UNIT 3 CASE STUDY - UNIX

Structure **Page Nos.**

3.0 INTRODUCTION

Operating system is a system software, which handles the interface to system hardware (input/output devices, memory, file system, etc), schedules tasks, and provides common core services such as a basic user interface. The purpose of using an Operating System is to provide:

- **Convenience**: It transforms the raw hardware into a machine that is more agreeable to users.
- **Efficiency**: It manages the resources of the overall computer system.

UNIX is a popular operating system, and is used heavily in a large variety of scientific, engineering, and mission critical applications. Interest in UNIX has grown substantially high in recent years because of the proliferation of the Linux (a Unix look-alike) operating system. The following are the uses of the UNIX operating system:

- Universally used for high-end number crunching applications.
- Wide use in CAD/CAM arena.
- Ideally suited for scientific visualization.
- Preferred OS platform for running internet services such as WWW, DNS, DHCP, NetNews, Mail, etc., due to networking being in the kernel for a long time now.

- Easy access to the core of the OS (the kernel) via C, C++, Perl, etc.
- Stability of the operating environment.
- The availability of a network extensible window system.
- The open systems philosophy.

UNIX was founded on what could be called a "small is good" philosophy. The idea is that each program is designed to do one job efficiently. Because UNIX was developed by different people with different needs it has grown to an operating system that is both flexible and easy to adapt for specific needs.

UNIX was written in a machine independent language and C language. So UNIX and UNIX-like operating systems can run on a variety of hardware. These systems are available from many different sources, some of them at no cost. Because of this diversity and the ability to utilize the same "user-interface" on many different systems, UNIX is said to be an open system.

At the time the first UNIX was written, most operating systems developers believed that an operating system must be written in an assembly language so that it could function effectively and gain access to the hardware. The UNIX Operating System is written in C.

The C language itself operates at a level that is just high enough to be portable to a variety of computer hardware. Most publicly distributed UNIX software is written in C and must be complied before use. In practical terms this means that an understanding of C can make the life of a UNIX system administrator significantly easier.

In the earlier units, we have studied various function of OS in general. In this unit we will study a case study on UNIX and how the UNIX handles various operating system functions. In the MCSL-045, section – 1, the practical sessions are given to provide you the hands-on experience.

3.1 OBJECTIVES

After going through this unit you should be able to:

- describe the features and brief history of unix;
- describe how UNIX executes a command;
- define the use of processes;
- describe the process management, memory management handled by UNIX;
- discuss the UNIX file system and the directory structure, and
- explain the CPU scheduling in UNIX.

3.2 FEATURES OF UNIX OS

UNIX is one of the most popular OS today because of its abilities like multi-user, multi-tasking environment, stability and portability. The fundamental features which made UNIX such a phenomenally successful operating systems are:

- **Multi-user:** More than one user can use the machine at a time supported via terminals (serial or network connection).
- **Multi-tasking:** More than one program can be run at a time.
- **Hierarchical file system: T**o support file organization and maintenance in a easier manner.
- **Portability:** Only the kernel (<10%) is written in assembler. This means that the operating system could be easily converted to run on different hardware.
- **Tools for program development:** Supports a wide range of support tools (debuggers, compilers).

3.3 BRIEF HISTORY

In the late 1960s, General Electric, MIT and Bell Labs commenced a joint project to develop an ambitious multi-user, multi-tasking OS for mainframe computers known as MULTICS. MULTICS was unsuccessful, but it did motivate Ken Thompson, who was a researcher at Bell Labs, to write a simple operating system himself. He wrote a simpler version of MULTICS and called his attempt UNICS (Uniplexed Information and Computing System).

To save CPU power and memory, UNICS (finally shortened to UNIX) used short commands to lessen the space needed to store them and the time needed to decode them - hence the tradition of short UNIX commands we use today, e.g., ls, cp, rm, mv etc.

Ken Thompson then teamed up with Dennis Ritchie, the creator of the first C compiler in 1973. They rewrote the UNIX kernel in C and released the Fifth Edition of UNIX to universities in 1974. The Seventh Edition, released in 1978, marked a split in UNIX development into two main branches: SYS V (System 5) and BSD (Berkeley Software Distribution).

Linux is a free open source UNIX OS. Linux is neither pure SYS V nor pure BSD. Instead, it incorporates some features from each (e.g. SYSV-style startup files but BSD-style file system layout) and plans to conform to a set of IEEE standards called POSIX (Portable Operating System Interface). You can refer to the MCSL-045 Section – 1 for more details in a tabular form. This tabular form will give you a clear picture about the developments.

