GENERAL REVIEW OF THE SYSTEM:

The UNIX system has become quite popular since its inception in 1969, running on machines of varying processing power from microprocessors to mainframes and providing a common execution environment across them. The system is divided into two parts. The first part consists of programs and services that have made the UNIX system environment so popular; it is the part readily apparent to users, including such programs as the shell, mail, text processing packages, and source code control systems. The second part consists of the operating system that supports these programs and services.

HISTORY

In 1965, Bell Telephone Laboratories joined an effort with the General Electric Company and Project MAC of the Massachusetts Institute of Technology to develop a new operating system called Multics. The goals of the Multics system were to provide simultaneous computer access to a large community of users, to supply ample computation power and data storage, and to allow users to share their data easily, if desired.

Many people who later took part in the early development of the UNIX system participated in the Multics work at Bell Laboratories. Although a primitive version of the Multics system was running on a GE 645 computer by 1969, it did not provide the general service computing for which it was intended, nor was it clear when its development goals would be met.

Consequently, Bell Laboratories ended its participation in the project. With the end of their work on the Multics project, members of the Computing Science Research Center at Bell Laboratories were left without a "convenient interactive computing service". In an attempt to improve their programming environment, Ken Thompson, Dennis Ritchie, and others sketched a paper design of a file system that later evolved into an early version of the UNIX file system.

Thompson wrote programs that simulated the behavior of the proposed file system and of programs in a demand-paging environment, and he even encoded a simple kernel for the GE 645 computer. At the same time, he wrote a game program, "Space Travel," in Fortran for a GECOS system (the Honeywell 635), but the program was unsatisfactory because it was difficult to control the "space ship" and the program was expensive to run. Thompson later found a little-used PDP-7 computer that provided good graphic display and cheap executing power.

Programming "Space Travel" for the PDP-7 enabled Thompson to learn about the machine, but its environment for program development required cross-assembly of the program on the GECOS machine and carrying paper tape for input to the PDP-7.

To create a better development environment, Thompson and Ritchie implemented their system design on the PDP-7, including an early version of the UNIX file system, the process subsystem, and a small set of utility programs.

Eventually, the new system no longer needed the GECOS system as a development environment but could support itself. **The new system was given the name UNIX** a pun on the name Multics coined by another member of the Computing Science Research Center, Brian Kernighan.

Several reasons have been suggested for the popularity and success of the UNIX system:

- ✓ The system is written in a high-level language, making it easy to read, understand, change, and move to other machines.
- ✓ It has a simple user interface that has the power to provide the services that users want.
- ✓ It provides primitives that permit complex programs to be built from simpler programs.
- ✓ It uses a hierarchical file system that allows easy maintenance and efficient implementation.
- ✓ It uses a hierarchical file system that allows easy maintenance and efficient implementation.
- ✓ It provides a simple, consistent interface to peripheral devices.
- ✓ It is a multi-user, multiprocess system; each user can execute several processes simultaneously.
- ✓ It hides the machine architecture from the user, making it easier to write programs that run on different hardware implementations.

Although the operating system and many of the command programs are written in C, UNIX systems support other languages, including FORTRAN, Basic, Pascal, Ada, COBOL, Lisp, and Prolog. The UNIX system can support any language that has a compiler or interpreter and a system interface that maps user requests for operating system services to the standard set of requests used on UNIX systems.

SYSTEM STRUCTURE (Topic beyond Syllabus):

Figure 1.1 depicts the high-level architecture of the UNIX system. The hardware at the center of the diagram provides the operating system with basic services. The operating system interacts directly with the hardware, providing common services to programs and insulating them from hardware idiosyncrasies.

Viewing the system as a set of layers, the operating system is commonly called the system kernel, or just the kernel, emphasizing its isolation from user programs. Because programs are independent of the underlying hardware, it is easy to move them between UNIX systems running on different hardware if the programs do not make assumptions about the underlying hardware.

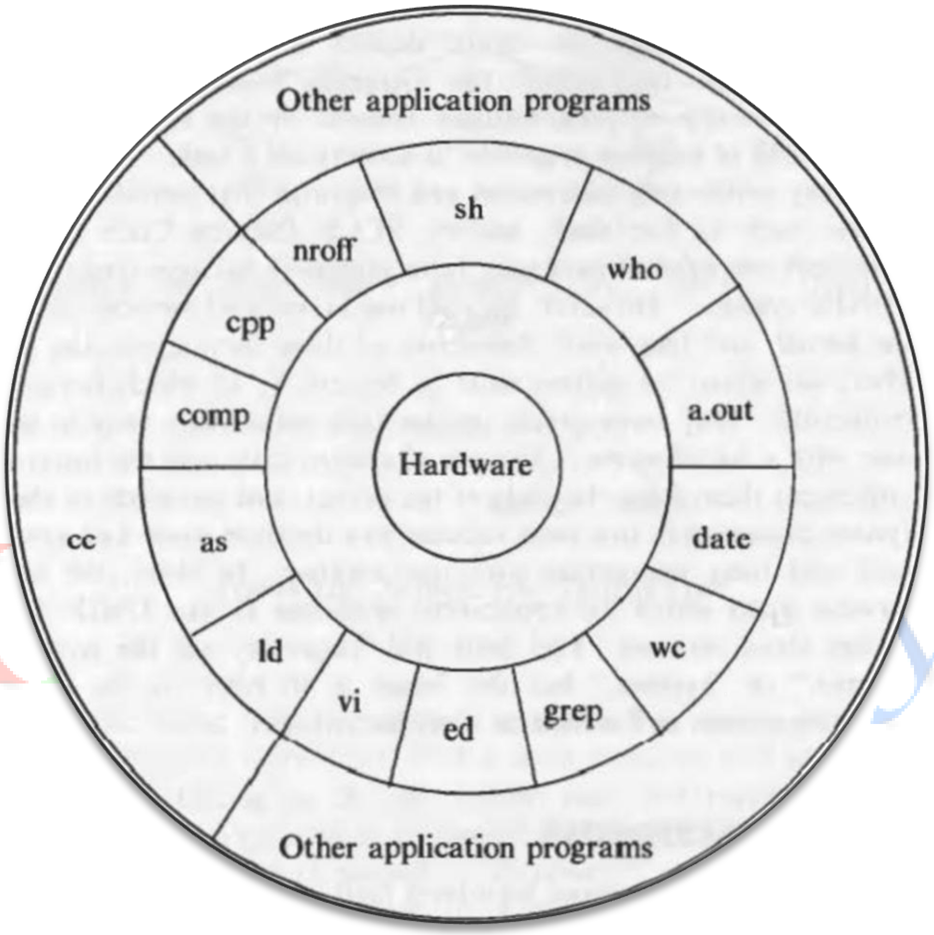


Figure 1.1 Architecture of UNIX System

For instance, programs that assume the size of a machine word are more difficult to move to other machines than programs that do not make this assumption. Programs such as the shell and editors (ed and vi) shown in the outer layers interact with the kernel by invoking a well defined set of system calls.

The system calls instruct the kernel to do various operations for the calling program and exchange data between the kernel and the program. Several programs shown in the figure are in standard system configurations and are known as commands, but private user programs may also exist in this layer as indicated by the program whose name is a.out, the standard name for executable files produced by the C compiler.

Other application programs can build on top of lower-level programs, hence the existence of the outermost layer in the figure. For example, the standard C compiler, cc, is in the outermost layer of the figure: it invokes a C preprocessor, two-pass compiler, assembler, and loader Oink-editor), all separate lower-level programs.

ARCHITECTURE OF THE UNIX OPERATING SYSTEM:

Figure 1.2 gives a block diagram of the kernel, showing various modules and their relationships to each other. In particular, it shows the file subsystem on the left and the process control subsystem on the right, the two major components of the kernel. The diagram serves as a useful logical view of the kernel, although in practice the kernel deviates from the model because some modules interact with the internal operations of others.

