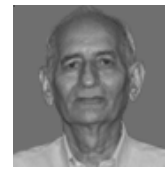
# UNIX: Genesis and Design Features

*Pramod Chandra P Bhatt*

**UNIX was primarily designed by Ken Thompson and Dennis Ritchie at Bell Labs. It was, and continues to be, regarded as a seminal contribution to operating systems design methodology, architecture and research. The primary strengths of UNIX lie in the soundness of underlying design principles. This article aims at providing insights into the making of UNIX and in particular, how the notion of open software helped in taking a utility- and tool-driven approach. This approach provides both usability[1] and extensibility besides maintainability. In fact, we see that the designers used best software architecture practices. We discuss UNIX shell and its operating environment, kernel, file systems, process creation and management, version management and documentation standard. Every effort has been made to quote from authentic sources to trace history.**



**Pramod Chandra P Bhatt retired as a Professor of Computer Science and Engineering from IIT Delhi. Later he worked at IIIT Bangalore as a senior professor. He also taught at IISc, Bangalore. Currently he operates as a freelance consultant in software engineering related areas.**

## 1. The Story of UNIX – How It All Happened

Let us trace the development of computer systems from the mid-1940s when John Von Neumann proposed 'stored program' architecture [1]. The commercial production of computer systems commenced in the 1950s. The 1960s witnessed frenetic activity as the electronic systems development then made stored program computers realizable on industrial scale.

The stored program notion posed intellectual challenges in programming at multiple levels ranging from application programs to system programs. In the systems programming area it provided opportunities for software control programs to regulate and optimally use the system's resources. This kind of programming effort leads to an Operating System (or OS). UNIX development is considered a significant milestone in OS design. The methodology adopted in creating UNIX has stood the test of time and has also found universal acceptance.

[1] Usability is in the context of programmer convenience. At that time, computers were primarily for programmers.

**Keywords**
Operating system design, UNIX shell, UNIX kernel, file systems, process management, interprocess communication, versioning, documentation and high level language C, tools and utilities.

In 1965, MIT, AT&T Bell labs and GE were working on an experimental OS called MULTICS (MULTiplexed Information and Computing Services) to run on GE 645 [2–3] computer. Thompson from Bell Labs was a member of the team. MULTICS later became a commercial product – albeit not attracting many customers.

With many a lesson learned, Thompson returned to Bell Labs to experiment and develop some OS design ideas further on a little used PDP-7 computer produced by DEC[2]. Ken's initial idea was to develop a *unics* (*uni* as *multi* is in MULTICS) system for a single user [3, 5]; *unics* was written in *asm*, the assembly language of PDP-7. The main objective set for *unics* was to offer an interpreted command language interface to the user. This mode of operation is what is nowadays referred to as *command line mode* in a PC environment.

[2] Digital Equipment Corporation at that time produced a range of mini-computers called 'PDP machines'.

Unics later became UNIX – still rhyming with MULTICS. Bell labs had a fairly active user group using this system. This period also marked the beginning of a long collaboration between Thompson and Ritchie. UNIX was aimed to support multiple users in time-sharing mode of operation. The command line mode of operation remained the primary mode of operation. UNIX datelines [4, 6, 8] suggest the year to be around 1969.

Thompson and Ritchie were 'thinking' programmers. Thinking programmers not only carry the lessons learned from task 'A' to task 'B' but also have the capacity to understand motivations that defined the requirements in the first place. When thinking programmers set out to design, they work on the requirements with a certain depth of understanding. Their programs almost always attempt generalizations.

[3] See a companion article [7] by K Bhaskar on C in this issue.

The first impact of Ritchie's participation in UNIX development was through the programming language C. Much of UNIX was rewritten in C in 1973.

The first impact[3] of Ritchie's participation in UNIX development was through the programming language C. Much of UNIX was rewritten in C in 1973 [8]. From our standpoint, C provided an opportunity to write UNIX in a High-Level Language (HLL). C could be both powerful (in abstractions) and cryptic (in coding).

It has the advantage that one can code at function level – to support high level of abstractions and at the same time code using bit-level operators just as in assembler-level programming. The HLL encoding of UNIX offered two major advantages. One is obvious: portability. The other one, which is not-so-obvious, is that HLL programming of OS facilitates translation of the requirements into implementable programs (by employing abstraction as the vehicle – a mechanism for the thinking programmers).

Dennis Ritchie in his Turing award lecture [9] also credits the working environment and the attitude of AT&T Bell Labs management for the success of UNIX. He observes: "Bell Labs has provided this commitment and more a rare and uniquely stimulating research environment for my colleagues and me". He goes on to add "… it will remain possible for a future Ken Thompson to find a little – used CRAY/1 computer and fashion a system as creative and as influential as UNIX". Elsewhere in this lecture, he states, "Bell labs has always had a considerable commitment to the world and does not fear edicts commanding us to be practical".

Thompson and Ritchie also consciously encouraged larger creative participation. To encourage this, they developed a set of generic productivity tools which could be used to quickly develop programs[4]. Also, they developed UNIX as an open system [10] that would permit extensibility. Added to this was the fact that UNIX was freely distributed to the universities. The open systems and easy availability stimulated the OS research community. The churn resulted in building many tools and systems quickly. For example, text and string processing functions in C were the primary basis for AWK[5] tool [10] available in UNIX environment. The free distribution of UNIX also led to university-based activities resulting in development of networking software at University of California at Berkeley and GUI (Graphical User Interface) using X-windows at MIT [10].

However, to get a true perspective of the enormity of what was aimed by the UNIX duo, we should also examine the machine

Thompson and Ritchie developed UNIX as an open system that would permit extensibility. Added to this was the fact that UNIX was freely distributed to the universities.

[4] The modern Software Development Kits (SDKs) derive much from this methodology of tools-based development.