3.4 STRUCTURE OF UNIX OS

UNIX is a layered operating system. The innermost layer is the hardware that provides the services for the OS. The following are the components of the UNIX OS.

The Kernel

The operating system, referred to in UNIX as the **kernel**, interacts directly with the hardware and provides the services to the user programs. These user programs don't need to know anything about the hardware. They just need to know how to interact with the kernel and it's up to the kernel to provide the desired service. One of the big appeals of UNIX to programmers has been that most well written user programs are independent of the underlying hardware, making them readily portable to new systems.

User programs interact with the kernel through a set of standard **system calls**. These system calls request services to be provided by the kernel. Such services would include accessing a file: open close, read, write, link, or execute a file; starting or updating accounting records; changing ownership of a file or directory; changing to a new directory; creating, suspending, or killing a process; enabling access to hardware devices; and setting limits on system resources.

UNIX is a **multi-user**, **multi-tasking** operating system. You can have many users logged into a system simultaneously, each running many programs. It is the kernel’s job to keep each process and user separate and to regulate access to system hardware, including CPU, memory, disk and other I/O devices.

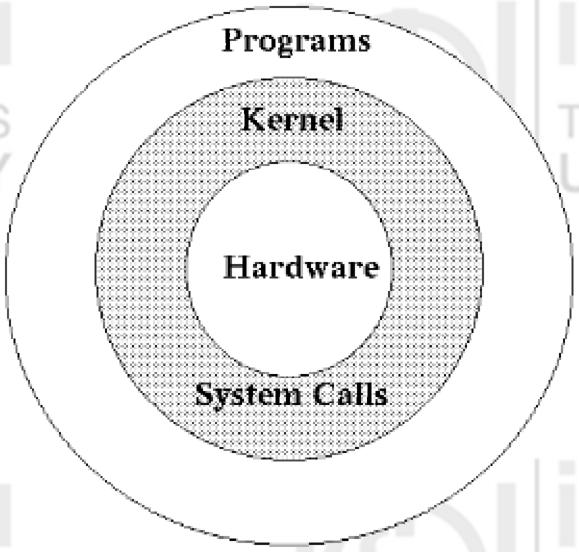

Figure 1: UNIX structure

The Shell

The shell is often called a command line shell, since it presents a single prompt for the user. The user types a command; the shell invokes that command, and then presents the prompt again when the command has finished. This is done on a line-by-line basis, hence the term "command line".

The shell program provides a method for adapting each user's setup requirements and storing this information for re-use. The user interacts with /bin/sh, which interprets each command typed. Internal commands are handled within the shell (set, unset), external commands are cited as programs (ls, grep, sort, ps).
There are a number of different command line shells (user interfaces).

- Bourne (sh)
- Korn (krn)
- C shell (csh)

In Windows, the command interpreter is based on a graphical user interface.
In UNIX, there is a line-orientated command interpreter. More details related to the shells are provided in MCSL-045, Section 1.

System Utilities

The system utilities are intended to be controlling tools that do a single task exceptionally well (e.g., *grep* finds text inside files while *wc* counts the number of words, lines and bytes inside a file). Users can solve problems by integrating these tools instead of writing a large monolithic application program.
Like other UNIX flavours, Linux's system utilities also embrace server programs called **daemons** that offer remote network and administration services (e.g. telnetd provides remote login facilities, httpd serves web pages).

Application Programs

Some application programs include the emacs editor, gcc (a C compiler), g++ (a C++ compiler), xfig (a drawing package), latex (a powerful typesetting language).

UNIX works very differently. Rather than having kernel tasks examine the requests of a process, the process itself enters *kernel space*. This means that rather than the process waiting "outside" the kernel, it enters the kernel itself (i.e. the process will start executing kernel code for itself).

This may sound like a formula for failure, but the ability of a process to enter kernel space is strictly prohibited (requiring hardware support). For example, on *x86*, a process enters kernel space by means of system calls - well known points that a process must invoke in order to enter the kernel.

When a process invokes a system call, the hardware is switched to the kernel settings. At this point, the process will be executing code from the kernel image. It has full powers to wreak disaster at this point, unlike when it was in user space. Furthermore, the process is no longer *pre-emptible*.