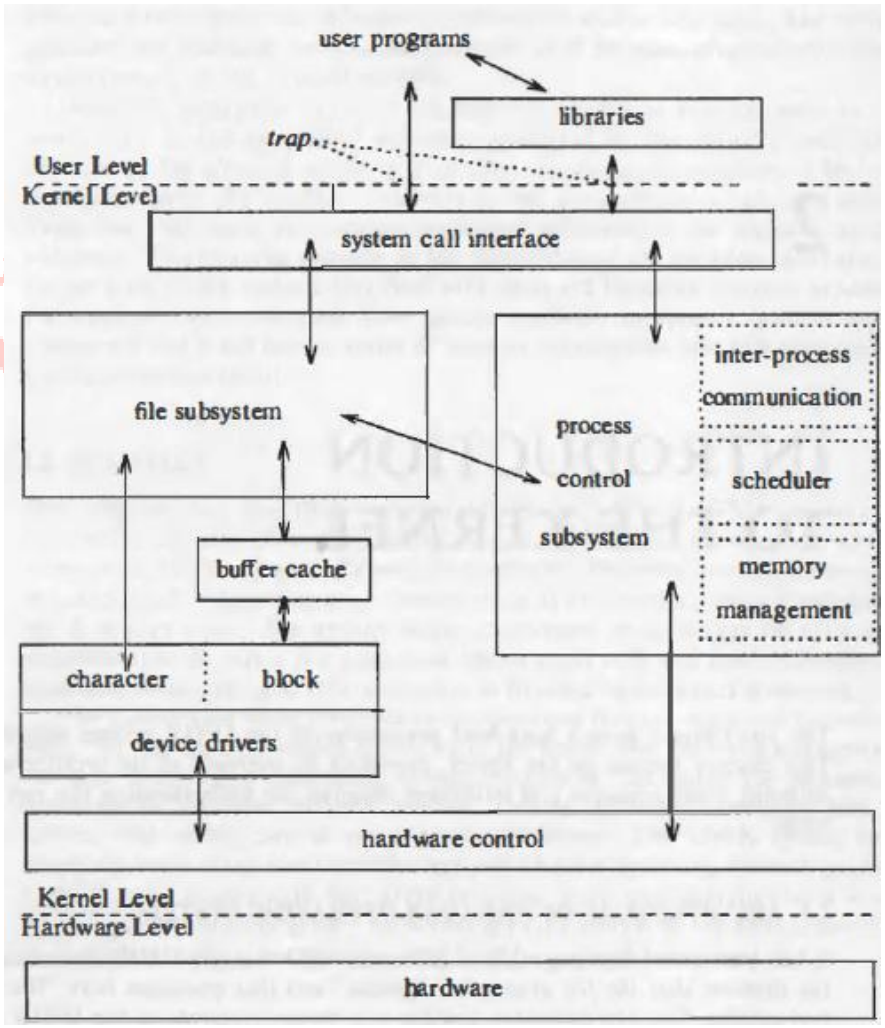


Figure 1.2 Block Diagram of the System Kernel

Figure 1.2 shows three levels: user, kernel, and hardware. The system call and library interface represent the border between user programs and the kernel depicted in Figure 1.1.

System calls look like ordinary function calls in C programs, and libraries map these function calls to the primitives needed to enter the operating system.

Assembly language programs may invoke system calls directly without a system call library, however. Programs frequently use other libraries such as the standard I/O library to provide a more sophisticated use of the system calls. The libraries are linked with the programs at compile time.

The figure partitions the set of system calls into those that interact with the file subsystem and those that interact with the process control subsystem.

The file subsystem manages files, allocating file space, administering free space, controlling access to files, and retrieving data for users.

Processes interact with the file subsystem via a specific set of system calls, such as open (to open a file for reading or writing), close, read, write, stat (query the attributes of a file), chown (change the record of who owns the file), and chmod (change the access permissions of a file).

The file subsystem accesses file data using a buffering mechanism that regulates data flow between the kernel and secondary storage devices. The buffering mechanism interacts with block I/O device drivers to initiate data transfer to and from the kernel. Device drivers are the kernel modules that control the operation of peripheral devices. Block I/O devices are random access storage devices; alternatively, their device drivers make them appear to be random access storage devices to the rest of the system.

For example, a tape driver may allow the kernel to treat a tape unit as a random access storage device. The file subsystem also interacts directly with "raw" I/O device drivers without the intervention of a buffering mechanism. Raw devices, sometimes called character devices, include all devices that are not block devices.

The process control subsystem is responsible for process synchronization, inter process communication, memory management, and process scheduling. The file subsystem and the process control subsystem interact when loading a file into memory for execution where the process subsystem reads executable files into memory before executing them.

Some of the system calls for controlling processes are: fork (create a new process), exec (overlay the image of a program onto the running process), exit (finish executing a process), wait (synchronize process execution with the exit of a previously forked process), brk (control the size of memory allocated to a process), and signal (control process response to extraordinary events).

The memory management module controls the allocation of memory. If at any time the system does not have enough physical memory for all processes, the kernel moves them between main memory and secondary memory so that all processes get a fair chance to execute.

The scheduler module allocates the CPU to processes. It schedules them to run in turn until they voluntarily relinquish the CPU while awaiting a resource or until the kernel preempts them when their recent run time exceeds a time quantum. The scheduler then chooses the highest priority eligible process to run; the original process will run again when it is the highest priority eligible process available.

USER PERSPECTIVE:

This will briefly review high-level features of the UNIX system such as the file system, the processing environment, and building block primitives (for example, pipes).

The File System:

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.

The file system is organized as a tree with a single root node called root (written as “/"); every non-leaf node of the file system structure is a directory of files, and files at the leaf nodes of the tree are either directories, regular files, or special device files.

The name of a file is given by a path name that describes how to locate the file in the file system hierarchy.

A ***path name*** is a sequence of component names separated by slash characters; a component is a sequence of characters that designates a file name that is uniquely contained in the previous (directory) component.

A *full path name* starts with a slash character and specifies a file that can be found by starting at the file system root and traversing the file tree, following the branches that lead to successive component names of the path name.

Thus, the path names “/etc/passwd”, “/bin/who”, and “/usr/src/cmd/who.c” designate files in the tree shown in Figure 1.3, but “/bin/passwd” and “/usr/src/date.c” do not.

A path name does not have to :start from root but can be designated relative to the current directory of an executing process, by omitting the initial slash in the path name.

Thus, starting from directory “/dev”, the path name “tty01” designates the file whose full path name is “/dev/ttyo1”.

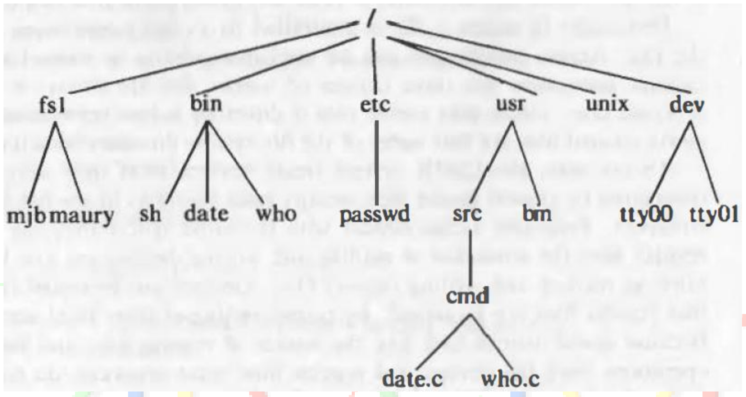


Figure 1.3 Sample File System Tree

Programs in the UNIX system have no knowledge of the internal format in which the kernel stores file data, treating the data as an unformatted stream of bytes. Programs may interpret the byte stream as they wish, but the interpretation bas no bearing on how the operating system stores the data. Thus, the syntax of accessing the data in a file is defined by the system and is identical for all programs, but the semantics of the data are imposed by the program.

