INTRODUCTION TO UNIX FILE SYSTEM:

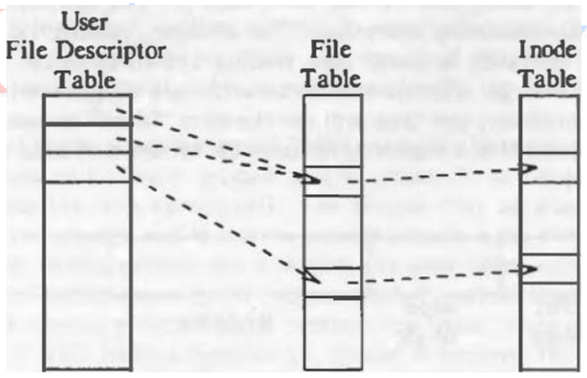
The internal representation of a file is given by an inode, which contains a description of the disk layout of the file data and other information such as the file owner, access permissions, and access times.

The term inode is a contraction of the term index node and is commonly used in literature on the UNIX system. Every file has one inode, but it may have several names, all of which map into the inode.

Each name is called a link. When a process refers to a file by name, the kernel parses the file name one component at a time, checks that the process has permission to search the directories in the path, and eventually retrieves the inode for the file.

The kernel contains two other data structures, the file table and the user file descriptor table. The file table is a global kernel structure, but the user file descriptor table is allocated per process.

When a process opens or creates a file, the kernel allocates an entry from each table, corresponding to the file's inode. Entries in the three structures as shown in figure below – user file descriptor table, file table, and inode table – maintain the state of the file and the user's access to it.



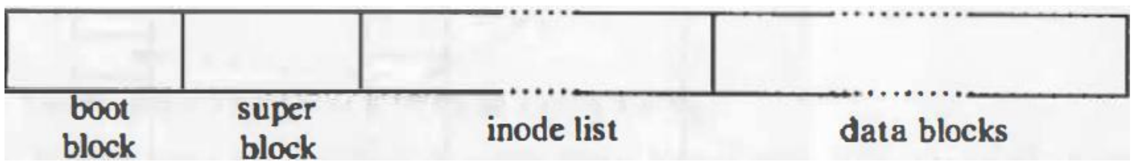
The file table keeps track of the byte offset in the file where the user's next read or write will start, and the access rights allowed to the opening process. The user file descriptor table identifies all open files for a process.

The kernel returns a file descriptor for the open and creat system calls, which is an index into the user file descriptor table. When executing read and write system calls, the kernel uses the file descriptor to access the user file descriptor table, follows pointers to the file table and inode table entries and, from the inode, finds the data in the file.

A file system consists of a sequence of logical blocks, each containing 512, 1024, 2048, or any convenient multiple of 512 bytes, depending on the system implementation. The size of a logical block is homogeneous within a file system but may vary between different file systems in a system configuration.

Using large logical blocks increases the effective data transfer rate between disk and memory, because the kernel can transfer more data per disk operation and therefore make fewer time-consuming operations.

A file system has the following structure



The ***boot block*** occupies the beginning of a file system, typically the first sector, and may contain the bootstrap code that is read into the machine to boot, or initialize, the operating system. Although only one boot block is needed to boot the system, every file system has a (possibly empty) boot block.

The ***super block*** describes the state of a file system - how large it is, how many files it can store, where to find free space on the file system, and other information.

The ***inode list*** is a list of inode's that follows the super block in the file system. Administrators specify the size of the inode list when configuring a file system. The kernel references inode's by index into the inode list.

The ***data blocks*** start at the end of the inode list and contain file data and administrative data. An allocated data block can belong to one and only one file in the file system.

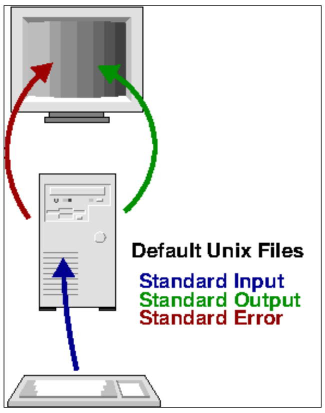
JKDirectory!

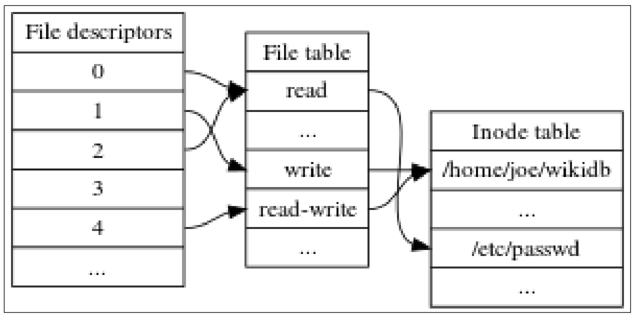
FILE DESCRIPTORS:

In UNIX and related computer operating systems, a file descriptor (FD, less frequently fildes) is an abstract indicator (handle) used to access a file or other input/output resource, such as a pipe or network socket.

A file descriptor is a non-negative integer, although, it is usually represented in C programming language as the type int, negative values are being reserved to indicate "no value" or an error condition.

Integer value	Name	<unistd.h> symbolic constant	<stdio.h> file stream
0	Standard input	STDIN_FILENO	stdin
1	Standard output	STDOUT_FILENO	stdout
2	Standard error	STDERR_FILENO	stderr





In the traditional implementation of UNIX, file descriptors index into a per-process file descriptor table maintained by the kernel, that in turn indexes into a system-wide table of files opened by all processes, called the file table.

