1ST CHAPTER GENERAL OVERVIEW

In general the *UNIX* Operating System is divided in 2 parts....

1) **Programs & Services Provided By The System: -** This is a user level view of the system. It contains programs & utilities like editors, mail, shell, etc. which are familiar to the user and made mainly of them, by hiding complex features.
2) **The Proper Operating System: -** This part is much exposed to the end user. This is the part, which makes part 1 to operate. In further description, we will mainly consider *UNIX* System, produced by *AT&T Bell Labs*.

History of *UNIX*

In 1965, *AT&T Bell Lab* & *GE (General Electric Company)* & Project MAC of *MIT (Massachusetts Institute of Technology)* develop a new operating system called **Multics**. The primary goal of them, was to develop a multi-operating platform. *[MULTICS – **Mult**iplexed **I**nformation & **C**omputing **S**ervice].*

Due to failure of achievement as the goal & due to over-budget, Bell Lab withdrew its participation. But some of the members of the above project from the *Bell* like ***Ken Thompson*** & ***Dennis Ritchie*** took the idea *Multics* and did a paper design of new operating system in 1969. They implemented this design on *DEC PDP-7* machine and was named as **UNICS** – Where "**MULTI**" is replaced by "**UNI**" by another member ***Brian Kernighan*** *(**Uni**plexed **I**nformation & **C**omputing **S**ervice).* Later on UNICS is made shorter to ***UNIX***.

In 1971, *UNIX* was tried on more powerful machine *PDP-11*. System was very small, requiring 16 KB for system, 8 KB for user programs and maximum limit at 64 KB per file. Thus disk of 512 KB was enough to hold the complete system. (So we can say that the whole system can be stored on 1.2 or 1.44 floppy drive at the time)

Initially the whole system was written in the assembly language and thus not easy to become portable on another machines. In 1972, ***Dennis Ritchie*** developed **'C'** language and in 1973 the whole *UNIX* operating system was re-written in *C* (keeping very small assembly code in it).

The operating system was still in use only for *Bell Laboratory*. But in 1977, it was first ported (or say installed) on *Non-PDP* machine called *Interdata 8/32*.

Realizing its commercial value, *AT&T* released *UNIX* commercially in the world with its source code. But this leaded to chaos, because many institutes made their own *UNIX* standards by adding more and more functionality to it. From 1977 to 1982, AT&T itself created ***UNIX System III*** (for commercial purpose), ***UNIX System IV*** (only for its internal use) and in Jan 1983, the ***UNIX System V***. This last version i.e. ***System V*** was sold commercially and become the most popular one.

Due to availability of source code, *University California* in Berkeley developed their own *UNIX* standard for *VAX* machine, which was called as ***UNIX 4.3 BSD*** *(**B**erkeley **S**oftware **D**istribution).*