Directories are like regular files in this respect; the system treats the data in a directory as a byte stream, but the data contains the names of the files in the directory in a predictable format so that the operating system and programs such as Is (list the names and attributes of files) can discover the files in a directory.

Permission to access a file is controlled by access permissions associated with the file. Access permissions can be set independently to control read, write, and execute permission for three classes of users: the file owner, a file group, and everyone else. Users may create files if directory access permissions allow it. The newly created files are leaf nodes of the file system directory structure.

To the user, the UNIX system treats devices as if they were files. Devices, designated by special device files, occupy node positions in the file system directory structure.

Programs access devices with the same syntax they use when accessing regular files; the semantics of reading and writing devices are to a large degree the same as reading and writing regular files.

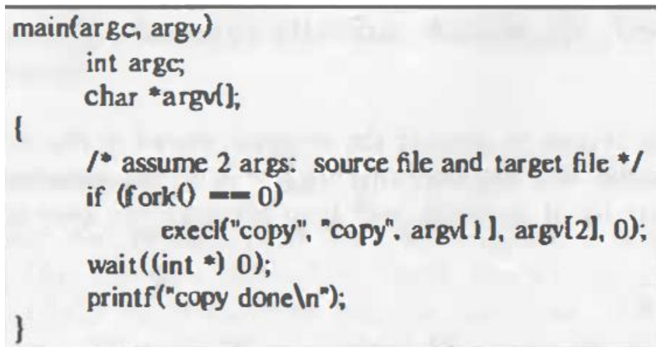
Devices are protected in the same way that regular files are protected: by proper setting of their (file) access permissions.

Processing Environment:

A program is an executable file, and a process is an instance of the program in execution. Many processes can execute simultaneously on UNIX systems (this feature is sometimes called multiprogramming or multitasking) with no logical limit to their number, and many instances of a program (such as copy) can exist simultaneously in the system.

Various system calls allow processes to create new processes, terminate processes, synchronize stages of process execution, and control reaction to various events. Subject to their use of system calls, processes execute independently of each other.