3.5 PROCESS MANAGEMENT

When the computer is switched on, the first thing it does is to activate resident on the system board in a ROM (read-only memory) chip. The operating system is not available at this stage so that the computer must "pull itself up by its own boot-straps". This procedure is thus often referred to as *bootstrapping*, also known as cold boot. Then the system initialization takes place. The system initialization usually involves the following steps. The kernel,

- tests to check the amount of memory available.
- probes and configures hardware devices. Some devices are usually compiled into the kernel and the kernel has the ability to autoprobe the hardware and load the appropriate drivers or create the appropriate entries in the /*dev* directory.
- sets up a number of lists or internal tables in RAM. These are used to keep track of running processes, memory allocation, open files, etc.

Depending on the UNIX version, the kernel now creates the first UNIX processes. A number of *dummy processes* (processes which cannot be killed) are created first to handle crucial system functions. A *ps -ef* listing on each OS shows will show you the existing processes. ***init*** is the last process created at boot time. It always has a process ID (PID) of 1. *init* is responsible for starting all subsequent processes. Consequently, it is the parent process of all (non-dummy) UNIX processes.

Don't confuse the *init* process with the system *init* command. The *init* command (usually found in /sbin or /usr/sbin) is used by root to put the system into a specific run level.

All subsequent processes are created by *init*. For example, one of the processes started by *init* is *inetd*, the internet super daemon. (*inetd*, in turn, creates many other processes, such as *telnetd*, on demand.) In the Unix process hierarchy, *init* is called the **parent** process and *inetd* the child of *init*. Any process can have any number of children (up to the kernel parameter *nproc*, the maximum allowed number of processes). If you kill the parent of a child process, it automatically becomes the child of *init*.

Each running process has associated with it a process ID or PID. In addition, each process is characterized by its parent's PID or PPID. Finally, each process runs at a default system priority (PRI). The smaller the numerical value of the PRI, the higher the priority and *vice versa*.

3.5.1 Management of the Processes by the Kernel

For each new process created, the kernel sets up an *address space* in the memory. This address space consists of the following logical segments:

- *text* - contains the program's instructions.
- *data* - contains initialized program variables.
- *bss* - contains uninitialized program variables.
- *stack* - a dynamically growable segment, it contains variables allocated locally and parameters passed to functions in the program.

Each process has two stacks: a *user stack* and a *kernel* stack. These stacks are used when the process executes in the user or kernel mode (described below).

Mode Switching

At least two different modes of operation are used by the UNIX kernel - a more privileged kernel mode, and a less privileged user mode. This is done to protect some parts of the address space from user mode access.

User Mode: Processes, created directly by the users, whose instructions are currently executing in the CPU are considered to be operating in the user-mode. Processes running in the user mode do not have access to code and data for other users or to other areas of address space protected by the kernel from user mode access.

Kernel Mode: Processes carrying out kernel instructions are said to be running in the kernel-mode. A user process can be in the kernel-mode while making a system call, while generating an exception/fault, or in case on an interrupt. Essentially, a mode switch occurs and control is transferred to the kernel when a user program makes a system call. The kernel then executes the instructions on the user's behalf.

While in the kernel-mode, a process has full privileges and may access the code and data of any process (in other words, the kernel can see the entire address space of any process).

3.5.2 The Context of a Process and Context Switching

The context of a process is essentially a snapshot of its current runtime environment, including its address space, stack space, etc. At any given time, a process can be in user-mode, kernel-mode, sleeping, waiting on I/O, and so on. The process scheduling subsystem within the kernel uses a time slice of typically 20ms to rotate among currently running processes. Each process is given its share of the CPU for 20ms, then left to sleep until its turn again at the CPU. This process of moving processes in and out of the CPU is called context switching. The kernel makes the operating system appear to be *multi-tasking* (i.e. running processes concurrently) via the use of efficient context-switching.

At each context switch, the context of the process to be swapped out of the CPU is saved to RAM. It is restored when the process is scheduled its share of the CPU again. All this happens very fast, in microseconds.

To be more precise, context switching may occur for a user process when

- a system call is made, thus causing a switch to the kernel-mode,
- a hardware interrupt, bus error, segmentation fault, floating point exception, etc. occurs,
- a process voluntarily goes to sleep waiting for a resource or for some other reason, and
- the kernel preempts the currently running process (i.e. a normal process scheduler event).