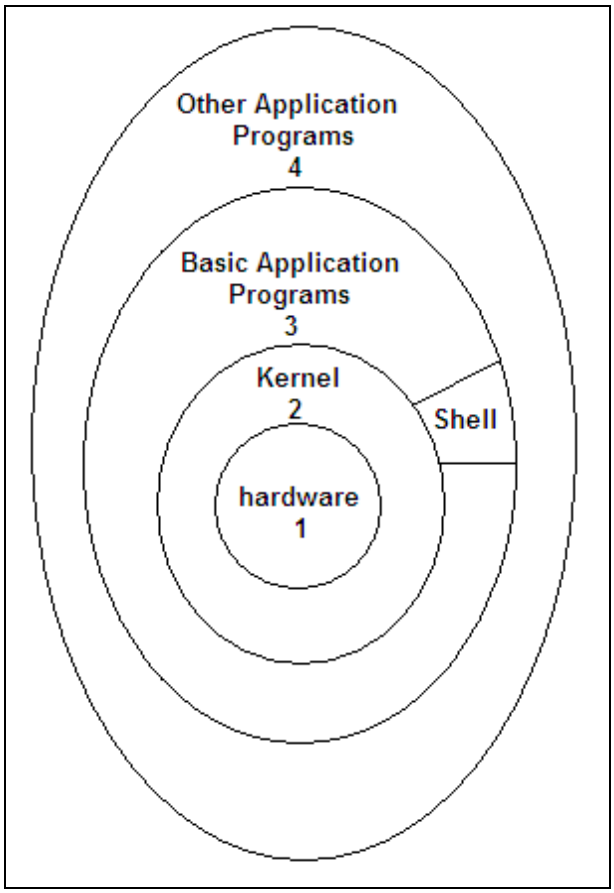
Main Reasons For Popularity Of *UNIX*

1) The system is written in ***C*** language, which is easy to read, easy to understand, easy to move to other machines.
2) It has easy user interface, i.e. more user-needed utilities can be added to it easily.

3) Complex programs can be built by combining existing simple programs (obviously by using pipe).
4) It uses ***Hierarchical*** file system, which is easy to maintain & implement. (*DOS* is influenced with this idea).
5) **In *UNIX*, everything storable is "*file*", means device, terminals, even directories are also files. *UNIX* considers "*file*" as stream of byte. Thus system programming becomes easy.**
6) Above idea gives simple interface to all attached peripheral devices like terminal, mouse, keyboard, printer, etc.
7) **It is Multi-user, Multi-tasking, Multi-threading operating system.**
8) It hides hardware (i.e. machine architecture) from both the programmer & end user. This makes easy to write programs to run on different hardware architecture.
9) Though system was written in *C*, it provides environmental support to other languages like *Basic, Fortran, Pascal, ADA, Cobol, Lisp, Prolog, C++, Java, etc.*
10) Even more languages, can be added to this system's supporting features, if those languages have their compilers or interpreters which can convert their programs into *UNIX* standard system calls.

The Basic System Architecture :

UNIX System Architecture



Above figure gives idea about *UNIX* system architecture.

! The ***hardware*** is at the center. The major goal is to hide it from the application programs.

- The second layer is ***Kernel*** which is set of such utilities which ...1) Connects application programs to the hardware (allowing programs not to worry about hardware) 2) Provide environment for application programs by effective use of CPU.
- The third layer is of ***basic application programs***, made up of system calls, which...1) On one hand, they provide environment to run various commands and other application programs and 2) On other hand, they communicate with *Kernel*.
- In the outermost layer, there are such application programs, which are built on basic application programs (like ***Shell***) of layer 3. Various compilers, ideas, editors, linkers also reside in this layer.

  - It is not restriction that, the 4th and last layer cannot be extended. You can extend the layers beyond this by building more applications based on the previous layer. So 5th layer (if exists) will contain those applications, which are built with the help of applications of 4th layer.
  - Note that, though you can extend the layers of application programs, all programs from layer 3 to the last, depend on the lower-level services provided by *Kernel*, and hence interaction of every program with the *Kernel* is mercy. This interaction (i.e. communication of application program with the *Kernel*) is done by a set of functions known as ***System Calls***. In ***UNIX System Release V***, there are such 64 system calls, out of which 32 system calls are used much frequently.