For example, a process executing the program in Figure 1.4 executes 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* to execute the program copy.



```
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 Programs that Creates a New Process to Copy Files

Building Block Primitives:

The philosophy of the UNIX system is to provide operating system primitives that enable users to write small, modular programs that can be used as building blocks to build more complex programs. One such primitive visible to shell users is the capability to 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.

Processes executing at a terminal typically use the terminal for these three files, but each may be "redirected" independently.

For instance, the command line ***ls*** lists all files in the current directory on the standard output. But the command line **ls > output** redirects the standard output to the file called "output" in the current directory.

The second building block primitive is 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.

When using the smaller programs as building blocks for a larger, more complex program, the programmer uses the pipe primitive and redirection of I/O to integrate the piece parts.

Indeed, the system tacitly encourages such programming style so that new programs can work with existing programs. For example, the program grep searches a set of files (parameters to grep) for a given pattern:

grep main a.c b.c c.c

searches the three files a.c, b.c, and c.c for lines containing the string "main" and prints the lines that it finds onto standard output. Sample output may be:

a.c: main(argc, argv)
c.c: /* here is the main loop in the program */
c.c: main()

The program we with the option -1 counts the number of lines in the standard input file. The command line ***grep main a.c b.c c.c | wc -l*** counts the number of lines in the files that contain the string "main"; the output from grep is "piped" directly into the wc command. For the previous sample output from grep, the output from the piped command is ***3.*** The use of pipes frequently makes it unnecessary to create temporary files.

THE BUFFER CACHE:

The kernel maintains files on mass storage devices such as disks, and it allows processes to store new information or to recall previously stored information.

When a process wants to access data from a file, the kernel brings the data into main memory where the process can examine it, alter it, and request that the data be saved in the file system again.

The kernel could read and write directly to and from the disk for all file system accesses, but system response time and throughput would be poor because of the slow disk transfer rate.

The kernel therefore attempts to minimize the frequency of disk access by keeping a pool of internal data buffers, called the ***buffer cache***, which contains the data in recently used disk blocks.

The position of the buffer cache module in the kernel architecture is in between the file subsystem and (block) device drivers.

When reading data from the disk, the kernel attempts to read from the buffer cache. If the data is already in the cache, the kernel does not have to read from the disk.

If the data is not in the cache, the kernel reads the data from the disk and caches it, using an algorithm that tries to save as much good data in the cache as possible.

Similarly, data being written to disk is cached so that it will be there if the kernel later tries to read it.

The kernel also attempts to minimize the frequency of disk write operations by determining whether the data must really be stored on disk or whether it is transient data that will soon be overwritten.

Higher-level kernel algorithms instruct the buffer cache module to pre-cache data or to delay-write data to maximize the caching effect.

BUFFER HEADERS:

During system initialization, the kernel allocates space for a number of buffers, configurable according to memory size and system performance constraints.

A buffer consists of two parts: a memory array that contains data from the disk and a buffer header that identifies the buffer.

The data in a buffer corresponds to the data in a logical disk block on a file system, and the kernel identifies the buffer contents by examining identifier fields in the buffer header.

The buffer is the in-memory copy of the disk block; the contents of the disk block map into the buffer, but the mapping is temporary until the kernel decides to map another disk block into the buffer. A disk block can never map into more than one buffer at a time.

If two buffers were to contain data for one disk block, the kernel would not know which buffer contained the current data and could write incorrect data back to disk.

For example, suppose a disk block maps into two buffers, A and B. If the kernel writes data first into buffer A and then into buffer B, the disk block should contain the contents of buffer B if all write operations completely fill the buffer.

However, if the kernel reverses the order when it copies the buffers to disk, the disk block will contain incorrect data.

The buffer header (Figure 1.5) contains a device number field and a block number field specifies the file system and the block number of the data on disk and uniquely identify the buffer.

Note: The buffer cache is a software structure that should not be confused with hardware caches that speed memory references.

The device number is the logical file system number, not a physical device (disk) unit number.

The buffer header also contains a pointer to a data array for the buffer, whose size must be at least as big as the size of a disk block, and a status field that summarizes the current status of the buffer.

The status of a buffer is a combination of the following conditions:

- The buffer is currently locked
- The buffer contains valid data
- The kernel must write the buffer contents to disk before reassigning the buffer; this condition is known as "delayed-write"
- The kernel is currently reading or writing the contents of the buffer to disk
- A process is currently waiting for the buffer to become free

The buffer header also contains two sets of pointers, used by the buffer allocation algorithms to maintain the overall structure of the buffer pool.

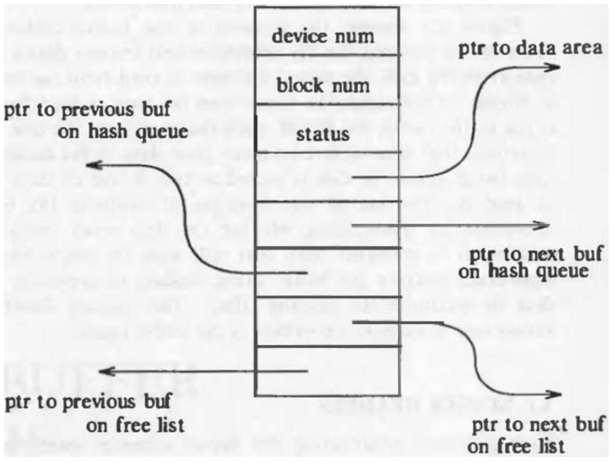

device num
block num
status
ptr to data area
ptr to previous buf on hash queue
ptr to next buf on hash queue
ptr to previous buf on free list
ptr to next buf on free list

Figure 1.5 Buffer Headers

STRUCTURE OF THE BUFFER POOL:

The kernel caches data in the buffer pool according to a least recently used algorithm: after it allocates a buffer to a disk block, it cannot use the buffer for another block until all other buffers have been used more recently.

The kernel maintains a free list of buffers that preserves the least recently used order. The free list is a doubly linked circular list of buffers with a dummy buffer header that marks its beginning and end (Figure 1.6).

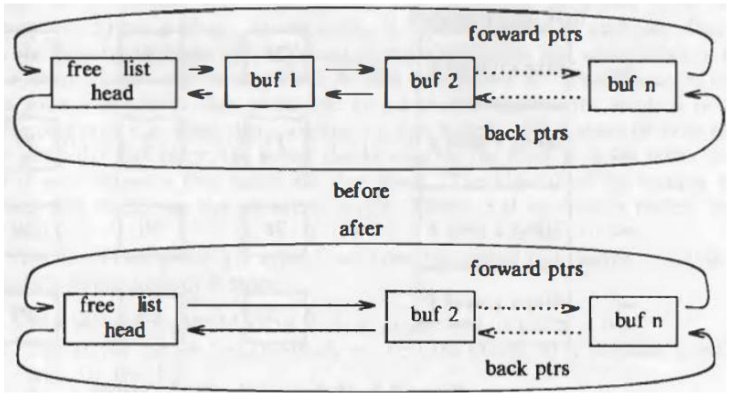


Figure 1.6 Free Lists of Buffers

Every buffer is put on the free list when the system is booted. The kernel takes a buffer from the head of the free list when it wants any free buffer, but it can take a buffer from the middle of the free list if it identifies a particular block in the buffer pool.

In both cases, it removes the buffer from the free list. When the kernel returns a buffer to the buffer pool, it usually attaches the buffer to the tail of the free list, occasionally to the head of the free list (for error cases), but never to the middle.

As the kernel removes buffers from the free list, a buffer with valid data moves closer and closer to head of the free list (Figure 1.6). Hence, the buffers that are closer to the head of the free list have not been used as recently as those that are further from the head of the free list.

When the kernel accesses a disk block, it searches for a buffer with the appropriate device-block number combination. Rather than search the entire buffer pool, it organizes the buffers into separate queues, hashed as a function of the device and block number.

The kernel links the buffers on a hash queue into a circular, doubly linked list, similar to the structure of the free list. The number of buffers on a hash queue varies during the lifetime of the system.

Figure 1.7 shows buffers on their hash queues: the headers of the hash queues are on the left side of the figure, and the squares on each row are buffers on a hash queue. Thus, squares marked 28, 4, and 64 represent buffers on the hash queue for "blkno 0 mod 4" (block number 0 modulo 4).

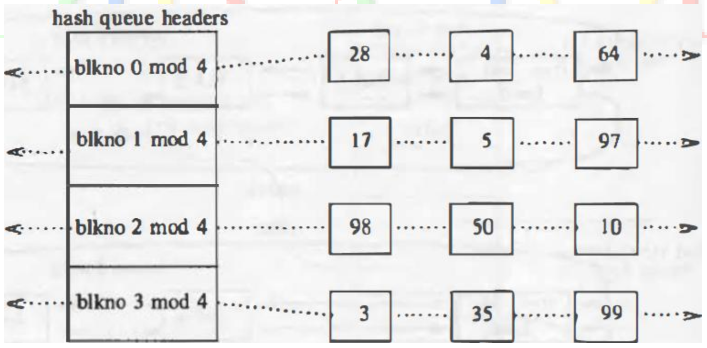


Figure 1.7 Buffers on Hash Queues

The dotted lines between the buffers represent the forward and back pointers for the hash queue. Each buffer always exists on a hash queue, but there is no significance to its position on the queue.

As stated above, no two buffers may simultaneously contain the contents of the same disk block; therefore, every disk block in the buffer pool exists on one and only one hash queue and only once on that queue. However, a buffer may be on the free list as well if its status is free.

SCENARIOS FOR RETRIEVAL OF A BUFFER:

High-level kernel algorithms in the file subsystem invoke the algorithms for managing the buffer cache. The high-level algorithms determine the logical device number and block number that they wish to access when they attempt to retrieve a block.

For example, if a process wants to read data from a file, the kernel determines which file system contains the file and which block in the file system contains the data. When about to read data from a particular disk block, the kernel checks whether the block is in the buffer pool and, if it is not there, assigns it a free buffer.

When about to write data to a particular disk block, the kernel checks whether the block is in the buffer pool, and if not, assigns a free buffer for that block.

There are five typical scenarios the kernel may follow in getblk to allocate a buffer for a disk block:

1. The kernel finds the block on its hash queue, and its buffer is free.
2. The kernel cannot find the block on the hash queue, so it allocates a buffer from the free list.
3. The kernel cannot find the block on the hash queue and, in attempting to allocate a buffer from the free list, finds a buffer on the free list that has been marked "delayed write". The kernel must write the "delayed write" buffer to disk and allocate another buffer.
4. The kernel cannot find the block on the hash queue, and the free list of buffers is empty.
5. The kernel finds the block on the hash queue, but its buffer is currently busy.

Figure 1.8 depicts the first scenario, where the kernel searches for block 4 on the hash queue marked "blkno 0 mod 4." Finding the buffer, the kernel removes it from the free list, leaving blocks 5 and 28 adjacent on the free list.

In the second scenario in algorithm getblk, the kernel searches the hash queue that should contain the block but fails to find it there. Since the block cannot be on another hash queue because it cannot "hash" elsewhere, it is not in the buffer cache.

So the kernel removes the first buffer from the free list instead; that buffer had been allocated to another disk block and is also on a hash queue.

If the buffer has not been marked for a delayed write, the kernel marks the buffer busy, removes it from the hash queue where it currently resides, reassigns the buffer header's device and block number to that of the disk block for which the process is searching, and places the buffer on the correct hash queue.

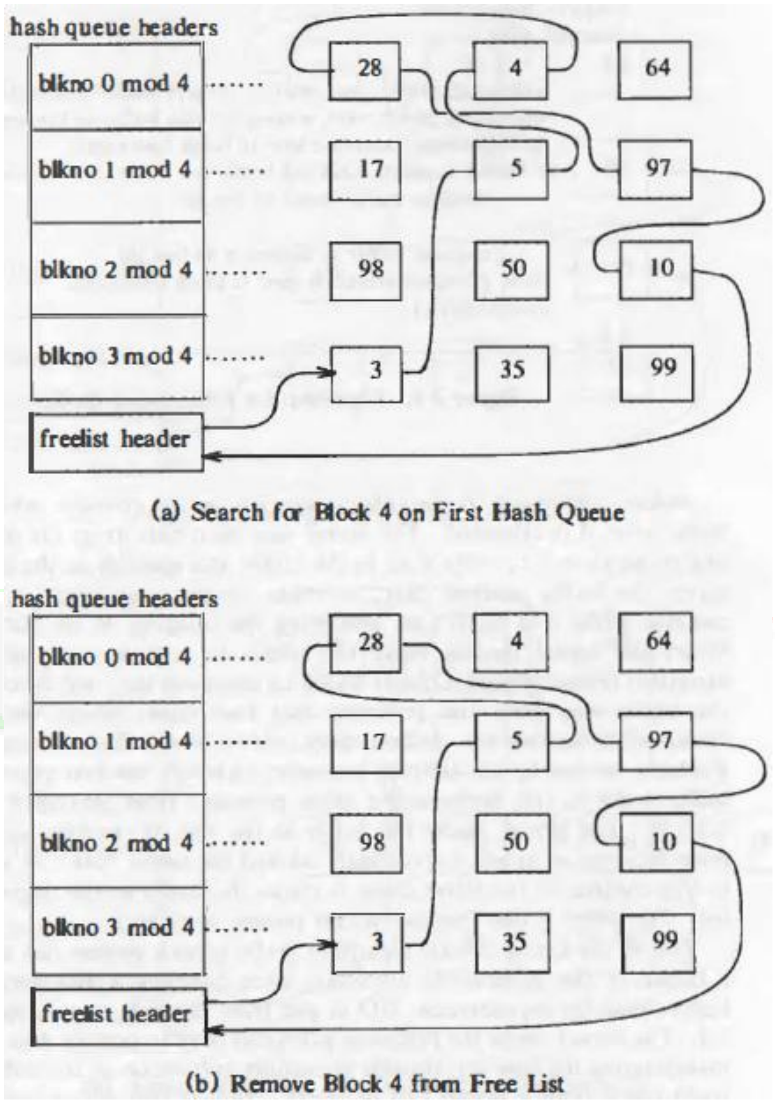


Figure 1.8 Scenario 1 in Finding a Buffer: Buffer on Hash Queue

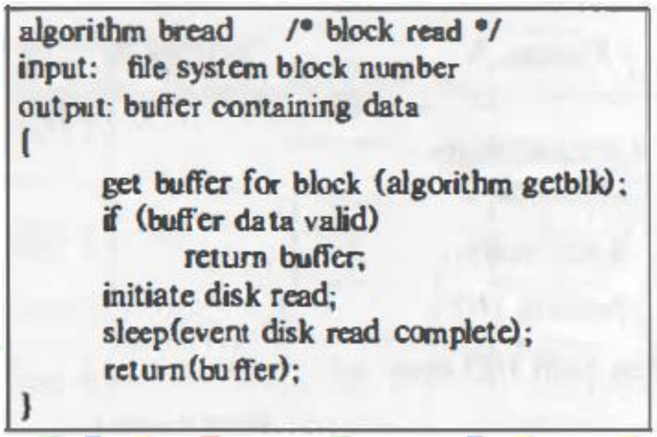
The kernel uses the buffer but has no record that the buffer formerly contained data for another disk block. A process searching for the old disk block will not find it in the pool and will have to allocate a new buffer for it from the free list.

In the third scenario in algorithm getblk, the kernel also has to allocate a buffer from the free list. However, it discovers that the buffer it removes from the free list has been marked for "delayed write," so it must write the contents of the buffer to disk before using the buffer.

The kernel starts an asynchronous write to disk and tries to allocate another buffer from the free list. When the asynchronous write completes, the kernel releases the buffer and places it at the head of the free list. The buffer had started at the end of the free list and had traveled to the head of the free list.

READING AND WRITING DISK BLOCKS:

To read a disk block (**Figure 1.9**), a process uses algorithm getblk to search for it in the buffer cache. If it is in the cache, the kernel can return it immediately without physically reading the block from the disk.



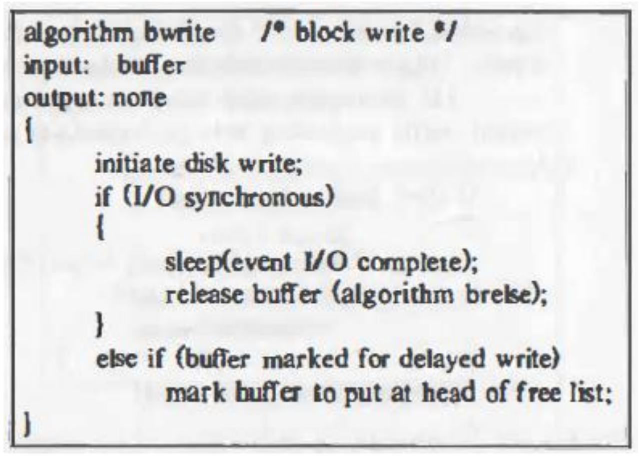
```
algorithm bread     /* block read */
input:  file system block number
output: buffer containing data
{
        get buffer for block (algorithm getblk);
        if (buffer data valid)
                return buffer;
        initiate disk read;
        sleep(event disk read complete);
        return(buffer);
}
```

Figure 1.9 Algorithm for Reading Disk Block

If it is not in the cache, the kernel calls the disk driver to "schedule" a read request and goes to sleep awaiting the event that the I/O completes. The disk driver notifies the disk controller hardware that it wants to read data, and the disk controller later transmits the data to the buffer.

Finally, the disk controller interrupts the processor when the I/O is complete, and the disk interrupt handler awakens the sleeping process; the contents of the disk block are now in the buffer. The modules that requested the particular block now have the data; when they no longer need the buffer they release it so that other processes can access it.

The algorithm for writing the contents of a buffer to a disk block is similar (**Figure 1.10**). The kernel informs the disk driver that it has a buffer whose contents should be output, and the disk driver schedules the block for I/O.