Context switching for a user process may occur also between *threads* of the same process. Extensive context switching is an indication of a CPU bottleneck.

3.5.3 Communication between the Running Processes

UNIX provides a way for a user to communicate with a running process. This is accomplished via *signals,* a facility which enables a running process to be notified about the occurrence of a) an error event generated by the executing process, or b) an asynchronous event generated by a process outside the executing process.

Signals are sent to the process ultimately by the kernel. The receiving process has to be programmed such that it can catch a signal and take a certain action depending on which signal was sent.

Here is a list of common signals and their numerical values:

SIGHUP	1	Hangup
SIGINT	2	Interrupt
SIGKILL	9	Kill (cannot be caught or ignore)
SIGTERM	15	Terminate (termination signal from SIGKILL).

3.6 MEMORY MANAGEMENT

One of the numerous tasks the UNIX kernel performs while the machine is up is to manage memory. In this section, we explore relevant terms (such as physical vs. virtual memory) as well as some of the basic concepts behind memory management.

3.6.1 Physical vs. Virtual Memory

UNIX, like other advanced operating systems, allows you to use all of the physical memory installed in your system as well as area(s) of the disk (called swap space) which have been designated for use by the kernel in case the physical memory is insufficient for the tasks at hand. Virtual memory is simply the sum of the physical memory (RAM) and the total swap space assigned by the system administrator at the system installation time.

Virtual Memory (VM) = Physical RAM + Swap space

3.6.2 Dividing Memory into Pages

The UNIX kernel divides the memory into manageable chunks called **pages**. A single page of memory is usually 4096 or 8192 bytes (4 or 8KB). Memory pages are laid down contiguously across the physical and the virtual memory.

3.6.3 Cache Memory

With increasing clock speeds for modern CPUs, the disparity between the CPU speed and the access speed for RAM has grown substantially. Consider the following:

- Typical CPU speed today: 250-500MHz (which translates into 4-2ns clock tick)
- Typical memory access speed (for regular DRAM): 60ns
- Typical disk access speed: 13ms

In other words, to get a piece of information from RAM, the CPU has to wait for 15-30 clock cycles, a considerable waste of time.

Fortunately, cache RAM has come to the rescue. The RAM cache is simply a small amount of very fast (and thus expensive) memory which is placed between the CPU and the (slower) RAM. When the kernel loads a page from RAM for use by the CPU, it also prefetches a number of adjacent pages and stores them in the cache. Since programs typically use sequential memory access, the next page needed by the CPU can now be supplied very rapidly from the cache. Updates of the cache are performed using an efficient algorithm which can enable cache hit rates of nearly 100% (with a 100% hit ratio being the ideal case).

CPUs today typically have hierarchical caches. The on-chip cache (usually called the L1 cache) is small but fast (being on-chip). The secondary cache (usually called the L2 cache) is often not on-chip (thus a bit slower) and can be quite large; sometimes as big as 16MB for high-end CPUs (obviously, you have to pay a hefty premium for a cache that size).

3.6.4 Memory Organisation by the Kernel

When the kernel is first loaded into memory at the boot time, it sets aside a certain amount of RAM for itself as well as for all system and user processes. Main categories in which RAM is divided are:

- *Text*: to hold the text segments of running processes.
- *Data*: to hold the data segments of running processes.
- *Stack*: to hold the stack segments of running processes.
- *Shared Memory*: This is an area of memory which is available to running programs if they need it. Consider a common use of shared memory: Let us assume you have a program which has been compiled using a shared library (libraries that look like *libxxx.so*; the C-library is a good example - all programs need it). Assume that five of these programs are running simultaneously. At run-time, the code they seek is made resident in the shared memory area. This way, a single copy of the library needs to be in memory, resulting in increased efficiency and major cost savings.
- *Buffer Cache*: All reads and writes to the file system are cached here first. You may have experienced situations where a program that is writing to a file doesn't seem to work (nothing is written to the file). You wait a while, then a *sync* occurs, and the buffer cache is dumped to disk and you see the file size increase.

3.6.5 The System and User Areas

When the kernel loads, it uses RAM to keep itself memory resident. Consequently, it has to ensure that user programs do not overwrite/corrupt the kernel data structures (or overwrite/corrupt other users' data structures). It does so by designating part of RAM as kernel or system pages (which hold kernel text and data segments) and user pages (which hold user stacks, data, and text segments). Strong memory protection is implemented in the kernel memory management code to keep the users from corrupting the system area. For example, only the kernel is allowed to switch from the user to the system area. During the normal execution of a Unix process, both system and user areas are used. A common system call when memory protection is violated is SIGSEGV.