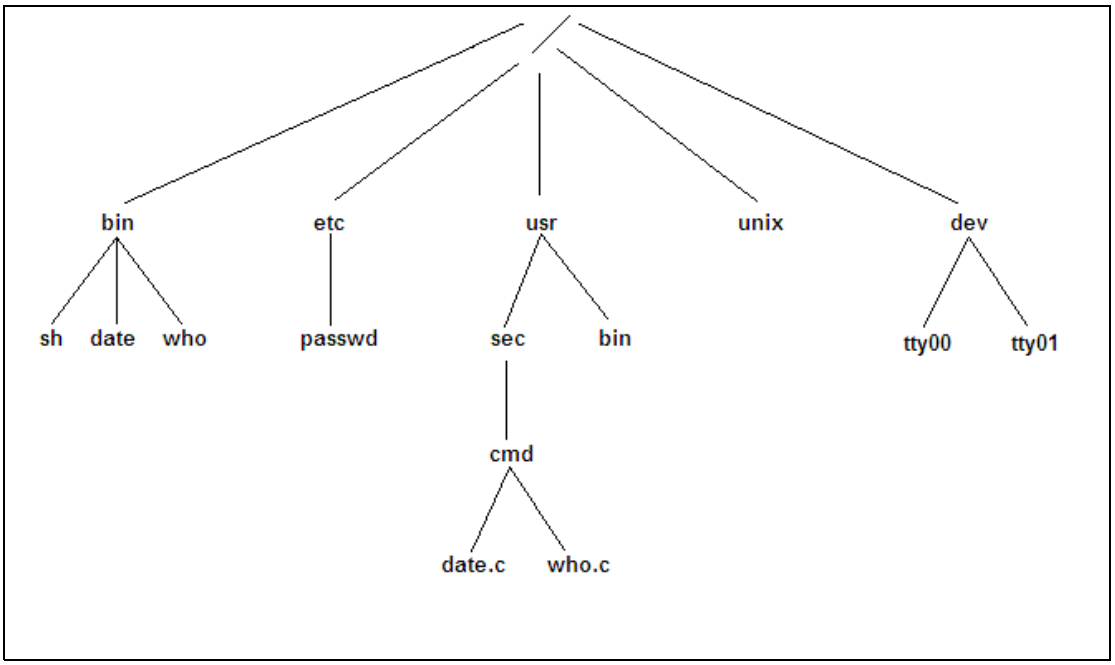
UNIX System : From The Eyes Of User

This includes high-level features of *UNIX* (i.e. users are not supposed to understand the low-level intricacies of the system)

The high-level features of *UNIX* system, mainly includes 3 points :

1) The File System.
2) The Processing Environment.
3) The Building Block Primitives.

1. The *UNIX* File System From The Eyes of User



The *UNIX* file system has following main characteristics: -

a) A hierarchical (i.e. tree like) structure.
b) Consistent treatment to file data (i.e. all files are stream of bytes).
c) Ability to create new & to delete old files.
d) Dynamic growth of files.
e) The protection of file data (by access permissions)
f) Peripheral devices & directories are also considered as files.

The *UNIX* file system is created like a tree structure having a single root node, written as " / ". The branches either end with leaf (i.e. no further division) or with a non-leaf node (i.e. which further may divide & re-divide). Every non-leaf node is considered as directory. These directories further may contain sub-directories (non-leaf) or just files (leaf). The leaf nodes indicate either empty directories (which further, in future may became non-leaf node by containing files in it, but at present they are empty) or regular files or special device files. (Note: regular files & device files always remain as leaf nodes. They can never become non-leaf like directories.)

Its Path Name gives the name of a file. *Path Name* is a sequence of component names, each separated by slash character. By using *Path Name*, location of a file in the file system hierarchy is found out. In *UNIX* everything, that is storable is *file*, hence the word "*component*" in the definition of *Path Name*, also indicates a file (may be regular or directory or device).

To traverse a location of file in the file system, you can start the searching from the top, i.e. from the root (or say " / "). Then follow the branches up to the desired file name. So if you want to locate the file *date.c* in the figure, the *Path Name* will be ***/usr/src/cmd/date.c*** .

In this path name, the beginning " / " is for root. ***usr, src, cmd*** are the components (or directory files) each separated by " / " character and *date.c* is the leaf that we want to locate.

Above path name (or simply path) is full path name, but it begins from the root. But it is not at all always necessary to locate a file from the root. If you are already switched in ***/usr/src*** directory, then the path of *date.c* will be just ***cmd/date.c***.

This is because of path of a file is relative to current directory in which you are switched. Such path name is termed as ***relative path name***.

A very important thing about file is that, *UNIX* treats all files (means even directory or device) as unformatted stream of bytes. Thus all files are same with the respect to *Kernel*. It is the program's duty to format the data when it accesses the file.

