Linux Programming

Read all record programs

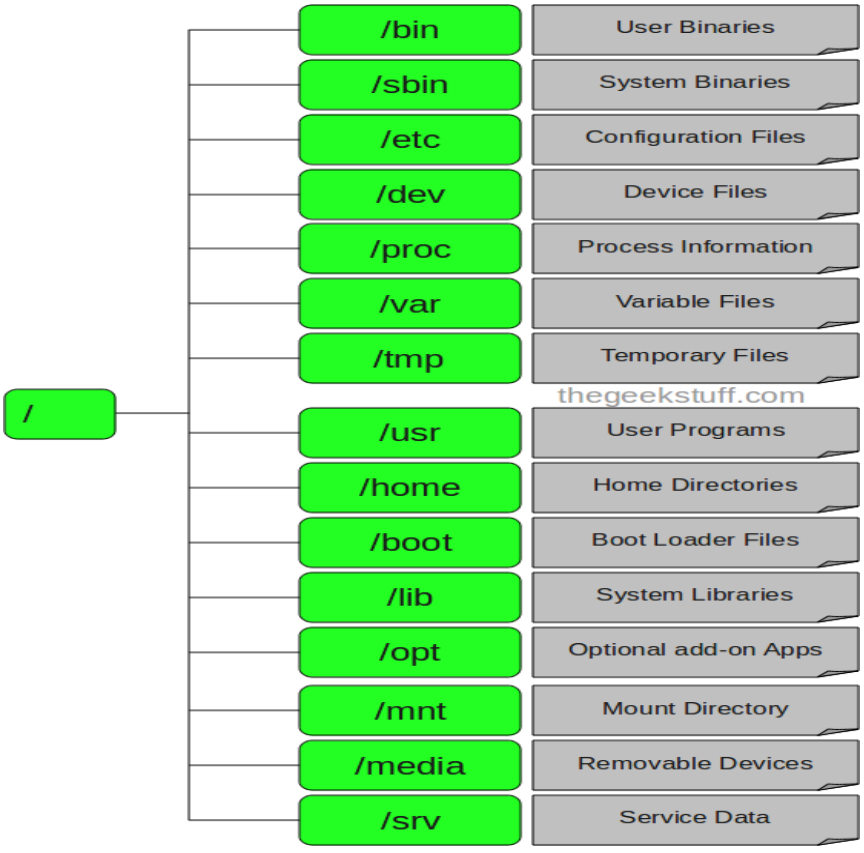
Unit 1

Files: Files concept, File System Structure, Inodes, File Attributes, File types, Library functions, the standard I/O and formatted I/O in C, stream errors, kernel support for files, System calls, file descriptors, low-level file Access- File structure related system calls(File APIs), file and record locking, file and directory management Directory file APIs, Symbolic links & hard links.

Files concept

In a Linux system, everything is a file and if it is not a file, it is a process. A file doesn't include only text files, images, and compiled programs but also includes partitions, hardware device drivers, and directories. Linux considers everything as a file. Files are always case sensitive.

File System Structure



The Linux File Hierarchy Structure or the Filesystem Hierarchy Standard (FHS) defines the directory structure and directory contents in Unix-like operating systems. It is

maintained by the Linux Foundation. All files and directories appear under the root directory /, even if they are stored on different physical or virtual devices.

./ (Root)

- Every single file and directory starts from the root directory
- Only the root user has the right to write under this directory
- /root is the root user's home directory, which is not the same as /

./bin

- Contains binary executables
- Common Linux commands you need to use in single-user modes are located under this directory.
- Commands used by all the users of the system are located here
- Examples: cat, ls, cp, ps, ping, grep

./dev

- Contains device files.
- These include terminal devices, usb, or any device attached to the system.
- For example: /dev/null, /dev/console, /dev/tty, /dev/usbmono

./etc

- Contains configuration files required by all programs.
- This also contains startup and shutdown shell scripts used to start/stop individual programs.
- For example: /etc/resolv.conf, /etc/logrotate.conf

./lib

- Contains library files that support the binaries located under /bin and /sbin
- Library filenames are either ld* or lib*.so.*
- For example: ld-2.11.1.so, libncurses.so.5.7

File system implementation

boot block

- A *boot block* is located in the first few sectors of a file system. The boot block contains the initial bootstrap program used to load the operating system.
- Typically, the first sector contains a bootstrap program that reads in a larger bootstrap program from the next few sectors, and so forth.

super block

- A *super block* describes the state of the file system: the total size of the partition, the block size, pointers to a list of free blocks, the inode number of the root directory, magic number, etc.

inode list

- A linear array of *inodes* (short for ``index nodes''). There is a one to one mapping of files to inodes and vice versa. An inode is identified by its ``inode number'', which contains the information needed to find the inode itself on the disk
- Thus, while users think of files in terms of file names, Unix thinks of files in terms of inodes.

data block

- *data blocks* blocks containing the actual contents of files

Inodes

- An Inode number is a uniquely existing number for all the files in Linux and all Unix type systems.
- When a file is created on a system, a file name and Inode number is assigned to it.
- Generally, to access a file, a user uses the file name but internally file name is first mapped with the respective Inode number stored in a table.
- An Inode is a data structure containing metadata about the files.
- Inode stores various information about a file like file size, owner of the file, access mode, date of creation, etc.

File Attributes

Type: Whether ordinary, directory, device, etc.
Permissions: Determines who can read, write, or execute a file.
Links: The number of hard links to the file. Several files in the file system can reference the same file on the drive.
Owner: A file is owned by a user, by default its creator. The owner can change many file attributes and set the permissions.
Group Owner: The group which owns the file. The owner by default belongs to this group.
File Size: The number of bytes of data contained.
File Time Stamps:

- Date and time of last modification
- Date and time of last access

File types

Regular files (-)

These are files data contain text, data, or program instructions and they are the most common type of files you can expect to find on a Linux system and they include:

1. Readable files
2. Binary files
3. Image files
4. Compressed files and so on.

Directory files (d)

These are special files that store both ordinary and other special files and they are organized on the Linux file system in a hierarchy starting from the root (/) directory. It is shown in blue color. It contains a list of files.

Special files

- **Block file (b):** These files are hardware files, and most of them are present in /dev. They are created either by fdisk command or by partitioning. They can transfer a large block of data and information at a given time.
- **Character device file (c):** The character device file provides a serial stream of input or output. Our terminals are a classic example of this type of file. They work by providing a way of communication with devices by transferring data one character at a time.
- **Named pipe file (p):** These are files that allow inter-process communication by connecting the output of one process to the input of another. A named pipe is a file that is used by two processes to communicate with each and it acts as a Linux pipe.

- **Symbolic link file (l):** These are linked files to other files. They are either Directory/Regular File. The inode number for this file and its parent files are the same. There are two types of link files available in Linux/Unix: soft and hard links.
- **Socket file (s):** These are files that provide a means of inter-process communication, but they can transfer data and information between processes running on different environments. This means that sockets provide data and information transfer between processes running on different machines on a network.

Library functions

Library functions are usually documented in section 3 of the manual pages and often have a standard include file associated with them, such as stdio.h for the standard I/O library. The functions which are a part of the standard C library are known as Library functions. For example the standard string manipulation functions like strcmp(), strlen() etc are all library functions.

Standard I/O in C

The standard I/O library (stdio) and its header file, stdio.h, provide a versatile interface to low-level I/O system calls. Three file streams are automatically opened when a program is started. They are stdin, stdout, and stderr. These are declared in stdio.h and represent the standard input, output, and error output, respectively, which correspond to the low-level file descriptors 0, 1, and 2 or STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO. Some functions in Standard I/O are:

fopen

You use it mainly for files and terminal input and output. If successful, fopen returns a non-null FILE * pointer. If it fails, it returns the value NULL, defined in stdio.h.

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

fopen opens the file named by the filename parameter and associates a stream with it. The mode parameter specifies how the file is to be opened. The b indicates that the file is a binary file rather than a text file.

- “r” or “rb”: Open for reading only
- “w” or “wb”: Open for writing, truncate to zero-length
- “a” or “ab”: Open for writing, append to end of file
- “r+” or “rb+” or “r+b”: Open for update (reading and writing)
- “w+” or “wb+” or “w+b”: Open for the update, truncate to zero-length
- “a+” or “ab+” or “a+b”: Open for the update, append to end of file

fread

The fread library function is used to read data from a file stream. Data is read into a data buffer given by ptr from the stream, stream. Both fread and fwrite deal with data records. These are specified by record size, size, and a count, nitems, of records to transfer. The function returns the number of items (rather than the number of bytes) successfully read into the data buffer. At the end of a file, fewer than nitems may be returned, including zero.

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);
```

fwrite

The fwrite library call has a similar interface to fread. It takes data records from the specified data buffer and writes them to the output stream. It returns the number of records successfully written.

```
#include <stdio.h>
size_t fwrite (const void *ptr, size_t size, size_t nitems, FILE *stream);
```

fclose

The fclose library function closes the specified stream, causing any unwritten data to be written. It's important to use fclose because the stdio library will buffer data. If the program needs to be sure that data has been completely written, it should call fclose. However, that fclose is called automatically on all file streams that are still open when a program ends normally, but then, of course, you do not get a chance to check for errors reported by fclose.