3.6.6 Paging vs. Swapping

Paging: When a process starts in UNIX, not all its memory pages are read in from the disk at once. Instead, the kernel loads into RAM only a few pages at a time. After the CPU digests these, the next page is requested. If it is not found in RAM, a **page fault** occurs, signaling the kernel to load the next few pages from disk into RAM. This is called **demand paging** and is a perfectly normal system activity in UNIX. (Just so you know, it is possible for you, as a programmer, to read in entire processes if there is enough memory available to do so.)

The UNIX daemon which performs the paging out operation is called *pageout*. It is a long running daemon and is created at boot time. The *pageout* process cannot be killed.

Swapping: Let's say you start ten heavyweight processes (for example, five xterms, a couple netscapes, a sendmail, and a couple pines) on an old 486 box running Linux with 16MB of RAM. Basically, you do not have enough physical RAM to accommodate the text, data, and stack segments of all these processes at once. Since the kernel cannot find enough RAM to fit things in, it makes use of the available virtual memory by a process known as **swapping**. It selects the least busy process and moves it in its entirety (meaning the program's in-RAM text, stack, and data segments) to disk. As more RAM becomes available, it swaps the process back in from disk into RAM. While this use of the virtual memory system makes it possible for you to continue to use the machine, it comes at a very heavy price. Remember, disks are relatively slower (by the factor of a million) than CPUs and you can feel this disparity rather severely when the machine is swapping. Swapping is not considered a normal system activity. It is basically a sign that you need to buy more RAM.

The process handling swapping is called *sched* (in other UNIX variants, it is sometimes called *swapper*). It always runs as process 0. When the free memory falls so far below *minfree* that *pageout* is not able to recover memory by page stealing, *sched* invokes the syscall *sched()*. Syscall *swapout* is then called to free all the memory pages associated with the process chosen for being swapping out. On a later invocation of *sched()*, the process may be swapped back in from disk if there is enough memory.

3.6.7 Demand Paging

Berkeley introduced demand paging to UNIX with BSD (Berkeley System) which transferred memory pages instead of processes to and from a secondary device; recent releases of UNIX system also support demand paging. Demand paging is done in a straightforward manner. When a process needs a page and the page is not there, a page fault to the kernel occurs, a frame of main memory is allocated, and then the process is loaded into the frame by the kernel.

The advantage of demand paging policy is that it permits greater flexibility in mapping the virtual address of a process into the physical memory of a machine, usually allowing the size of a process to be greater than the amount of availability of physical memory and allowing more Processes to fit into main memory. The advantage of a swapping policy is that is easier to implement and results in less system overhead.

3.7 FILE SYSTEM IN UNIX

Physical disks are partitioned into different *file systems*. Each file system has a maximum size, and a maximum number of files and directories that it can contain. The file systems can be seen with the df command. Different systems will have their file systems laid out differently. The / directory is called the *root* of the file system.

The UNIX file system stores all the information that relates to the long-term state of the system. This state includes the operating system kernel, the executable files, configuration information, temporary work files, user data, and various special files that are used to give controlled access to system hardware and operating system functions.
The constituents of UNIX file system can be one of the following types:

3.7.1 Ordinary files

Ordinary files can contain text, data, or program information. Files cannot contain other files or directories. UNIX filenames are not broken into a name part and an extension part, instead they can contain any keyboard character except for '/' and be up to 256 characters long. However, characters such as *,?,# and & have special meaning in most shells and should not therefore be used in filenames. Putting spaces in filenames also makes them difficult to manipulate, so it is always preferred to use the underscore '_'.