To clear this point we will take one example. When a text formatting program like ***troff*** accesses a file, it expects "null termination" of strings in the file and also expects "new line" character in the file data to display the file data line-by-line. So when *Kernel* stores a file by unformatted stream of bytes, it is the duty of ***troff*** program to find "new-line" character in that unformatted data and then arrange the file data in line-by-line format to give formatted output. But the changes made by ***troff*** to give you formatted output are temporary. No changes are made in the internal structure of file and internally the file remains unformatted stream of bytes. Now if you edit a line and add or delete some words in it and save the changes, then don't expect that *Kernel* will store it in formatted form. *Kernel* does not know about formatting and it again saves all changes with unformatted stream of bytes. When you again open the same file with some other text editor program, that program does the necessary formatting and gives you formatted output.

Though it seems that, this approach gives overhead to program, this is very necessary to maintain uniqueness for all files with reference to *Kernel*. And this gives freedom to program to use same system calls to access any type of file – whether regular or directory or device.

So keeping prior explanation in mind, we can say that directories are also files with unformatted stream of bytes. So directories are also just like regular files in respect with *Kernel*.

So we may wish to edit the directory file with any editor. But this will not happen. You will not be able to edit a directory file or device file. Why?
Reasons:
1st - Directories do not have give the permission to read by editor.
2nd - Editors do not have such format to get line-by-line output and…
3rd – System keeps the data in directory file and device file in such a way that the operating system (i.e. *Kernel*) and directory commands only can read them.

The word "permission" is also important from above explanation. Every file in *UNIX* system has a set of "Access Permission". This is due to "High Security" nature of *UNIX* system. There are 3 permissions for 3 classes of users of the system. The 3 permissions are ***read*** (octal 4), ***write*** (octal 3) and ***execute*** (octal 1). And the 3 user classes are ***File Owner***, ***File Group***, and *everybody else (i.e.* ***Others****)*. System assigns different permissions to all files, either system files or user-created files. If you are owner of a file you can change the default permissions by ***chmod*** command. If the file is binary executable, it gets executed by *execute* permission. System, by default does not give permission to read or write such executable file. A question may arise, *what mean by* ***execute*** *for directory file ?* : *execute* permissions for a directory file means allow to traverse its contents.

Similar to directory files, devices are also special types of files. They occupy *node position* in the file hierarchy structure. Programs access devices same as they access regular files. Reading & writing in concern with device files is also similar to the process of reading & writing of regular files. Not only that, but their permissions are also similar to regular files.

Discussion on page 5 & 6 is concern with a same point that "***Files in UNIX are unformatted stream of bytes***" and *system treats them with same system calls assessment.*

Thus "*copy oldfile newfile*" command will operate on any type of file, whether it is a regular file or directory file or device file.

2. Processing Environment : From The Eyes of User

A program is an executable file and a process is an instance of the "*program in execution*". Many processes can run simultaneously on *UNIX* system. This feature is called as ***Multi-tasking*** or ***Multi-programming***. Also many instances of one program also can run simultaneously on *UNIX* system. There are system calls to create a new process, terminate an old process, synchronization of stage of process execution and reaction of a process to an event. Note that different processes or different instances of same process get execute independently of each other.

The major 4 system calls for process management are ***fork()***, ***exec()***, ***wait()*** and ***exit()***, where *fork()* is used to create a new process, *exec()* is used to execute the new process, *wait()* is used to allow the old process to wait for the completion of the execution of new process and *exit()* is used for exiting the program.

Here the old process which creates new process (by executing *fork()* system call) is called as ***Parent Process*** and the newly created process is called as ***Child Process***.

When *exec()* executes a child process, the child process overlaps (or more precisely overlays) the address space (memory) of parent process. And thus there remains no memory for parent process to run, thus parent process must wait for the completion of execution of child.