```
#include <stdio.h>
int fclose(FILE *stream);
```

fflush

The fflush library function causes all outstanding data on a file stream to be written immediately. You can sometimes use it when you're debugging a program to make sure that the program is writing data and not hanging. Note that an implicit flush operation is carried out when fclose is called, so you don't need to call fflush before fclose.

```
#include <stdio.h>
int fflush(FILE *stream);
```

fseek

The fseek function sets the position in the stream for the next read or writes on that stream. fseek returns an integer: 0 if it succeeds, –1 if it fails, with errno set to indicate error.

```
#include <stdio.h>
int fseek(FILE *stream, long int offset, int whence);
```

The offset parameter is used to specify the position, and the whence parameter specifies how the offset is used. whence can be one of the following:

- SEEK_SET: offset is an absolute position
- SEEK_CUR: offset is relative to the current position
- SEEK_END: offset is relative to the end of the file

fgetc, getc, and getchar

- The fgetc function returns the next byte, as a character, from a file stream. When it reaches the end of the file or there is an error, it returns EOF. You must use ferror or feof to distinguish the two cases.
- The getc function is equivalent to fgetc, except that it may be implemented as a macro.
- The getchar function is equivalent to getc(stdin) and reads the next character from the standard input.

```
#include <stdio.h>
int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar();
```

fputc, putc, and putchar

- The fputc function writes a character to an output file stream. It returns the value it has written, or EOF on failure.
- The function putc is equivalent to fputc, but it may be implemented as a macro.

- The putchar function is equivalent to putc(c, stdout), writing a single character to the standard output. Note that putchar takes and getchar returns characters as ints, not char. This allows the end-offile (EOF) indicator to take the value –1, outside the range of character codes.

```
#include <stdio.h>
int fputc(int c, FILE *stream);
 int putc(int c, FILE *stream);
int putchar(int c);
```

fgets and gets

The fgets function reads a string from an input file stream. fgets writes characters to the string pointed to by s until a newline is encountered, n-1 characters have been transferred, or the end of file is reached, whichever occurs first.

The gets function is similar to fgets, except that it reads from the standard input and discards any newline encountered. It adds a trailing null byte to the receiving string.

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
char *gets(char *s);
```

Formatted I/O in C

There are several library functions for producing output in a controlled fashion. These functions include printf for printing values to a file stream, and scanf and others for reading values from a file stream.

printf, fprintf, and sprintf

The printf family of functions format and output a variable number of arguments of different types. The way each is represented in the output stream is controlled by the *format* parameter, which is a string that contains ordinary characters to be printed and codes called *conversion specifiers*, which indicate how and where the remaining arguments are to be printed.

The printf function produces its output on the standard output. The fprintf function produces its output on a specified stream. The sprintf function writes its output and a terminating null character into the string s passed as a parameter. This string must be large enough to contain all of the output.

The most commonly used conversion specifiers:

- %d, %i: Print an integer in decimal
- %o, %x: Print an integer in octal, hexadecimal
- %c: Print a character
- %s: Print a string
- %f: Print a floating-point (single precision) number

The printf functions return an integer, the number of characters written. This doesn't include the terminating null in the case of sprintf. On error, these functions return a negative value and set errno

```
#include <stdio.h>
int printf(const char *format, ...);
```

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

scanf, fscanf, and sscanf

The scanf family of functions works in a way similar to the printf group, except that these functions read items from a stream and place values into variables at the addresses they're passed as pointer parameters.

Conversion specifiers are

- %d: Scan a decimal integer
- %o, %x: Scan octal, hexadecimal integer
- %f, %e, %g: Scan a floating-point number
- %c: Scan a character (whitespace not skipped)
- %s: Scan a string
- %%: Scan a % character

The scanf functions return the number of items successfully read, which will be zero if the first item fails. If the end of the input is reached before the first item is matched, EOF is returned. If a read error occurs on the file stream, the stream error flag will be set and the error variable, errno, will be set to indicate the type of error.

```
#include <stdio.h>
int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *s, const char *format, ...);
```

Other Stream Functions

Several other stdio library functions use either stream parameters or the standard streams stdin, stdout, stderr:

- fgetpos: Get the current position in a file stream.
- fsetpos: Set the current position in a file stream.
- ftell: Return the current file offset in a stream.
- rewind: Reset the file position in a stream.
- freopen: Reuse a file stream.
- setvbuf: Set the buffering scheme for a stream.
- remove: Equivalent to unlink unless the path parameter is a directory, in which case it's equivalent to rmdir.

Stream errors

To indicate an error, many stdio library functions return out-of-range values, such as null pointers or the constant EOF. In these cases, the error is indicated in the external variable errno:

```
#include <errno.h>
extern int errno;
```

The ferror function tests the error indicator for a stream and returns nonzero if it's set, but zero otherwise.

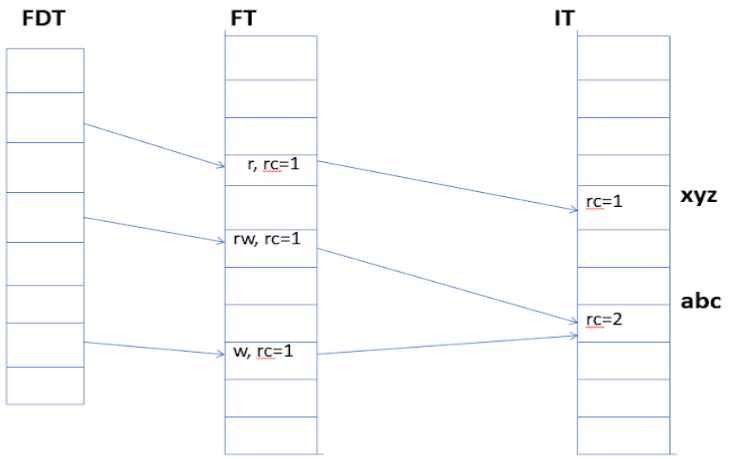
The feof function tests the end-of-file indicator within a stream and returns nonzero if it is set, zero otherwise.

The clearerr function clears the end-of-file and error indicators for the stream to which stream points. It has no return value and no errors are defined. You can use it to recover from error conditions on streams.

```
#include <stdio.h>
int ferror(FILE *stream);
int feof(FILE *stream);
void clearerr(FILE *stream);
```

Kernel support for files

1. In computing, the kernel is the main component of most computer operating systems.
2. It is a bridge between applications and the actual data processing done at the hardware level. The kernel's responsibilities include managing the system's resources.
3. The Kernel has a mechanism to keep track of all files opened by the process.
4. The Kernel maintains three data structures (run-time tables):
    a. File Descriptor Table
    b. File Table
    c. Inode Table
5. File Descriptor Table is a per-process basis. It records all files opened by the process.
6. There will be only one file table and Inode table for each process.
7. The Inode table contains a copy of the file Inodes most recently accessed.
8. When a process calls an open system call to open a file to read or write, it does the following:
    a. The Kernel will search the process File Descriptor table and look for the first unused entry. If an entry is found the entry will be used to reference the file.
    b. The Kernel will scan the File Table in its Kernel space to find an unused entry that can be assigned to reference the file.
    c. If an unused entry is found, the following events will occur:
        i. The process's File Descriptor Table entry will be set to point to file table entry.
        ii. The File Table entry will be set to point to the Inode Table entry.
        iii. The File Table entry will contain the current file pointer of the open file. This is an offset from the beginning of the file where the next read or write operation will occur.
        iv. The file table entry will contain an open mode that specifies that the file is opened for read-only, write-only, or read and write, etc.
        v. The reference count in the File Table entry is set to 1. The reference count keeps track of how many file descriptors from any process are referencing the entry.
        vi. The reference count of the in-memory Inode of the file is increased by 1. This count specifies how many file table entries are pointing to that Inode.
        vii. If either 1 or 2 points fails, the open function will return with a -1 failure status, and no file descriptor table or file table entry will be allocated.



9. Once the open call succeeds, the process can use the returned file descriptor for future reference.
10. The reference count in a file Inode record specifies how many file table entries are pointing to the file Inode record. If the count is not zero, it means that one or more processes are currently opening the file for access.
11. When the process attempts to read(or write) data from the file, it will use the descriptor as the first argument to read(or write) system call.
12. The kernel will use the file descriptor to index the process's file descriptor table to find the pointer to the file table entry of the opened file.
13. If the read(or write) operation is found compatible with the file's open mode, the kernel will use the pointer specified in the file table entry to access the file's Inode record(as stored in the Inode table).
14. When a process calls the close function to close an opened file, the sequence of events is as follows:
    a. The Kernel sets the corresponding file descriptor table entry to be unused.
    b. It decrements the reference count in the corresponding file table entry by 1. If the reference count is still non-zero, go to step f.
    c. The file table entry is marked as unused.
    d. The reference count in the corresponding file Inode table entry is decrement by 1. If the reference count is still non-zero, go to step f.
    e. If the hard-link count of the Inode is not zero, it returns to the caller with a success status. Otherwise, it marks the Inode table entry as unused and deallocates all the physical disk storage of the file, as all the file path names have been removed by some process.
    f. It returns to the process with a 0(success) status.

System calls

The functions which change the execution mode of the program from user mode to kernel mode are known as system calls. These calls are required in case some services are required by the program from the kernel. For example, if we want to change the date and time of the system or if we want to create a network socket then these services can only be provided by the kernel and hence these cases require system calls. socket() is a system call.