3.7.2 Directories

Directories are folders that can contain other files and directories. A **directory** is a collection of files and/or other directories. Because a directory can contain other directories, we get a directory **hierarchy**. The “top level” of the hierarchy is the **root directory.**

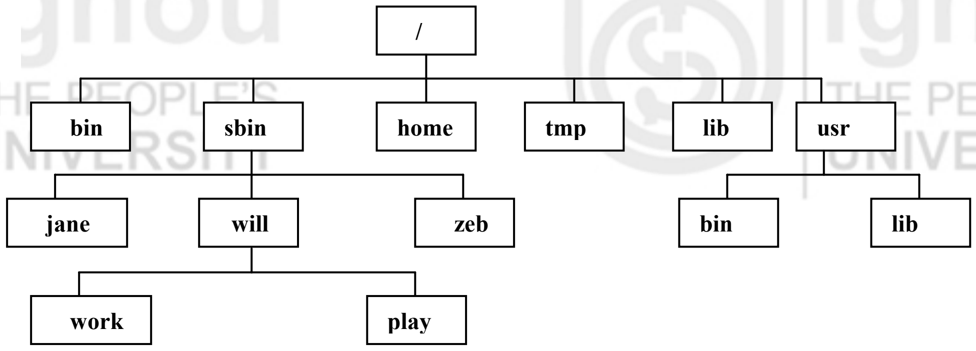

Figure 2: UNIX File System

The UNIX file system is a hierarchical tree structure with a top-level directory known as the root (designated by a slash '/'). Because of the tree structure, a directory can have many child directories, but only one parent directory. The layout is shown in Figure 2. The path to a location can be defined by an absolute path from the root /, or as a relative path from the current working directory. To specify an **absolute path,** each directory from the source to the destination must be included in the path, with each directory in the sequence being separated by a slash. To specify **a relative path,** UNIX provides the shorthand "." for the current directory and ".." for the parent directory e.g., The absolute path to the directory "play" is /sbin/will/play, while the relative path to this directory from "zeb" is ../will/play. Various UNIX directories and their contents are listed in the table given below.

Table 1: Typical UNIX directories

Directory	**Content**
/	The "root" directory
/bin	Essential low-level system utilities
/usr/bin	Higher-level system utilities and application programs
/sbin	Superuser system utilities (for performing system administration tasks)
/lib	Program libraries (collections of system calls that can be included in programs by a compiler) for low-level system utilities
/usr/lib	Program libraries for higher-level user programs
/tmp	Temporary file storage space (can be used by any user)
/home or /homes	User home directories containing personal file space for each user. Each directory is named after the login of the user.
/etc	UNIX system configuration and information files
/dev	Hardware devices
/proc	A pseudo-filesystem that is used as an interface to the kernel. Includes a sub-directory for each active program (or process).

When you log into UNIX, your current working directory is your user home directory.

You can refer to your home directory at any time as "~" and the home directory of other users as "~<login>".

3.7.3 Links

A link is a pointer to another file. There are two types of links - a **hard link** to a file cannot be distinguished from the file itself. A **soft link** or also called the **symbolic link** provides an indirect pointer or shortcut to a file.

Each file is assigned an ***inode number*** by the kernel. Attributes in a file table in the kernel include its name, permissions, ownership, time of last modification, time of last access, and whether it is a file, directory or some other type of entity.

A -i flag to *ls* shows the inode number of each entry.

A -i flag to *df* shows the number of inodes instead of the amount of space.

The *ls -l* command lists the number of links to the inode in the second column. If you remove one of the names associated with a particular inode, the other names are preserved.

The file is not removed from the filesystem until the "link count" goes to zero. All permissions and ownerships of a link are identical to the file that you have linked to. You may not link to a directory.

You can form a *symbolic link* to a file by giving *a -s* flag to *ln*. The symbolic link has its own inode number. To remove a symbolic link to a directory, use *rm*, not *rmdir*.

3.7.4 File and Directory Permissions

Each file or a directory on a UNIX system has three types of permissions, describing what operations can be performed on it by various categories of users. These permissions are read (r), write (w) and execute (x), and the three categories of users are user or owner (u), group (g) and others (o). These are summarized in the *Figure 3* given below:

Permission	**File**	**Directory**
Read	User can look at the contents of the file	User can list the files in the directory
Write	User can modify the contents of the file	User can create new files and remove existing files in the directory
Execute	User can use the filename as a UNIX command	User can change into the directory, but cannot list the files unless (s)he has read permission. User can read files if (s)he has read permission on them.

Figure 3:.Interpretation of permissions for files and directories

These file and directory permissions can only be modified by:

- their owners
- by the superuser (root)

by using the chmod(change [file or directory])mode system utility.