[5] The name AWK comes from the names of the Bell Labs scientists who developed it. The names are: Aho, Weinberger and Kernighan.

At Bell Labs, there have been 10 versions of UNIX that had evolved over time.

configurations available then. Also, note the constraint that UNIX was not an official project (with Bell labs financing it) until 1971. Once Bell Labs adopted UNIX, the project acquired PDP-11/20 of its own. A subsequent PDP-11/45 machine had UNIX posited in 42 Kbytes [9] to support amongst others a very large number of device drivers, IO buffers and systems tables. In fact, in their paper [8], the claim is that, under 50 Kbytes core image of UNIX, it was possible to support four swappable disk drives (with 2.5 MB removable cartridges), high speed paper tape reader-punch devices, nine track tape drives, DEC tape, console typewriter, 14-variable communications interfaces, attached to different types of data sets used as spoolers for printers, picture phone interface voice response units, photo-type setter, digital switching network and a satellite PDP-11/20 machine. This indeed is a formidable task.

UNIX was adapted and enhanced by both universities and commercial organizations.

At Bell Labs, there have been 10 versions of UNIX [11] that had evolved over time. Documentation on all of these is not easy to get, but there are tidbits that can be pooled together. In the years following 1978 and UNIX's 7th edition (a 32-bit edition), UNIX was adapted and enhanced by both universities and commercial organizations. This resulted in many manifestations. We summarize these efforts in *Figure* 1.

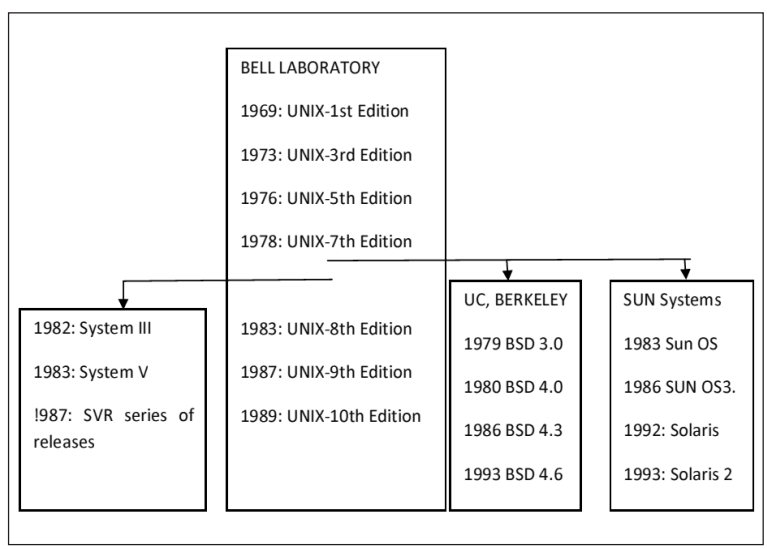



**Figure 1.** The UNIX family with some initial manifestations.

More details are given in [11]. Even so, the figure here identifies the main families. Other offshoots (from BSD 4.2) include the famous MACH system developed at CMU [12] and OSF [13].

A notable effort that needs to be mentioned is MINIX – a clone of UNIX written by Andrew Tanenbaum of Virje University, Netherlands [14]. Its circulation on a set of floppies made it easy to install and use. MINIX prompted Linus Torvalds [15] to develop LINUX – an effort to get UNIX-like operating environment on a PC[6]. The easy and free availability of Linux as an open source OS since the early 1990s has catapulted the open source development to new heights. Torvalds and a team currently maintain all updates and releases for open source Linux kernel.

The easy and free availability of Linux as an open source OS since the early 1990s has catapulted the open source development to new heights. Torvalds and a team currently maintain all updates and releases for open source Linux kernel.

Many people contributed to the UNIX effort with coordination from Thompson and Ritchie. The question is: How well did they coordinate between themselves? The following quote [16] from Ken Thompson says it all: "That brings me to Dennis Ritchie. Our collaboration has been a thing of beauty. In the ten years that we have worked together, I can recall only one case of mis-coordination of work. On that occasion, I discovered that we both had written the same 20-line assembly language program. I compared the sources and was astounded to find that they matched character-for-character. The result of our work together has been far greater than the work that we each contributed". A wonderful collaborative team effort indeed!

The references [1–9, 11–16] offer time-lines that give insights into the evolution of UNIX design. Over time, many individuals and organizations have contributed to UNIX. This article will limit itself to those UNIX design features that were enunciated by Thompson and Ritchie. These features have had long-lasting impact and are relevant even today.

## 2. UNIX Design Features

In this section, we shall deal mainly with some important design features in UNIX [10, 17]. The following key design ideas of

[6] Linus Torvalds' first e-mail to Andrew is available on the web. It is also reproduced in the 2nd edition of [10].

UNIX have had long-term impact on almost all other future OS designs:

- Concept of a shell
- Separation of kernel and user space
- File system and file management
- Unified view of input and output
- Process: creation, life cycle and management
- Inter-process communication
- Versioning and documentation

However, before discussing these in detail, it is important to understand user interaction in UNIX. To begin with, users get a login id with an identifiable home directory. Once logged in, users can begin interacting with UNIX. So, let us walk through operational stages that lead to the execution of a user program. In doing so, we will become familiar with many terms that UNIX uses. Let us consider the program given below [18] – the first program everyone learns in C.

```
main()
{
        printf("hello, world\n");
}
```

Most textbooks would recommend that we store the program as *hello.c*. The next is to compile the program using a *cc* command to generate an executable called *a.out*. In fact, the *cc* command used the *hello.c* as input file and produced *a.out* as output file. We note that *hello.c* is a text file and *a.out* is an executable file. Both are located in the user's directory. These file names show up when *ls* command is given to list all the files in the directory. To execute this program, we use *a.out* as if it is a command (like *ls*). For most people, that is the first experience at interactivity.

