.

exit() This system call is used to terminate the execution of the process.

abort() This system call allows one process to terminate another, but this facility can only be used by parent process.

In addition to fork(), exit(), abort() system calls, there are several other system calls that are used for different purposes, such as for controlling communication between processes etc.

2- File Management

These types of system calls are used for file management. For example; create file, delete file, open file, close file, read file, write file, reposition, get file attributes, and set file attributes etc.

3- Device Management

These types of system calls are used for device management. For example; request & release devices, get & set device attributes, logically attach & detach (remove) devices etc.

4- Communications

These types of system calls are used for communications. For example; create & delete communication connection, send & receive messages, transfer status information, attach & detach remote devices etc.