$ chmod *options files*

chmod accepts options in two forms. Firstly, permissions may be specified as a sequence of 3 octal digits. Each octal digit represents the access permissions for the user/owner, group and others respectively. The mapping of permissions onto their corresponding octal digits is as follows:

---	0
--x	1
-w-	2
-wx	3
r--	4
r-x	5
rw-	6
rwx	7

For example the command:

```
$ chmod 600 ignou.txt
```

sets the permissions on *ignou.txt* to rw------- (i.e., only the owner can read and write to the file). chmod also supports a -R option which can be used to recursively modify file permission by using *chgrp* (change group) .

$ chgrp *group files*

This command can be used to change the group that a file or directory belongs to.

3.7.5 File Compression and Backup

UNIX systems usually support a number of utilities for backing up and compressing files. The most useful are:

tar (tape archiver)

tar backs up entire directories and files onto a tape device or into a single disk file known as an archive.

To create a disk file tar archive, use
$ tar -cvf *archivename filenames*

where *archivename* will usually have *.tar* extension. Here the *c* option means create, *v* means verbose (output filenames as they are archived), and *f* means file.

- To list the contents of a tar archive, use:
  $ tar -tvf *archivename*
- To restore files from a tar archive, use:

  $ tar -xvf *archivename*

compress, gzip

compress and gzip are utilities for compressing and decompressing individual files (which may be or may not be archive files).

- To compress files, use:
  $ compress *filename* **or** $ gzip *filename*
- To reverse the compression process, use:
  $ compress -d filename **or** $ gzip -d filename

3.7.6 Handling Removable Media

UNIX supports tools for accessing removable media such as CDROMs and floppy disks with the help of mount and umount commands.

The *mount* command serves to attach the file system found on some device to the file system tree. Conversely, the *umount* command will detach it again (it is very important to remember to do this when removing the floppy or CDROM). The file /etc/fstab contains a list of devices and the points at which they will be attached to the main filesystem:

```
$ cat /etc/fstab ⟵
/dev/fd0   /mnt/floppy  auto    rw,user,noauto  0 0
/dev/hdc   /mnt/cdrom   iso9660 ro,user,noauto  0 0
```

In this case, the mount point for the floppy drive is /mnt/floppy and the mount point for the CDROM is /mnt/cdrom. To access a floppy we can use:

```
$ mount /mnt/floppy ⟵
$ cd /mnt/floppy ⟵
$ ls
```

To force all changed data to be written back to the floppy and to detach the floppy disk from the file system, we use:

```
$ umount /mnt/floppy
```

3.7.7 Pipes and Filters

Unix systems allow the output of one command to be used as input for another command. Complex tasks can be performed by a series of commands one piping input into the next.

Basically a pipe is written generator | consumer.

Sometimes, the use of intermediately files is undesirable. Consider an example where you are required to generate a report, but this involves a number of steps. First you have to concatenate all the log files. The next step after that is strip out comments, and then to sort the information. Once it is sorted, the results must be printed.

A typical approach is to do each operation as a separate step, writing the results to a file that then becomes an input to the next step.

Pipelining allows you to connect two programs together so that the output of one program becomes the input to the next program.

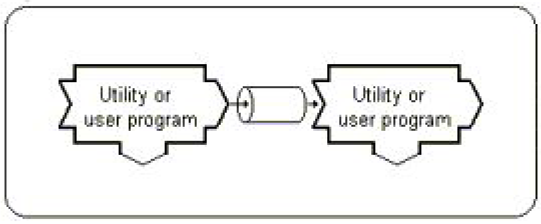

Figure 4: Pipes in UNIX

The symbol | (vertical bar) represents a pipe. Any number of commands can be connected in sequence, forming a pipeline .All programs in a pipeline execute at the same time.

The pipe (|) operator is used to create concurrently executing processes that pass data directly to one another. For example:

```
$ cat hello.txt | sort | uniq ⟵
```

creates three processes (corresponding to cat, sort and uniq) that execute concurrently. As they execute, the output of the first process is passed on to the sort process which is in turn passed on to the uniq process. uniq displays its output on the screen (a sorted list of users with duplicate lines removed).

3.7.8 Redirecting Input and Output

The output from programs is usually written to the screen, while their input usually comes from the keyboard (if no file arguments are given). In technical terms, we say that processes usually write to **standard output** (the screen) and take their input from **standard input** (the keyboard). There is in fact another output channel called **standard error**, where processes write their error messages; by default error messages are also sent to the screen.