File descriptors

To the kernel, all open files are referred to by file descriptors. Afile descriptor is a non-negative integer. When we open an existing file or create a new file, the kernel returns a file descriptor to the process. When we want to read or write a file, we identify the file with the file descriptor that was returned by open or creat as an argument to either read or write.

By convention, UNIX System shells associate file descriptor 0 with the standard input of a process, file descriptor 1 with the standard output, and file descriptor 2 with the standard error. File descriptors range from 0 through OPEN_MAX−1.

Although their values are standardized by POSIX.1, the magic numbers 0, 1, and 2 should be replaced in POSIX-compliant applications with the symbolic constants STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO to improve readability. These constants aredefined in the<unistd.h>header.

Low-level file Access- File structure related system calls(File APIs)

Each running program, called a process, has several file descriptors associated with it. These are small integers that you can use to access open files or devices. When a program starts, it usually has three of these descriptors already opened. These are:

- 0: Standard input
- 1: Standard output
- 2: Standard error

write

The write system call arranges for the first nbytes bytes from buf to be written to the file associated with the file descriptor fildes. It returns the number of bytes written. If the function returns 0, it means no data was written; if it returns –1, there has been an error in the write call, and the error will be specified in the errno global variable.

```
#include <unistd.h>
size_t write(int fildes, const void *buf, size_t nbytes);
```

read

The read system call reads up to nbytes bytes of data from the file associated with the file descriptor fildes and places them in the data area buf. It returns the number of data bytes read, which may be less than the number requested. If a read call returns 0, it had nothing to read; it reached the end of the file. An error on the call will cause it to return –1.

```
#include <unistd.h>
size_t read(int fildes, void *buf, size_t nbytes);
```

open

To create a new file descriptor, you need to use the open system call. The name of the file or device to be opened is passed as a parameter, path; the oflags parameter is used to specify actions to be taken on opening the file. The oflags are specified as a combination of a mandatory file access mode and other optional modes.

- O_RDONLY Open for read-only
- O_WRONLY Open for write-only
- O_RDWR Open for reading and writing
- O_APPEND: Place written data at the end of the file.
- O_TRUNC: Set the length of the file to zero, discarding existing contents.
- O_CREAT: Creates the file, if necessary, with permissions given in mode.

- O_EXCL: Used with O_CREAT, ensures that the caller creates the file. If the file already exists, the open will fail.

When you create a file using the O_CREAT flag with open, you must use the three-parameter form. mode, the third parameter, is made from a bitwise OR of the flags defined in the header file sys/stat.h. These are:

- S_IRUSR: Read permission, owner
- S_IWUSR: Write permission, owner
- S_IXUSR: Execute permission, owner
- S_IRGRP: Read permission, group
- S_IWGRP: Write permission, group
- S_IXGRP: Execute permission, group
- S_IROTH: Read permission, others
- S_IWOTH: Write permission, others
- S_IXOTH: Execute permission, others

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
int open(const char *path, int oflags);
int open(const char *path, int oflags, mode_t mode);
```

umask

The umask is a system variable that encodes a mask for file permissions to be used when a file is created. You can change the variable by executing the umask command to supply a new value. The value is a three-digit octal value. Each digit is the result of ORing values from 1, 2, or 4.

close

You use close to terminate the association between a file descriptor, fildes, and its file. The file descriptor becomes available for reuse. It returns 0 if successful and –1 on error.

```
#include <unistd.h>
int close(int fildes);
```

ioctl

ioctl is a bit of a ragbag of things. It provides an interface for controlling the behavior of devices and their descriptors and configuring underlying services. Terminals, file descriptors, sockets, and even tape drives may have ioctl calls defined for them and you need to refer to the specific device's man page for details. POSIX defines the only ioctl for streams.

```
#include <unistd.h>
int ioctl(int fildes, int cmd, ...);
```

ioctl performs the function indicated by cmd on the object referenced by the descriptor fildes. It may take an optional third argument, depending on the functions supported by a particular device.

For example, the following call to ioctl on Linux turns on the keyboard LEDs:

```
ioctl(tty_fd, KDSETLED, LED_NUM|LED_CAP|LED_SCR);
```

Other System Calls for Managing Files

There are several other system calls that operate on these low-level file descriptors. These allow a program to control how a file is used and to return status information.

lseek

The lseek system call sets the read/write pointer of a file descriptor, fildes; that is, you can use it to set wherein the file the next read or write will occur. You can set the pointer to an absolute location in the file or to a position relative to the current position or the end of the file.

```
#include <unistd.h>
#include <sys/types.h>
off_t lseek(int fildes, off_t offset, int whence);
```

The offset parameter is used to specify the position, and the whence parameter specifies how the offset is used. whence can be one of the following:

- SEEK_SET: offset is an absolute position
- SEEK_CUR: offset is relative to the current position
- SEEK_END: offset is relative to the end of the file

lseek returns the offset measured in bytes from the beginning of the file that the file pointer is set to, or –1 on failure. The type off_t, used for the offset in seek operations, is an implementation-dependent integer type defined in sys/types.h.

fstat, stat, and lstat

The fstat system call returns status information about the file associated with an open file descriptor. The information is written to a structure, buf, the address of which is passed as a parameter.

```
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
int fstat(int fildes, struct stat *buf);
int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

The related functions stat and lstat return status information for a named file. They produce the same results, except when the file is a symbolic link. lstat returns information about the link itself, and stat returns information about the file to which the link refers.

- st_mode File permissions and file-type information
- st_ino The inode associated with the file
- st_dev The device the file resides on
- st_uid The user identity of the file owner
- st_gid The group identity of the file owner
- st_atime The time of last access
- st_ctime The time of last change to permissions, owner, group, or content
- st_mtime The time of last modification to contents
- st_nlink The number of hard links to the file

The st_mode flags returned in the stat structure also have several associated macros defined in the header file sys/stat.h. These macros include names for permission and file-type flags and some masks to help with testing for specific types and permissions. Some of them are

- S_ISBLK: Test for block special file
- S_ISCHR: Test for character special file
- S_ISDIR: Test for the directory
- S_ISFIFO: Test for FIFO

- S_ISREG: Test for regular file
- S_ISLNK: Test for symbolic link

dup and dup2

The dup system calls provide a way of duplicating a file descriptor, giving two or more different descriptors that access the same file. These might be used for reading and writing to different locations in the file.

The dup system call duplicates a file descriptor, fildes, returning a new descriptor. The dup2 system call effectively copies one file descriptor to another by specifying the descriptor to use for the copy. These calls can also be useful when you're using multiple processes communicating via pipes.

```
#include <unistd.h>
int dup(int fildes); int dup2(int fildes, int fildes2);
```

Symbolic links and hard links

A link in UNIX is a pointer to a file. Like pointers in any programming language, links in UNIX are pointers pointing to a file or a directory. Creating links is a kind of shortcut to access a file. Links allow more than one file name to refer to the same file, elsewhere. There are two types of links :

- Soft Link or Symbolic links
- Hard Links

These links behave differently when the source of the link (what is being linked to) is moved or removed. Symbolic links are not updated (they merely contain a string which is the pathname of its target); hard links always refer to the source, even if moved or removed.

For example, if we have a file a.txt. If we create a hard link to the file and then delete the file, we can still access the file using the hard link. But if we create a soft link of the file and then delete the file, we can't access the file through the soft link and the soft link becomes dangling. A hard link increases the reference count of a location while soft links work as a shortcut (like in Windows).

Hard Links

- Each hard-linked file is assigned the same Inode value as the original, therefore they reference the same physical file location. Hard links more flexible and remain linked even if the original or linked files are moved throughout the file system, although hard links are unable to cross different file systems.
- ls -l command shows all the links with the link column shows several links.
- Links have actual file contents
- Removing any link just reduces the link count, but doesn't affect other links.
- We cannot create a hard link for a directory to avoid recursive loops.
- If the original file is removed then the link will still show the content of the file.
- Command to create a hard link is:
    - $ ln [original filename] [link name]

Soft Links

- A soft link is similar to the file shortcut feature which is used in Windows Operating systems. Each soft linked file contains a separate Inode value that points to the

original file. As similar to hard links, any changes to the data in either file are reflected in the other. Soft links can be linked across different file systems, although if the original file is deleted or moved, the soft linked file will not work correctly (called hanging link).

- ls -l command shows all links with first column value l? and the link points to the original file.
- Soft Link contains the path for the original file and not the contents.
- Removing a soft link doesn't affect anything but removing the original file, the link becomes a "dangling" link which points to a nonexistent file.
- A soft link can link to a directory.
- The link across filesystems: If you want to link files across the filesystems, you can only use symlinks/soft links.
- The command to create a Soft link is
    - $ ln -s [original filename] [link name]

Unit 2

Process: Process concept, Kernel support for process, process attributes, process control - process creation, waiting for a process, process termination, zombie process, orphan process APIs.
Signals: Introduction to signals, Signal generation and handling, Kernel support for signal, Signal function, unreliable signal, reliable signal, kill, raise, alarm, pause, abort, sleep functions.

Process and Thread

Process

Process means any program is in execution. Process control block controls the operation of any process. Process control block contains information about processes for example Process priority, process id, process state, CPU, register, etc. A process can create other processes which are known as Child Processes. The process takes more time to terminate and it is isolated means it does not share the memory with any other process.
The process can have the following states like new, ready, running, waiting, terminated, suspended.