```
algorithm bwrite     /* block write */
input:  buffer
output: none
{
        initiate disk write;
        if (I/O synchronous)
        {
                sleep(event I/O complete);
                release buffer (algorithm brelse);
        }
        else if (buffer marked for delayed write)
                mark buffer to put at head of free list;
}
```

Figure 1.10 Algorithm for Writing a Disk Block

If the write is synchronous, the calling process goes to sleep awaiting I/O completion and releases the buffer when it awakens. If the write is asynchronous, the kernel starts the disk write but does not wait for the write to complete. The kernel will release the buffer when the I/O completes.

A delayed write is different from an asynchronous write. When doing an asynchronous write, the kernel starts the disk operation immediately but does not wait for its completion. For a "delayed write," the kernel puts off the physical write to disk as long as possible; then, recalling the third scenario in algorithm getblk, it marks the buffer "old" and writes the block to disk asynchronously.

ADVANTAGES AND DISADVANTAGES OF THE BUFFER CACHE:

Use of the buffer cache has several advantages and, unfortunately, some disadvantages:

- The use of buffers allows uniform disk access, because the kernel does not need to know the reason for the I/O. Instead, it copies data to and from buffers, regardless of whether the data is part of a file, an inode, or a super block.
- The system places no data alignment restrictions on user processes doing I/O, because the kernel aligns data internally.
    - Hardware implementations frequently require a particular alignment of data for disk l/0, such as aligning the data on a two-byte boundary or on a four-byte boundary in memory. Without a buffer mechanism, programmers would have to make sure that their data buffers were correctly aligned.
- Use of the buffer cache can reduce the amount of disk traffic, thereby increasing overall system throughput and decreasing response time.

  - Processes reading from the file system may find data blocks in the cache and avoid the need for disk I/O.
  - The kernel frequently uses "delayed write" to avoid unnecessary disk writes, leaving the block in the buffer cache and hoping for a cache hit on the block.

- The buffer algorithms help insure file system integrity, because they maintain a common, single image of disk blocks contained in the cache. If two processes simultaneously attempt to manipulate one disk block, the buffer algorithms (getblk for example) serialize their access, preventing data corruption.
- Reduction of disk traffic is important for good throughput and response time, but the cache strategy also introduces several disadvantages.
  - Since the kernel does not immediately write data to the disk for a delayed write, the system is vulnerable to crashes that leave disk data in an incorrect state.
- Use of the buffer cache requires an extra data copy when reading and writing to and from user processes.
  - A process writing data copies the data into the kernel, and the kernel copies the data to disk; a process reading data has the data read from disk into the kernel and from the kernel to the user process.
  - When transmitting large amounts of data, the extra copy slows down performance, but when transmitting small amounts of data, it improves performance because the kernel buffers the data (using algorithms getblk and delayed write) until it is economical to transmit to or from the disk.

UTILITIES:

GENERAL PURPOSE UTILITIES:

1. cal - Displays a calendar
2. date - print or set the system date and time
3. bc - An arbitrary precision calculator language
4. echo - Display a line of text.
5. printf - Format and print data
6. passwd - update user's authentication tokens

7. who - Show who is logged on

8. w - Show who is logged on and what they are doing

cal: Displays a calendar

Syntax

cal [−smjy13] [[[day] month] year]

Description: View calendar of specific month or year.

Options:

−1 Display single month output. (This is the default.)

−3 Display previous/current/next month output

−s Display Sunday as the first day of the week

−m Display Monday as the first day of the week

−j Display Julian dates (days one-based, numbered from January 1).

−y Display a calendar for the current year.

Eg: $cal