The *UNIX Kernel* contains very few system calls. This is contrast to the *Kernels* of other operating systems. Other operating system *Kernels* are rich in functions. *UNIX Kernel* is comparatively very small because using system call functions users or programmers can create their own programs to do their desired tasks. One of such program (Not the part of *UNIX* Operating System, but nowadays distributed along with *UNIX Operating System Software Distribution Package*) is *Shell*. This program is called ***Command Interpreter*** (like DOS has its Command Interpreter as ***command.com*** file.) of the *UNIX* system. This program gets automatically executed when you log into the system. Means appearance of command prompt (# or $ or % or something else) with blinking cursor is the indication, that the command interpreter program i.e. the *Shell* is started.

You can execute 3 types of commands on the *Shell* prompt (another name for command prompt) :

1) An executable program (just by its name of file. Extension is not necessary)
2) The internal command of the *Shell* itself. (*Shell* program itself has some commands known as its internal commands)
3) Executable programs & internal commands of *Shell* can be grouped together (to do a specific job) in a file called as ***Shell Script*** and this *Shell Script* itself can be used as a third type of command.

When you write a command at command prompt (any of the above) and press enter to execute it:

- *Shell* considers first word of your command as command name (which may be a program or internal command or *Shell Script*).
- As you know *Shell* itself is a program, and the command is another program, you can say that *Shell* is a parent process and the command is a child process. In other words, *Shell* program uses *fork()* system call to start command as child process. In words of programming, *Shell* forks child process and child process uses *exec()* system call to execute the command.
- This is about the first word of command line. Then what about the other words of command line? : Those other words are considered as command line arguments (i.e. *argv[ ]*) of the first word (i.e. of the command itself). In other words, the second third, fourth and etc. words are the parameters of the first word.
- The *Shell* (i.e. parent process of command) usually executes its command (i.e. child process) synchronously; means *Shell* waits until the command execution gets completed. When command execution completes, *Shell* prompt re-appears, indicating that the command execution completed.

```
#                       → Empty command prompt
# who                   → Command is typed and Enter pressed.
  vijay tty01....       → Output of who command.
#                       → Shell prompt re-appears indicating who is finished.
```

- But you can allow the *Shell* to behave asynchronously, means you can allow *Shell* to display its command prompt quickly to enter new command, before the previous command is yet not finished. This is mainly done when a task is a big job (say printing a large file) and you want to continue with some other tasks while the big job is running in background. To do this, you have to do a minute change in the big job's command. Just post fix the command (of big job) with "&" and then press enter.

```
#                    → Empty command prompt
# who &              → New command who runs in background.
#                    → Shell prompt immediately appears, allowing you to enter
                       new command. Note yet who is not completed.
```

This method is "Asynchronous" because *Shell* does not wait for previous command to get finished and immediately allows to enter new command..

Execution Environment – Every process running in *UNIX* system has execution environment, which includes the current directory. You can change the directory by using ***cd*** command. "**..**" these 2 dots represent parent directory of the current directory.

2. Building Block Primitives (i.e. Redirection & Pipe)

Before, we stated that you could use small *UNIX* programs (or utilities or commands) to develop big, sophisticated programs. To do this *UNIX* system provides you 2 building block primitives, which are used with small programs to develop big programs. :

A) I/O Redirection
B) Pipe

A) I/O Redirection

Every process in *UNIX* (or say every program or every command) has access to 3 files. One standard input file from which process sends it input to standard output file to which process writes its output and the standard error file to which it writes its error messages, if errors occurred. For the sake of unity, system uses the ***terminal*** as representative of these 3 files. Means a process treats *terminal* (i.e. Console) as standard input file and reads its input from the *terminal*. Also process treats *terminal* as standard output file and writes its output to *terminal*. Similarly, process treats *terminal* as its standard error file and writes process's error message to the same *terminal*. In other words "***terminal***" device file is treated as the 3 above files.

But by using Redirection I/O property, process can change this default behavior and can read from or can write to or can put error to the redirected target.

E.g.: If we issue following command..

 # who

Then as there is no redirection used, by default output will go to ***terminal*** (to console) because the standard output file is the ***terminal*** by default. So output will be seen on *Screen*.

For using Redirection Operators "**< , & , >**" are used "to read from" and "to write to" respectively. Thus if we issue the same above command as...