A few points need to be made here in the context of this example:

- The file *a.out* is an executable file. So, it has a binary or machine executable format.

- The binary code in *a.out* creates a 'core image' for execution.
- Upon giving the command *a.out*, we seek to schedule its execution.
- The command *a.out* is given in the operating environment offered to us by a UNIX shell.
- When executing, *a.out* manifests its dynamic behaviour. It is considered to be a process.
- At any time there could be many processes in the system.
- The processes compete to get the processor for execution.
- Processes may seek IO as and when required.

Every legitimate user is offered interactive operating environment by UNIX shell. Offering shell as an operating environment was a very clever idea indeed.

The fourth point above mentions UNIX shell. The point to note is interactivity. Every legitimate user is offered interactive operating environment by UNIX shell. Offering shell as an operating environment was a very clever idea indeed. Let us explore it further.

### 2.1 *The Concept of Shell in UNIX*

Thompson's intent was to offer an interactive OS. The shell concept precisely meets that objective. The UNIX shell is essentially an interactive command interpreter. The reader might have noted the extent of interaction in the elementary example given earlier. *Figure* 2 shows the positioning of the shell in the context of the rest of the system. In particular, observe how the user command or a built-in command gets interpreted and an interactive response is given. It also shows possible usage of the tools suite available to the shell.

The UNIX shell is essentially an interactive command interpreter.

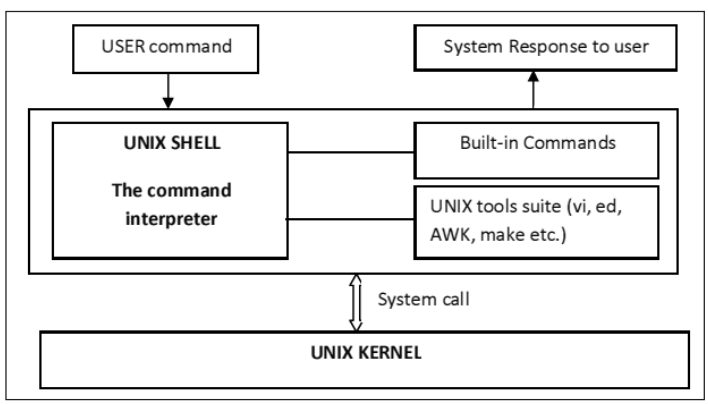



**Figure 2.** UNIX shell and its interaction with the rest of the system.

UNIX shell supports a very rich set of commands [10, 19–21] to support user interactivity. Basically, these commands allow users to create and manage programs. In general, commands may accept arguments. In Appendix-1 we tabulate some commands – a few of these require arguments which could be file names. References [19–21] contain a fuller list of commands[7].

[7] Though there may be a common minimal set, commands differ for different UNIX versions, families and shells.

Shell also offers built-in tools or user-created tools with their own command repertoire. For example, take the edit tools like *ed*, *sed* or *vi*. Each has its own edit capabilities. As an open system, UNIX encouraged users to develop tools to enrich UNIX tools suite. One such useful tool, AWK, was created by Aho, Weinberger and Krernighan [10]. AWK is used when files bear a formatted structure. It offers many useful commands to process structured data-files.

UNIX command interpreter can be used to automate routine command chains. This can be done by writing a command script [5,10,18,19]. The command chain is scripted separately as a text file. To execute the script, the shell is simply directed to accept the input from the script file.

Isolation of shell from the kernel was a very clever design idea. It permitted every user to operate in his login shell – a 'sand-box'. As a result, users could customize individual operating environment even while there is a multi-user operation. Also, the sand-box offers security. A shell can invoke sub-shells recursively. The isolation of shell from kernel gave an opportunity to evolve several specialized shells such as Bourne shell and C-shell later. This was primarily possible because of standard system call interface with the kernel, i.e., for a UNIX kernel interface, different shell formulations could be designed. Next we discuss how the system and user programs co-habit.

Isolation of shell from the kernel was a very clever design idea. It permitted every user to operate in his login shell – a 'sand-box'. As a result, users could customize individual operating environment even while there is a multi-user operation. Also, the sand-box offers security.

### 2.2 *Separation of Kernel and User Privileges*

Stored program principle entails that the program instructions that execute, reside in main memory. This means that program instructions for users and system management (i.e., OS utilities

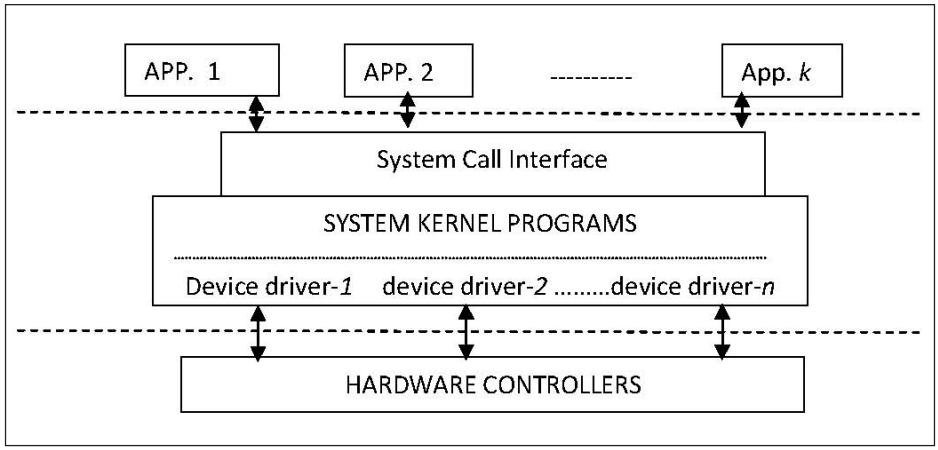



**Figure 3.** The layered architecture of UNIX.

and programs) both reside in the main memory. Note that their spheres of activity are different. A well-designed OS usually isolates execution of programs that provision system services. A clean design methodology employs a kernel to direct all system services. UNIX marks out isolated space in memory to carry out the kernel's activities. In fact, UNIX has a layered architecture as shown in *Figure* 3.