This table records the mode with which the file (or other resource) has been opened: for reading, writing, appending, reading and writing, and possibly other modes. It also indexes into a third table called the inode table that describes the actual underlying files.

To perform input or output, the process passes the file descriptor to the kernel through a system call, and the kernel will access the file on behalf of the process. The process does not have direct access to the file or inode tables.

JKDirectory!

On Linux, the set of file descriptors open in a process can be accessed under the path /proc/PID/fd/, where PID is the process identifier.

In Unix-like systems, file descriptors can refer to any UNIX file type named in a file system. As well as regular files, this includes directories, block and character devices (also called "special files"), UNIX domain sockets, and named pipes. File descriptors can also refer to other objects that do not normally exist in the file system, such as anonymous pipes and network sockets.

INODE REPRESENTATION:

Inodes exist in a static form on disk, and the kernel reads them into an in-core inode to manipulate them. Disk inodes consist of the following fields:

1. **File owner identifier**. Ownership is divided between an individual owner and a "group" owner and defines the set of users who have access rights to a file. The super user has access rights to all files in the system.
2. **File type**. Files may be of type regular, directory, character or block special, or FIFO (pipes).
3. **File access permissions**. The system protects files according to three classes: the owner and the group owner of the file, and other users; each class has access rights to read, write and execute the file, which can be set individually.

4. **File access times**, giving the time the file was last modified, when it was last accessed, and when the inode was last modified.
5. **Number of links to the file**, representing the number of names the file has in the directory hierarchy.
6. **Table of contents for the disk addresses of data in a file**. Although users treat the data in a file as a logical stream of bytes, the kernel saves the data in discontiguous disk blocks. The inode identifies the disk blocks that contain the file's data.
7. **File size**. Data in a file is addressable by the number of bytes from the beginning of the file, starting from byte offset 0, and the file size is l greater than the highest byte offset of data in the file.

For example, if a user creates a file and writes only 1 byte of data at byte offset 1000 in the file, the size of the file is 1001 bytes.

owner mjb
group os
type regular file
perms rwxr-xr-x
accessed Oct 23 1984 1:45 P.M.
modified Oct 22 1984 10:30 AM.
inode Oct 23 1984 1:30 P.M.
size 6030 bytes
disk addresses

In a Unix-style file system, the inode is a data structure used to represent a file system object, which can be one of various things including a file or a directory. Each inode stores the attributes and disk block location(s) of the file system object's data. The File-attributes consist of the following:

Field No.	Stat Name	UNIX	Win98/NT	MacOS
1	st_dev	Device number of file system	Drive number	vRefNum
2	st_ino	Inode number	Always 0	fileID/dirID
3	st_mode	File mode	File mode	777 dirs/apps; 666 docs; 444 locked docs
4	st_nlink	Number of links to the file	Number of link (only on NTFS)	Always 1
5	st_uid	Owner ID	Always 0	Always 0
6	st_gid	Group ID	Always 0	Always 0
7	st_rdev	Device ID for special files	Drive No.	Always 0

8	st_size	File size in bytes	File size in bytes	Data fork file size in bytes
9	st_blksize	Preferred block size	Always 0	Preferred block size
10	st_blocks	Number of blocks allocated	Always 0	Number of blocks allocated
11	st_atime	Last access time since epoch	Last access time since epoch	Last access time -66 years
12	st_mtime	Last modify time since epoch	Last modify time since epoch	Last access time -66 years
13	st_ctime	Inode change time since epoch	File create time since epoch	File create time -66 years

A file's inode number can be found using the ls -i command. The ls -i command prints the i-node number in the first column of the report.

SUPER BLOCK:

The superblock is part of various file systems of the operating system UNIX and its derivatives. It typically includes the following management information of the file system:

- Size of the file system
- The number of free blocks in the file system
- A list of free blocks available on the file system - Pointer to free list
- Pointer to the first free block in the free list
- Size of the inode list
- Number of free inodes
- Pointer to list of free inodes
- Pointer to the next free inode in the list of free inodes
- Barrier-fields for list of free blocks / inodes (eg for bad blocks)
- Display whether superblock is changed

JKDirectory!

The kernel periodically writes the super block to disk if it had been modified so that it is consistent with the data in the file system.

SYSTEM CALLS AND LIBRARY FUNCTIONS:

A system call is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on. This may include hardware-related services for example, accessing a hard disk drive, creation and execution of new processes, and communication with integral kernel services such as process scheduling. System calls provide an essential interface between a process and the operating system.

In most systems, system calls are possible to be made only from user space processes, while in some systems, OS/360 and successors for example, privileged system code also issues system calls.

System calls can be roughly grouped into five major categories:

Process Control:

- ✓ load
- ✓ execute
- ✓ end, abort
- ✓ create process
- ✓ terminate process
- ✓ get/set process attributes
- ✓ wait for time, wait event, signal event
- ✓ allocate, free memory

File management:

- ✓ create file, delete file
- ✓ open, close
- ✓ read, write, reposition
- ✓ get/set file attributes

JKDirectory!

Device Management:

- ✓ request device, release device
- ✓ read, write, reposition
- ✓ get/set device attributes
- ✓ logically attach or detach devices

Information Maintenance:

- ✓ get/set time or date
- ✓ get/set system data
- ✓ get/set process, file, or device attributes

Communication:

- ✓ create, delete communication connection
- ✓ send, receive messages
- ✓ transfer status information
- ✓ attach or detach remote devices

Libraries are a set of common functions that can be used by application programs. These common functions are termed as ‘Library functions’. In most cases, the library functions adhere to certain standards. The ANSI C standard library is a good example. Hence, they are portable, which is a clear advantage over system calls.