Thread

Thread is the segment of a process means a process can have multiple threads and these multiple threads are contained within a process. A thread has 3 states: running, ready, and blocked. The thread takes less time to terminate as compared to process and like process threads do not isolate.

Process	Thread
Process means any program is in execution.	Thread means a segment of a process.
The process takes more time to terminate.	The thread takes less time to terminate.
It takes more time for creation.	It takes less time for creation.
It also takes more time for context switching.	It takes less time for context switching.
The process is less efficient in terms of communication.	Thread is more efficient in terms of communication.
The process consumes more resources.	Thread consumes fewer resources.
The process is isolated.	Threads share memory.

The process is called the heavyweight process.	Thread is called a lightweight process.
Process switching uses an interface in the operating system.	Thread switching does not require to call an operating system and causes an interrupt to the kernel.
If one server process is blocked no other server process can execute until the first process unblocked.	The second thread in the same task could run, while one server thread is blocked.
The process has its own Process Control Block, Stack, and Address Space.	Thread has Parents' PCB, its Thread Control Block and Stack, and common Address space.

fork and vfork

fork()

fork() is a system call that is used to create a new process. The new process created by the fork() system call is called a child process and the process that invoked the fork() system call is called the parent process. The Code of the child process is the same as the code of its parent process. Once the child process is created, both parent and child processes start their execution from the next statement after fork() and both processes get executed simultaneously.

vfork()

vfork() is also a system call that is used to create a new process. The new process created by the vfork() system call is called the child process and the process that invoked vfork() system call is called the parent process. The Code of a child process is the same as the code of its parent process. The child process suspends execution of the parent process until the child process completes its execution as both processes share the same address space.

fork()	vfork()
In the fork() system call, child and parent process have separate memory space.	While in vfork() system call, child and parent process share the same address space.
The child process and parent process gets executed simultaneously.	Once the child process is executed then the parent process starts its execution.
The fork() system call uses copy-on-write as an alternative.	While vfork() system call does not use copy-on-write.
Child process does not suspend parent process execution in fork() system call.	Child process suspends parent process execution in vfork() system call.

Page of one process is not affected by a page of other processes.	Page of one process is affected by a page of other processes.
fork() system call is more used.	vfork() system call is less used.
There is a wastage of address space.	There is no wastage of address space.
If the child process alters a page in address space, it is invisible to the parent process.	If the child process alters the page in the address space, it is visible to the parent process.

Introduction to signals

Signals are software interrupts. They are classic examples of asynchronous events. Most nontrivial application programs need to deal with signals. Signals provide a way of handling asynchronous events—for example, a user at a terminal typing the interrupt key to stop a program or the next program in a pipeline terminating prematurely.

Signals have been provided since the early versions of the UNIX System, but the signal model provided with systems such as Version 7 was not reliable. Signals could get lost, and it was difficult for a process to turn off selected signals when executing critical regions of code. Both 4.3BSD and SVR3 made changes to the signal model, adding what are called reliable signals. But the changes made by Berkeley and AT&T were incompatible. Fortunately, POSIX.1 standardized the reliable-signal routines.

Signal names are all defined by positive integer constants (the signal number) in the header<signal.h>. No signal has a signal number of 0. The kill function uses the signal number of 0 for a special case. POSIX.1 calls this value the null signal.

Signal generation and handling

Generation

- The terminal-generated signals occur when users press certain terminal keys. Pressing the DELETE key on the terminal (or Control-C on many systems) normally causes the interrupt signal (SIGINT) to be generated. This is how to stop a runaway program.
- Hardware exceptions generate signals: divide by 0, invalid memory reference, and the like. These conditions are usually detected by the hardware, and the kernel is notified. The kernel then generates the appropriate signal for the process that was running at the time the condition occurred. For example, SIGSEGV is generated for a process that executes an invalid memory reference.
- The kill(2) function allows a process to send any signal to another process or process group. Naturally, there are limitations: we have to be the owner of the process that we're sending the signal to, or we have to be the superuser.
- The kill(1) command allows us to send signals to other processes. This program is just an interface to the kill function. This command is often used to terminate a runaway background process.
- Software conditions can generate signals when a process should be notified of various events. These aren't hardware-generated conditions (as is the divide by-0 condition), but software conditions. Examples are SIGURG (generated when out-of-band data arrives over a network connection), SIGPIPE (generated when a

process writes to a pipe that has no reader), and SIGALRM (generated when an alarm clock set by the process expires).

Handling

The process can't simply test a variable (such as errno) to see whether a signal has occurred; instead, the process has to tell the kernel ‘‘if and when this signal occurs, do the following.’’ We can tell the kernel to do one of three things when a signal occurs. We call this the disposition of the signal, or the action associated with a signal.

- Ignore the signal. This works for most signals, but two signals can never be ignored: SIGKILL and SIGSTOP. The reason these two signals can't be ignored is to provide the kernel and the superuser with a surefire way of either killing or stopping any process. Also, if we ignore some of the signals that are generated by a hardware exception (such as illegal memory reference or divide by 0), the behavior of the process is undefined.
- Catch the signal. To do this, we tell the kernel to call a function of ours whenever the signal occurs. In our function, we can do whatever we want to handle the condition. If we're writing a command interpreter, for example, when the user generates the interrupt signal at the keyboard, we probably want to return to the main loop of the program, terminating whatever command we were executing for the user. If the SIGCHLD signal is caught, it means that a child process has terminated, so the signal-catching function can call waitpid to fetch the child's process ID and termination status.
- Let the default action apply. Every signal has a default action. Note that the default action for most signals is to terminate the process.

Signal function

The simplest interface to the signal features of the UNIX System is the signal function.

```
#include <signal.h>
void (*signal(int signo,void (*func)(int)))(int);
```

It returns the previous disposition of signal if OK, SIG_ERR on error.

The prototype for the signal function states that the function requires two arguments and returns a pointer to a function that returns nothing (void). The signo argument is just the name of the signal. The value of func is

- the constant SIG_IGN,
- the constant SIG_DFL
- the address of a function to be called when the signal occurs.

If we specify SIG_IGN, we are telling the system to ignore the signal. When we specify SIG_DFL, we are setting the action associated with the signal to its default value. When we specify the address of a function to be called when the signal occurs, we are arranging to ‘‘catch’’ the signal. We call the function either the signal handler or the signal-catching function.

If we examine the system's header<signal.h>, we will probably find declarations of the form

- #define SIG_ERR (void (*)()) -1
- #define SIG_DFL (void (*)()) 0
- #define SIG_IGN (void (*)()) 1

These constants can be used in place of the ‘‘pointer to a function that takes an integer argument and returns nothing,’’ the second argument to signal, and the return value from a signal. The three values used for these constants need not be −1, 0, and 1. They must be three values that can never be the address of any declarable function.

Unreliable signal

In earlier versions of the UNIX System, signals were unreliable. By this we mean that signals could get lost: a signal could occur and the process would never know about it. Also, a process had little control over a signal: a process could catch the signal or ignore it. Sometimes, we would like to tell the kernel to block a signal: don't ignore it, just remember if it occurs, and tell us later when we're ready.

One problem with these early versions was that the action for a signal was reset to its default each time the signal occurred. The code that was described usually looked like

```
int sig_int(); /* my signal handling function */
        .
        .
        .
signal(SIGINT, sig_int); /* establish handler */
        .
        .
        .
sig_int() {
        signal(SIGINT, sig_int); /* reestablish handler for next time */
                .
                ./ *process the signal ... */
                .
}
```

The problem with this code fragment is that there is a window of time—after the signal has occurred, but before the call to signal in the signal handler—when the interrupt signal could occur another time. This second signal would cause the default action to occur, which for this signal terminates the process. This is one of those conditions that works correctly most of the time, causing us to think that it is correct when it isn't.

Another problem with these earlier systems was that the process was unable to turn a signal off when it didn't want the signal to occur. All the process could do was ignore the signal. There are times when we would like to tell the system ‘‘prevent the following signals from interrupting me, but remember if they do occur.’’The classic example that demonstrates this flaw is shown by a piece of code that catches a signal and sets a flag for the process that indicates that the signal occurred:

```
int sig_int(); /* my signal handling function */
int sig_int_flag; /* set nonzero when signal occurs */
main() {
        signal(SIGINT, sig_int); /* establish handler */
                .
                .
                .
        while (sig_int_flag == 0)
                pause(); /* go to sleep, waiting for signal */
                .
                .
                .
}
sig_int() {
        signal(SIGINT, sig_int); /* reestablish handler for next time */
        sig_int_flag = 1; /* set flag for main loop to examine */
}
```

Here, the process is calling the pause function to put it to sleep until a signal is caught. When the signal is caught, the signal handler just sets the flag sig_int_flag to a nonzero value. The process is automatically awakened by the kernel after the signal handler returns, notices that the flag is nonzero, and does whatever it needs to do. But there is a window of time when things can go wrong. If the signal occurs after the test of sig_int_flag but before the call to pause, the process could go to sleep forever (assuming that the signal is never generated again). This occurrence of the signal is lost. This is another example of some code that isn't right, yet it works most of the time. Debugging this type of problem can be difficult.