```
July 2016

Su  mo  tu we  th  fr  sa
1    2    3   4    5   6   7
8    9   10 11   12  13 14
15 16  17 18  19   20 21
22 23  24  25  26  27  28
29 30  31
```

date: Print or set the system date and time

Syntax:

date [OPTION]... [+FORMAT]

date [-u|--utc|--universal] [MMDDhhmm[[CC]YY][.ss]]

Description: Display the current time in the given FORMAT, or set the system date.

Options:

-d, --date=STRING

display time described by STRING, not `now'

-f, --file=DATEFILE

like --date once for each line of DATEFILE

-r, --reference=FILE

display the last modification time of FILE

-s, --set=STRING

set time described by STRING

-u, --utc, --universal

print or set Coordinated Universal Time

Formats

%d	day of month (e.g, 01)
%m	month (01..12)
%Y	year
%F	full date; same as %Y-%m-%d
%H	hour (00..23)
%M	minute (00..59)
%S	second (00..60)
%T	time; same as %H:%M:%S

Eg: $date

Tue Jul 19 18:37:07 IST 2016

bc: An arbitrary precision calculator language

Syntax:

bc [-hlwsqv] [long-options] [file ...]

Description:

Options:

-l, --mathlib

Define the standard math library.

-w, --warn

Give warnings for extensions to POSIX bc.

-s, --standard

Process exactly the POSIX bc language.

-q, --quiet

Do not print the normal GNU bc welcome.

Eg:
```
$bc –q
  Var=100
  Var
  100
  20<10
  0
```

echo: Display a line of text.

Syntax:

echo [SHORT-OPTION]... [STRING]...

echo LONG-OPTION

Description: Echo the STRING(s) to standard output.

Options:

-n do not output the trailing newline

-e enable interpretation of backslash escapes

-E disable interpretation of backslash escapes (default)

Escape Sequence

If -e is in effect, the following sequences are recognized:

\\	backslash
\a	alert (BEL)
\b	backspace
\n	new line
\r	carriage return
\t	horizontal tab
\v	vertical tab

Eg:
```
$echo "Unix is precious"
    Unix is precious
```

who: Show who is logged on

Syntax:

who [OPTION]... [FILE | ARG1 ARG2]

Description: Print information about users who are currently logged in.

Options:

-a, --all

same as -b -d --login -p -r -t -T -u

-b, --boot

time of last system boot

-H, --heading

print line of column headings

-l, --login

print system login processes

Eg:
```
$who
    14121a05c4      tty        2016-07-19     12:30  (:0)
    14121a05c7      pts/0      2016-07-19     12:32 (:0.0)
```

w: Show who is logged on and what they are doing

Syntax:

w - [husfV] [user]

Description: The command w on many Unix-like operating systems provides a quick summary of every user logged into a computer, what each user is currently doing.

Options:

-h Don’t print the header.

-u Ignores the username while figuring out the current process and CPU times.

-s Use the short format. Don’t print the login time, JCPU or PCPU times.

Eg: $w

```
USER       TTY     FROM     LOGIN@     IDLE    JCPU    PCPU    WHAT
Panther     tty1       :0           12:30      33:28         60:53s    0.09s  pam:gdm
```

passwd: update user's authentication tokens

Syntax:

passwd [-k] [-l] [-u [-f]] [-d] [-n mindays] [-x maxdays] [-w warndays] [-i inactivedays] [-S] [--stdin] [username]

Description: The passwd utility is used to update user's authentication token(s)

Options:

-k The option -k, is used to indicate that the update should only be for expired authentication tokens (passwords); the user wishes to keep their non-expired tokens as before.

-l This option is used to lock the specified account and it is available to root only. The locking is performed by rendering the encrypted password into an invalid string (by prefixing the encrypted string with an !).

```
Eg:   $passwd
        changing password for user eleiss.
```

printf: Format and print data

Syntax:

printf FORMAT [ARGUMENT]...

printf OPTION

Description: Print ARGUMENT(s) according to FORMAT, or execute according to OPTION

Options:

--help Display help and exit

Eg:
```
$ printf ”unix\v is\v precious\n”
 unix
is
 precious
```

FILE HANDLING UTILITIES:

1. pwd - print name of current/working/present directory
2. cd - Change the current directory
3. mkdir - make directories
4. rmdir - remove empty directories
5. ls - list directory contents
6. cat - concatenate files and print on the standard output
7. cp - copy files and directories
8. rm - remove files or directories
9. mv - move (rename) files
10. wc - print newline, word, and byte counts for each file
11. cmp - compare two files
12. comm - compare two sorted files line by line
13. diff - find differences between two files

pwd: Print name of current/working/present directory

Syntax: pwd [OPTION]...

Description: Print the full filename of the current working directory.

Options:

-L, --logical

use PWD from environment, even if it contains symlinks

-P, --physical

avoid all symlinks

Eg: $pwd

/home/eleiss

cd: Change the current directory

Syntax: cd [-L|-P] [dir]

Description: The cd command, also known as chdir (change directory), is a command-line OS shell command used to change the current working directory in operating systems such as UNIX.

Options:

-P option to the set built-in command

-L option forces symbolic links to be followed

Eg: $cd /etc

```
    $pwd
      /etc
```

mkdir: make directories

Syntax: mkdir [OPTION]... DIRECTORY...

Description: Create the DIRECTORY(ies), if they do not already exist.

Options:

-m, --mode=MODE

set file mode (as in chmod), not a=rwx - umask

-p, --parents

no error if existing, make parent directories as needed

-v, --verbose

print a message for each created directory

Eg: mkdir -v linux

Mkdir: created directory 'linux'

rmdir: remove empty directories

Syntax: rmdir [OPTION]... DIRECTORY...

Description: Remove the DIRECTORY(ies), if they are empty.

Options:

--ignore-fail-on-non-empty

ignore each failure that is solely because a directory is non-empty

-p, --parents

remove DIRECTORY and its ancestors; e.g., 'rmdir -p a/b/c' is similar to 'rmdir a/b/c a/b a'

Eg: rmdir friends

Rmdir: failed to removes 'friends':directory not empty

ls: list directory contents

Syntax: ls [OPTION]... [FILE]...

Description: List information about the FILEs (the current directory by default). Sort entries alphabetically if none of -cftuvSUX nor --sort.

Options:

-a, --all

do not ignore entries starting with .

-A, --almost-all

do not list implied . and ..

-c with -lt: sort by, and show, ctime (time of last modification of file status information) with -l: show ctime and sort by name otherwise: sort by ctime

-C list entries by columns

Eg: ls

```
City   country    friends    fruits    linux    states    tree
```

cat: Concatenate files and print on the standard output

Syntax: cat [OPTION]... [FILE]...

Description: Concatenate FILE(s), or standard input, to standard output.

Options:

-b, --number-nonblank

number nonempty output lines

-E, --show-ends

display $ at end of each line

-s, --squeeze-blank

suppress repeated empty output lines

-T, --show-tabs

display TAB characters as ^I

-v, --show-nonprinting

use ^ and M- notation, except for LFD and TAB

Eg: cat india