In contrast to system call, the library functions are linked to application programs. Library functions always execute in user space (also termed as ‘user mode‘). Hence, they cannot interact directly with the hardware.

Library functions in-turn may utilize system calls for performing certain tasks which can only be carried out only in ‘kernel mode’.

The time consumed to execute a library function which does not in-turn invoke a system call is always lesser than the time consumed to execute a system call because the application program need not context switch between user space and kernel space.

LOW LEVEL FILE ACCESS:

OPEN:

The open system call is the first step a process must take to access the data in a file. The syntax for the open system call is

JKDirectory!

fd = open(pathname, flags, modes);

Where pathname is a file name, flags indicate the type of open (such as for reading or writing), and modes give the file permissions if the file is being created.

The open system call returns an integer called the user file descriptor. Other file operations, such as reading, writing, seeking, duplicating the file descriptor, setting file I/O parameters, determining file status, and closing the file, use the file descriptor that the open system call returns.

READ:

The syntax of the read system call is:

number = read(fd, buffer, count);

Where fd is the file descriptor returned by open, buffer is the address of a data structure in the user process that will contain the read data on successful completion of the call, count is the number of bytes the user wants to read, and number is the number of bytes actually read.

WRITE:

The syntax for the write system call is:

number = write(fd, buffer, count);

Where the meaning of the variables fd, buffer, count, and number are the same as they are for the read system call. The algorithm for writing a regular file is similar to that for reading a regular file.

However, if the file does not contain a block that corresponds to the byte offset to be written, the kernel allocates a new block using algorithm alloc and assigns the block number to the correct position in the inode's table of contents.

If the byte offset is that of an indirect block, the kernel may have to allocate several blocks for use as indirect blocks and data blocks.

CLOSE:

A process closes an open file when it no longer wants to access it. The syntax for the close system call is:

close(fd);

Where fd is the file descriptor for the open file. The kernel does the close operation by manipulating the file descriptor and the corresponding file table and inode table entries. If the reference count of the file table entry is greater than 1 because of dup or fork calls, then other user file descriptors reference the file table entry, as will be seen; the kernel decrements the count and the close completes.

JKDirectory!

When the close system call completes, the user file descriptor table entry is empty. Attempts by the process to use that file descriptor result in an error until the file descriptor is reassigned as a result of another system call.

LSEEK:

The ordinary use of read and write system calls provides sequential access to a file, but processes can use the lseek system call to position the I/O and allow random access to a file.

The syntax for the system call is:

position = lseek(fd, offset, reference);

Where fd is the file descriptor identifying the file, offset is a byte offset, and reference indicates whether offset should be considered from the beginning of the file, from the current position of the read/write offset, or from the end of the file. The return value, position, is the byte offset where the next read or write will start.

STAT AND FSTAT:

The system calls stat and fstat allow processes to query the status of files, returning information such as the file type, file owner, access permissions, file size, number of links, inode number, and file access times. The syntax for the system calls is:

stat(pathname, statbuffer);

fstat(fd, statbuffer);

Where pathname is a file name, fd is a file descriptor returned by a previous open call, and statbuffer is the address of a data structure in the user process that will contain the status information of the file on completion of the call. The system calls simply write the fields of the inode into statbuffer.

LSTAT:

lstat is a system call that is used to determine information about a file based on its filename. lstat is exactly the same as the stat system call. The only difference between the two is when the filename refers to a link. When this is the case, lstat returns information about the link itself, whereas stat returns information about the actual file. The Syntax for the system call is:

lstat(pathname, lstatbuffer);

Where the pathname is the path of the file that is being inquired and lstatbuffer is a structure where data about the file will be stored. The lstat system call returns a positive value on success and negative value on failure.

IOCTL:

JKDirectory!

The ioctl system call is a generalization of the terminal-specific ***stty*** (set terminal settings) and ***gtty*** (get terminal settings) system calls available in earlier versions of the UNIX system. It provides a general, catch-all entry point for device specific commands, allowing a process to set hardware options associated with a device and software options associated with the driver.

The specific actions specified by the ioctl call vary per device and are defined by the device driver. Programs that use ioctl must know what type of file they are dealing with, because they are device specific. This is an exception to the general rule that the system does not differentiate between different file types. The syntax of the system call is

ioctl (fd, command, arg);

Where fd is the file descriptor returned by a prior open system call, command is a request of the driver to do a particular action, and arg is a parameter (possibly a pointer to a structure) for the command.

UMASK:

The umask command is used to set up the file creation mask. The umask value is used every time a file is created as a security measure to make sure that the processes you run cannot create new files with any more access permission than you want to give. This is done by having the permission bits in the umask value reset in the mode bits of any file that gets created on your behalf.

The umask command is implemented internally by a call to an underlying system call of the same name. The syntax of umask system call is:

umask(mask);

A call to the umask() system call sets the file creation mask to the specified mask value. Only the permission bits in mask are saved, the values of the other bits are ignored. The return value from umask() is just the previous value of the file creation mask.

DUP:

The dup system call copies a file descriptor into the first free slot of the user file descriptor table, returning the new file descriptor to the user. It works for all file types. The syntax of the system call is

newfd = dup(fd);

Where fd is the file descriptor being duped and newfd is the new file descriptor that references the file. Because dup duplicates the file descriptor, it increments the count of the corresponding file table entry, which now has one more file descriptor entry that points to it.

DUP2:

JKDirectory!

The dup2 is a system call similar to dup in that it duplicates one file descriptor, making them aliases, and then deleting the old file descriptor. This becomes very useful when attempting to redirect output, as it automatically takes care of closing the old file descriptor, performing the redirection in one elegant command.