 # who > user.txt

Then output will not go to default ***terminal***, but will go to ***user.txt*** and will be written on ***user.txt*** file. In other words using "**>**" redirection output operator, we change the "stardard output file" from the default ***terminal*** to ***user.txt*** file. Thus obviously we will not be able to see the output on *Screen* (i.e. ***terminal***) instead we have to open the file ***user.txt*** to see the output of ***who*** command.

Similarly.....

 # mail vijay

command expects you to type a letter at ***terminal*** using keyboard, which will be then posted to user ***vijay***. But if you issue…….

 # mail vijay < letter.txt

Then this time command does not expect you type letter from keyboard, but it reads its input from ***letter.txt*** file and then sends it to user ***vijay***. This happens because using Redirection input operators "<", you changed the default "standard input file" i.e. the ***terminal***, to ***letter.txt***.

Similarly, if an error occurs, it is displayed on ***terminal*** by default, as ***terminal*** is the default standard error file. But if you want to store error message in a file, then you can use Redirection operator ">" with ***file descriptor*** 2 (which represents error) as "***2>***".

 # cp 2> error.txt

In above command, ***cp*** (which is *UNIX*'s copy command) expects 2 paramters, we have not passed any, hence obviously an error will occur. But now error will not be displayed on ***terminal***, but due to ***2>*** operator, it will be redirected to ***error.txt*** file and to see the above error, we have to open the ***error.txt*** file.

We used "<", ">", & "***2>***" separately in above three examples. But flexibility of *UNIX* allows us to combine any of them at a time in one single command line. Such as…

 # wc < vijay.txt > count.txt 2> error.txt

In above command, ***wc*** (**w**ord **c**ount) reads ***vijay.txt*** as its input, counts number of characters in it and writes its output to ***count.txt*** and if any error occurs then writes its errors to ***error.txt*** file.

All above discussion tells another important fact that, if neither of < or > or ***2>*** are used, *Shell* takes default standard input file, default standard output file and default standard error file as only and only the ***terminal***. But if any of above operator is used, then *Shell* changes its default behavior and assign 3 standard files respectively, before executing the actual process (i.e. command).

B) Pipe

This is the second building block primitive provided by *UNIX* system. This is denoted by " | " symbol. Pipe allows the flow of data to be passed between a reader and a writer process. Means a process can redirect its output to a pipe and the same pipe is used as input by another process and both these operations are done in one single command line.

To understand the concept, we will take a simple example. It is obvious that, when we want to store some command's output, we will use redirection and will store output in some will deserve file.

But suppose, we want to count the number of words in the output of ***who*** command, then…

 # who > users.txt

 # wc < users.txt

Above two commands will do out job by using redirection. But we know that ***who*** command outputs the current users (one or many) on the system and next time the numbers & users will not be same. So creating ***users.txt*** file is temporary and after getting the job done, we will delete the ***users.txt*** file to avoid un-necessary waste of storage.

So, for such tasks, redirection adds two un-necessary efforts: 1) to create a temporary file and 2) to delete that temporary file.

So, there must be some way by which this intermediate file creation can be avoided. And ***the way is Pipe***.

We will do the same above job by using pipe in one single command line…
 # who | wc

This command executes ***who***, writes it's output to pipe, the same pipe is used as input by ***wc*** command and ***wc*** counts words in the pipe and displays the count on the ***terminal***. No intermediate file creation is done and obviously no deletion of intermediate file is necessary.

Conceptually, data written by the first process (1st command) on pipe becomes input by the second process (2nd command).

You can use as much you want the number of pipes in your command line. Only the point of syntax to remember is that, *the command on left side of the pipe must be "a output giving" command (i.e. writer command) and the command on the right side of the pipe must "a input taking" command (i.e. reader command)*.

Even if you wish, you can combine redirection & pipe together in a single command.

Operating System Functions

Kernel is heart of every Operating System and *UNIX* is not an exception to this fact. Rather in *UNIX*, it is said, "***Kernel is Operating System and Operating System is Kernel***". The most important functions of any Operating System (obviously including *UNIX*) are….