To redirect standard output to a file instead of the screen, we use the > operator:

```
$ echo hello ⟵
hello
$ echo hello > output ⟵
$ cat output ⟵
hello
```

In this case, the contents of the file output will be destroyed if the file already exists. If instead we want to append the output of the echo command to the file, we can use the >> operator:

```
$ echo bye >> output ↵
$ cat output ↵
hello
bye
```

To capture standard error, prefix the > operator with a 2 (in UNIX the file numbers 0, 1 and 2 are assigned to standard input, standard output and standard error respectively), e.g.:

```
$ cat nonexistent 2>errors ↵
$ cat errors ↵
cat: nonexistent: No such file or directory
$
```

Standard input can also be redirected using the < operator, so that input is read from a file instead of the keyboard:

```
$ cat < output ↵
hello
bye
```

You can combine input redirection with output redirection, but be careful not to use the same filename in both places. For example:

```
$ cat < output > output ↵
```

will destroy the contents of the file output. This is because the first thing the shell does when it sees the > operator is to create an empty file ready for the output.

3.8 CPU SCHEDULING

CPU scheduling in UNIX is designed to benefit interactive processes. Processes are given small CPU time slices by a priority algorithm that reduces to round-robin scheduling for CPU-bound jobs.

The scheduler on UNIX system belongs to the general class of operating system schedulers known as round robin with multilevel feedback which means that the kernel allocates the CPU time to a process for small time slice, pre-empts a process that exceeds its time slice and feed it back into one of several priority queues. A process may need much iteration through the "feedback loop" before it finishes. When kernel does a context switch and restores the context of a process, the process resumes execution from the point where it had been suspended.

Each process table entry contains a priority field. There is a process table for each process which contains a priority field for process scheduling. The priority of a process is lower if they have recently used the CPU and vice versa.

The more CPU time a process accumulates, the lower (more positive) its priority becomes, and vice versa, so there is negative feedback in CPU scheduling and it is difficult for a single process to take all the CPU time. Process aging is employed to prevent starvation.

Older UNIX systems used a one second quantum for the round- robin scheduling. 4.33SD reschedules processes every 0.1 second and recomputes priorities every second. The round-robin scheduling is accomplished by the time-out mechanism, which tells the clock interrupt driver to call a kernel subroutine after a specified interval; the subroutine to be called in this case causes the rescheduling and then resubmits a time-out to call itself again. The priority recomputation is also timed by a subroutine that resubmits a time-out for itself event. The kernel primitive used for this purpose is called sleep (not to be confused with the user-level library routine of the same name.) It takes an argument, which is by convention the address of a kernel data structure related to an event that the process wants to occur before that process is awakened. When the event occurs, the system process that knows about it calls wakeup with the address corresponding to the event, and all processes that had done a sleep on the same address are put in the ready queue to be run.

Check Your Progress 1

1) In which programming language UNIX is written and mention the important features of UNIX OS.

………………………………………………………………………………………

………………………………………………………………………………………

2) What is a file structure in UNIX? What is its importance in UNIX?

………………………………………………………………………………………

………………………………………………………………………………………

3.9 SUMMARY

In this unit we discussed issues broadly related to features of UNIX OS, boot process, system initialization, process management, memory management, file system and CPU scheduling in UNIX operating system. In this unit we discussed several theoretical concepts of UNIX operating system in general, it is often useful to use them in your lab for practice. Refer to the Section – 1 of MCSL-045, in which we have covered the practical component of the UNIX operating system.

3.10 SOLUTIONS / ANSWERS

Check Your Progress 1

1) UNIX is written in high-level language, C. This makes the system highly portable. It supports good programming environment. It allows complex programming to be built from smaller programs. It uses a hierarchical file system that allows easy maintenance and efficient implementation.

2) UNIX system uses a hierarchical file system structure. Any file may be located by tracing a path from the root directory. Each supported device is associated with one or more special files. Input/Output to special files is done in the same manner as with ordinary disk files, but these requests cause activation of the associated devices.

3.11 FURTHER READINGS

1) Brain W. Kernighan, Rob Pike, *The UNIX Programming Environment*, PHI, 1996

2) Sumitabha Das, *Your UNIX – The Ultimate Guide*, TMGH, 2002

3) Sumitabha Das, *UNIX Concepts and Applications*, Second Edition, TMGH, 2003

4) K.Srirengan, *Understanding UNIX*, PHI, 2002.

5) Behrouz A. Foroujan, Richard F.Gilberg, *UNIX and Shell Programming*, Thomson Press, 2003.