For example, if you wanted to redirect standard output to a file, then you would simply call dup2, providing the open file descriptor for the file as the first command and 1 (standard output) as the second command. The syntax for the dup2 system call is:

dup2(oldfd, newfd);

The oldfd is the source file descriptor that remains open after the call to dup2 and the newfd is the destination file descriptor that will points to the same file as oldfd after this call returns. It returns the value of the newfd up on success. A negative value will be returned when error occurs.

THE STANDARD I/O LIBRARY:

FOPEN:

The fopen() function is used to open a file and associates an I/O stream with it. This function takes two arguments. The first argument is a pointer to a string containing name of the file to be opened while the second argument is the mode in which the file is to be opened. The mode can be:

- ✓ ‘r’ : Open text file for reading. The stream is positioned at the beginning of the file.

- ✓ 'r+' : Open for reading and writing. The stream is positioned at the beginning of the file.
- ✓ 'w' : Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.
- ✓ 'w+' : Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.
- ✓ 'a' : Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.
- ✓ 'a+' : Open for reading and appending (writing at end of file). The file is created if it does not exist. The initial file position for reading is at the beginning of the file, but output is always appended to the end of the file.

The fopen() function returns a FILE stream pointer on success while it returns NULL in case of a failure.

Syntax:

FILE *fopen(const char *path, const char *mode);

FREAD() AND FWRITE():

JKDirectory!

The functions fread/fwrite are used for reading/writing data from/to the file opened by fopen function. These functions accept three arguments. The first argument is a pointer to buffer used for reading/writing the data. The data read/written is in the form of 'nmemb' elements each 'size' bytes long.

In case of success, fread/fwrite return the number of bytes actually read/written from/to the stream opened by fopen function. In case of failure, a lesser number of byes (then requested to read/write) is returned.

The syntax for fread and fwrite are:

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);

FCLOSE():

The fclose() function first flushes the stream opened by fopen() and then closes the underlying descriptor. Upon successful completion this function returns 0 else end of file (eof) is returned. In case of failure, if the stream is accessed further then the behavior remains undefined.

The syntax for fclose is:

int fclose(FILE *fp);

FSEEK():

The fseek() function is used to set the file position indicator for the stream to a new position. This function accepts three arguments. The first argument is the FILE stream pointer returned by the fopen() function. The second argument 'offset' tells the amount of bytes to seek.

The third argument 'whence' tells from where the seek of 'offset' number of bytes is to be done. The available values for whence are SEEK_SET, SEEK_CUR, or SEEK_END. These three values (in order) depict the start of the file, the current position and the end of the file. Upon success, this function returns 0, otherwise it returns -1.

The syntax for fseek is:

int fseek(FILE *stream, long offset, int whence);

FFLUSH():

The function **fflush**() forces a write of all buffered data for the given output or update *stream* via the stream's underlying write function. The open status of the stream is unaffected. If the *stream* argument is NULL, **fflush**() flushes *all* open output streams.

Syntax:

fflush(*FILE *stream*);

JKDirectory!

Upon successful completion 0 is returned. Otherwise, EOF is returned and the global variable *errno* is set to indicate the error.

FGETC():

fgetc() reads the next character from *stream* and returns it as an *unsigned char* cast to an *int*, or **EOF** on end of file or error.

Syntax:

int fgetc(FILE ***stream*);**

fgetc() return the character read as an *unsigned char* cast to an *int* or **EOF** on end of file or error.

FGETS():

fgets() reads in at most one less than *size* characters from *stream* and stores them into the buffer pointed to by *s*. Reading stops after an **EOF** or a newline. If a newline is read, it is stored into the buffer. A terminating null byte ('\0') is stored after the last character in the buffer.

Syntax:

char *fgets(char ***s*, int** *size***, FILE ***stream***);**

fgets() returns *s* on success, and NULL on error or when end of file occurs while no characters have been read.

FPUTC():

fputc() writes the character *c*, cast to an *unsigned char*, to *stream*.

Syntax:

int fputc(int *c*, **FILE** ****stream*);**

fputc() return the character written as an *unsigned char* cast to an *int* or **EOF** on error.

FPUTS():

fputs() writes the string *s* to *stream*, without its terminating null byte ('\0'). **fputs**() return a nonnegative number on success, or **EOF** on error.

Syntax:

int fputs(const char **s*, **FILE** **stream***);**

FORMATTED INPUT AND OUTPUT:

JKDirectory!

PRINTF():

printf inserts arguments into a user-defined string of text, creating formatted output. printf prints a *formatted string* to the standard output. Upon successful completion, it returns the number of bytes transmitted.

Syntax:

int printf(const char ****format*, arguments ...);**

Format

The *FORMAT* string contains three types of objects:

- ✓ Ordinary characters, which are copied exactly to the output.
- ✓ Interpreted character sequences, which are escaped with a backslash ("\").
- ✓ Conversion specifications, which define the way in which ARGUMENTs will be expressed as part of the output.

Example: printf "My name is \"%s\".\nIt's a pleasure to meet you." "John"

Output: My name is "John". It's a pleasure to meet you.

The power of **printf** lies in the fact that for any given *FORMAT* string, the *ARGUMENT*s can be changed to affect the output. For example, the output of the command in the above example can be altered just by changing the argument, "John". If used in a script, this argument can be set to a variable. For instance, the command

```
printf "Hi, I'm %s.\n" $LOGNAME
```

...will insert the value of the environment variable **$LOGNAME**, which is the username of whoever ran the command.