1) Process creation, termination, suspension and inter-process communication.
2) CPU scheduling for multiple processes in "time-shared" manner. (here it is assumed that system has one CPU)
3) Memory allocation (either by swapping or paging) for executing process/processes.
4) File I/O, by creating file system.
5) Insulates processes from hardware of the machine.

★ Two Modes Of A Process:

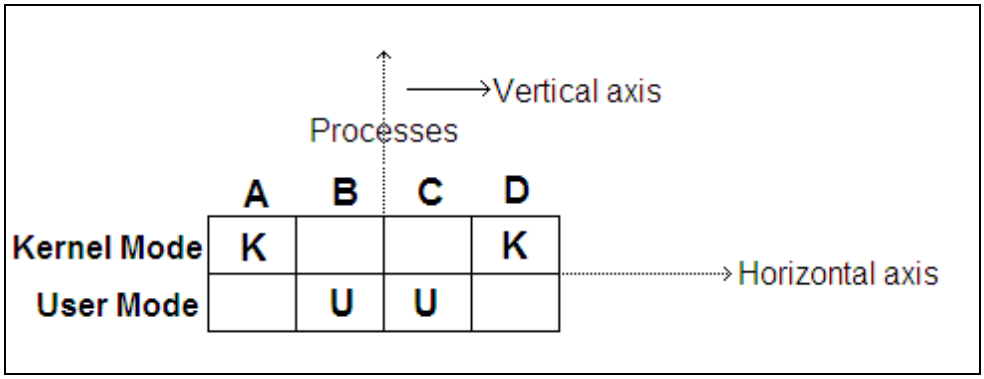
Usually a program (process) contains "two type" code. One is the part of system calls and other is the part of comparison, assignment, loop like operations.

According to this strategy, a process (i.e. running program) runs in two modes: - *Kernel Mode* & *User Mode*.

Kernel Mode: When a process makes a system call, then it said that, now the process is running in *Kernel Mode*.

User Mode: When the process is doing operations like assignments, comparisons, looping, then it is said that, now the process is running in *User Mode*.

The program's code is mixed with system calls & operations and thus a process of that program must switch from *Kernel Mode* to *User Mode* and from *User Mode* to *Kernel Mode* whenever necessary.



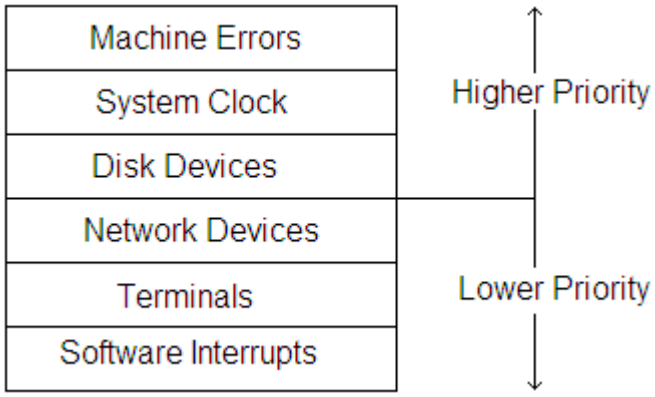
In above figure, it is assumed that processes A, B, C & D are currently running simultaneously on the system. Out of the 4 processes, A & D are running in *Kernel Mode* (denoted by K), means they are making some system calls. While processes B & C are running in *User Mode* (denoted by U), means they are doing some operations like assignment, comparison, looping, etc.

Kernel distinguishes the 4 processes on horizontal axis of the figure, while hardware distinguishes the 4 processes on vertical axis of the figure.

⋆ Interrupts:

While a process is running in *Kernel Mode*, means while a system is making system calls, then some system calls are such that, they allow devices like peripheral I/O devices (disk) or system clock (timer) to interrupt the CPU asynchronously. On receiving such interrupt, *Kernel* saves its current context (***a frozen image of what the process is doing currently***), then it determines the cause of interrupt and first services interrupt by suspending the execution of the process. After completion of servicing the interrupt *Kernel* restores the context (saved previously as frozen image) and re-starts the execution of the blocked/suspended process (from the point where it was stopped) as if nothing had happened.