Reliable signal

We need to define some of the terms used throughout our discussion of signals. First, a signal is generated for a process (or sent to a process) when the event that causes the signal occurs. The event could be a hardware exception (e.g., divide by 0), a software condition (e.g., an alarm timer expiring), a terminal-generated signal, or a call to the kill function. When the signal is generated, the kernel usually sets a flag of some form in the process table.

We say that a signal is delivered to a process when the action for a signal is taken. During the time between the generation of a signal and its delivery, the signal is said to be pending.

A process has the option of blocking the delivery of a signal. If a signal that is blocked is generated for a process, and if the action for that signal is either the default action or to catch the signal, then the signal remains pending for the process until the process either

- unblocks the signal
- changes the action to ignore the signal.

The system determines what to do with a blocked signal when the signal is delivered, not when it's generated. This allows the process to change the action for the signal before it's delivered. The sigpending function (Section 10.13) can be called by a process to determine which signals are blocked and pending.

What happens if a blocked signal is generated more than once before the process unblocks the signal? POSIX.1 allows the system to deliver the signal either once or more than once. If the system delivers the signal more than once, we say that the signals are queued. Most UNIX systems, however, do not queue signals unless they support the real-time extensions to POSIX.1. Instead, the UNIX kernel simply delivers the signal once.

Each process has a signal mask that defines the set of signals currently blocked from delivery to that process. We can think of this mask as having one bit for each possible signal. If the bit is on for a given signal, that signal is currently blocked. A process can examine and change its current signal mask by calling sigprocmask.

Since the number of signals can exceed the number of bits in an integer, POSIX.1 defines a data type, called sigset_t, that holds a signal set. The signal mask, for example, is stored in one of these signal sets.

kill and raise

The kill function sends a signal to a process or a group of processes. The raise function allows a process to send a signal to itself.

```
#include <signal.h>
int kill(pid_t pid, int signo);
int raise(int signo);
```

The call raise(signo); is equivalent to the call kill(getpid(), signo);

There are four different conditions for the pid argument to kill.

- pid > 0: The signal is sent to the process whose process ID is pid.
- pid == 0: The signal is sent to all processes whose process group ID equals the process group ID of the sender and for which the sender has permission to send the signal. The set of processes excludes certain system processes. This set of system processes includes the kernel processes and init(pid 1).
- pid < 0: The signal is sent to all processes whose process group ID equals the absolute value of pid and for which the sender has permission to send the signal. The set of processes excludes certain system processes.
- pid == −1: The signal is sent to all processes on the system for which the sender has permission to send the signal. The set of processes excludes certain system processes.

POSIX.1 defines signal number 0 as the null signal. If the signo argument is 0, then the normal error checking is performed by kill, but no signal is sent. This technique is often used to determine if a specific process still exists. If we send the process the null signal and it doesn't exist, kill returns −1and errno is set to ESRCH.

If the call to kill causes the signal to be generated for the calling process and if the signal is not blocked, either signo or some other pending, an unblocked signal is delivered to the process before kill returns.

alarm and pause

The alarm function allows us to set a timer that will expire at a specified time in the future. When the timer expires, the SIGALRM signal is generated. If we ignore or don't catch this signal, its default action is to terminate the process.

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

It returns 0 or a number of seconds until previously set alarm.

The seconds value is the number of clock seconds in the future when the signal should be generated. When that time occurs, the signal is generated by the kernel, although additional time could elapse before the process gets control to handle the signal, because of processor scheduling delays.

There are only one of these alarm clocks per process. If, when we call alarm, a previously registered alarm clock for the process has not yet expired, the number of seconds left for that alarm clock is returned as the value of this function. That previously registered alarm clock is replaced by the new value.

If a previously registered alarm clock for the process has not yet expired and if the seconds value is 0, the previous alarm clock is canceled. The number of seconds left for that previous alarm clock is still returned as the value of the function.

Although the default action for SIGALRMis to terminate the process, most processes that use an alarm clock catch this signal. If the process then wants to terminate, it can perform whatever cleanup is required before terminating. If we intend to catch SIGALRM, we need to be careful to install its signal handler before calling the alarm. If we call alarm first and are sent SIGALRM before we can install the signal handler, our process will terminate.

The pause function suspends the calling process until a signal is caught.

```
#include <unistd.h>
int pause(void);
```

The only time pause returns are if a signal handler is executed and that handler returns. In that case, pause returns −1 with errno set to EINTR.

abort

The abort function causes abnormal program termination.

```
#include <stdlib.h>
void abort(void);
```

This function sends the SIGABRT signal to the caller.ISO C states that calling abort will deliver an unsuccessful termination notification to the host environment by calling raise(SIGABRT).

ISO C requires that if the signal is caught and the signal handler returns, abort still doesn't return to its caller. If this signal is caught, the only way the signal handler can't return is if it calls exit, _exit, _Exit, longjmp, or siglongjmp. POSIX.1 also specifies that abort overrides the blocking or ignoring of the signal by the process.

The intent of letting the process catch the SIGABRT is to allow it to perform any cleanup that it wants to do before the process terminates. If the process doesn't terminate itself from this signal handler, POSIX.1 states that when the signal handler returns, abort terminates the process.

The ISO C specification of this function leaves it up to the implementation as to whether output streams are flushed and whether temporary files are deleted. POSIX.1 goes further and allows an implementation to call fclose on open standardI/O streams before terminating if the call to abort terminates the process.

sleep functions

sleep

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
```

This function causes the calling process to be suspended until either

- The amount of wall clock time specified by seconds has elapsed.
- A signal is caught by the process and the signal handler returns.

As with an alarm signal, the actual return may occur at a time later than requested because of other system activity.

In case 1, the return value is 0. When sleep returns early because of some signal being caught (case 2), the return value is the number of unslept seconds (the requested time minus the actual time slept).

Although sleep can be implemented with the alarm function, this isn't required. If an alarm is used, however, there can be interactions between the two functions. The POSIX.1 standard leaves all these interactions unspecified.

nanosleep

The nanosleep function is similar to the sleep function but provides nanosecond-level granularity.

```
#include <time.h>
int nanosleep(const struct timespec *reqtp,struct timespec *remtp);
```

It returns 0 if slept for requested time or −1 on error.

This function suspends the calling process until either the requested time has elapsed or the function is interrupted by a signal. The reqtp parameter specifies the amount of time to sleep in seconds and nanoseconds. If the sleep interval is interrupted by a signal and the process doesn't terminate, the timespec structure pointed to by the remtp parameter will be set to the amount of time left in the sleep interval. We can set this parameter to NULL if we are uninterested in the time unslept.

If the system doesn't support nanosecond granularity, the requested time is rounded up. Because the nanosleep function doesn't involve the generation of any signals, we can use it without worrying about interactions with other functions.

clock_nanosleep

We need a way to suspend the calling thread using a delay time relative to a particular clock. The clock_nanosleepfunction provides us with this capability.

```
#include <time.h>
int clock_nanosleep(clockid_t clock_id, int flags, const struct timespec *reqtp, struct timespec *remtp);
```

It returns 0 if slept for requested time or error number on failure.

The clock_id argument specifies the clock against which the time delay is evaluated. Identifiers for clocks are listed in Figure6.8. The flags argument is used to control whether the delay is absolute or relative. When flags is set to 0, the sleep time is relative (i.e., how long we want to sleep). When it is set to TIMER_ABSTIME, the sleep time is absolute (i.e., we want to sleep until the clock reaches the specified time).

The other arguments, reqtp, and remtp, are the same as in the nanosleep function. However, when we use an absolute time, the remtp argument is unused, because it isn't needed; we can reuse the same value for the reqtp argument for additional calls to clock_nanosleepuntil the clock reaches the specified absolute time value.

Note that except for error returns, the call clock_nanosleep(CLOCK_REALTIME, 0, reqtp, remtp); has the same effect as the call nanosleep(reqtp, remtp);

Unit 3

IPC: Introduction to IPC, Pipes, FIFOs, Introduction to three types of IPC – message queues, semaphores and shared memory.
Message Queues: Kernel support for messages, Unix system V APIs for messages, client/server example.
Semaphores: Kernel support for semaphores, Unix system V APIs for semaphores.

Unnamed pipes and FIFOs(named pipes)

A pipe is an important mechanism in Unix-based systems that allows us to communicate data from one process to another without storing anything on the disk. In Linux, we have two types of pipes: pipes (also known as anonymous or unnamed pipes) and FIFO's (also known as named pipes).

Unnamed Pipes	Named Pipes
As suggested by their names, a named type has a specific name that can be given to it by the user. Named pipe if referred through this name only by the reader and writer. All instances of a named pipe share the same pipe name.	Unnamed pipes is not given a name. It is accessible through two file descriptors that are created through the function pipe(fd[2]), where fd[1] signifies the write file descriptor, and fd[0] describes the read file descriptor.
An unnamed pipe is only used for communication between a child and its parent process	A named pipe can be used for communication between two unnamed processes as well. Processes of different ancestry can share data through a named pipe.
An unnamed pipe vanishes as soon as it is closed, or one of the processes (parent or child) completes execution.	A named pipe exists in the file system. After input-output has been performed by the sharing processes, the pipe still exists in the file system independently of the process and can be used for communication between some other processes.
Unnamed pipes are always local; they cannot be used for communication over a network.	Named pipes can be used to provide communication between processes on the same computer or between processes on different computers across a network, as in the case of a distributed system.
An unnamed pipe is a one-way pipe that typically transfers data between a parent process and a child process.	A Named pipe can have multiple processes communicating through it, like multiple clients connected to one server.

Message Queues

Message queues provide a reasonably easy and efficient way of passing data between two unrelated processes. They have the advantage over named pipes in that the message queue exists independently of both the sending and receiving processes, which

removes some of the difficulties that occur in synchronizing the opening and closing of named pipes.

Message queues provide a way of sending a block of data from one process to another. Additionally, each block of data is considered to have a type, and a receiving process may receive blocks of data having different type values independently.

The good news is that you can almost totally avoid the synchronization and blocking problems of named pipes by sending messages. Even better, you can “look ahead” for messages that are urgent in some way. The bad news is that, just like pipes, there’s a maximum size limit imposed on each block of data and also a limit on the maximum total size of all blocks on all queues throughout the system.