In *Figure* 3, user programs are identified as App.*i*. These communicate with the kernel through a system call interface to obtain system-related services. Depending upon the nature of the service sought, the kernel system programs determine if a IO is required, some memory needs to be allocated or processor scheduling is to be done, etc.

One other interesting aspect of UNIX kernel design is its modular structure. There is a separate module with a clear service role defined, as shown in *Figure* 4.The two major aspects relate to file-management and process control with each having its own sub-systems and modules. For more detailed kernel system programs, please see [8, 10, 17]. However, the services most often used by user programs are (i) to interact with the program-related files and IO, and (ii) to obtain a degree of flexibility to communicate with other processes. As we will see in the next sub-section, kernel operates the entire file management system.

Another interesting aspect of UNIX kernel design is its modular structure. There is a separate module with a clear service role defined.The two major aspects relate to file-management and process control with each having its own sub-systems and modules.

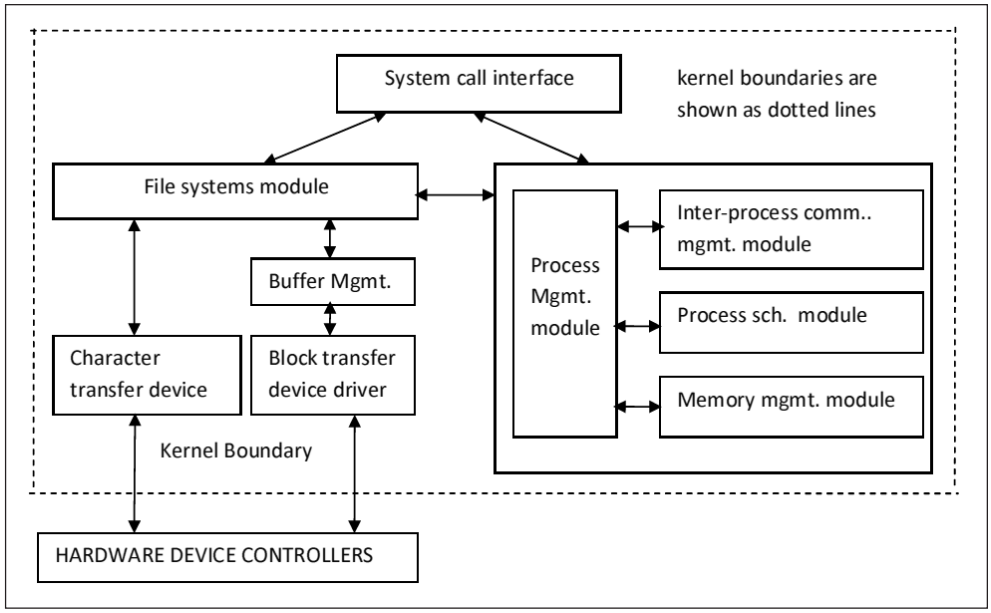



**Figure 4.** The kernel modules with their sub-systems.

The isolation of kernel services follows an established software design principle, e.g., 'separation of concerns' [22]. Thompson and Ritchie enforced it in UNIX. In fact, user program normally cannot access the kernel space. This form of isolation provides good system level robustness. Thompson and Ritchie reckoned that instruction interpretation happens at full throttle, i.e., at the processor speed. This is alright when users execute their program's logical instructions. In case an error occurs, a user could seek a re-run with corrections incorporated. However, while executing instructions that engage in system related activities, user errors should not be accepted. Such errors can lead to the disruption of system activities affecting other users. It can also destroy the system support processes and even the operating system itself. So, the best principle is that system related instructions be carefully shepherded and executed in a protected mode of a Kernel's privileges. That rationalizes the isolation of user and kernel privileges and space.

In fact, user program normally cannot access the kernel space. This form of isolation provides good system level robustness.

### 2.3 *File System and File Management*

As thinking programmers, both Thompson and Ritchie were

acutely aware of user woes arising from unfriendly OS environments. Therefore, a conscious effort was made to invoke simplicity in organization to enhance file-system usability. The hierarchical tree data structure used is a master-stroke in design.

A file is essentially some organized information [10] in the form of a sequence of bytes. Simple file examples are the user program and data files. The OS too manages files for system entities (libraries, device drivers, etc.) using the file system. The file-system provisions creation, opening, storage, update, access control, and even disposal of files in a safe and convenient manner. The file system, therefore, is a very elaborate system. We describe only one key design feature – the hierarchical file organization.

Let us consider a simple example of user files. Suppose a teacher teaches two courses every semester and has to maintain two kinds of files, one that provides material for teaching and another for administration of courses. *Figure* 5 shows a possible hierarchical structure.

In *Figure* 5 we see that a tree structure with node …/myHome/Courses branches out to four nodes that group the files for each course 'admin' and 'lectures'. The file name descriptors help to identify files. An alternate arrangement could have had two semester-wise groups rather than four, i.e., the user has complete freedom and convenience to organize files that he owns.

Thompson and Ritchie were acutely aware of user woes arising from unfriendly OS environments. Therefore, a conscious effort was made to invoke simplicity in organization to enhance file-system usability. The hierarchical tree data structure used is a master-stroke in design.