While handing the interrupts, *Kernel* gives first preference to high priority interrupts and then to low priority interrupts. Then a question may arise that *who decides the high or low priority interrupts ?*. The priority of the interrupt is usually decided by the hardware itself. For example – Machine errors, system clock, disk devices are high priority interrupts hardware and software interrupts, terminals & network devices are low priority interrupt hardware.



(Interrupts are not serviced by special programs or say special processes, but they are serviced by some special functions in the *Kernel*, which are called by the *Kernel* in the context of currently running process.)

⋆ Exceptions:

Exception is an un-excepted event caused by a process. E.g. – Addressing illegal memory by using pointer, execution of un-allowed instruction (like dividing by 0) etc. Note that, interrupts & exceptions are different terms. Interrupts occur by the events which are external to the process while exception occurs "in the middle" of the process due to code instruction.

It is also said that, interrupt occurs between the two instructions and exception occurs at the instruction (obviously at illegal instruction).

For interrupt, system starts the next instruction normally after handling the interrupt.
But for exception, system handles the exception, and then either re-starts the same instruction (at which exception occurred) and again gives the exception or allows user to terminate (abort) the current process.

⋆ Processor Execution Levels:

As seen on the previous page, the figure shows the priority of interrupts. You can set the processor execution level to one of the level in that figure, so that the leveled interrupt & the lower to the set level are bypassed and only high priority levels are taken into consideration.

E.g. If a processor execution level is set to "Disk Interrupt Level" (see in figure), then only "System Clock" & "Machine Error" interrupts are processed and remaining all lower level interrupts (including the set level) are prevented.

This type of processor execution level setting is necessary, particularly for 2 cases…

1) During a Critical Activity: Suppose a *Kernel* is currently processing a linked list in a process and somewhere in between a disk interrupt occurs, then to avoid corrupting of data and pointers, *Kernel* prevents the interrupt.
2) Privileged Instructions: Some instructions (code) in a process are so important that they must be dealt first. Such instructions are called as privileged instructions. During processing of such privileged instructions, the processor execution level is set and certain interrupts (set level interrupt & lower to that) are prevented, allowing only higher priority interrupts to handle.

⋆ Memory Management:

Note that whenever a *UNIX* machine boots, its *Kernel* always resides in main memory (i.e. RAM).

Now when a compiler compiles a program, it creates an object code of that program. In that object code, it creates addresses of variables, addresses of data structures, addresses of functions and machine instructions. As compiler can be ported to any machine, it is machine hardware independent. Thus the addresses created by the compiler are based on an imagination of machine, which is called as ***Virtual Machine*** (not real, but taken as ideal by the compiler or for the compiler). Thus the object code is created for a virtual machine and in that machine there is no space in its main memory to run multiple programs, but to run only this program.

But when actually the program executes on a real machine, now its duty of the operating system (i.e. *Kernel*) to translate those virtual machine addresses to real machine addresses. This is because unlike virtual machine, real machine is made operational by the Operating System to run multiple processes and not the only program. Thus *Kernel* first has to remember which processes are running currently

(except the program's process), whether they fully occupied main memory, if not, then can I assign the remaining memory to the program's process and while doing this, can I be able to translate the program's virtual addresses to remaining real memory addresses and so on. The capability of *Kernel* to map program's virtual addresses to real memory addresses depends on what sized memory chip is installed on the machine. So this is hardware dependent thing. If sufficient memory is available, then *Kernel* can assign memory to the program's process by methods of memory management, like swapping or demand paging.

⋆ Why the *UNIX* kernel is so small? :

UNIX Kernel is made very small by using small number of system calls. This is very much contradictory to other Operating System's *Kernels*, which use many system calls. *UNIX* designers did it intentionally, because they design the system for programmer as their target audience. They gave freedom to programmer to develop new utilities and also allowed them to add those utilities to the Operating System as if they were the part of system designer team.