



Atria Institute of Technology
Department of Information Science and Engineering
Bengaluru-560024



ACADEMIC YEAR: 2021-2022
ODD SEMESTER NOTES

Semester : 5th Semester

Subject Name : Unix Programming

Subject Code : 18CS56

Faculty Name : Mrs. Kavitha Vasanth

Module-1

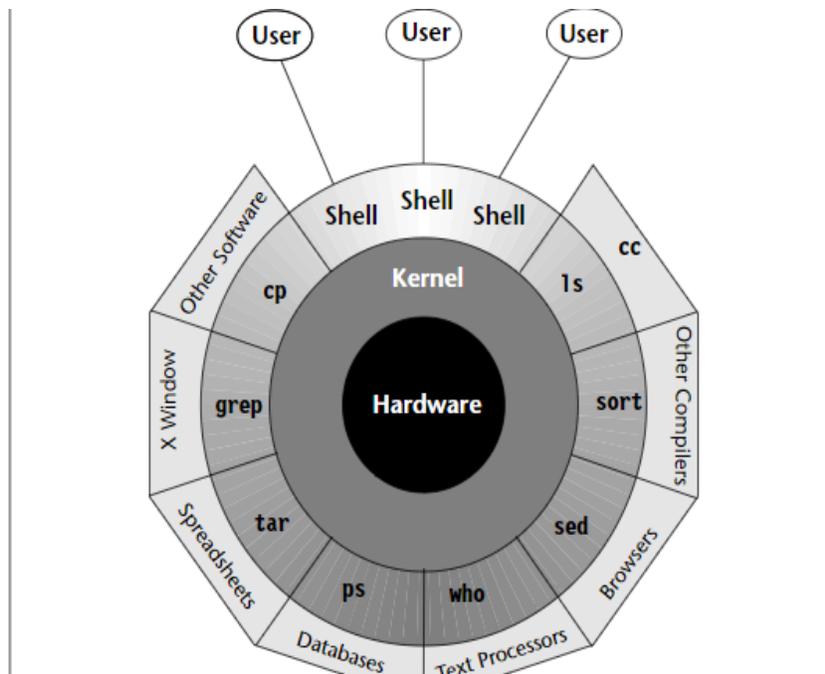
Introduction

Unix Components/Architecture

The success of UNIX, according to Thompson and Ritchie, “lies not so much in new inventions but rather in the full exploitation of a carefully selected set of fertile ideas, and especially in showing that they can be keys to the implementation of a small and yet powerful operating system.

Need to understand its software architecture – its foundation.

1. Division of labor: kernel and shell
2. The file and process
3. The system calls



Division of Labor: Kernel and Shell

the division of labor between two agencies—the kernel and the shell. The kernel interacts with the machine’s hardware, and the shell with the user. You have seen both of them in

action in the hands-on session, though the kernel wasn't mentioned by name. Their relationship is depicted in Fig.

The kernel is the core of the operating system. The system's bootstrap program (a small piece of program code) loads the kernel into memory at startup. The kernel comprises a set of routines mostly written in C that communicate with the hardware directly. User programs (the applications) that need to access the hardware (like the hard disk or the terminal) communicate with the kernel using a set of functions called system calls. The kernel has work to do even if no user program is running. It is often called the operating system—a program's gateway to the computer's resources.

Computers don't have any inherent ability to translate user commands into action. That requires an interpreter, and that job in UNIX is handled by the "outer part" of the operating system—the shell. It is actually the interface between the user and the kernel. Even though there's only one kernel running on the system, there could be several shells in action—one for each user who is logged in. When you enter a command through the keyboard, the shell thoroughly examines the keyboard input for special characters. If it finds any, it rebuilds a simplified command line, and finally communicates with the kernel to see that the command is executed.

The File and Process

Two simple entities support the UNIX system — the file and process - "Files have places and processes have life." Files are containers for storing static information. Even directories and devices are considered files. A file is related to another file by being part of a single hierarchical structure called the file system. The second entity is the process, which represents a program in execution. Like files, processes also form a hierarchy, and are best understood when we consider one process as the child of another.

The System Calls

The UNIX system—comprising the kernel, shell, and applications—is written in C. Though there are over a thousand different commands in the system, they often need to

carry out certain common tasks—like reading from or writing to disk. The code for performing disk I/O operations is not built into the programs but is available in the kernel. Programs access these kernel services by invoking special functions called system calls. Often the same system call can access both a file and a device; the open system call opens both.

POSIX specifies the system calls that all UNIX systems must implement. Once software has been developed on one UNIX system using the calls mandated by POSIX, it can be easily moved to another UNIX machine.

Features of Unix

The following sections present the major features of this operating system.

1. A Multiuser System
2. Multitasking System
3. The Building-Block Approach
4. Unix toolkit
5. Pattern Matching
6. Programming Facility
7. Documentation

A Multiuser System – UNIX is a multiprogramming system. It permits multiple programs to remain in memory and compete for the attention of the CPU. These programs can be run by different users; UNIX is also a multiuser system. This feature often baffles Windows users as Windows is essentially a single-user system where the CPU, memory, and hard disk are all dedicated to a single user.

Multitasking System – UNIX is a multitasking system. It is common for a user to edit a file, print another one on the printer, send email to a friend, and browse the World Wide Web—all without leaving any of the applications. The X Window system exploits the multitasking feature by allowing you to open multiple windows on your desktop. In a multitasking environment, a user sees one job running in the foreground; the rest run in the background. You can switch jobs between background and foreground, suspend, or even terminate them.

The Building-Block Approach – A complex task can be broken into a finite number of simple ones. The shell offers a mechanism called the pipe that allows the output of one command to serve as input to another. To take advantage of this feature a special set of commands (called filters) were designed where each command did “one thing well.” By interconnecting these tools using the piping mechanism, you can solve very complex text manipulation problems.

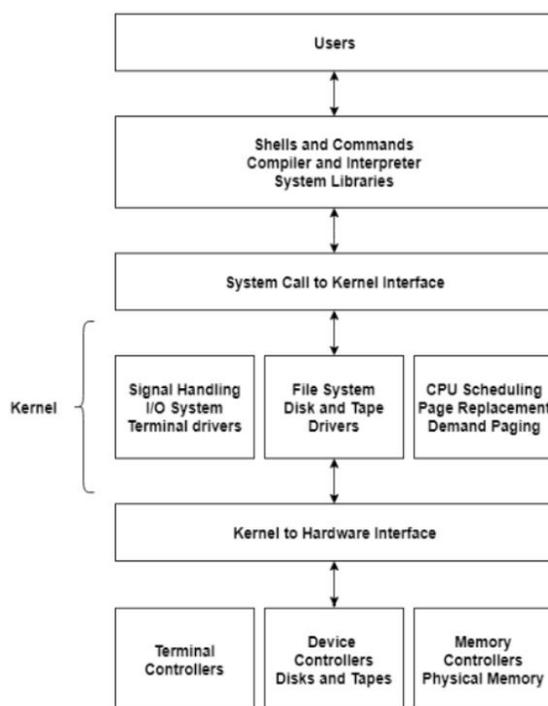
Unix Toolkit – New tools are being added and the older ones are being removed or modified. The shell and utilities form part of the POSIX specification. There are open-source version for most of these utilities.

Pattern Matching – Many commands use filenames as arguments, and these filenames often have a common string. For instance, all C programs have the .c extension, and to back them up to tape with the tar command, we need not specify all of their filenames to tar. Instead, we can simply use a pattern *.c. The * is a special character (known as a metacharacter) that is used by the shell to match a number of characters.

Programming Facility – The UNIX shell is also a programming language; it was designed for a programmer, not a casual end user. It has all the necessary ingredients, like control structures, loops, and variables, that establish it as a powerful programming language in its own right. These features are used to design shell scripts—programs that run UNIX commands in a batch.

Documentation – The principal online help facility available is the man command, which remains the most important reference for commands and their configuration files. Apart from the online ocean of the Internet. on UNIX queries in case problem.