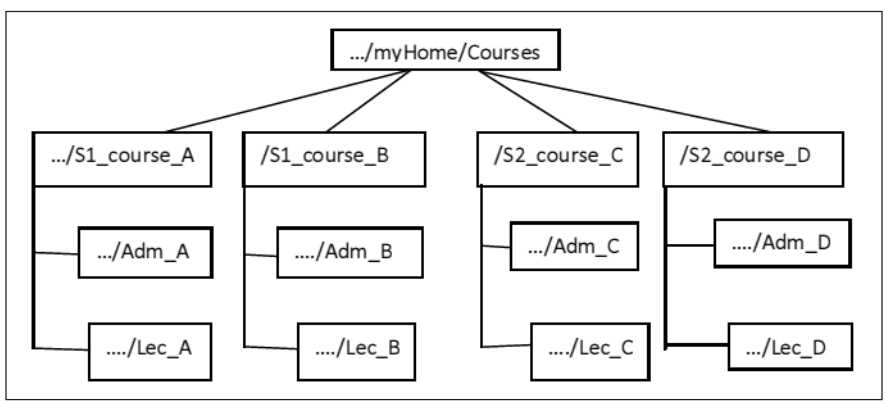



**Figure 5.** Two possible tree structures.

Some important directories are /bin which stores executable files, /dev which caters to the devices and communication with devices, /home which subtends user home directories like myHome.

In fact, the entire file system supports a hierarchy beginning with root node identified by "/". The node …/myHome in the example is the user's node to create his own file tree structure. In fact, this node is called home directory. Further down in the tree, file Adm_A may be a structured file suitable for AWK processing. Lec_A could be a directory with a sub-tree of individual lecture files. For example, Lec_A directory could support a sub-tree – a collection of files such as Lec_A1, Lec_A2, etc., A specification such as …/myHome/Courses/S1_course_A/Lec_A/lec_A5 would identify 5th lecture for Course_A. Files are specified by traversing the tree path to the desired file.

The directory hierarchy from the root node is shown in *Figure* 6. Some important directories are /bin which stores executable files, /dev which caters to the devices and communication with devices, /home which subtends user home directories like myHome. More details can be found in [8, 17, 19, 23].

One feature which UNIX introduced was 'access privileges'. These ensure that the file system permits only legitimate access. The owner of a file determines the privileges for his files.

The user-created files are maintained in user space with a file-descriptor and characterization that helps its likely context of use. A program may be a text file while an object file may be an executable. The context of use is distinguished by file type.

One feature which UNIX introduced was 'access privileges'. These ensure that the file system permits only legitimate access. The owner of a file determines the privileges for his files. This also means that system files, owned by kernel, are pretty much secured. No accidental damage of catastrophic proportion can be inflicted on the OS by a user either by mistake or with malicious

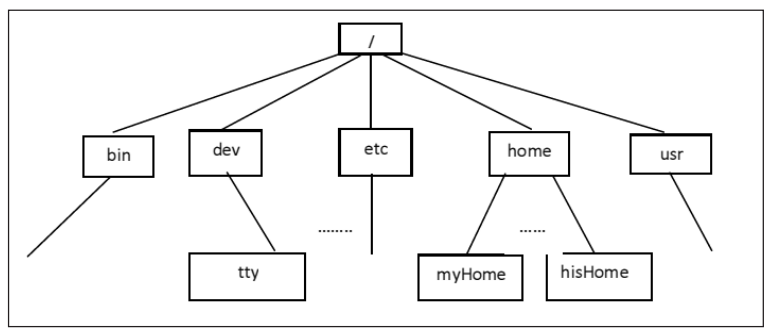



**Figure 6.** The UNIX directory structure.

intent. The access privileges may enable: i) secure access of the files for exclusive use (by the owner), ii) secure access of the files for limited sharing with other users (by a group) and iii) open access of the files for use by any one.

No accidental damage of catastrophic proportion can be inflicted on the OS by a user either by mistake or with malicious intent.

### 2.4 *Unified View of Input and Output*

UNIX designers attempted a form of generalization for IO [5, 10]. Input is similar to read operation and output to write operation. With this generalization, IO device communication is modeled to be similar to a file read or write. For usual IO, UNIX defines a default standard input known as *stdin*, standard output as *stdout* and IO error messaging as *stderr*. These definitions are inherited by every UNIX shell. However, to input or output to any other device, the communication can be directed by symbols "<" for input and ">" for output. Yet another form of IO is required when a process output is to be fed as input to another process. Such piping is directed by "|" symbol. We shall describe pipe plumbing between processes in Section 2.8.

IO devices are categorized as those that communicate i) one byte at a time – such as a keyboard and ii) devices that minimally require a block of bytes for IO such as a tape or disk drive. For block devices, the kernel manages the buffer. IO is supported for all forms like polling, interrupt, DMA. All communication to or from the devices is regulated by kernel as shown in *Figure* 4.

Another point on which UNIX scores high is the uniformity in protocol for access to devices. The device files can be 'mounted'[8] locally or remotely. All one needs is the presence of the file on a valid qualified path name (a path from root downwards). What is interesting is that a remote mount on networked systems is conceptually similar, i.e., a qualified hierarchical path traces the system in the network, finally leading to the remote file.

[8] The term 'mount' is derived from the practice of mounting a detachable tape or disk pack from a drive.

### 2.5 *Processes*: *Creation, Life Cycle and Management*

A program in execution is, in fact, a process [8, 10, 17]. It is what gives dynamics of operation in computer systems. UNIX shell is

[9] *a.out* is the default executable generated on compilation. However, the user may choose a name for the executable.