The conversion characters themselves, which tell **printf** what kind of argument to expect, are as follows:

conversion character	argument type
d, **i**	An integer, expressed as a decimal number.
O	An integer, expressed as an unsigned octal number.
x, **X**	An integer, expressed as an unsigned hexadecimal number
U	An integer, expressed as an unsigned decimal number.
C	An integer, expressed as a character. The integer corresponds to the character's ASCII code.
S	A string.
F	A floating-point number, with a default precision of 6.
e, **E**	A floating-point number expressed in scientific notation, with a default precision of 6.
P	A memory address pointer.
%	No conversion; a literal percent sign ("**%**") is printed instead.

JKDirectory!

The following character sequences are interpreted as special characters by **printf**:

\"	prints a double-quote (**"**)
\	prints a backslash (****)
\a	issues an alert (plays a bell)
\b	prints a backspace
\c	instructs **printf** to produce no further output
\e	prints an escape character (ASCII code 27)
\n	Prints a new line
\t	Prints a horizontal tab
\v	Prints a vertical tab

FPRINTF():

The *fprintf()* function places output on the named output *stream*. Upon successful completion, it returns the number of bytes transmitted.

Syntax:

int fprintf(FILE ****stream*, const char ****format***, ...);**

Example: To print a date and time in the form "Sunday, July 3, 10:02", where weekday and month are pointers to strings:

```
#include <stdio.h>
fprintf(stdout, "%s, %s %d, %.2d:%.2d\n", weekday, month, day, hour, min);
```

SPRINTF():

The *sprintf()* function places output followed by the null byte, '\0', in consecutive bytes starting at **s*; it is the user's responsibility to ensure that enough space is available. Upon successful completion, it returns the number of bytes transmitted excluding the terminating null.

Syntax:

int sprintf(char *s, const char **format*, ...);

Upon successful completion, these functions return the number of bytes transmitted excluding the terminating null in the case of *sprintf()* or a negative value if an output error was encountered.

Example:

```
#include <stdio.h>
#include <math.h>
int main()
{
  char str[80];
  sprintf(str, "Value of Pi = %f", M_PI);
  puts(str);
  return(0);
}
```

Let us compile and run the above program, this will produce the following result −

```
Value of Pi = 3.141593
```

JKDirectory!

SCANF():

The **scanf**() function reads input from the standard input stream *stdin.* The **scanf**() family of functions scans input according to a format which may contain conversion specifiers; results from such conversions, if any, are stored through the *pointer* arguments. The following conversions are available:

% Matches a literal `%'. That is, `%%' in the format string matches a single input `%' character. No conversion is done, and assignment does not occur.

d Matches an optionally signed decimal integer; the next pointer must be a pointer to *int*.

o Matches an octal integer; the next pointer must be a pointer to *unsigned int*.

f Matches an optionally signed floating-point number; the next pointer must be a pointer to *float*.

s Matches a sequence of non-white-space characters.

Syntax:

int scanf(const char ****format*, ...);**

scanf reads characters from the standard input, interprets them according to the specification in format, and stores the results through the remaining arguments.

scanf stops when it exhausts its format string, or when some input fails to match the control specification. It returns as its value the number of successfully matched and assigned input items. This can be used to decide how many items were found.

On the end of file, EOF is returned; note that this is different from 0, which means that the next input character does not match the first specification in the format string. The next call to scanf resumes searching immediately after the last character already converted.

The *format* string consists of a sequence of *directives* which describe how to process the sequence of input characters.

If processing of a directive fails, no further input is read, and **scanf**() returns. A "failure" can be either of the following: *input failure*, meaning that input characters were unavailable, or *matching failure*, meaning that the input was inappropriate.

JKDirectory!

A directive is one of the following:

- ✓ A sequence of white-space characters (space, tab, newline).
- ✓ An ordinary character (i.e., one other than white space or '%')
- ✓ A conversion specification, which commences with a '%' (percent) character

Each *conversion specification* in *format* begins with either the character '%' or the character sequence "**%*n*$**" followed by:

An optional '*' assignment-suppression character: **scanf**() reads input as directed by the conversion specification, but discards the input.

An optional 'm' character: **scanf**() allocates a buffer of sufficient size, and assigns the address of this buffer to the corresponding *pointer* argument, which should be a pointer to a *char ** variable.

An optional decimal integer: It specifies the *maximum field width.* Reading of characters stops either when this maximum is reached or when a nonmatching character is found, whichever happens first.

Example:

Suppose we want to read input lines that contain dates of the form 25 Dec 1988

The scanf statement is

```
int day, year;
char monthname[20];
scanf("%d %s %d", &day, monthname, &year);
```

Note: No & is used with monthname, since an array name is a pointer.

FSCANF():

The fscanf function is just like the scanf function, except that the first argument of fscanf specifies a stream from which to read, whereas scanf can only read from standard input.

Syntax:

int fscanf(*FILE *stream*, *const char *format*, ...);

Here is a code example that generates a text file containing five numbers with fprintf, then reads them back in with fscanf.

JkDirectory!

```
#include <stdio.h>
#include <errno.h>

int main()
{
  float f1, f2;
  int i1, i2;
  FILE *my_stream;
  char my_filename[] = "snazzyjazz.txt";
  my_stream = fopen (my_filename, "w");
  fprintf (my_stream, "%f %f %#d %#d", 23.5, -12e6, 100, 5);
  /* Close stream; */
  fclose (my_stream);
  my_stream = fopen (my_filename, "r");
  fscanf (my_stream, "%f %f %i %i", &f1, &f2, &i1, &i2);
  /* Close stream; */
  fclose (my_stream);
  printf ("Float 1 = %f\n", f1);
  printf ("Float 2 = %f\n", f2);
  printf ("Integer 1 = %d\n", i1);
  printf ("Integer 2 = %d\n", i2);
  return 0;
}
```

This code example prints the following output on the screen:

```
Float 1 = 23.500000
Float 2 = -12000000.000000
Integer 1 = 100
Integer 2 = 5
```

If you examine the text file snazzyjazz.txt, you will see it contains the following text:

```
23.500000 -12000000.000000 100 5
```

SSCANF():

sscanf() reads its input from the string instead of standard input. It scans the string according to the format in format and stores the resulting values through arg1, arg2, etc. These arguments must be pointers.