Unix Structure



documentation, there’s a vast UNIX resources available on the Internet. There are several newsgroups where you can post your queries in case you are stranded with a

Environment and Unix

Unix is a multiuser, multitasking operating system that was developed by Bell Laboratories in 1969. In a multiuser system, many users can use the system simultaneously. A multitasking system is capable of doing multiple jobs. Each user interacts with their own shell instance in this type of operating system and can start applications as required.

Kernel – The kernel provides a bridge between the hardware and the user. It is a software application that is central to the operating system. The kernel handles the files, memory, devices, processes and the network for the operating system. It is the responsibility of the kernel to make sure all the system and user tasks are performed correctly.

Shell - The program between the user and the kernel is known as the shell. It translates the many commands that are typed into the terminal session. These commands are known as the shell script. There are two major types of shells in Unix. These are Bourne shell and C Shell. The Bourne shell is the default shell for version 7 Unix. The character \$ is the default prompt for the Bourne shell. The C shell is a command processor that is run in a text window. The character % is the default prompt for the C shell.

Applications - The applications and utility layer in Unix includes the word processors, graphics programs, database management programs, commands etc. The application programs provide an application to the end users. For example, a web browser is used to find information while gaming software is used to play games. The requests for service and application communication systems used in an application by a programmer is known as an application program interface (API).

Posix and Single Unix Specification

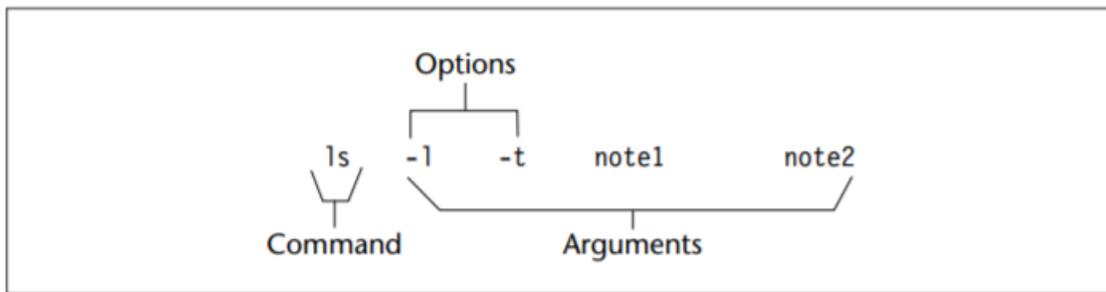
The Dennis Ritchie rewrote UNIX in C to make it portable, that didn't quite happen. UNIX fragmentation and the absence of a single conforming standard adversely affected the development of portable applications. To address the issue, AT&T created the System V Interface Definition (SVID). Later, X/Open (now The Open Group), a consortium of vendors and users, created the X/Open Portability Guide (XPG). Products conforming to this specification were branded UNIX95, UNIX98, or UNIX03 depending on the version of the specification.

Two of the most-cited standards from the POSIX family are known as POSIX.1 and POSIX.2. POSIX.1 specifies the C application program interface—the system calls. POSIX.2 deals with the shell and utilities.

The joint initiative of X/Open and IEEE resulted in the unification of the two standards. This is the Single UNIX Specification, Version 3 (SUSV3) that is also known as IEEE 1003.1:2001 (POSIX.1). The “write once, adopt everywhere” approach to this development means that once software has been developed on any POSIX-compliant UNIX system, it can be easily ported to another POSIX-compliant UNIX machine with minimum modifications.

General features of Unix commands / command structure

A command is an instruction given by a user telling a computer to do something, such a run a single program or a group of linked programs.

FIGURE 2.1 *Structure of a UNIX Command*

commands that had multiple words (like `mkdir` scripts) and one that had an embedded minus sign (`ls -F`). It's time we subjected a typical UNIX command to a dissective study. The structure of such a command is shown in Fig. 2.1. This command sequence has five words. The first word is the command itself and the remaining ones are its arguments. The `ls` command is specified here with four arguments. Two of the arguments begin with a hyphen (`-l` and `-t`) and are appropriately called options. The entire line is referred to as the command line. A command line is executed only after you hit [Enter].

Command arguments and options

Options – `ls` is a Linux shell command that lists directory contents of files and directories.

Syntax:

`$ls [option] [filename]`

`ls -t`: It sorts the file by modification time, showing the last edited file first. `head -1` picks up this first file. To open the last edited file in the current directory, use the combination of `ls` and `head` commands.

`ls -l`: To show long listing information about the file/directory.

Filename Arguments – Many Unix commands use a filename as a argument so the command can take input from file. If a command uses a filename as argument at all, it will generally be its

last argument – and after all options. It's also quite common to see many commands working with multiple filenames as arguments:

```
ls -lat chap01 chap02 chap03
```

```
cp chap01 chap02      cp- copies files
```

Exception – there are commands (pwd) that don't accept any arguments, and some (who) that may or may not be specified with arguments. The ls command can run without arguments (ls), with only options (ls -l), with only filenames (ls chap01 chap 02), or using a combination of both (ls -la chap01 chap02).

Basic Unix Commands such as echo, printf, ls, who, date, passwd, cal, combining commands.

echo command - echo command in linux is used to display line of text/string that are passed as an argument. This is a built-in command that is mostly used in shell scripts and batch files to output status text to the screen or a file.

Syntax:

echo [option] [string]

NOTE: -e here enables the interpretation of backslash escapes

1. \b: it removes all the spaces in between the text.
2. \c: suppress trailing new line with backspace interpreter '-e' to continue without emitting new line.
3. \n: this option creates new line from where it is used.
4. \t: this option is used to create horizontal tab spaces.
5. \r: carriage return with backspace interpreter '-e' to have specified carriage return in output.
6. \v: this option is used to create vertical tab spaces.
7. \a: alert return with backspace interpreter '-e' to have sound alert.
8. echo *: this command will print all files/folders, similar to ls command.

9. -n: this option is used to omit echoing trailing newline.

printf command - “*printf*” command in Linux is used to display the given string, number or any other format specifier on the terminal window. It works the same way as “printf” works in programming languages like C.

Syntax:

\$printf [-v var] format [arguments]

Note: *printf* can have format specifiers, escape sequences or ordinary characters.

Format Specifiers: The most commonly used printf specifiers are %s, %b, %d, %x and %f.

1. *%s* specifier: It is basically a string specifier for string output.
2. *%b* specifier: It is same as string specifier but it allows us to interpret escape sequences with an argument.
3. *%d* specifier: It is an integer specifier for showing the integral values.
4. *%f* specifier: It is used for output of floating point values.
5. *%x* specifier: It is used for output of lowercase hexadecimal values for integers and for padding the output

ls command - ls is a Linux shell command that lists directory contents of files and directories.

Syntax:

\$ls [option] [filename]

1. ls -t - Open Last Edited File
2. ls -l - Display One File Per Line
3. ls -l - Display All Information About Files/Directories
4. ls -lh - Display File Size in Human Readable Format
5. ls -ld - Display Directory Information
6. ls -lt - Order Files Based on Last Modified Time
7. ls -ltr - Order Files Based on Last Modified Time (In Reverse Order)
8. ls -a - Display Hidden Files

9. ls -R - Display Files Recursively
10. ls -i - Display File Inode Number
11. ls -q - Hide Control Characters
12. ls -n - Display File UID and GID
13. ls -F - Visual Classification of Files With Special Characters
14. ls --color=auto - Visual Classification of Files With Colors.

-rw-rw-r-- 1 maverick maverick 1176 Feb 16 00:19 1.c

1st Character – File Type: First character specifies the type of the file. In the example above the hyphen (-) in the 1st character indicates that this is a normal file. Following are the possible file type options in the 1st character of the ls -l output.