```
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
int msgget(key_t key, int msgflg);
int msgrcv(int msqid, void *msg_ptr, size_t msg_sz, long int msgtype, int msgflg);
int msgsnd(int msqid, const void *msg_ptr, size_t msg_sz, int msgflg);
```

msgget

You create and access a message queue using the msgget function:

```
int msgget(key_t key, int msgflg);
```

The program must provide a key-value that, as with other IPC facilities, names a particular message queue. The special value IPC_PRIVATE creates a private queue, which in theory is accessible only by the current process. As with semaphores and messages, on some Linux systems, the message queue may not actually be private. Because a private queue has very little purpose, that’s not a significant problem.

The second parameter, msgflg, consists of nine permission flags. A special bit defined by IPC_CREAT must be bitwise ORed with the permissions to create a new message queue. The IPC_CREAT flag is silently ignored if the message queue already exists.

The msgget function returns a positive number, the queue identifier, on success or –1 on failure.

msgsnd

The msgsnd function allows you to add a message to a message queue:

```
int msgsnd(int msqid, const void *msg_ptr, size_t msg_sz, int msgflg);
```

The structure of the message is constrained in two ways. First, it must be smaller than the system limit, and second, it must start with a long int, which will be used as a message type in the receive function. When you’re using messages, it’s best to define your message structure something like this:

```
struct my_message {
long int message_type; /* The data you wish to transfer */
}
```

The first parameter, msqid, is the message queue identifier returned from a msgget function.

The second parameter, msg_ptr, is a pointer to the message to be sent, which must start with a long int type as described previously.

The third parameter, msg_sz, is the size of the message pointed to by msg_ptr. This size must not include the long int message type.

The fourth parameter, msgflg, controls what happens if either the current message queue is full or the systemwide limit on queued messages has been reached. If msgflg has the IPC_NOWAIT flag set, the function will return immediately without sending the message and the return value will be –1. If the msgflg has the IPC_NOWAIT flag clear, the sending process will be suspended, waiting for space to become available in the queue.

On success, the function returns 0, on failure –1. If the call is successful, a copy of the message data has been taken and placed on the message queue.

msgrcv

The msgrcv function retrieves messages from a message queue:

int msgrcv(int msqid, void *msg_ptr, size_t msg_sz, long int msgtype, int msgflg);

The first parameter, msqid, is the message queue identifier returned from a msgget function.

The second parameter, msg_ptr, is a pointer to the message to be received, which must start with a long int type as described previously in the msgsnd function.

The third parameter, msg_sz, is the size of the message pointed to by msg_ptr, not including the long int message type.

The fourth parameter, msgtype, is a long int, which allows a simple form of reception priority to be implemented. If msgtype has the value 0, the first available message in the queue is retrieved. If it’s greater than zero, the first message with the same message type is retrieved. If it’s less than zero, the first message that has a type the same as or less than the absolute value of msgtype is retrieved.

If you simply want to retrieve messages in the order in which they were sent, set msgtype to 0. If you want to retrieve only messages with a specific message type, set msgtype equal to that value. If you want to receive messages with a type of n or smaller, set msgtype to -n.

The fifth parameter, msgflg, controls what happens when no message of the appropriate type is waiting to be received. If the IPC_NOWAIT flag in msgflg is set, the call will return immediately with a return value of –1. If the IPC_NOWAIT flag of msgflg is clear, the process will be suspended, waiting for an appropriate type of message to arrive.

On success, msgrcv returns the number of bytes placed in the receive buffer, the message is copied into the user-allocated buffer pointed to by msg_ptr, and the data is deleted from the message queue. It returns –1 on error.

msgctl

The final message queue function is msgctl, which is very similar to that of the control function for shared memory:

```
int msgctl(int msqid, int command, struct msqid_ds *buf);
```

The msqid_ds structure has at least the following members:

```
struct msqid_ds {
        uid_t  msg_perm.uid;
        uid_t  msg_perm.gid;
        mode_t msg_perm.mode;
}
```

The first parameter, msqid, is the identifier returned from msgget.

The second parameter, command, is the action to take. It can take three values:

- IPC_STAT Sets the data in the msqid_ds structure to reflect the values associated with the message queue.
- IPC_SET If the process has permission to do so, this sets the values associated with the message queue to those provided in the msqid_ds data structure.
- IPC_RMID Deletes the message queue.

0 is returned on success, –1 on failure. If a message queue is deleted while a process is waiting in an msgsnd or msgrcv function, the send or receive function will fail.

Semaphores

When you write programs that use threads operating in multiuser systems, multiprocessing systems, or a combination of the two, you may often discover that you have critical sections of code, where you need to ensure that a single process (or a single thread of execution) has exclusive access to a resource.

Semaphores have a complex programming interface. Fortunately, you can easily provide a much-simplified interface that is sufficient for most semaphore-programming problems.

To prevent problems caused by more than one program simultaneously accessing a shared resource, you need a way of generating and using a token that grants access to only one thread of execution in a critical section at a time.

One possible solution that you've already seen is to create files using the O_EXCL flag with the open function, which provides atomic file creation. This allows a single process to succeed in obtaining a token: the newly created file. This method is fine for simple problems but rather messy and very inefficient for more complex examples.

A more formal definition of a semaphore is a special variable on which only two operations are allowed; these operations are officially termed wait and signal. Because “wait” and “signal” already have special meanings in Linux programming, we’ll use the original notation:

- P(semaphore variable) for wait
- V(semaphore variable) for signal

Definition

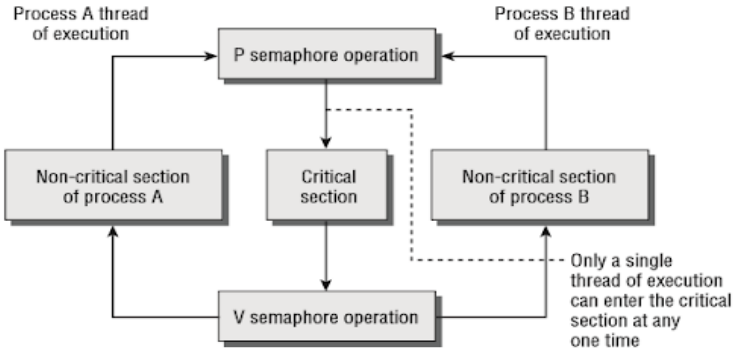
The simplest semaphore is a variable that can take only the values 0 and 1, a binary semaphore. This is the most common form. Semaphores that can take many positive values are called general semaphores. Here we concentrate on binary semaphores.

The definitions of P and V are surprisingly simple. Suppose you have a semaphore variable sv. The two operations are then defined as follows:

- P(sv) If sv is greater than zero, decrement sv. If sv is zero, suspend execution of this process.
- V(sv) If some other process has been suspended waiting for sv, make it resume execution. If no process is suspended waiting for sv, increment sv.

Pseudocode

```
semaphore sv = 1;
loop forever
{
        P(sv);
        critical code section;
        V(sv);
        noncritical code section;
}
```



Linux Semaphore Facilities

The semaphore function definitions are

```
#include <sys/sem.h>
int semctl(int sem_id, int sem_num, int command, ...);
int semget(key_t key, int num_sems, int sem_flags);
int semop(int sem_id, struct sembuf *sem_ops, size_t num_sem_ops);
```

semget

The semget function creates a new semaphore or obtains the semaphore key of an existing semaphore:

```
int semget(key_t key, int num_sems, int sem_flags);
```

The first parameter, key, is an integral value used to allow unrelated processes to access the same semaphore. All semaphores are accessed indirectly by the program supplying a key, for which the system then generates a semaphore identifier. The semaphore key is used only with semget. All other semaphore functions use the semaphore identifier returned from semget.

There is a special semaphore key-value, IPC_PRIVATE, that is intended to create a semaphore that only the creating process could access, but this rarely has any useful purpose. You should provide a unique, non-zero integer value for a key when you want to create a new semaphore.

The num_sems parameter is the number of semaphores required. This is almost always 1.

The sem_flags parameter is a set of flags, very much like the flags to the open function. The lower nine bits are the permissions for the semaphore, which behave like file permissions. Also, these can be bitwise ORed with the value IPC_CREAT to create a new semaphore. It will return an error if the semaphore already exists.

The semget function returns a positive (nonzero) value on success; this is the semaphore identifier used in the other semaphore functions. On error, it returns –1.

semop

The function semop is used for changing the value of the semaphore:

```
int semop(int sem_id, struct sembuf *sem_ops, size_t num_sem_ops);
```

The first parameter, sem_id, is the semaphore identifier, as returned from semget. The second parameter, sem_ops, is a pointer to an array of structures, each of which will have at least the following members:

```
struct sembuf {
        short sem_num;
        short sem_op;
        short sem_flg;
}
```

The first member, sem_num, is the semaphore number, usually 0 unless you're working with an array of semaphores. The sem_op member is the value by which the semaphore should be changed. In general, only two values are used, –1, which is your P operation to wait for a semaphore to become available, and +1, which is your V operation to signal that a semaphore is now available.

The final member, sem_flg, is usually set to SEM_UNDO. This causes the operating system to track the changes made to the semaphore by the current process and, if the process terminates without releasing the semaphore, allows the operating system to automatically release the semaphore if it was held by this process. It's good practice to set sem_flg to SEM_UNDO unless you specifically require different behavior.

semctl

The semctl function allows direct control of semaphore information:

```
int semctl(int sem_id, int sem_num, int command, ...);
```

The first parameter, sem_id, is a semaphore identifier, obtained from semget. The sem_num parameter is the semaphore number. You use this when you're working with arrays of semaphores. Usually, this is 0, the first and only semaphore. The command parameter is the action to take, and a fourth parameter, if present, is a union semun, which according to the X/OPEN specification must have at least the following members:

```
union semun {
        int val;
        struct semid_ds *buf;
        unsigned short *array;
}
```

The two common values of command are:

- SETVAL: Used for initializing a semaphore to a known value. The value required is passed as the val member of the union semun. This is required to set the semaphore up before it's used for the first time.
- IPC_RMID: Used for deleting a semaphore identifier when it's no longer required.

The semctl function returns different values depending on the command parameter. For SETVAL and IPC_RMID it returns 0 for success and –1 on error.

Unit 4

Shared Memory: Kernel support for shared memory, Unix system V APIs for shared memory, semaphore and shared memory example.
Multithreaded Programming: Differences between threads and processes, Thread structure and uses. Threads and Lightweight Processes, POSIX Thread APIs, Creating Threads. Thread Attributes. Thread Synchronization with semaphores and with Mutexes, Example programs.

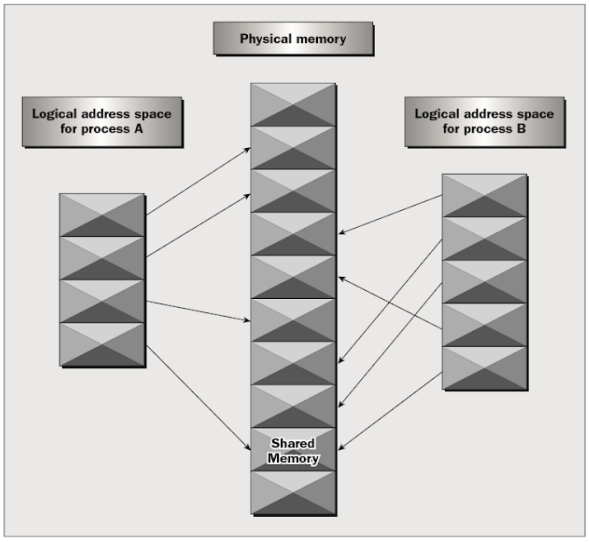
Shared Memory

Shared memory is the second of the three IPC facilities. It allows two unrelated processes to access the same logical memory. Shared memory is a very efficient way of transferring data between two running processes.

Shared memory is a special range of addresses that are created by IPC for one process and appear in the address space of that process. Other processes can then “attach” the same shared memory segment into their own address space. All processes can access the memory locations just as if the memory had been allocated by malloc. If one process writes to the shared memory, the changes immediately become visible to any other process that has access to the same shared memory.

Shared memory provides an efficient way of sharing and passing data between multiple processes. By itself, shared memory doesn’t provide any synchronization facilities. Because it provides no synchronization facilities, you usually need to use some other mechanism to synchronize access to the shared memory.

There are no automatic facilities to prevent a second process from starting to read the shared memory before the first process has finished writing to it. It’s the responsibility of the programmer to synchronize access.



The arrows show the mapping of the logical address space of each process to the physical memory available. In practice, the situation is more complex because the available

memory consists of a mix of physical memory and memory pages that have been swapped out to disk.

The functions for shared memory resemble those for semaphores:

```
#include <sys/shm.h>
void *shmat(int shm_id, const void *shm_addr, int shmflg);
int shmctl(int shm_id, int cmd, struct shmid_ds *buf);
int shmdt(const void *shm_addr);
int shmget(key_t key, size_t size, int shmflg);
```

shmget

You create shared memory using the shmget function:

```
int shmget(key_t key, size_t size, int shmflg);
```

As with semaphores, the program provides a key, which effectively names the shared memory segment, and the shmget function returns a shared memory identifier that is used in subsequently shared memory functions. There's a special key-value, IPC_PRIVATE, that creates shared memory private to the process.

The second parameter, size, specify the amount of memory required in bytes.

The third parameter, shmflg, consists of nine permission flags that are used in the same way as the mode flags for creating files. A special bit defined by IPC_CREAT must be bitwise ORed with the permissions to create a new shared memory segment.

The permission flags are very useful with shared memory because they allow a process to create shared memory that can be written by processes owned by the creator of the shared memory but only read by processes that other users have created. You can use this to provide efficient read-only access to data by placing it in shared memory without the risk of it being changed by other users.

If the shared memory is successfully created, shmget returns a non-negative integer, the shared memory identifier. On failure, it returns –1.

shmat

When you first create a shared memory segment, it's not accessible by any process. To enable access to the shared memory, you must attach it to the address space of a process. You do this with the shmat function:

```
void *shmat(int shm_id, const void *shm_addr, int shmflg);
```

The first parameter, shm_id, is the shared memory identifier returned from shmget.

The second parameter, shm_addr, is the address at which the shared memory is to be attached to the current process. This should almost always be a null pointer, which allows the system to choose the address at which the memory appears.

The third parameter, shmflg, is a set of bitwise flags. The two possible values are SHM_RND, which, in conjunction with shm_addr, controls the address at which the shared memory is attached, and SHM_RDONLY, which makes the attached memory read-only.

If the shmat call is successful, it returns a pointer to the first byte of shared memory. On failure –1 is returned.

The shared memory will have read or write access depending on the owner (the creator of the shared memory), the permissions, and the owner of the current process. Permissions on shared memory are similar to the permissions on files.

An exception to this rule arises if shmflg & SHM_RDONLY is true. Then the shared memory won't be writable, even if permissions would have allowed write access.

shmdt

The shmdt function detaches the shared memory from the current process. It takes a pointer to the address returned by shmat. On success, it returns 0, on error –1. Note that detaching the shared memory doesn't delete it; it just makes that memory unavailable to the current process.

shmctl

The control functions for shared memory are (thankfully) somewhat simpler than the more complex ones for semaphores:

```
int shmctl(int shm_id, int command, struct shmid_ds *buf);
```

The shmid_ds structure has at least the following members:

```
struct shmid_ds {
        uid_t shm_perm.uid;
        uid_t shm_perm.gid;
        mode_t shm_perm.mode;
}
```

The first parameter, shm_id, is the identifier returned from shmget.

The second parameter, command, is the action to take. It can take three values:

1. IPC_STAT: Sets the data in the shmid_ds structure to reflect the values associated with the shared memory.
2. IPC_SET: Sets the values associated with the shared memory to those provided in the shmid_ds data structure if the process has permission to do so.
3. IPC_RMID: Deletes the shared memory segment.

The third parameter, buf, is a pointer to the structure containing the modes and permissions for the shared memory

Unit 5

Sockets: Introduction to Sockets, Socket Addresses, Socket system calls for connection oriented protocol and connectionless protocol, example, client/server programs.
Advanced I/O: Introduction, Non-Blocking I/O, Record Locking, I/O Multiplexing, select and pselect Functions, Poll Function, Asynchronous I/O, POSIX Asynchronous I/O readv and writev functions, readn and written-functions, Memory-Mapped I/O

Introduction to Sockets

A socket is an abstraction of a communication endpoint. Just as they would use file descriptors to access files, applications use socket descriptors to access sockets. Socket descriptors are implemented as file descriptors in the UNIX System. Indeed, many of the functions that deal with file descriptors, such as read and write, will work with a socket descriptor. To create a socket, we call the socket function.

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

It returns file (socket) descriptor if OK, −1 on error. The domain argument determines the nature of the communication, including the address format. The constants start with AF_ because each domain has its own format for representing an address.

- AF_INET IPv4: Internet domain
- AF_INET6 IPv6: Internet domain
- AF_UNIX UNIX: domain
- AF_UNSPEC: unspecified

The type argument determines the type of the socket, which further determines the communication characteristics. The socket types defined by POSIX.1 are summarized below, but implementations are free to add support for additional types.

- SOCK_DGRAM: fixed-length, connectionless, unreliable messages
- SOCK_RAW: datagram interface to IP
- SOCK_SEQPACKET: fixed-length, sequenced, reliable, connection-oriented messages
- SOCK_STREAM: sequenced, reliable, bidirectional, connection-oriented byte streams

The protocol argument is usually zero, to select the default protocol for the given domain and socket type. When multiple protocols are supported for the same domain and socket type, we can use the protocol argument to select a particular protocol. The default protocol for a SOCK_STREAM socket in the AF_INET communication domain is
TCP (Transmission Control Protocol). The default protocol for a SOCK_DGRAM socket in the AF_INET communication domain is UDP (User Datagram Protocol).

- IPPROTO_IP: IPv4 Internet Protocol
- IPPROTO_IPV6: IPv6 Internet Protocol (optional in POSIX.1
- IPPROTO_ICMP: Internet Control Message Protocol
- IPPROTO_RAW: Raw IP packets protocol (optional in POSIX.1)
- IPPROTO_TCP: Transmission Control Protocol
- IPPROTO_UDP: User Datagram Protocol

Communication on a socket is bidirectional. We can disable I/O on a socket with the shutdown function.