Syntax:

int sscanf(*const char *str*, *const char *format*, arg1, arg2, ...);

JKDirectory!

The format string usually contains conversion specifications, which are used to control conversion of input. The format string may contain:

- ✓ Blanks or tabs, which are not ignored.
- ✓ Ordinary characters (not %), which are expected to match the next non-white space character of the input stream.
- ✓ Conversion specifications, consisting of the character %, an optional assignment suppression character *, an optional number specifying a maximum field width, an optional h, l or L indicating the width of the target, and a conversion character.

The following example shows the usage of sscanf() function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
  int day, year;
  char weekday[20], month[20], dtm[100];
  strcpy( dtm, "Saturday March 25 1989" );
  sscanf( dtm, "%s %s %d  %d", weekday, month, &day, &year );
```

```
    printf("%s %d, %d = %s\n", month, day, year, weekday );
    return(0);
  }
```

Let us compile and run the above program that will produce the following result:

```
March 25, 1989 = Saturday
```

FILE AND DIRECTORY MAINTENANCE:

CHMOD:

chmod is used to change the permissions of files or directories. On Linux and other Unix-like operating systems, there is a set of rules for each file which defines who can access that file, and how they can access it. These rules are called file permissions or file modes. The command name chmod stands for "change mode", and it is used to define the way a file can be accessed.

In general, chmod commands take the form:

chmod options permissions filename

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

JKDirectory!

Permissions defines the permissions for the owner of the file (the "user"), members of the group who owns the file (the "group"), and anyone else ("others"). 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 user can read, write, and execute it;
2. Members of your group can read and execute it; and
3. Others may only read 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. The commas separate the different classes of permissions, and there are no spaces in between them.

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).

chmod syntax

chmod [OPTION]... MODE[,MODE]... FILE...

chmod [OPTION]... OCTAL-MODE FILE...

chmod [OPTION]... --reference=RFILE FILE...

CHOWN:

The chown command changes the owner and owning group of files. Chown changes the user and/or group ownership of each given file.

JKDirectory!

- ✓ If only an owner (a user name or numeric user ID) is given, that user is made the owner of each given file, and the files' group is not changed.
- ✓ If the owner is followed by a colon and a group name (or numeric group ID), with no spaces between them, the group ownership of the files is changed as well.
- ✓ If a colon but no group name follows the user name, that user is made the owner of the files and the group of the files is changed to that user's login group.
- ✓ If the colon and group are given, but the owner is omitted, only the group of the files is changed; in this case, chown performs the same function as chgrp.
- ✓ If only a colon is given, or if the entire operand is empty, neither the owner nor the group is changed.

chown syntax:

chown [OPTION]... [OWNER][:[GROUP]] FILE...

chown [OPTION]... --reference=RFILE FILE...

chown examples:

Set the owner of file file.txt to user chope: ***chown chope file.txt***

Recursively grant ownership of the directory /files/work, and all files and subdirectories, to user chope.

```
chown -R chope /files/work
```

CHGRP:

Changes group ownership of a file or files.

chgrp syntax:

chgrp [OPTION]... GROUP FILE...

chgrp [OPTION]... --reference=RFILE FILE...

Chgrp examples:

Change the owning group of the file file.txt to the group named hope.

```
chgrp hope file.txt
```

Change the owning group of /office/files, and all subdirectories, to the group staff.

```
chgrp -R staff /office/files
```

UNLINK:

The unlink command calls and directly interfaces with the unlink system function, which removes a specified file.

JKDirectory!

unlink syntax:

unlink FILE

unlink OPTION

unlink examples

```
unlink hope.txt
```

Removes the filename **hope.txt**, and if there is no other hard link to the file data, the file data itself is removed from the system.

LINK:

The **link** utility is a Unix command line program that creates a hard link from an existing directory entry to a new directory entry. Link command calls the link function to create a link to a file. It does not perform error checking before attempting to create the link.

It returns an exit status that indicates whether the link was created (0 if successful, >0 if an error occurred). Creating a link to a directory entry that is itself a directory requires elevated privileges.

The **link** command creates a hard link named *FILE2* which shares the same index node as the existing file *FILE1*. Since *FILE1* and *FILE2* share the same index node, they will point to the same data on the disk, and modifying one will be functionally the same as modifying the other.

This is distinct from creating a "soft" symbolic link to a file, which creates its own index node and therefore does not directly point to the same data. For example, a user cannot create a hard link which links to a directory; but this can be accomplished using a symbolic link.

link syntax:

link *FILE1 FILE2*
link *OPTION*

SYMLINK:

The **symlink** command lists, creates and removes Unix-style symbolic links (symlinks).

The symlink make command is equivalent to "ln -s" on a UNIX platform. Symlink can manipulate symlinks created by "ln -s" on UNIX and vice versa.

Note: The symlink commands are only available on Windows platforms. On other platforms, use the "ln -s" command

JKDirectory!

A symbolic link, also termed a soft link, is a special kind of file that points to another file, much like a shortcut in Windows. Unlike a hard link, a symbolic link does not contain the data in the target file.

It simply points to another entry somewhere in the file system. This difference gives symbolic links certain qualities that hard links do not have, such as the ability to link to directories, or to files on remote computers.

Also when you delete a target file, symbolic links to that file become unusable, whereas hard links preserve the contents of the file. To create a symbolic link in UNIX, at the Unix prompt, enter:

```
ln -s source_file myfile
```

Replace source_file with the name of the existing file for which you want to create the symbolic link (this file can be any existing file or directory across the file systems). Replace myfile with the name of the symbolic link.

The ln command then creates the symbolic link. After you've made the symbolic link, you can perform an operation on or execute myfile, just as you could with the source_file. You can use normal file management commands (e.g., cp, rm) on the symbolic link.

Note: If you delete the source file or move it to a different location, your symbolic file will not function properly. You should either delete or move it. If you try to use it for other purposes (e.g., if you try to edit or execute it), the system will send a "file nonexistent" message.

HARDLINK: A hard link is essentially a label or name assigned to a file. Conventionally, we think of a file as consisting of a set of information that has a single name. However, it is possible to create a number of different names that all refer to the same contents. Commands executed upon any of these different names will then operate upon the same file contents. To make a hard link to an existing file, enter: ***ln oldfile newlink***

MKDIR:

To create a subdirectory in the home directory of your UNIX account, use the mkdir command.

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

For example, to create a subdirectory called work, at the UNIX prompt from within your home directory, enter:

```
mkdir work
```

The mkdir command requires at least one argument and will take multiple arguments. To create three directories named work, programs, and reports, enter:

JKDirectory!

```
mkdir work programs reports
```

If you're currently in your home directory and you want to make a new directory in another directory, you don't need to use the cd command first. You may specify a pathname with the mkdir command, for example: ***mkdir bin/work***

The mkdir command in conjunction with the -p option creates the directories specified in a path. This is very useful when you want to make a directory structure that is several directories deep, and none of the subdirectories exist yet. Without the -p option, this command would have returned an error.

For example, the following command will create the work, work/programs, work/programs/reports, and work/programs/reports/completed subdirectories:

```
mkdir -p work/programs/reports/completed
```

Create the mydir directory, and set its permissions such that all users may read, write, and execute the contents.

```
mkdir -m a=rwx mydir
```

RMDIR:

The rmdir utility removes the directory entry specified by each directory argument, provided the directory is empty.

Arguments are processed in the order given. To remove both a parent directory and a subdirectory of that parent, the subdirectory must be specified first so the parent directory is empty when rmdir tries to remove it.

Syntax: **rmdir [-p] directory...**

-p – Each directory argument is treated as a pathname of which all components will be removed, if they are empty, starting with the last most component.

This command will remove a subdirectory. To remove a subdirectory named oldstuff, enter:

```
rmdir oldstuff
```

Note: The directory you specify for removal must be empty. To clean it out, switch to the directory and use the ls and rm commands to inspect and delete files.

rmdir examples: Removes the directory mydir. rmdir mydir

CD:

The cd command, which stands for "change directory", changes the shell's current working directory.

JKDirectory!

The cd command is one of the commands you will use the most at the command line in Linux. It allows you to change your working directory. You use it to move around within the hierarchy of your file system.

To help you organize your files, your file system contains special files called directories. Think of them like folders in a file cabinet: they have names, just like files, but their function is to "contain" other files, and other directories. In this way, you can keep the files on your system separate and sorted according to their function or purpose.

All files and directories on your system stem from one main directory: the root directory. There are no directories "above" the root directory; all other directories are "below" the root directory.

Any directory which is contained inside another directory is called a subdirectory. Subdirectories "branch" off the "root" of the directory "tree." Unlike a real tree, directory trees are upside-down: the root is at the top and the branches reach down.

When you move into a subdirectory, you are moving "down" the tree; when you move into a directory's parent directory; you are moving "up" the tree. All directories on your file system are subdirectories of the root directory.

Note: By default, when you open a terminal and begin using the command line, you are placed in your home directory.

Directories are separated by a forward slash ("/"). For instance, the directory name "documents/work/accounting" means "the directory named accounting, which is in the directory named work, which is in the directory named documents which is in the current directory."

To change into this directory, and make it our working directory, we would use the command: **Syntax:**

```
cd documents/work/accounting
```

If the first character of a directory name is a slash, that denotes that the directory path begins in the root directory. So, in contrast to the example above, the directory name "/documents/work/accounting" (note the beginning slash) means "the directory named accounting, which is in the directory named work, which is in the directory named documents which is in the root directory."

The current directory, regardless of which directory it is, is represented by a single dot ("."). So, running this command:

```
cd .
```

JKDirectory!

...would change us into the directory we're already in. In other words, it would do nothing, but it would do it successfully.

The parent directory of the current directory — in other words, the directory one level up from the current directory, which contains the directory we're in now — is represented by two dots ("..").

So, If we were in the directory /home/username/documents, and we executed the command:

```
cd ..
```

...we would be placed in the directory /home/username.

Your home directory is the directory you're placed in, by default, when you open a new terminal session. It's the directory that holds all your settings, your mail, your default documents and downloads folder, and many other personal items. It has a special representation: a tilde ("~").

So, if our username is username, and our home directory is /home/username, the command:

```
cd ~
```

...is functionally the same as the command: **cd /home/username**

Syntax:

cd [-L|-P] *directory*

Options

-L → Force symbolic links to be followed. This is the default behavior of cd.

-P → Use the physical directory structure without following symbolic links. This is the opposite of the -L option, and if they are both specified, this option will be ignored.

SCANNING DIRECTORIES:

Opendir:

It is used to open a directory. The *opendir*() function shall open a directory stream corresponding to the directory named by the *dirname* argument. The directory stream is positioned at the first entry. If the type **DIR** is implemented using a file descriptor, applications shall only be able to open up to a total of {OPEN_MAX} files and directories.

Upon successful completion, *opendir*() shall return a pointer to an object of type **DIR**. Otherwise, a null pointer shall be returned and *errno* set to indicate the error.

JKDirectory!

Syntax:

#include <dirent.h>
DIR *opendir(const char *dirname);

The following program fragment demonstrates how the *opendir*() function is used.