```
India is a great country. Jai Hind.
```

cp: Copy files and directories

Syntax: cp [OPTION]... [-T] SOURCE DEST

cp [OPTION]... SOURCE... DIRECTORY

cp [OPTION]... -t DIRECTORY SOURCE...

Description: Copy SOURCE to DEST, or multiple SOURCE(s) to DIRECTORY.

Options:

-a, --archive

same as -dR --preserve=all

-d same as --no-dereference --preserve=links

-i, --interactive

prompt before overwrite (overrides a previous -n option)

-H follow command-line symbolic links in SOURCE

-l, --link

link files instead of copying

-p same as --preserve=mode, ownership, timestamps

-R, -r, --recursive

copy directories recursively

Eg: cp –v /etc/passwd.

‘/etc/passwd\’->’./paswd\’

rm: Remove files or directories

Syntax: rm [OPTION]... FILE...

Description: rm removes each specified file. By default, it does not remove directories.

Options:

-f, --force

ignore nonexistent files, never prompt

-i prompt before every removal

-I prompt once before removing more than three files, or when removing recursively. Less intrusive than -i, while still giving protection against most mistakes

-r, -R, --recursive

remove directories and their contents recursively

-v, --verbose

explain what is being done

Eg: rm –v divya/file1

Removed’divya/file1’

mv: Move or Rename files

Syntax: mv [OPTION]... [-T] SOURCE DEST

mv [OPTION]... SOURCE... DIRECTORY

mv [OPTION]... -t DIRECTORY SOURCE...

Description: Rename SOURCE to DEST, or move SOURCE(s) to DIRECTORY.

Options:

-f, --force

do not prompt before overwriting

-i, --interactive

prompt before overwrite

-t, --target-directory=DIRECTORY

move all SOURCE arguments into DIRECTORY

-T, --no-target-directory

treat DEST as a normal file

-u, --update

move only when the SOURCE file is newer than the destination file or when the destination file is missing

Eg: mv -v /tmp/shell_scripting.

‘/tmp/shell_scripting’->’./shell_scripting’

wc: print newline, word, and byte counts for each file

Syntax: wc [OPTION]... [FILE]...

wc [OPTION]... --files0-from=F

Description: Print newline, word, and byte counts for each FILE, and a total line if more than one FILE is specified. With no FILE, or when FILE is -, read standard input.

Options:

-c, --bytes

print the byte counts

-m, --chars

print the character counts

-l, --lines

print the newline counts

-L, --max-line-length

print the length of the longest line

-w, --words

print the word counts

Eg: wc eleiss

```
 3   30   223   eleiss
```

cmp: Compare two files

Syntax: cmp [-l | -s] file1 file2 [skip1 [skip2]]

Description: cmp is used to compare two files byte by byte. If a difference is found, it reports the byte and line number where the first difference is found. If no differences are found, by default, cmp returns no output.

Options:

-l Print the byte number (decimal) and the differing byte values (octal) for each difference.

-s Print nothing for differing files; return exit status only.

Exit Status

0 The files are identical.

1 The files are different; this includes the case where one file is identical to the first part of the other. In the latter case, if the -s option has not been specified, cmp writes to standard output that EOF was reached in the shorter file (before any differences were found).

>1 An error occurred.

Eg: $cmp file1 file2

file1:this is file1, file2:this is file 1

$echo $?

0

comm: Compare two sorted files line by line

Syntax: comm [OPTION]... FILE1 FILE2

Description: Compare sorted files FILE1 and FILE2 line by line. With no options, produce three-column output. Column one contains lines unique to FILE1, column two contains lines unique to FILE2, and column three contains lines common to both files.

Options:

-1 suppress column 1 (lines unique to FILE1)

-2 suppress column 2 (lines unique to FILE2)

-3 suppress column 3 (lines that appear in both files)

Eg:
```
cat team-A
arpit
jimmy
Cat team_B
digvijay
hardik

$ comm team_A team_B
arpit                        digvijay
jimmy                        hardik
```

diff: Find differences between two files

Syntax: diff [options] from-file to-file

Options:

-a Treat all files as text and compare them line-by-line, even if they do not seem to be text.

-b Ignore changes in amount of white space.

-B Ignore changes that just insert or delete blank lines.

-i Ignore changes in case; consider upper- and lower-case letters equivalent.

-r When comparing directories, recursively compare any subdirectories found.

Eg: diff source.txt dest.txt

```
 1,10c1,5
     <hello
     -------
     >world
```

SECURITY BY FILE PERMISSIONS:

Every file in UNIX has the following attributes –

- **Owner permissions**: The owner's permissions determine what actions the owner of the file can perform on the file.
- **Group permissions**: The group's permissions determine what actions a user, who is a member of the group that a file belongs to, can perform on the file.
- **Other (world) permissions**: The permissions for others indicate what action all other users can perform on the file.

To access a file or directory in UNIX every type of user has some modes of conditions those are:

1. READ(r) - Grants the capability to read ie. view the contents of the file.
2. EXECUTE (x) - Grants the capability to modify, or remove the content of the file.
3. WRITE (w) - User with execute permissions can run a file as a program.

- The first three characters (2-4) represent the permissions for the file's owner. For example -rwxr-x**r--** represents that owner has read (r), write (w) and execute (x) permission.

- The second group of three characters (5-7) consists of the permissions for the group to which the file belongs. For example -rwxr-x**r--**represents that group has read (r) and execute (x) permission but no write permission.
- The last group of three characters (8-10) represents the permissions for everyone else. For example -rwxr-x**r--** represents that other world has read (r) only permission.

Viewing Permissions in the File Listing:

A quick and easy way to list a file's permissions are with the long listing (**-l**) option of the **ls** command. For example, to view the permissions of **file.txt**, you could use the command:

```
ls -l file.txt
```

```
OUTPUT::        -rwxrw-r-- 1   hope   hopestaff  123   Feb 03 15:36   file.txt
```

Here first column represents different access mode i.e. permission associated with a file or directory. Here's what each part of this information means:

-	The first character represents the file type: "**-**" for a regular file, "**d**" for a directory, "**l**" for a symbolic link.
rwx	The next three characters represent the permissions for the file's owner: in this case, the owner may **r**ead from, **w**rite to, and/or e**x**ecute the file.
rw-	The next three characters represent the permissions for members of the group that the file belongs to. In this case, any member of the file's owning group may **r**ead from or **w**rite to the file. The final dash is a placeholder; group members do not have permission to execute this file.
r--	The permissions for "others" (everyone else). Others may only **r**ead this file.
1	The number of hard links to this file.
hope	The file's owner.
hopestaff	The group to whom the file belongs.
123	The size of the file in blocks.
Feb 03 15:36	The file's mtime (date and time when the file was last modified).
file.txt	The name of the file.

Changing Permissions:

To change file or directory permissions, you use the **chmod** (change mode) command. There are two ways to use chmod: symbolic mode and absolute mode. In general, **chmod** commands take the form: chmod *options permissions filename*

chmod syntax:

chmod [option] … MODE [MODE] … FILE …

chmod [option] … OCTAL MODE FILE….

chmod [option] .. reference =RFILE FILE…

If no *options* are specified, **chmod** modifies the permissions of the file specified by *filename* to the permissions specified by *permissions*.

There are two ways to represent these permissions: with symbols (alphanumeric characters), or with octal numbers (the digits **0** through **7**).

Let's say you are the owner of a file named **myfile**, and you want to set its permissions so that:

1. the **u**ser can **r**ead, **w**rite, and e**x**ecute it;
2. members of your **g**roup can **r**ead and e**x**ecute it; and
3. **o**thers may only **r**ead it.

This command will do the trick:

chmod u=rwx,g=rx,o=r myfile

This is an example of using symbolic permissions notation. The letters **u**, **g**, and **o** stand for "**user**", "**group**", and "**other**". The equals sign ("**=**") means "set the permissions exactly like this," and the letters "**r**", "**w**", and "**x**" stand for "read", "write", and "execute", respectively. Here is the equivalent command using octal permissions notation:

chmod 754 myfile

Here the digits **7**, **5**, and **4** each individually represent the permissions for the user, group, and others, in that order. Each digit is a combination of the numbers **4**, **2**, **1**, and**0**:

- **4** stands for "read",
- **2** stands for "write",

- **1** stands for "execute", and
- **0** stands for "no permission."

So **7** is the combination of permissions **4**+**2**+**1** (read, write, and execute), **5** is **4**+**0**+**1**(read, no write, and execute), and **4** is **4**+**0**+**0** (read, no write, and no execute).

Using chmod in Symbolic Mode

The easiest way for a beginner to modify file or directory permissions is to use the symbolic mode. With symbolic permissions you can add, delete, or specify the permission set you want by using the operators in the following table.

Chmod operator	Description
+	Adds the designated permission(s) to a file or directory.
-	Removes the designated permission(s) from a file or directory.
=	Sets the designated permission(s).

Here's an example using testfile. Running ls -1 on testfile shows that the file's permissions are as follows −