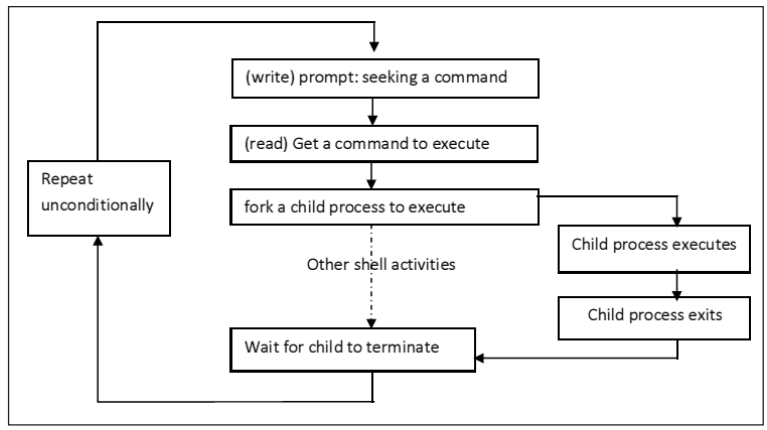



**Figure 7.** Shell operation including launch of child processes.

itself a process. When we give a command or execute *a.out*[9] file, UNIX launches a child process under the shell (parent process). The child process inherits properties from the parent and executes in that environment. A *fork()* system call in UNIX launches a child process. In *Figure* 7 we show the operation of shell and launching of child process which may be a consequence of a shell command or *a.out* (or any named executable).

Thus, we have demonstrated that multiple processes can co-exist in UNIX environment. Right away, we can see that UNIX shell and *a.out* are two separate processes.

Suppose we have a process that is engaging a processor. Now suppose it seeks to perform IO – then it would be imperative for the processor to wait till IO is completed. Given the difference in the processor and IO speeds (which could be of at least 2 to 3 orders), we should anticipate long waiting periods for the central processing unit (CPU). If there is only one resident process in the system, then we shall have a large number of idling processor cycles during IO. This is no different from the idling of car engines at a stationary point. To mitigate such wastage and utilize the processor cycles better, UNIX supports residency of multiple processes. In fact, these processes may belong to different users – in that case, we have a time-shared system on hand.

UNIX supports residency of multiple processes. In fact, these processes may belong to different users – in that case, we have a time-shared system on hand.

Time-shared systems would have multiple users sharing a computer system [8, 10]. Processes owned by different applications or

users take turns to share the CPU. A process which is currently engaging the processor is 'active', that which is awaiting release of processor by active process is 'ready to run', and the one which is awaiting an event which may be an indication of completion of IO is said to be in 'wait mode'.

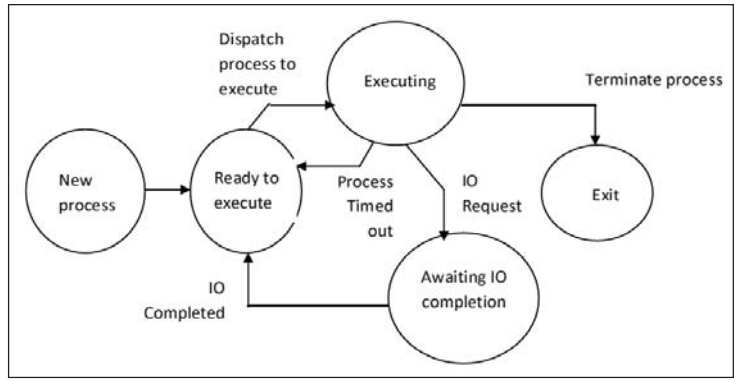



**Figure 8.** Process states in UNIX.

A typical process life cycle is shown in *Figure* 8. A process initially begins as a new process and joins the pool of processes that are ready to execute. Once the scheduler selects the process, it begins its execution by engaging the CPU. The process is allocated a processor time slice for execution. Upon completion of time slice, it is timed out and another ready-to-run process may begin engaging the processor. It is also possible that during the time slice, it may seek IO. The only other possibility is that the process terminates. In that case it exits. In *Figure* 8, we have captured the complete lifecycle of a process with the possible state transitions.

UNIX scheduler follows a policy which allows a long waiting process to gradually have its priority revised to higher level to get scheduled. The UNIX scheduler generally follows a fair scheduling policy. The policy was primarily designed to support general purpose computing. Note that this is one reason why UNIX is not considered to be suitable for real-time systems. UNIX designers operated with mutual trust and cooperation. They desired the same to transcend to the end users as well. This is reflected in a choice they offered to the users, i.e., users could offer to have their processes run in lower priority mode as *nice* processes, i.e., operate at lower levels of priority and yield computing resources to other executing processes.

UNIX scheduler follows a policy which allows a long waiting process to gradually have its priority revised to higher level to get scheduled. The UNIX scheduler generally follows a fair scheduling policy. The policy was primarily designed to support general purpose computing. Note that this is one reason why UNIX is not considered to be suitable for real-time systems.

There are many situations when processes need to communicate. For example, the kernel communicates to the user process about

One method of communication is when the output of a process is fed as input to another process. UNIX called such a communication a pipe.

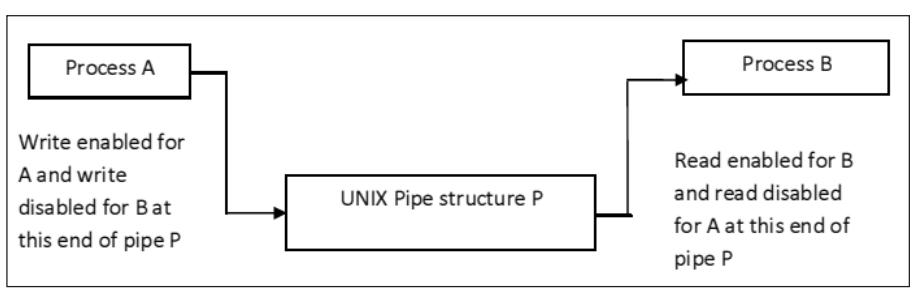
the completion of IO. More advanced applications such as banking may require a complex protocol of communication. If one withdraws an amount from a bank, there are two processes: the user process and the bank's accounting system. Such an activity involves multiple stages of operation to ensure that the entire accounting process is consistent. The basic point is that user processes and applications need to communicate.

### 2.6 *Inter-Process Communication*

The inter-process communication (IPC) suite of programs [10, 17, 24] is a major contribution of UNIX. UNIX style of IPC programming is almost a folklore now. IPC program listings are far too long for this article – so readers should look up [10, 24] for detailed IPC program description. We will limit ourselves here only to the mechanisms used. We discuss pipes, shared files, shared memory and messages for establishing communication between processes.

*Pipes*: One method of communication is when the output of a process is fed as input to another process. UNIX called such a communication a pipe. Pipe is a data structure defined in UNIX with two ends. Suppose A and B are two processes that wish to use pipe P with two ends – end1 and end2 – for communication. We assume that process A output is fed as input to B. In that case, A chooses end1 of P to write (closing read operation at end2) and B chooses end2 to read (closing the write operation at the end1). The scheme is shown in *Figure* 9. Pipes have limitations; for example, for a two way communication between process A and process B, we need two pipes. Secondly, it is not amenable to broadcast.

**Figure 9.** The UNIX pipe.



*Shared files*: This method allows processes to share a file. Usually the paradigm ensures that one process writes into the file and the other reads from it. This may be easily done by having the definition of the file in the environment of the two communicating processes. The processes may use file pointer value to locate the file to be shared. However, in this case, care needs to be taken that readers and writers cooperate. UNIX has mechanisms to allow such cooperation.

Shared memory is yet another method of Inter Process Communication.

*Shared memory*: This is yet another method of IPC. As the name implies, shared memory is a part of the memory made available to the two communicating processes. Clearly, careful mechanisms need to be set up to let UNIX processes access shared memory without conflict.

For both shared files and shared memory methods of IPC, care needs to be taken that write operations are not concurrent. This is to prevent the shared data updates from becoming inconsistent. There are paradigms that can support simultaneity in read operation though. Note however that write is performed as an exclusive operation. This means that there is no read operation while there is write operation.

*Messages*: This is another form of IPC wherein messages are communicated between the processes. Invariably the message has a structure to it. The structure may provide a header and a body. By analysing the message header, the intent for the body can be interpreted. It may seek to trigger an appropriate event at the receiver end. The message exchange has also led to a parallel processing paradigm called message passing interface (MPI) in distributed environment [25]. The IPC mechanism is also extensively used by the protocol processing in Internet [26].

Messages is another form of Inter Process Communication wherein messages are communicated between the processes.

We have covered the most commonly used IPCs. In practice, setting up the IPC is now considered a standard procedure. This is because of the availability of the code patterns for almost all forms of communication. In addition to these mechanisms, UNIX support signals for IPC. There are many kinds of signals that support IPC.

Versioning and documentation are part of the best software engineering practices for any large software development project as it helps in maintenance of the programs over its lifetime.

### 2.7 *Versioning and Documentation*

UNIX versioning and documentation was very meticulous and is worthy of emulation. Versioning and documentation are part of the best software engineering practices for any large software development project as it helps in maintenance of the programs over its lifetime. Every major software development evolves over time and undergoes multiple revisions. Sometimes, one may be able to offer multiple combinations by incorporating different options. UNIX offered a numbering system to keep track of program version during the development as shown in *Figure* 10.

Note that as the versioning tree depicts, different choices could lead to varying configurations. The versioning system in SCCS (source code control system) is a UNIX tool that precisely permits one to do this [5, 10]. SCCS has evolved over time. In its current avatar for change management, Linux employs CVS (Concurrent Versioning System) and git[10]. Details of CVS can be found in [10].

[10] git is a slang in English language. IT professionals adapted it, as this task is somewhat unpleasant but very important.

An associated concern of the developers is documentation. During a large software system development effort, multiple teams work in tandem and end up using each other's programs. Besides being developed in parts by different teams, UNIX also permitted users to add their own tools (as an open system). So, a documentation standard had to be created. UNIX documentation is presently often the model. It is famously referred to as 'man' pages, where man is a short form of manual which is available on-line. UNIX is credited with a very thorough on-line manual. *Figure* 11 shows how the UNIX command man pages are organized.

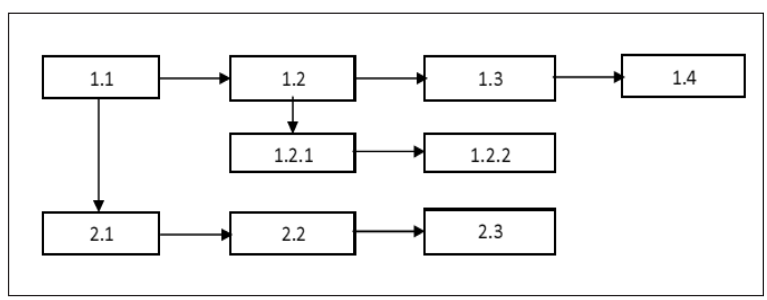



**Figure 10.** The version numbering in UNIX.

**Name:** Usually a short name with one line description
**Synopsis:** Describing the command with the options and arguments
**Description:** Essentially a functional description
**Example:** A simple example of one of the common usages
**See also:** Which are the related items in man pages

**Figure 11.** Organisation of UNIX man pages.

UNIX classified manuals as User Manual, Programmer's Reference Manual and Administrator's Manual, etc. UNIX created tools and offered conventions to help generate these documents. The tradition of offering on-line man is continuing with UNIX's latest avatar, namely, Linux.

### 2.8 *Some Additional Aspects of UNIX and C*

One of the interesting aspects of Thompson and Ritchie's attitude to work was to gather enough statistics to check if they made correct choices or how their tools performed. Between UNIX and C programming environments, there were enough tools provided to collect statistics. The idea was to monitor program activity, including that of UNIX [8]. This reference cites statistics to show CPU activity, access patterns for loads created by different forms of usages.

One of the interesting aspects of Thompson and Ritchie's attitude to work was to gather enough statistics to check if they made correct choices or how their tools performed. Between UNIX and C programming environments, there were enough tools provided to collect statistics.

### 2.9 *An Aside*

Finally, one aside, without which the author's tribute to Ritchie would be incomplete. Dennis Ritchie was always considered to be a quiet and serious programmer. Even his humour was serious and the prank that he played on his Nobel Laureate boss is well-documented [28] – serious as he was, it was video-graphed. The video is worth viewing. No wonder, he loved working for Bell Labs. As he put it: "Bell Labs was a place that believed in creating practical artifacts that were useful for the world at large".

## 3. Conclusions

In this article, we have looked at the history of UNIX. We also elaborated key design features. An effort was made to explore the

Thompson and Ritchie's decision to adhere to an open system and take tools driven approach is now a universal practice.

rationale that prompted the choices made by Thompson and Ritchie. In particular, their decision to adhere to an open system and take tools-driven approach is now a universal practice. We have also shown how Ken Thompson and Dennis Ritchie – the thinking programmers – used the software architecture principle and even provided some of the best practices as guidelines for future OS developers.

## Suggested Reading


**[1]** **John von Neumann, First Draft of a Report on the EDVAC ContractNo.W-670-ORD-4926, Between the United States Army Ordnance Department and the University of Pennsylvania Moore School of Electrical Engineering, University of Pennsylvania, June 30, 1945.**

**[2]** **http://www.livinginternet.com/i/iw_UNIX_dev.htm**

**[3]** **http://news.softpedia.com/news/40-Years-of-UNIX-119827.shtml**

**[4]** **http://www.UNIX.org/what_is_UNIX/history_timeline.html**

**[5]** **Eric Foster Johnson, *UNIX Programming Tools*, M&T Books Press, 1997.**

**[6]** **Dennis M Ritchie's home page at http://cm.bell-labs/who/dmr**

**[7]** **K Bhaskar, C – Past, Presesent, and Future – A Perspective, *Resonance*, Vo.17, No.8, pp.748–758, 2012.**

**[8]** **Dennis M Ritchie and Ken Thompson, The UNIX Time Sharing System, *Communications of ACM*, Vol.17, No.7, pp.365–375, 1974.**

**[9]** **Dennis M Ritchie, Reflections on Software Research, *Communications of ACM*, Vol.27, No.8, pp. 758–760, 1984.**

**[10]** **Pramod Chandra P Bhatt, *An Introduction to Operating Systems: Concepts and Practice*, 3rd Edition, PHI Learning, 2010.**

**[11]** **http://www.UNIX.org/what_is_UNIX/history_timeline.html**

**[12]** **http://www.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html**

**[13]** **http://en.wikipedia.org/wiki/Tru64_UNIX**

**[14]** **en.wikipedia.org/wiki/Andrew_S._Tanenbaum**

**[15]** **http://en.wikipedia.org/wiki/Linus_Torvalds**

**[16]** **Ken Thompson, Reflection on Trusting Trust, *Communications from ACM*, Vol.27, No.8, pp.761–763, 1984.**

**[17]** **Maurice J Bach, *The Design of UNIX Operating System*, PHI Learning, 9th Printing, May 1994.**

**[18]** **Brian W Kernighan and Dennis M Ritchie, *The C Programming Language*, PHI Learning, 1985.**

**[19]** **Graham Glass and King Ables, *UNIX for Programmers and Users*, Prentice Hall, 1999.**

**[20]** **http://www.computerhope.com/UNIX/overview.htm**

**[21]** **http://linux.about.com/od/commands/l/blcmds.htm**

**[22]** **http://www.sei.cmu.edu/library/assets/Architecture_and_Design.pdf**

**[23]** **Brian W Kernigham, and Rob Pike, *The UNIX Programming Environment*, Prentice Hall, 1987.**



*Address for Correspondence*
Pramod Chandra P Bhatt
Vidha Software Solutions
393 (Phase-2) Palm Meadows
Whitefield
Bangalore 560 066, India.
Email:
bhatt_pcp@consultant.com

**[24] Chris Brown, *UNIX: Distributed Programming*, Prentice Hall, UK, 1994.**

**[25] William Gropp, et al, *Using MPI*, MIT Press, 1994.**

**[28] James F Kurose and Keith W Ross, *Computer Networking*, 3rd Edition, Pearson, 2005.**

**[29] http://www.fiction.net/tidbits/computer.true_story_UNIX.html**

## Appendix – 1

A few illustrative shell commands:

In Section 1, we stated that UNIX shell provides the user a command interpreter interface. Here, we describe some representative and useful commands. We will assume that ":" is a UNIX prompt to seek a command input and - shows the response from UNIX. Let us illustrate how a system responds to a command *date*. We offer some explanations within curly brackets.

: *date*

- Wednesday March 27; 12:27 GST 2012

: *echo* This command reproduces the arguments of echo

- This command reproduces the arguments of echo

: *wc* –l MyFile

- 923 MyFile { MyFile is a text file with 923 words in it}

: *who*

- bhatt tty03 March 27 09:36
- vivek tty07 March 27 10:33 {Users bhatt and vivek are using the system currently}

: *cat* MyFile1 MyFile2 > MyFile3

{cat takes MyFile1 and MyFile2 as input and outputs MyFile3 as the concatenated file}

: *chmod* 751 MyFile

- {This sets the new access privileges for MyFile; see the next *ls* command}[11]. "r" stands for read, "w" for write and "x" for execute rights.

: *ls* –l MyFile

- r——wx—x bhatt {shows user group and world access privileges for bhatt's file MyFile}[12]

: *kill* process-id

- {kills the process with name process-id}

It is too voluminous to list all UNIX commands here. Users may refer to [20, 21] for c-shell command list of the recent Linux system. Each distribution of UNIX comes with documentation listing all commands of the the specific shell. In addition, the documentation often provides details of system call (API) as well as administrator commands.

[11] 7 here is for rwx, 5 for r-x, and 1 for x, i.e., rwxr-wx—x

[12] *ls* command offers options to also get such details as size and date of creation of files.