```
#include <sys/socket.h>
int shutdown(int sockfd,int how);
```

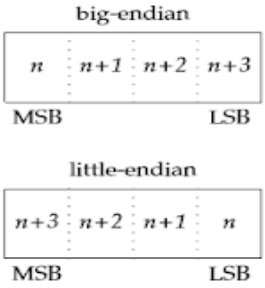
If how isSHUT_RD, then reading from the socket is disabled. If how isSHUT_WR, then we can't use the socket for transmitting data. We can use SHUT_RDWR to disable both data transmission and reception. It returns 0 if OK, −1 on error.

Socket Addresses

The machine's network address helps us identify the computer on the network we wish to contact, and the service, represented by a port number, helps us identify the particular process on the computer.

Byte Ordering

When communicating with processes running on the same computer, we generally don't have to worry about byte ordering. The byte order is a characteristic of the processor architecture, dictating how bytes are ordered within larger data types, such as integers. The below figure shows how the bytes within a 32-bit integer are numbered.



Operating system	Processor architecture	Byte order
FreeBSD 8.0	Intel Pentium	little-endian
Linux 3.2.0	Intel Core i5	little-endian
Mac OS X 10.6.8	Intel Core 2 Duo	little-endian
Solaris 10	Sun SPARC	big-endian

Four functions are provided to convert between the processor byte order and the network byte order for TCP/IP applications.

```
#include <arpa/inet.h>
uint32_t htonl(uint32_t hostint32);   /* Returns: 32-bit integer in network byte order */
uint16_t htons(uint16_t hostint16); /* Returns: 16-bit integer in network byte order  */
uint32_t ntohl(uint32_t netint32);   /* Returns: 32-bit integer in host byte order  */
uint16_t ntohs(uint16_t netint16);  /* Returns: 16-bit integer in host byte order  */
```

Address Formats

An address identifies a socket endpoint in a particular communication domain. The address format is specific to the particular domain. So that addresses with different formats can be passed to the socket functions, the addresses are cast to a generic sockaddr address structure:

```
struct sockaddr {
        sa_family_t sa_family; /* address-family */
        char sa_data[]; /* variable-length address */
          .
          .
          .
};
```

Internet addresses are defined in <netinet/in.h>. In the IPv4 Internet domain (AF_INET), a socket address is represented by a sockaddr_instructure:

```
struct in_addr {
        in_addr_t s_addr; /* IPv4 address */
};
struct sockaddr_in {
        sa_family_t sin_family; /* address family */
        in_port_t sin_port; /* port number */
        struct in_addr sin_addr; /* IPv4 address */
};
```

The in_port_t data type is defined to be a uint16_t. The in_addr_t data type is defined to be a uint32_t. These integer data types specify the number of bits in the data type and aredefined in<stdint.h>.

In contrast to the AF_INET domain, the IPv6 Internet domain (AF_INET6)socket address is represented by a sockaddr_in6structure:

```
struct in6_addr {
        uint8_t s6_addr[16]; /* IPv6 address */
};
struct sockaddr_in6 {
        sa_family_t sin6_family; /* address family */
        in_port_t sin6_port; /* port number */
        uint32_t sin6_flowinfo; /* traffic class and flow info */
        struct in6_addr sin6_addr; /* IPv6 address */
        uint32_t sin6_scope_id; /* set of interfaces for scope */
};
```

Socket system calls for connection oriented protocol and connectionless protocol

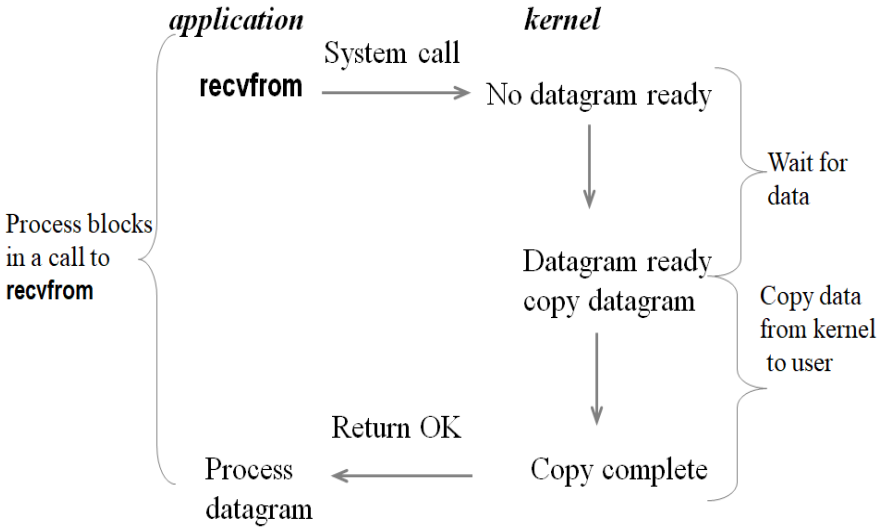
https://www.cse.iitk.ac.in/users/dheeraj/cs425/lec18.html

Blocking & Non-Blocking I/O

System calls are divided into two categories: the ‘‘slow’’ ones and all the others. The slow system calls are those that can block forever. They include

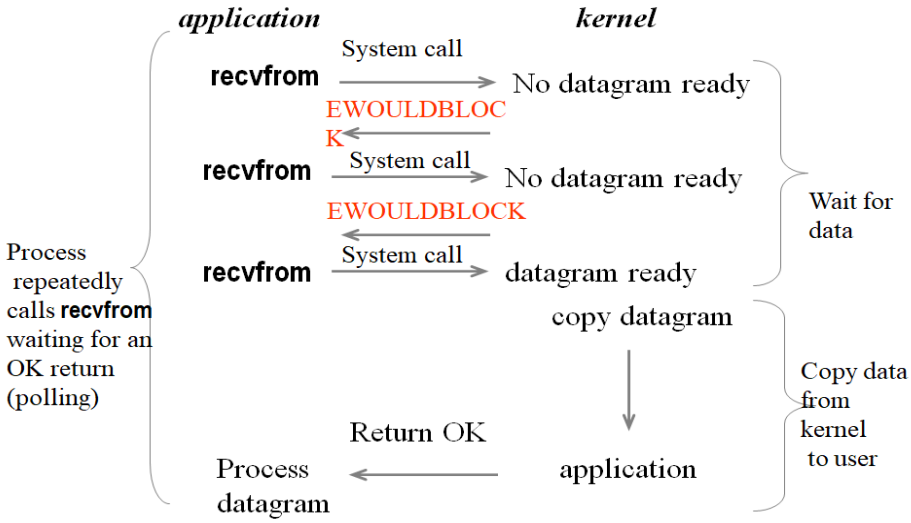
- Reads that can block the caller forever if data isn’t present with certain file types (pipes, terminal devices, and network devices)
- Writes that can block the caller forever if the data can’t be accepted immediately by these same file types (e.g., no room in the pipe, network flow control)

- Opens that block until some condition occurs on certain file types (such as an open of a terminal device that waits until an attached modem answers the phone, or an open of a FIFO for writing only, when no other process has the FIFO open for reading)
- Reads and writes of files that have mandatory record locking enabled.
- Certainioctloperations
- Some of the interprocess communication functions



Nonblocking I/O lets us issue an I/O operation, such as an open, read, or write, and not have it block forever. If the operation cannot be completed, the call returns immediately with an error noting that the operation would have blocked. There are two ways to specify nonblocking I/O for a given descriptor.

- If we call open to get the descriptor, we can specify the O_NONBLOCK flag.
- For a descriptor that is already open, we call fcntl to turn on the O_NONBLOCK file status flag.



Example for non-blocking

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
```

```
char buf[500000];
int main(void)
{
	int ntowrite, nwrite;
	char *ptr;
	ntowrite = read(STDIN_FILENO, buf, sizeof(buf));
	fprintf(stderr, "read %d bytes\n", ntowrite);
	set_fl(STDOUT_FILENO, O_NONBLOCK); /* set nonblocking */
	ptr = buf;
	while (ntowrite > 0)
	{
		errno = 0;
		nwrite = write(STDOUT_FILENO, ptr, ntowrite);
		fprintf(stderr, "nwrite = %d, errno = %d\n", nwrite, errno);
		if (nwrite > 0)
			ptr += nwrite; ntowrite -= nwrite;
	}
	clr_fl(STDOUT_FILENO, O_NONBLOCK); /* clear nonblocking */
	exit(0);
}
```