```
#include <sys/types.h>
#include <dirent.h>
#include <libgen.h>
...
  DIR *dir;
  struct dirent *dp;
...
  if ((dir = opendir (".")) == NULL) {
     perror ("Cannot open .");
     exit (1);
  }
```

The opendir() function should be used in conjunction with readdir(), closedir(), and rewinddir() to examine the contents of the directory. The opendir() function should be used in conjunction with readdir(), closedir(), and rewinddir() to examine the contents of the directory. This method is recommended for portability.

The *opendir*() function shall fail if: Search permission is denied, The length of the *dirname* argument exceeds {PATH_MAX}, A component of *dirname* is not a directory, etc..

Readdir:

It is used to read the directory entry. The readdir() function shall return a pointer to a structure representing the directory entry at the current position in the directory stream specified by the argument *dirp*, and position the directory stream at the next entry.

It shall return a null pointer upon reaching the end of the directory stream. The structure dirent defined in the <dirent.h> header describes a directory entry. The readdir() function shall not return directory entries containing empty names.

If entries for dot or dot-dot exist, one entry shall be returned for dot and one entry shall be returned for dot-dot; otherwise, they shall not be returned. The pointer returned by readdir() points to data which may be overwritten by another call to readdir() on the same directory stream. This data is not overwritten by another call to readdir() on a different directory stream.

Upon successful completion, readdir() shall return a pointer to an object of type struct dirent. When an error is encountered, a null pointer shall be returned and errno shall be set to indicate the error. When the end of the directory is encountered, a null pointer shall be returned and errno is not changed.

JKDirectory!

Syntax:

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dirp);
```

The following sample program searches the current directory for each of the arguments supplied on the command line.

```
#include <dirent.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>
static void lookup(const char *arg)
{
   DIR *dirp;
   struct dirent *dp;
   if ((dirp = opendir(".")) == NULL) {
      perror("couldn't open '.'");
      return;
   }
```

```
        do {
            errno = 0;
            if ((dp = readdir(dirp)) != NULL) {
                if (strcmp(dp->d_name, arg) != 0)
                    continue;
                (void) printf("found %s\n", arg);
                (void) closedir(dirp);
                    return;
            }
        } while (dp != NULL);
        if (errno != 0)
            perror("error reading directory");
        else
            (void) printf("failed to find %s\n", arg);
        (void) closedir(dirp);
        return;
    }
    int main(int argc, char *argv[])
    {
        int i;
        for (i = 1; i < argc; i++)
            lookup(argv[i]);
        return (0);
    }
```

JKDirectory!

Telldir:

It tells about current location of a named directory stream. The telldir() function obtains the current location associated with the directory stream specified by dirp. If the most recent operation on the directory stream was a seekdir(), the directory position returned from the telldir() is the same as that supplied as a loc argument for seekdir().

Upon successful completion, telldir() returns the current location of the specified directory stream. No errors are defined.

Syntax:

```
#include <dirent.h>
long int telldir(DIR *dirp);
```

Seekdir:

It set position of directory stream. The seekdir() function sets the position of the next readdir() operation on the directory stream specified by dirp to the position specified by loc.

The value of loc should have been returned from an earlier call to telldir(). The new position reverts to the one associated with the directory stream when telldir() was performed.

If the value of loc was not obtained from an earlier call to telldir() or if a call to rewinddir() occurred between the call to telldir() and the call to seekdir(), the results of subsequent calls to readdir() are unspecified.

The seekdir() function returns no value.

Syntax:

```
#include <sys/types.h>
#include <dirent.h>
void seekdir(DIR *dirp, long int loc);
```

Rewinddir:

It reset position of directory stream to the beginning of a directory. The rewinddir() function resets the position of the directory stream to which dirp refers to the beginning of the directory. It also causes the directory stream to refer to the current state of the corresponding directory, as a call to opendir() would have done. If dirp does not refer to a directory stream, the effect is undefined.

JKDirectory!

The rewinddir() function does not return a value.

Syntax:

```
#include <sys/types.h>
#include <dirent.h>
void rewinddir(DIR *dirp);
```

The rewinddir() function should be used in conjunction with opendir(), readdir() and closedir() to examine the contents of the directory. This method is recommended for portability.

Closedir:

It is used to close a directory stream. The closedir() function shall close the directory stream referred to by the argument dirp. Upon return, the value of dirp may no longer point to an accessible object of the type DIR. If a file descriptor is used to implement type DIR, that file descriptor shall be closed.

Upon successful completion, closedir() shall return 0; otherwise, -1 shall be returned and errno set to indicate the error. The closedir() function may fail if: The dirp argument does not refer to an open directory stream.

Syntax:

```
#include <dirent.h>
int closedir(DIR *dirp);
```

The following program fragment demonstrates how the closedir() function is used.

```
...
   DIR *dir;
   struct dirent *dp;
...
   if ((dir = opendir (".")) == NULL) {
...
   }
   while ((dp = readdir (dir)) != NULL) {
...
   }
   closedir(dir);
...
```

JKDirectory!