Field Explanation

1. - normal file
2. d : directory
3. s : socket file
4. l : link file

Field 1 – File Permissions: Next 9 character specifies the files permission. The every 3 characters specifies read, write, execute permissions for user(root), group and others respectively in order. Taking above example, -rw-rw-r-- indicates read-write permission for user(root) , read permission for group, and no permission for others respectively. If all three permissions are given to user(root), group and others, the format looks like -rwxrwxrwx

Field 2 – Number of links: Second field specifies the number of links for that file. In this example, 1 indicates only one link to this file.

Field 3 – Owner: Third field specifies owner of the file. In this example, this file is owned by username ‘maverick’.

Field 4 – Group: Fourth field specifies the group of the file. In this example, this file belongs to ”maverick” group.

Field 5 – Size: Fifth field specifies the size of file in bytes. In this example, ‘1176’ indicates the file size in bytes.

Field 6 – Last modified date and time: Sixth field specifies the date and time of the last modification of the file. In this example, ‘Feb 16 00:19’ specifies the last modification time of the file.

Field 7 – File name: The last field is the name of the file. In this example, the file name is l.c.

who command - who command is used to find out the following information are:

1. Time of last system boot
2. Current run level of the system
3. List of logged in users and more

Description: The who command is used to get information about currently logged in user on to system.

Syntax: \$who [options] [filename]

Example:

1. The who command displays the following information for each user currently logged in to the system if no option is provided :

1. Login name of the users
2. Terminal line numbers
3. Login time of the users in to system
4. Remote host name of the user

\$ who

hduser tty7 2018-03-18 19:08 (:0)

date command - date command is used to display the system date and time. date command is also used to set date and time of the system. By default the date command displays the date in the time zone on which unix/linux operating system is configured.

You must be the super-user (root) to change the date and time.

Syntax:

```
$date [OPTION]... [+FORMAT] date [-u|--utc|--universal]
[MMDDhhmm[[CC]YY][.ss]]
```

Options in date command are:

1. date (no option) – With no options, the date command displays the current date and time
2. -u Option – Displays the time in GMT (Greenwich Mean Time)/UTC(Coordinated Universal Time)time zone.
3. -date or -d Option – Displays the given date string in the format of date. But this will not affect the system's actual date and time value.
4. -s or --set Option – To set the system date and time -s or --set option is used.
5. --file or -f Option – This is used to display the date string present at each line of file in the date and time format.
6. -r Option – This is used to display the last modified timestamp of a datefile.

Cal command - If a user wants a quick view of calendar in Linux terminal, cal is the command for you. By default, cal command shows current month calendar as output.

cal command is a calendar command in Linux which is used to see the calendar of a specific month or a whole year.

Syntax: cal [[month] year]

Rectangular bracket means it is optional, so if used without option, it will display a calendar of current month and year.

1. cal: Shows current month calendar on the terminal.
2. cal 08 2000: Shows calendar of selected month and year.

3. `cal 2018`: Shows the whole calendar of the year.
4. `cal 2018 | more`: But year may not be visible in the same screen use `more` with `cal` use spacebar to scroll down.
5. `cal -3`: Shows calendar of previous, current and next month.

passwd command – `passwd` command in Linux is used to change the user account passwords. The root user reserves the privilege to change the password for any user on the system, while a normal user can only change the account password for his or her own account.

Syntax: `passwd [options] [username]`

```
hp@DESKTOP-:~$ passwd
Changing password for hp.
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
hp@DESKTOP-:~$
```

Combining commands – Combining two or more commands on the command line is also known as “command chaining”. We’ll show you different ways you can combine commands on the command line.

Option One: The Semicolon (;) Operator

The semicolon (;) operator allows you to execute multiple commands in succession, regardless of whether each previous command succeeds.

`$ls ; pwd ; whoami`

- You don’t have to put spaces between the semicolons and the commands, either.
- You can enter the three commands as `ls ; pwd ; whoami`. However, spaces make the combined command more readable, which is especially useful if you’re putting a combined command into a shell script.

Option Two: The Logical AND Operator (&&)

If you want the second command to only run if the first command is successful, separate the commands with the logical AND operator, which is two ampersands (&&).

For example, we want to make a directory called MyFolder and then change to that directory—provided it was successfully created. So, we type the following on the command line and press Enter.

```
$mkdir MyFolder && cd MyFolder
```

- The folder was successfully created, so the cd command was executed and we are now in the new folder.
- We recommend using the logical AND operator rather than the semicolon operator most of the time (;). This ensures that you don't do anything disastrous.

Option Three: The Logical OR Operator (||)

Sometimes you might want to execute a second command only if the first command does *not* succeed. To do this, we use the logical OR operator, or two vertical bars (||).

```
[$ -d ~/MyFolder ] || mkdir ~/MyFolder
```

Be sure there is a space after the first bracket and before the second bracket or the first command that checks if the directory exists will not work.

Combining Multiple Operators

You can combine multiple operators on the command line, too. For example, we want to first check if a file exists ([-f ~/sample.txt]). If it does, we print a message to the screen saying so (echo "File exists."). If not, we create the file (touch ~/sample.txt). So, we type the following at the command prompt and press Enter.

```
[$ -f ~/sample.txt ] && echo "File exists." || touch ~/sample.txt
```

Here's a useful summary of each of the operators used to combine commands:

- A ; B – Run A and then B, regardless of the success or failure of A

- A && B – Run B only if A succeeded
- A || B – Run B only if A failed

Meaning of Internal and External Command

The UNIX system is command-based *i.e* things happen because of the commands that you key in. All UNIX commands are seldom more than four characters long.

They are grouped into two categories:

1. **Internal Commands:** Commands which are built into the shell. For all the shell built-in commands, execution of the same is fast in the sense that the shell doesn't have to search the given path for them in the PATH variable, and also no process needs to be spawned for executing it.

Examples: source, cd, fg, etc.

2. **External Commands:** Commands which aren't built into the shell. When an external command has to be executed, the shell looks for its path given in the PATH variable, and also a new process has to be spawned and the command gets executed. They are usually located in /bin or /usr/bin. For example, when you execute the "cat" command, which usually is at /usr/bin, the executable /usr/bin/cat gets executed.

Examples: ls, cat etc.

For instance, take **echo** command:

\$type echo

echo is a shell built-in

How to find out whether a command is internal or external?

In addition to this you can also find out about a particular command *i.e*. whether it is internal or external with the help of **type** command :

\$type cat

cat is /bin/cat

//specifying that cat is external type//

\$type cd

cd is a shell built-in

//specifying that cd is internal type//

The type command: knowing the type of a command and locating it.

The **type** command is used to describe how its argument would be translated if used as commands. It is also used to find out whether it is built-in or external binary file.

Syntax: type [Options] command names

1. type -a - This option is used to find out whether it is an alias, keyword or a function and it also displays the path of an executable.
2. type -t - This option will display a single word as an output.

Example:

- type -t pwd (Print working directory)
 - type -t cp (copy)
 - type -t ls (listing the contents of directories)
3. type -p - This option displays the name of the disk file which would be executed by the shell. It will return nothing if the command is not a disk file.

Root Login and su command

The root is the user name or account that by default has access to all commands and files on a Linux or other Unix-like operating system. It is also referred to as the root account, root user, and the superuser.

PRIVILEGES AND PERMISSIONS

- Root privileges are the powers that the root account has on the system. The root account is the most privileged on the system and has absolute power over it (i.e., complete access to all files and commands). Among root's powers are the ability to modify the system in any way desired and to grant and revoke access permissions (i.e., the ability to read, modify and execute specific files and directories) for other users, including any of those that are by default reserved for root.
- The permissions system in Unix-like operating systems is set by default to prevent access by ordinary users to critical parts of the system and to files and directories belonging to other users. This is because it is very easy to damage a Unix-like system with root access. However, an important principle of Unix-like operating systems is the provision of maximum flexibility to configure the system, and thus the root user is fully empowered.
- **Linux superuser**
In Linux and Unix like computer operating systems, root is the conventional name of the user who has all rights or permissions (to all files and programs) in all modes (single- or multi-user). The root user can do many things an ordinary user cannot, such as changing the ownership of files and binding to ports numbered below 1024. The etymology of the term may be that root is the only user account with permission to modify the root directory of a Unix system.
- **Linux Login as Superuser**
You need to use any one of the following command to log in as superuser / root user on Linux:
 - su command – Run a command with substitute user and group ID in Linux
 - sudo command – Execute a command as another user on Linux.

How to become Superuser in Linux using su

Under Linux (and other Unixish operating systems) you use command called su. It is used is used to become another user during a login session or to login as super user. If invoked without a username, su defaults to becoming the super user. It is highly recommend that you use argument - to su command. It is used to provide an environment similar to what the user root

would expect had the user logged in directly. Type su command as follows:

```
$ su -
```

Sample outputs:

```
Password: <TYPE ROOT PASSWORD> #
```

Exiting from su

You simply need to type the following exit command or logout command:

```
$exit
```

OR

```
$logout
```

Unix Files

Naming of Files

A filename can consist of up to 255 characters. Files may or may not have extensions, and can consist of practically any ASCII character except the / and the NULL character (ASCII value 0). As a general rule you should avoid using unprintable characters in filenames. Further, since the shell has a special treatment for characters like \$, `, ?, *, & among others, it is recommended that only the following characters be used in filenames:

- Alphabetic characters and numerals.
- The period (.), hyphen (-), and underscore (_).

Example

All file names are case sensitive. So filename vivek.txt Vivek.txt VIVEK.txt all are three different files.

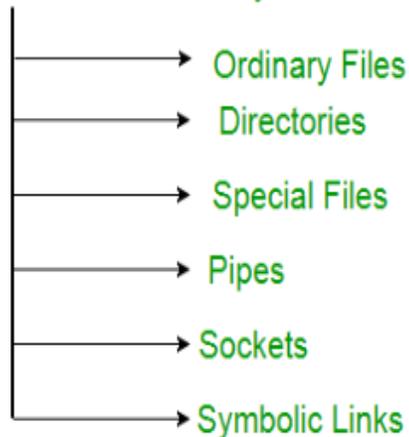
Most modern Linux and UNIX limit filename to 255 characters (255 bytes). However, some older version of UNIX system limits filenames to 14 characters only.

A filename must be unique inside its directory. For example, inside /home/vivek directory you cannot create a demo.txt file and demo.txt directory name. However, other directory may have files with the same names. For example, you can create demo.txt directory in /tmp.

Basic file types/categories

Types of Unix files – The UNIX files system contains several different types of files:

Classification of Unix File System :



- 1) Ordinary files - An ordinary file is a file on the system that contains data, text, or program instructions and used to store your information, such as some text you have written or an image you have drawn. This is the type of file that you usually work with and always located within/under a directory file.

In long-format output of `ls -l`, this type of file is specified by the “-” symbol.

- 2) Directories – Directories store both special and ordinary files. For users familiar with Windows or Mac OS, UNIX directories are equivalent to folders. A directory file contains an entry for every file and subdirectory that it houses. If you have 10 files in a directory, there will be 10 entries in the directory. Each entry has two components.

- The Filename
- A unique identification number for the file or directory (called the inode number)
 - Branching points in the hierarchical tree.
 - Used to organize groups of files.
 - May contain ordinary files, special files or other directories.
 - Never contain “real” information which you would work with (such as text). Basically, just used for organizing files.
 - All files are descendants of the root directory, (named /) located at the top of the tree.
 - In long-format output of `ls -l` , this type of file is specified by the “d” symbol.

- 3) Special Files – Used to represent a real physical device such as a printer, tape drive or terminal, used for Input/Output (I/O) operations. Device or special files are used for device Input/Output(I/O) on UNIX and Linux systems. They appear in a file system just like an ordinary file or a directory.

On UNIX systems there are two flavors of special files for each device, character special files and block special files :

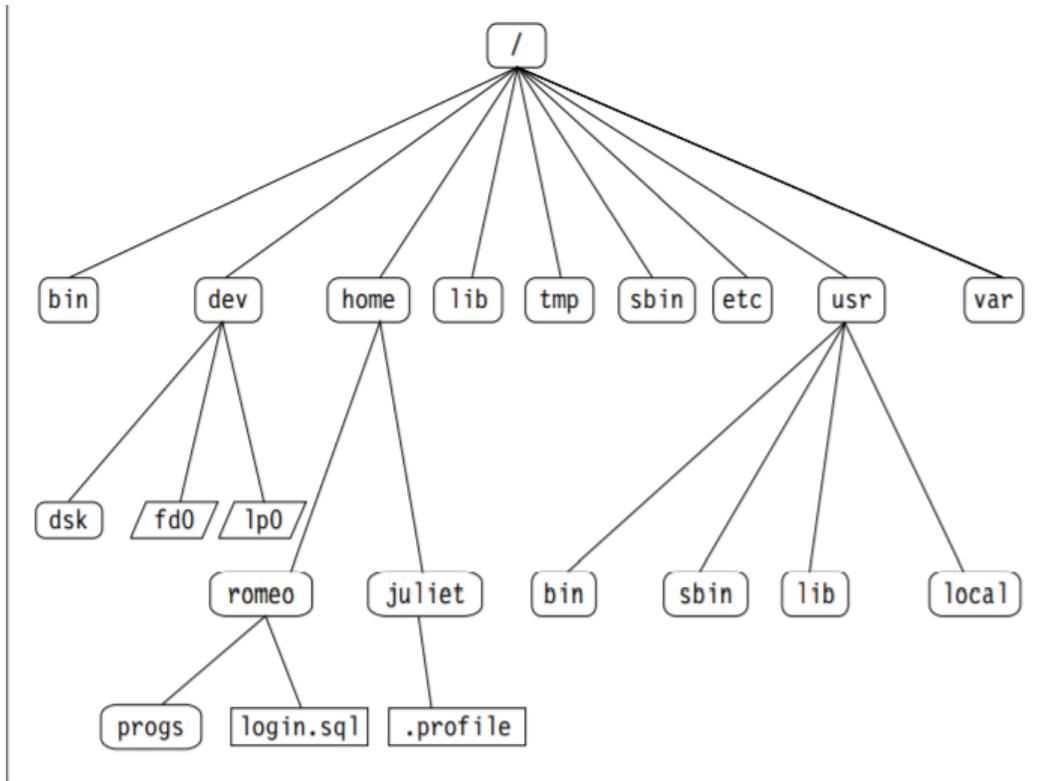
- When a character special file is used for device Input/Output(I/O), data is transferred one character at a time. This type of access is called raw device access.
- When a block special file is used for device Input/Output(I/O), data is transferred in large fixed-size blocks. This type of access is called block device access.

- 4) Pipes – UNIX allows you to link commands together using a pipe. The pipe acts a temporary file which only exists to hold data from one command until it is read by another. A Unix pipe provides a one-way flow of data. The output or result of the first command sequence is used as the input to the second command sequence. To make a pipe, put a vertical bar (|) on the command line between two commands. For example: `who | wc -l`

- 5) Sockets – A Unix socket (or Inter-process communication socket) is a special file which allows for advanced inter-process communication. A Unix Socket is used in a client-server application framework. In essence, it is a stream of data, very similar to network stream (and network sockets), but all the transactions are local to the filesystem.

- 6) Symbolic Link – Symbolic link is used for referencing some other file of the file system. Symbolic link is also known as Soft link. It contains a text form of the path to the file it references. To an end user, symbolic link will appear to have its own name, but when you try reading or writing data to this file, it will instead reference these operations to the file it points to. If we delete the soft link itself, the data file would still be there. If we delete the source file or move it to a different location, symbolic file will not function properly. In long-format output of `ls -l`, Symbolic link are marked by the “l” symbol.

Organization of files



Unix file system is a logical method of **organizing and storing** large amounts of information in a way that makes it easy to manage. A file is a smallest unit in which the information is stored. Unix file system has several important features. All data in Unix is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the file system.

Files in Unix System are organized into multi-level hierarchy structure known as a directory tree. At the very top of the file system is a directory called “root” which is represented by a “/”. All other files are “descendants” of root.

Hidden files

UNIX allows users to have files which are not listed, by default, by the **ls** command. These are called hidden files and are distinguishable from other files by the fact that their filenames begin

with a dot (.). Such a file is .profile which is executed every time you log in to the system. Hidden files are listed by adding the **-a** option to the **ls** command.

Standard directories

The File system Hierarchy Standard (FHS) defines the structure of file systems on Linux and other UNIX-like operating systems. However, Linux file systems also contain some directories that aren't yet defined by the standard.

1) / – The Root Directory

Everything on your Linux system is located under the / directory, known as the root directory. You can think of the / directory as being similar to the C:\ directory on Windows – but this isn't strictly true, as Linux doesn't have drive letters.

2) /bin – Essential User Binaries

The /bin directory contains the essential user binaries (programs) that must be present when the system is mounted in single-user mode. Applications such as Firefox are stored in /usr/bin, while important system programs and utilities such as the bash shell are located in /bin.

3) /boot – Static Boot Files

The /boot directory contains the files needed to boot the system – for example, the GRUB boot loader's files and your Linux kernels are stored here. The boot loader's configuration files aren't located here, though – they're in /etc with the other configuration files.

4) /cdrom – Historical Mount Point for CD-ROMs

The /cdrom directory isn't part of the FHS standard, but you'll still find it on Ubuntu and other operating systems. It's a temporary location for CD-ROMs inserted in the system. However, the standard location for temporary media is inside the /media directory.

5) /dev – Device Files

Linux exposes devices as files, and the /dev directory contains a number of special files that represent devices. These are not actual files as we know them, but they appear as files – for example, /dev/sda represents the first SATA drive in the system. This directory

also contains pseudo-devices, which are virtual devices that don't actually correspond to hardware.

6) **/etc – Configuration Files**

The /etc directory contains configuration files, which can generally be edited by hand in a text editor. Note that the /etc/ directory contains system-wide configuration files – user-specific configuration files are located in each user's home directory.

7) **/home – Home Folders**

The /home directory contains a home folder for each user. For example, if your user name is bob, you have a home folder located at /home/bob. This home folder contains the user's data files and user-specific configuration files. Each user only has write access to their own home folder and must obtain elevated permissions (become the root user) to modify other files on the system.

8) **/lib – Essential Shared Libraries**

The /lib directory contains libraries needed by the essential binaries in the /bin and /sbin folder. Libraries needed by the binaries in the /usr/bin folder are located in /usr/lib.

9) **/lost+found – Recovered Files**

Each Linux file system has a lost+found directory. If the file system crashes, a file system check will be performed at next boot. Any corrupted files found will be placed in the lost+found directory, so you can attempt to recover as much data as possible.

10) **/media – Removable Media**

The /media directory contains subdirectories where removable media devices inserted into the computer are mounted. For example, when you insert a CD into your Linux system, a directory will automatically be created inside the /media directory. You can access the contents of the CD inside this directory.

11) **/mnt – Temporary Mount Points**

Historically speaking, the /mnt directory is where system administrators mounted temporary file systems while using them. For example, if you're mounting a Windows partition to perform some file recovery operations, you might mount it at /mnt/windows. However, you can mount other file systems anywhere on the system.

12) **/opt – Optional Packages**

The `/opt` directory contains subdirectories for optional software packages. It's commonly used by proprietary software that doesn't obey the standard file system hierarchy – for example, a proprietary program might dump its files in `/opt/application` when you install it.

13) `/proc` – Kernel & Process Files

The `/proc` directory is similar to the `/dev` directory because it doesn't contain standard files. It contains special files that represent system and process information.

14) `/root` – Root Home Directory

The `/root` directory is the home directory of the root user. Instead of being located at `/home/root`, it's located at `/root`. This is distinct from `/`, which is the system root directory.

15) `/run` – Application State Files

The `/run` directory is fairly new, and gives applications a standard place to store transient files they require like sockets and process IDs. These files can't be stored in `/tmp` because files in `/tmp` may be deleted.

16) `/sbin` – System Administration Binaries

The `/sbin` directory is similar to the `/bin` directory. It contains essential binaries that are generally intended to be run by the root user for system administration.

17) `/selinux` – SELinux Virtual File System

If your Linux distribution uses SELinux for security (Fedora and Red Hat, for example), the `/selinux` directory contains special files used by SELinux. It's similar to `/proc`. Ubuntu doesn't use SELinux, so the presence of this folder on Ubuntu appears to be a bug.

18) `/srv` – Service Data

The `/srv` directory contains “data for services provided by the system.” If you were using the Apache HTTP server to serve a website, you'd likely store your website's files in a directory inside the `/srv` directory.

19) `/tmp` – Temporary Files

Applications store temporary files in the `/tmp` directory. These files are generally deleted whenever your system is restarted and may be deleted at any time by utilities such as `tmpwatch`.

20) /usr – User Binaries & Read-Only Data

The /usr directory contains applications and files used by users, as opposed to applications and files used by the system. For example, non-essential applications are located inside the /usr/bin directory instead of the /bin directory and non-essential system administration binaries are located in the /usr/sbin directory instead of the /sbin directory. Libraries for each are located inside the /usr/lib directory. The /usr directory also contains other directories – for example, architecture-independent files like graphics are located in /usr/share.

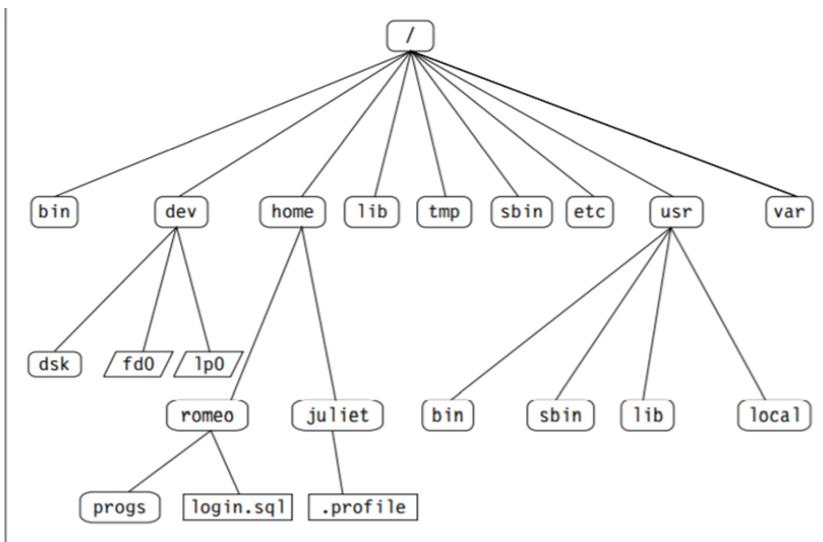
21) /var – Variable Data Files

The /var directory is the writable counterpart to the /usr directory, which must be read-only in normal operation. Log files and everything else that would normally be written to /usr during normal operation are written to the /var directory. For example, you'll find log files in /var/log.

Parent Child Relationship

The implicit feature of unix file system is that there is a top, which serves as a reference point for all files. This top is called root and is represented by a / (frontslash). root is actually a directory. It is conceptually different from the user-id root used by the system administrator to log in. The root directory (/) has a number of subdirectories under it. These subdirectories in turn have more subdirectories and other files under them. For instance, bin and usr are two directories directly under /, while a second bin and Romeo are subdirectories under usr.

Thus, the home directory is the parent of Romeo, while / is the parent of home, and the grandparent of Romeo. If you create a file login.sql under the Romeo directory, then Romeo will be the parent of this file. In these parent – child relationship, the parent is always a directory. home and Romeo are both directories as they are both parents of at least one file or directory. login.sql is simply an ordinary file; it can't have any directory under it.



- *bin* - short for binaries, this is the directory where many commonly used executable commands reside
- *dev* - contains device specific files
- *etc* - contains system configuration files
- *home* - contains user directories and files
- *lib* - contains all library files
- *mnt* - contains device files related to mounted devices
- *proc* - contains files related to system processes
- *root* - the root users' home directory (note this is different than /)
- *sbin* - system binary files reside here. If there is no sbin directory on your system, these files most likely reside in etc
- *tmp* - storage for temporary files which are periodically removed from the filesystem
- *usr* - also contains executable commands

The home directory and the HOME variable

A home directory, also called a login directory, is the directory on Unix-like operating systems that serves as the repository for a user's personal files, directories and programs. It is

also the directory that a user is first in after logging into the system. A home directory is created automatically for every ordinary user in the directory called */home*. A standard subdirectory of the root directory, */home* has the sole purpose of containing users' home directories. The root directory, which is designated by a forward slash (/), is the directory that contains all other directories and their subdirectories as well as all files on the system.

The name of a user's home directory is by default identical to that of the user. Thus, for example, a user with a user name of *mary* would typically have a home directory named *mary*. It would have an *absolute pathname* of */home/mary*. An absolute pathname is the location of a directory or file relative to the root directory, and it always starts with the root directory (i.e., with a forward slash).

The only user that will by default have its home directory in a different location is the *root* (i.e., administrative) user, whose home directory is */root*. */root* is another standard subdirectory of the root directory, and it should not be confused with the root directory (although it sometimes is by new users). For security purposes, even system administrators should have ordinary accounts with home directories in */home* into which they routinely log in, and they should use the root account only when absolutely necessary.

There are several easy ways for a user to return to its home directory regardless of its *current directory* (i.e., the directory in which it is currently working in). The simplest of these is to use the *cd* (i.e., *change directory*) command without any options or *arguments* (i.e., input files), i.e., by merely typing the following and then pressing the ENTER key:

```
$cd
```

The absolute pathname of a user's home directory is stored in that user's *\$HOME environmental variable*. Environmental variables are a class of variables that tell the *shell* (i.e., the program that provides the text-only user interface for entering commands) how to behave as a user works at the *command line* (i.e., all-text mode). Thus a third way for a user to return to its home directory is to use *\$HOME* as an argument to *cd*, i.e.,

```
$cd $HOME
```

```
/home/atria
```

The PATH variable

The PATH environment variable has a special format. Let's see what it looks like:

```
$ echo $PATH /usr/local/bin:/bin:/usr/bin:/sbin:/usr/sbin.
```

It's essentially a `:`-separated list of directories. When you execute a command, the shell **searches through each of these directories, one by one**, until it finds a directory where the executable exists. Remember that we found `ls` in `/bin`, right? `/bin` is the second item in the PATH variable. So let's remove `/bin` from PATH. We can do this by using the `export` command:

```
$ export PATH=/usr/local/bin:/usr/bin:/sbin:/usr/sbin.
```

Make sure that the variable is set correctly:

```
$ echo $PATH /usr/local/bin:/usr/bin:/sbin:/usr/sbin.
```

Now, if we try to run `ls`, the shell no longer knows to look in `/bin`!

```
$ ls
```

```
-bash: ls: command not found
```

Manipulating your PATH variable

- The PATH variable contains the search path for executing commands and scripts. To see your PATH, enter:
- ```
$ echo $PATH
```

```
/home/khess/.local/bin:/home/khess/bin:/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin
```

Temporarily change your PATH by entering the following command to add `/opt/bin`:

- ```
$ PATH=$PATH:/opt/bin $ echo $PATH
```

```
/home/khess/.local/bin:/home/khess/bin:/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/opt/bin
```

The change is temporary for the current session. It isn't permanent because it's not entered into the `.bashrc` file. To make the change permanent, enter the command `PATH=$PATH:/opt/bin` into your home directory's `.bashrc` file.

- When you do this, you're creating a new `PATH` variable by appending a directory to the current `PATH` variable, `$PATH`. A colon (`:`) separates `PATH` entries.

Relative and Absolute Pathnames

An absolute path is defined as specifying the location of a file or directory from the root directory (`/`). In other words, that an absolute path is a complete path from start of actual file system from `/` directory.

Example for Absolute path

```
$pwd
```

```
/home/atria
```

```
$cd /home/atria/abc
```

```
$pwd
```

```
/home/atria/abc
```

Relative path is defined as the path related to the present working directory(`pwd`). It starts at your current directory and never starts with a `/`.

Example for Relative path

```
$pwd
```

```
/home/atria
```

```
$cd abc
```

```
$pwd
```

```
/home/atria/abc
```

Directory commands

- `pwd`(present or print working directory)

- cd(current directory)
- mkdir(make a directory)
- rmdir(remove directory)

pwd - pwd stands for Print Working Directory. It prints the path of the working directory, starting from the root.

- pwd is shell built-in command(pwd) or an actual binary(/bin/pwd).
- \$PWD is an environment variable which stores the path of the current directory.

This command has two flags.

pwd -L: Prints the symbolic path.

pwd -P: Prints the actual path.

Built-in pwd - In the given example the directory is a symbolic link for a target directory.

Example for built-in pwd

```
$ pwd
```

```
/home/atria
```

```
$ cd /home/atria/abc
```

```
$ pwd -L
```

```
/home/atria/abc
```

```
$ pwd -P          /var/abc
```

Binary pwd (/bin/pwd) - The default behavior of Built-in pwd is same as pwd -L. and the default behavior of /bin/pwd is same as pwd -P.

Example for binary pwd

```
$ /bin/pwd
```

```
/var/abc
```

```
$ /bin/pwd -P
```

```
/var/abc
```

```
$ /bin/pwd -L
```

```
/home/atria/abc.
```

cd - cd command in Linux known as change directory command. It is used to change current working directory.

Syntax:

```
$ cd [directory]
```

To move inside a subdirectory: to move inside a subdirectory in linux we use

```
$ cd [directory_name]
```

Different functionalities of cd command:

- **cd /:** this command is used to change directory to the root directory, The root directory is the first directory in your filesystem hierarchy.

```
$ cd /
```

- **cd dir_1/dir_2/dir_3:** This command is used to move inside a directory from a directory

```
$ cd dir_1/dir_2/dir_3
```

Example – cd /Documents/D1

- **cd ~ :** this command is used to change directory to the home directory.

```
$ cd ~
```

- **cd .. :** this command is used to move to the parent directory of current directory, or the directory one level up from the current directory. “..” represents parent directory.

```
$ cd ..
```

- **cd “dir name”:** This command is used to navigate to a directory with white spaces. Instead of using double quotes we can use single quotes then also this command will work.

```
$ cd "dir name"
```

mkdir - mkdir command in Linux allows the user to create directories (also referred to as folders in some operating systems). This command can create multiple directories at once as well as set the permissions for the directories. It is important to note that the user executing this command must have enough permissions to create a directory in the parent directory, or he/she may receive a 'permission denied' error.

Syntax:

\$mkdir [options...] [directories ...]

- **--version** - It displays the version number, some information regarding the license and exits.

Syntax: \$mkdir --version

- **--help** - It displays the help related information and exits.

Syntax: mkdir --help

- **-v or --verbose** - It displays a message for every directory created.

Syntax: \$mkdir -v [directories]

- **-p** - A flag which enables the command to create parent directories as necessary. If the directories exist, no error is specified.

Syntax: \$mkdir -p [directories]

- **-m** - This option is used to set the file modes, i.e. permissions, etc. for the created directories. The syntax of the mode is the same as the chmod command.

Syntax: \$mkdir -m a=rwx [directories]

rmdir - rmdir command is used to remove empty directories from the filesystem in Linux. The rmdir command removes each and every directory specified in the command line only if these directories are empty. So if the specified directory has some sub-directories or files in it then this cannot be removed by *rmdir* command.

Syntax:

```
$rmdir [-p] [-v | -verbose] [-ignore-fail-on-non-empty] directories ...
```

The above syntax specifies that the directories created give access to all the users to read from, write to and execute the contents of the created directories. You can use 'a=r' to only allow all the users to read from the directories and so on.

- **rmdir -p:** In this option each of the directory argument is treated as a pathname of which all components will be removed, if they are already empty, starting from the last component.
- **rmdir -v, -verbose:** This option displays verbose information for every directory being processed.
- **rmdir -ignore-fail-on-non-empty:** This option do not report a failure which occurs solely because a directory is non-empty. Normally, when rmdir is being instructed to remove a non-empty directory, it simply reports an error. This option consists of all those error messages.
- **rmdir -version:** This option is used to display the version information and exit.

The dot(.) and double dot(..) notations to represent present and parent directories and their usage in relative path names.

unix offers a shortcut – the relative pathname – that uses either the current or parent directory as reference and specifies the path relative to it. A relative pathname uses one of these cryptic symbols:

. (a single dot) – this represent the current directory.

.. (two dots) – this represents the parent directory.

Now use the (..) to frame relative pathnames. Assuming that you are placed in /home/atria/progs/data:

```
$ pwd
```

```
/home/atria/progs/data/text
```

```
$ cd ..           //moves one level up
```

```
$ pwd
```

```
/home/atria/progs/data
```

The command `cd ..` translates to this: “change your directory to the parent of the current directory.” you can combine any number of such sets of `..` separated by `/s`. however, when a `/` is used with `..` it acquires a different meaning; instead of moving down a level, it moves one level up. For instance, to move to `/home`, you can always use `cd /home`. Alternatively you can also use a relative pathname:

```
$ pwd
```

```
/home/atria/abc
```

```
$cd ../../
```

```
$ pwd
```

```
/home
```

The solitary dot that refers to the current directory. Any command which uses the current directory as argument can also work with a single dot. This means that the `cp` command which also uses a directory as the last argument can be used with a dot:

```
cp ../abc/.profile .           //a filename can begin with a dot.
```

This copies the file `.profile` to the current directory (`.`). note that you didn’t have to specify the filename of the copy; it’s the same as the original one. This dot is also implicitly included whenever we use a filename as argument, rather than a pathname. For instance, `cd progs` is same as `cd ./progs`.

File related commands – cat, mv, rm, cp, wc and od commands

cat command

Cat(concatenate) command is very frequently used in Linux. It reads data from the file and gives their content as output. It helps us to create, view, concatenate files. So let us see some frequently used cat commands.

1) **To view a single file**

Command: \$cat filename

Output:

- It will show content of given filename.

2) **To view multiple files**

Command:

- \$cat file1 file2

Output:

This will show the content of file1 and file2.

3) **To view contents of a file preceding with line numbers.**

Command:

\$cat -n filename

Output:

It will show content with line number

4) **Create a file**

Command:

\$ cat >newfile

Output:

Will create and a file named newfile

5) **Copy the contents of one file to another file.**

Command:

```
$cat [filename-whose-contents-is-to-be-copied] > [destination-filename]
```

Output:

The content will be copied in destination file

6) Cat command can suppress repeated empty lines in output**Command:**

```
$cat -s geeks.txt
```

Output:

Will suppress repeated empty lines in output

7) Cat command can append the contents of one file to the end of another file.**Command:**

```
$cat file1 >> file2
```

Output:

- Will append the contents of one file to the end of another file

8) Cat command can display content in reverse order using tac command.**Command:**

```
$tac filename
```

Output:

- Will display content in reverse order

9) Cat command can highlight the end of line.**Command:**

```
$cat -E "filename"
```

Output:

- Will highlight the end of line

10) If you want to use the -v, -E and -T option together, then instead of writing -vET in the command, you can just use the -A command line option.

Command

- `$cat -A "filename"`

11) Cat command to display the content of all text files in the folder.

Command:

```
$cat *.txt
```

Output:

- Will show the content of all text files present in the folder.

mv - mv stands for move. mv is used to move one or more files or directories from one place to another in file system like UNIX. It has two distinct functions:

(i) It rename a file or folder.

(ii) It moves group of files to different directory.

- No additional space is consumed on a disk during renaming. This command normally works silently means no prompt for confirmation.

Syntax:

```
$mv [Option] source destination
```

Let us consider 5 files having name **a.txt**, **b.txt** and so on till **e.txt**.

- To rename the file **a.txt** to **geek.txt(not exist)**:

```
$ ls
```

```
a.txt b.txt c.txt d.txt
```

```
$ mv a.txt geek.txt
```

```
$ ls
```

```
b.txt c.txt d.txt geek.txt
```

If the destination file **doesn't exist**, it will be created. In the above command **mv** simply replaces the source filename in the directory with the destination filename(new name). If the destination file **exist**, then it will be **overwrite** and the source file will be deleted.

Let's try to understand with example, moving **geeks.txt** to **b.txt(exist)**:

```
$ ls
```

```
b.txt c.txt d.txt geek.txt
```

```
$ cat geek.txt
```

```
India
```

```
$ cat b.txt
```

```
geeksforgeeks
```

```
$ mv geek.txt b.txt
```

```
$ ls
```

```
b.txt c.txt d.txt
```

```
$ cat b.txt
```

```
India
```

Options:

1. -i (Interactive): Like in cp, the -i option makes the command ask the user for confirmation before moving a file that would overwrite an existing file, you have to press **y** for confirm moving, any other key leaves the file as it is. This option doesn't work if the file doesn't exist, it simply rename it or move it to new location.

```
$ ls
```

```
b.txt c.txt d.txt geek.txt
```

```
$ cat geek.txt
```

```
India
```

File Related Command: mv

```
$ cat b.txt
```

```
geeksforgeeks
```

```
$ mv -i geek.txt b.txt
```

```
mv: overwrite 'b.txt'? y
```

```
$ ls
```

```
b.txt c.txt d.txt
```

```
$ cat b.txt
```

```
India
```

rm - rm stands for remove here. rm command is used to remove objects such as files, directories, symbolic links and so on from the file system like UNIX. To be more precise, rm removes references to objects from the filesystem, where those objects might have had multiple references (for example, a file with two different names). By default, it does not remove directories.

This command normally works silently and you should be very careful while running rm command because once you delete the files then you are not able to recover the contents of files and directories.

Syntax:

```
$rm [OPTION]... FILE...
```

File Related Command: rm

Syntax:

```
$rm [OPTION]... FILE...
```

Let us consider 5 files having name a.txt, b.txt and so on till e.txt.

- \$ ls

```
a.txt b.txt c.txt d.txt e.txt
```

File Related Command: rm

Removing one file at a time

```
$ rm a.txt
```

```
$ ls
```

```
b.txt c.txt d.txt e.txt
```

Removing more than one file at a time

```
$ rm b.txt c.txt $ ls
```

```
d.txt e.txt
```

Note: No output is produced by **rm**, since it typically only generates messages in the case of an error.

File Related Command: rm

Options:

1. -i (Interactive Deletion): Like in cp, the -i option makes the command ask the user for confirmation before removing each file, you have to press **y** for confirm deletion, any other key leaves the file un-deleted.

```
$ rm -i d.txt
```

```
rm: remove regular empty file 'd.txt'? y
```

```
$ ls
```

```
e.txt
```

2. -f (Force Deletion): **rm** prompts for confirmation removal if a file is **write protected**. The **-f** option overrides this minor protection and removes the file forcefully.

```
$ ls -l
```

```
total 0 -r--r--r--+ 1 User User 0 Jan 2 22:56 e.txt
```

```
$ rm e.txt
```

```
rm: remove write-protected regular empty file 'e.txt'? n
```

```
$ ls
```

```
e.txt
```

```
$ rm -f e.txt
```

```
$ ls
```

Note: **-f** option of **rm** command will not work for write-protect directories.

cp – copy command copies a file or a group of files. It creates an exact image of the file on disk with a different name.

```
$ cp a.txt b.txt
```

If the destination file(b.txt) doesn't exist, it will first be created before copying takes place. If not it will be overwritten. So for that you just check with the **ls** command whether or not the file exists.

Options:

1. -i(interactive): **i** stands for Interactive copying. With this option system first warns the user before overwriting the destination file. **cp** prompts for a response, if you press **y** then it overwrites the file and with any other option leave it uncopied.

```
$ cp -i a.txt b.txt
```

```
cp: overwrite 'b.txt'? y
```

```
$ cat b.txt
```

```
GFG
```

2. -b(backup): With this option **cp** command creates the backup of the destination file in the same folder with the different name and in different format.

```
$ ls
```

```
a.txt b.txt
```

```
$ cp -b a.txt b.txt
```

```
$ ls
```

```
a.txt b.txt b.txt~
```

3. -r or -R: Copying directory structure. With this option **cp** command shows its recursive behavior by copying the entire directory structure recursively.

Suppose we want to copy **geeksforgeeks** directory containing many files, directories into **gfg** directory(not exist).

```
$ ls geeksforgeeks/
```

```
a.txt b.txt b.txt~ Folder1 Folder2
```

Without -r option, error

```
$ cp geeksforgeeks gfg
```

cp: -r not specified; omitting directory 'geeksforgeeks' With -r, execute successfully

```
$ cp -r geeksforgeeks gfg
```

```
$ ls gfg/
```

```
a.txt b.txt b.txt~ Folder1 Folder2
```

wc - wc stands for word count. As the name implies, it is mainly used for counting purpose. It is used to find out number of lines, word count, byte and characters count in the files specified in the file arguments. By default it displays four-columnar output. First column shows number of lines present in a file specified, second column shows number of words present in the file, third column shows number of characters present in file and fourth column itself is the file name which are given as argument.

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

File Related Command: wc

Let us consider two files having name **state.txt** and **capital.txt** containing 5 names of the Indian states and capitals respectively.

```
$ cat state.txt
```

```
Andhra Pradesh
```

```
Arunachal Pradesh
```

```
$ cat capital.txt
```

Hyderabad

Raipur

- **Passing only one file name in the argument.**

```
$ wc state.txt
```

Passing more than one file name in the argument.

```
$ wc state.txt capital.txt
```

```
5 7 63 state.txt
```

```
5 5 45 capital.txt
```

```
10 12 108 total
```

Note: When more than file name is specified in argument then command will display four-columnar output for all individual files plus one extra row displaying total number of lines, words and characters of all the files specified in argument, followed by keyword **total**.

Options:

1. **-l:** This option prints the **number of lines** present in a file.
2. **-w:** This option prints the **number of words** present in a file.
3. **-c:** This option displays **count of bytes** present in a file.
4. **-m:** Using **-m** option 'wc' command displays **count of characters** from a file.
5. **-L:** The 'wc' command allow an argument **-L**, it can be used to print out the length of longest (number of characters) line in a file.
6. **-version:** This option is used to display the version of **wc** which is currently running on your system.

Applications of wc Command

1. To count all files and folders present in directory: As we all know ls command in unix is used to display all the files and folders present in the directory, when it is piped with **wc** command with **-l** option it display count of all files and folders present in current directory.

2. Display number of word count only of a file: We all know that this can be done with `wc` command having `-w` option, `wc -w file_name`, but this command shows two-columnar output one is count of words and other is file name.

od - `od` command in Linux is used to convert the content of input in different formats with octal format as the default format. This command is especially useful when debugging Linux scripts for unwanted changes or characters. If more than one file is specified, `od` command concatenates them in the listed order to form the input. It can display output in a variety of other formats, including hexadecimal, decimal, and ASCII. It is useful for visualizing data that is not in a human-readable format, like the executable code of a program.

Syntax:

`$od [OPTION]... [FILE]...`

Module-2: File attributes & permissions

The ls command with options

Display All Information About Files/Directories Using ls -l. The list is preceded by the words of total 100, which indicates that a total of 100 blocks are occupied by these files on disk, each block consisting of 512 bytes(1024 in linux).

\$ ls -l: To show long listing information about the file/directory.

This option displays most attributes of a file – like its permissions, size and ownership details.

```

atria@atria-VirtualBox:~$ ls
10.c  abc.txt  Documents  ghl.txt  Public  Untitled Folder
2.c  bef.txt  Downloads  IS2      Rajitha  USN
3.c  capital.txt  examples.desktop  Music  state.txt  Videos
8.c  Desktop  geek.c     Pictures  Templates
atria@atria-VirtualBox:~$ ls -l
total 100
-rwxr-xr-x 1 atria atria 52 Sep 28 14:55 10.c
-rwxr-xr-x 1 atria atria 175 Sep 24 11:50 2.c
-rwxr-xr-x 1 atria atria 52 Sep 24 11:49 3.c
-rwxr-xr-x 1 atria atria 175 Sep 28 14:53 8.c
-rwxr-xr-x 1 atria atria 6 Sep 25 08:46 abc.txt
-rwxr-xr-x 1 atria atria 6 Sep 25 09:13 bef.txt
-rwxr-xr-x 1 atria atria 39 Sep 24 19:00 capital.txt
drwxr-xr-x 3 atria atria 4096 Sep 24 11:36 Desktop
drwxr-xr-x 3 atria atria 4096 Sep 19 08:32 Documents
drwxr-xr-x 2 atria atria 4096 Sep 15 11:04 Downloads
-rwxr-xr-x 1 atria atria 8980 Sep 15 10:58 examples.desktop
-rwxr-xr-x 1 atria atria 52 Sep 18 10:37 geek.c
-rwxr-xr-x 1 atria atria 6 Sep 25 10:32 ghl.txt
drwxr-xr-x 2 atria atria 4096 Sep 19 09:47 IS2
drwxr-xr-x 2 atria atria 4096 Sep 15 11:04 Music
drwxr-xr-x 2 atria atria 4096 Sep 28 15:33 Pictures
drwxr-xr-x 2 atria atria 4096 Sep 15 11:04 Public
drwxr-xr-x 2 atria atria 4096 Sep 28 14:36 Rajitha
-rwxr-xr-x 1 atria atria 58 Sep 24 18:59 state.txt
drwxr-xr-x 2 atria atria 4096 Sep 15 11:04 Templates
drwxr-xr-x 2 atria atria 4096 Sep 28 14:38 Untitled folder
drwxr-xr-x 3 atria atria 4096 Sep 28 14:42 USN
drwxr-xr-x 2 atria atria 4096 Sep 15 11:04 Videos
atria@atria-VirtualBox:~$
    
```

```

-rwxr-xr-x 1 atria atria 52 Sep 28 14:55 10.c
-rwxr-xr-x 1 atria atria 175 Sep 24 11:50 2.c
-rwxr-xr-x 1 atria atria 52 Sep 24 11:49 3.c
-rwxr-xr-x 1 atria atria 175 Sep 28 14:53 8.c
-rwxr-xr-x 1 atria atria 6 Sep 25 08:46 abc.txt
-rwxr-xr-x 1 atria atria 6 Sep 25 09:13 bef.txt
-rwxr-xr-x 1 atria atria 39 Sep 24 19:00 capital.txt
drwxr-xr-x 3 atria atria 4096 Sep 24 11:36 Desktop
-rwxr-xr-x 3 atria atria 4096 Sep 19 08:32 Documents
    
```

-rw-rw-r-- 1 atria atria 1176 Feb 16 00:19 1.c

Field 1 – File Permissions: Next 9 character specifies the files permission. Every 3 characters specifies read, write, execute permissions for user(root), group and others respectively in order. Taking above example, -rw-rw-r-- indicates read-write permission for user(root), read permission for group, and no permission for others respectively. If all three permissions are given to user(root), group and others, the format looks like -rwxrwxrwx

Field 2 – Number of links: Second field specifies the number of links for that file. In this example, 1 indicates only one link to this file.

Field 3 – