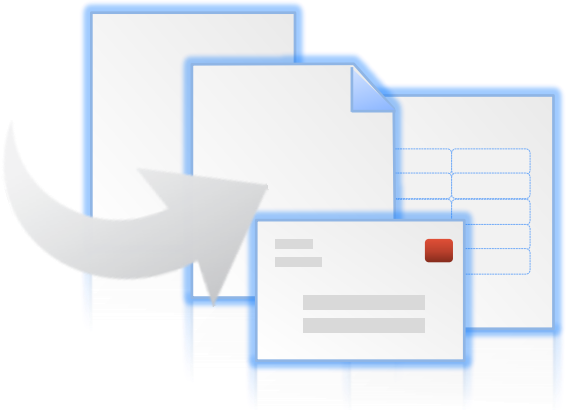
ADVANCED OPERATING SYSTEMS

UNIT I INTRODUCTION TO UNIX/LINUX KERNEL

BY

MR.PRASAD SAWANT



PREREQUISITES:

1. Working knowledge of C programming.
2. Basic Computer Architecture concepts.
3. Basic algorithms and data structure concepts.

OUT LINE OF UNIT

1) System Structure
2) User Perspective
3) Assumptions about Hardware
4) Architecture of UNIX Operating System
5) Concepts of Linux Programming
6) Files and the File system
7) Processes
8) Users and Groups
9) Permissions
10) Signals
11) Interprocess Communication

SYSTEM STRUCTURE

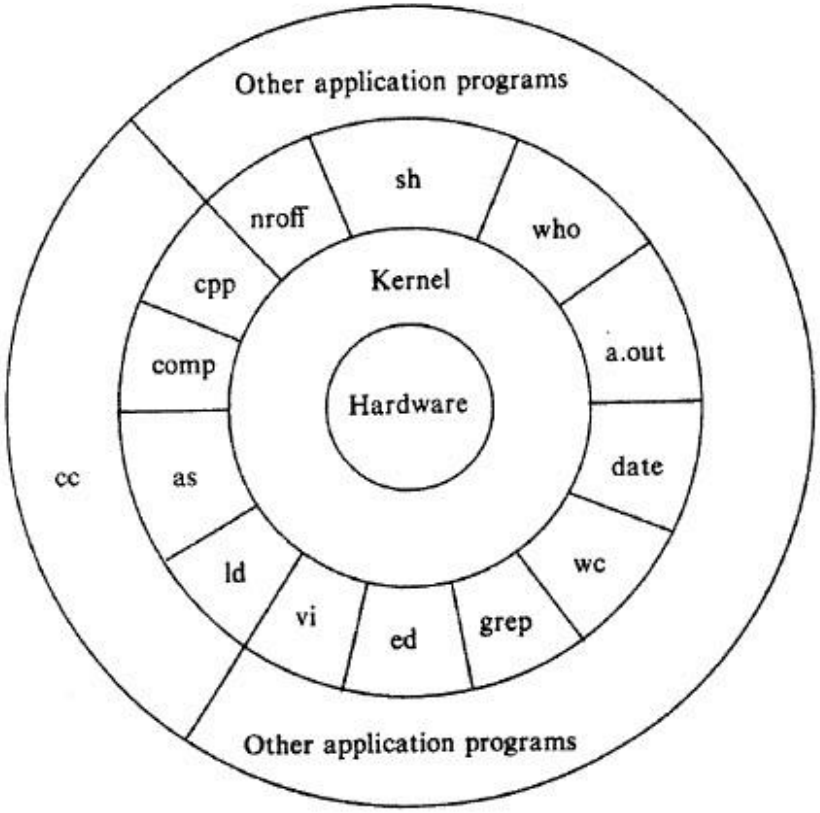


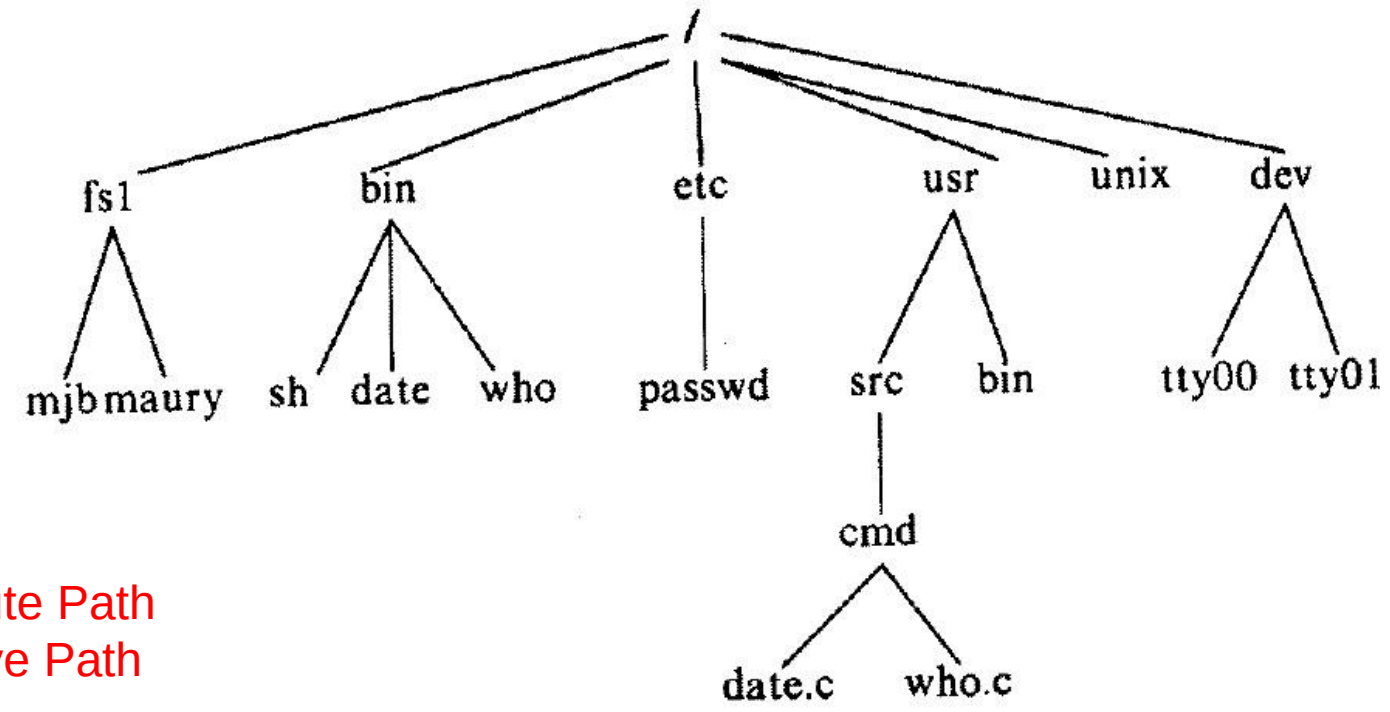
Figure 1.1. Architecture of UNIX Systems

USER PERSPECTIVE

The UNIX file system is characterized by

- a hierarchical structure,
- consistent treatment of file data,
- the ability to create and delete files,
- dynamic growth of files,
- the protection of file data,
- the treatment of peripheral devices (such as terminals and tape units) as files.

SAMPLE FILE SYSTEM TREE



Absolute Path
Relative Path

PROCESSING ENVIRONMENT

A *program* is an executable file, and a *process* is an instance of the program in execution .Many process can execute simultaneously on UNIX system with no logical limit to their number ,and many instances of a program can exist simultaneously in the system .

User Perspective

- the *fork* system call to create *a* new process. The new process, called the *child* process, gets a 0 return value from *fork* and invokes *execl*
- The *execl* call overlays the address space of the child process with the file "copy“.
- If the *execl* call succeeds, it never returns because the process executes in a new address space meanwhile, the process that had invoked *fork* (the parent) receives a non-0 return from the call, calls *wait:*, suspending its execution until *copy* finishes, prints the message "copy done," and *exits*

```
main(argc, argv)
    int argc;
    char *argv[];
{
    /* assume 2 args: source file and target file */
    if (fork() == 0)
        execl("copy", "copy", argv[1], argv[2], 0);
    wait((int *) 0);
    printf("copy done\n");
}
```

Figure 1.4. Program that Creates a New Process to Copy Files

USER PERSPECTIVE: SHELL

The shell allows three types of commands.

- First, a command can be an executable file that contains object code produced by compilation of source code (a C program for example).
- Second, a command can be an executable file that contains a sequence of shell command lines
- The internal commands make the shell a programming language in addition to a command interpreter and include commands for looping

USER PERSPECTIVE :
BUILDING BLOCK PRIMITIVES

redirect I/O

Processes conventionally have access to three files: they read from their *standard input* file, write to their *standard output* file, and write error messages to their *standard error* file.

```
[root@localhost ch1]# ls
a.out  demo1.c  demo2.c  output
[root@localhost ch1]#  ls > output1
[root@localhost ch1]# cat output1
a.out
demo1.c
demo2.c
output
output1
[root@localhost ch1]#
```


USER PERSPECTIVE :
BUILDING BLOCK PRIMITIVES

redirect I/O

```
[root@localhost ch1]# gcc demo2.c 2> error.txt
[root@localhost ch1]# cat error.txt
demo2.c: In function 'main':
demo2.c:9: error: expected ';' before '}' token
[root@localhost ch1]#
```


USER PERSPECTIVE : **BUILDING BLOCK PRIMITIVES**

Pipe

- The *pipe*, a mechanism that allows a stream of data to be passed between reader and writer processes. Processes can redirect their standard **output to a pipe to be read by other** processes that have redirected their standard input to come from the pipe.
- The data that the first processes write into the pipe is the input for the second processes. The second processes could also redirect their output, and so on, depending on programming need. Again, the processes need not know what type of file their standard **output is;they work regardless of whether their standard output** is a regular file, a pipe, or a device

USER PERSPECTIVE :
BUILDING BLOCK PRIMITIVES

Pipe

```
[root@localhost ch1]# grep main demo1.c demo2.c
demo1.c:main()
demo2.c:main()
[root@localhost ch1]#
```

```
[root@localhost ch1]# grep main demo1.c demo2.c
demo1.c:main()
demo2.c:main()
[root@localhost ch1]# grep main demo1.c demo2.c | wc -l
2
[root@localhost ch1]#
```


ASSUMPTIONS ABOUT HARDWARE

The execution of user processes on UNIX systems is divided into two levels: user and kernel. When a process executes a system call, the *execution mode* of the process changes from *user mode* to *kernel mode:* the operating system executes and attempts to service the user request, returning an error code if it fails. Even if the user makes no explicit requests for operating system services, the operating system still does bookkeeping operations that relate to the user process, handling interrupts, scheduling processes, managing memory, and so on. Many machine architectures (and their operating systems) support more levels than the two outlined here, but the two modes, user and kernel, are sufficient for UNIX systems.

ASSUMPTIONS ABOUT HARDWARE

USER VS KERNEL

1. Processes in user mode can access their own instructions and data but not kernel instructions and data (or those of other processes). Processes in kernel mode,however, can access kernel and user addresses.
2. Some machine instructions are privileged and result in an error when executed in user mode.

CONCEPTS OF LINUX PROGRAMMING
FILES AND THE FILE SYSTEM

1. The file is the most basic and fundamental abstraction in Linux. Linux follows the *everything-is-a-file*
2. In order to be accessed, a file must first be opened. Files can be opened for reading, writing, or both
3. An open file is referenced via a unique descriptor, a mapping from the metadata associated with the open file back to the specific file itself. Inside the Linux kernel, this descriptor is handled by an integer (of the C type int) called the *file descriptor*, abbreviated *fd*.

FILES AND THE FILESYSTEM

Regular files

What most of us call “files” are what Linux labels *regular files*. A regular file contains bytes of data, organized into a linear array called a byte stream. In Linux, no further organization or formatting is specified for a file.

Directories and links

Accessing a file via its inode number is cumbersome so files are always opened from user space by a name, not an inode number.

Directories are used to provide the names with which to access files. A directory acts as a mapping of human-readable names to inode numbers. A name and inode pair is called a *link*. The physical on-disk form of this mapping a simple table, a hash, or Whatever is implemented and managed by the kernel code that supports a given filesystem. Conceptually, a directory is viewed like any normal file, with the difference that it contains only a mapping of names to inodes. The kernel directly uses this mapping to perform name-to-inode resolutions.

FILES AND THE FILE SYSTEM

Hard links

Conceptually, nothing covered thus far would prevent multiple names resolving to the same inode. Indeed, this is allowed. When multiple links map different names to the same inode, we call them *hard links*.

Hard links allow for complex filesystem structures with multiple pathnames pointing to the same data. The hard links can be in the same directory, or in two or more different directories.

FILES AND THE FILE SYSTEM

Symbolic links

Hard links cannot span filesystems because an inode number is meaningless outside of the inode's own filesystem. To allow links that can span filesystems, and that are a bit simpler and less transparent, Unix systems also implement *symbolic links* (often shortened to *symlinks*).

Symbolic links look like regular files. A symlink has its own inode and data chunk, which contains the complete pathname of the linked-to file. This means symbolic links can point anywhere, including to files and directories that reside on different filesystems, and even to files and directories that do not exist. A symbolic link that points to a nonexistent file is called a *broken link*.

USERS AND GROUPS

Authorization in Linux is provided by *users* and *groups*. Each user is associated with a unique positive integer called the *user ID* (uid). Each process is in turn associated with exactly one uid, which identifies the user running the process, and is called the process' *real uid*. Inside the Linux kernel, the uid is the only concept of a user. Users themselves, however, refer to themselves and other users through *usernames*, not numerical values. Usernames and their corresponding uids are stored in */etc/passwd*, and library routines map user-supplied usernames to the corresponding uids.

PERMISSIONS

Table 1-1. Permission bits and their values

Bit	Octal value	Text value	Corresponding permission
8	400	`r--------`	Owner may read
7	200	`-w-------`	Owner may write
6	100	`--x------`	Owner may execute
5	040	`---r-----`	Group may read
4	020	`----w----`	Group may write
3	010	`-----x---`	Group may execute
2	004	`------r--`	Everyone else may read
1	002	`-------w-`	Everyone else may write
0	001	`--------x`	Everyone else may execute

SIGNALS

Signals are a mechanism for one-way asynchronous notifications. A signal may be sent from the kernel to a process, from a process to another process, or from a process to itself. Signals typically alert a process to some event, such as a segmentation fault, or the user pressing Ctrl-C.

INTERPROCESS COMMUNICATION

Allowing processes to exchange information and notify each other of events is one of an operating system's most important jobs.