```
$ls -l testfile
-rwxrwxr--  1 amrood   users 1024  Nov 2 00:10  testfile
```

Then each example chmod command from the preceding table is run on testfile, followed by ls -l so you can see the permission changes −

$chmod o+wx testfile

```
$ls -l testfile
-rwxrwxrwx     1 amrood   users 1024  Nov 2 00:10  testfile
```

$chmod u-x testfile

```
$ls -l testfile
-rw-rwxrwx  1 amrood   users 1024  Nov 2 00:10  testfile
```

```
$chmod g=rx testfile
```

```
$ls -l testfile
```

```
-rw-r-xrwx  1 amrood   users 1024  Nov 2 00:10  testfile
```

Here's how you could combine these commands on a single line:

```
$chmod o+wx,u-x,g=rx testfile
```

```
$ls -l testfile
```

```
-rw-r-xrwx  1 amrood   users 1024  Nov 2 00:10  testfile
```

Changing Owners and Groups

While creating an account on Unix, it assigns a owner ID and a group ID to each user. All the permissions mentioned above are also assigned based on Owner and Groups. Two commands are available to change the owner and the group of files:

- **chown** − The chown command stands for "change owner" and is used to change the owner of a file.
- **chgrp** − The chgrp command stands for "change group" and is used to change the group of a file.

Changing Ownership:

The chown command changes the ownership of a file. The basic syntax is as follows −

```
$ chown user filelist
```

The value of user can be either the name of a user on the system or the user id (uid) of a user on the system.

Following example −

```
$ chown amrood testfile
```

Changes the owner of the given file to the user **amrood**

NOTE: The super user, root, has the unrestricted capability to change the ownership of a any file but normal users can change only the owner of files they own.

Changing Group Ownership:

The chrgp command changes the group ownership of a file. The basic syntax is as follows −

```
$ chgrp group filelist
```

The value of group can be the name of a group on the system or the group ID (GID) of a group on the system.

Following example −

```
$ chgrp special testfile
$
```

Changes the group of the given file to **special** group

PROCESS UTILITIES:

ps: Report a snapshot of the current processes

SYNOPSIS: ps [*options*]

DESCRIPTION

ps displays information about a selection of the active processes. Ps reports the statuses of current processes in a system.

Options:-

-d:- Select all processes except session leaders.

-A:- Select all processes. Identical to **-e**.

-N:- Select all processes except those that fulfill the specified conditions. (negates the selection) Identical to **--deselect**.

KILL:-

Name:-kill - terminate a process

Synopsis:- **kill** [**-s** *signal*|**-p**][**--**] *pid*...
kill -l [*signal*]

Description:-The command **kill** sends the specified signal to the specified process or process group. If no signal is specified, the TERM signal is sent. The TERM signal will kill processes which do not catch this signal. For other processes, it may be necessary to use the KILL (9) signal, since this signal cannot be caught.

Options:-

-p:-Specify that **kill** should only print the process id (pid) of the named processes, and not send any signals.

-l:-Print a list of signal names. These are found in */usr/include/linux/signal.h*

-a:-Do not restrict the command name-to-pid conversion to processes with the same uid as the present process.

DISK UTILITIES:

df:

It displays the amount of disk space available on the file system containing each file name argument. With no file name, available space on all currently mounted file systems is shown. More on using df to display the amount of disk space available.

df [-k] [-P|-t] [-del] [file...]

Options:-

-k :Use 1024-byte units, instead of the default 512-byte units, when writing space figures.

-P :Use a standard, portable, output format

-t :If XSI compliant, show allocated space as well

-h :Display in KB, MB, or GB

du:

du (abbreviated from disk usage) is a standard Unix program used to estimate file space usage—space used under a particular directory or files on a file system.

Options:-

-a, In addition to the default output, include information for each non-directory entry

-d #, the depth at which summing should occur. -d 0 sums at the current level, -d 1 sums at the subdirectory, -d 2 at sub-subdirectories, etc.

-H, calculate disk usage for link references specified on the command line

-k, show sizes as multiples of 1024 bytes, not 512-byte

-s, report only the sum of the usage in the current directory, not for each file

Example:

Sum of directories (-s) in kilobytes (-k):

```
$du -sk *
152304  directoryOne
1856548 directoryTwo
```

Network Commands:-

Def: A network consists of several computers connected together. The network can be as simple as a few computers connected in your home or office, or as complicated as a large university network or even the entire Internet. The network commands also include information on tools for network configuration, file transfer and working with remote machines.

Networking Commands Example in Unix and Linux:-

• finding host/domain name and IP address - **hostname**
• test network connection – **ping**
• getting network configuration – **ifconfig**
• Network connections, routing tables, interface statistics – **netstat**
• query DNS lookup name – **nslookup**
• communicate with another hostname – **telnet**
• outing steps that packets take to get to network host – **traceroute**
• view user information – **finger**
• checking status of destination host - **telnet**

hostname

hostname with no options displays the machine's hostname.
hostname**–d** displays the domain name the machine belongs to.
hostname –**f** displays the fully qualified host and domain name.
hostname **–i** displays the IP address for the current machine.

ping

It sends packets of information to the user-defined source. If the packets are received, the destination device sends packets back. Ping can be used for two purposes

1. To ensure that a network connection can be established

2. Timing information as to the speed of the connection.
If you **do ping www.yahoo.com** it will display its IP address. Use ctrl+C to stop the test.

ifconfig: View network configuration, it displays the current network adapter configuration. It is handy to determine if you are getting transmit (TX) or receive (RX) errors.

Netstat:

Most useful and very versatile for finding a connection to and from the host

netstat -g can find out all the multicast groups (network) subscribed by this host by issuing.
netstat -nap | grep port will display process id of application which is using that port.
netstat -a or netstat –all will display all connections including TCP and UDP .
netstat --tcp or netstat –t will display only TCP connection.
netstat --udp or netstat –u will display only UDP connection.
netstat -g will display all multicast network subscribed by this host.

nslookup:

If you know the IP address it will display hostname. To find all the IP addresses for a given domain name, the command nslookup is used. You must have a connection to the internet for this utility to be useful, e.g.

$ **nslookup blogger.com**

You can also use the **nslookup** to convert hostname to IP Address and from IP Address from the hostname.

traceroute:

A handy utility to view the number of hops and response time to get to a remote system or website is traceroute. Again you need an internet connection to make use of this tool.

Finger:

View user information, displays a user's login name, real name, terminal name and write status. This is pretty old Unix command and rarely used nowadays.

telnet:

Connects destination host via the telnet protocol, if telnet connection establishes on any port means connectivity between two hosts is working fine.

$ telnet hostname port

…will telnet hostname with the port specified. Normally it is used to see whether the host is alive and the network connection is fine or not.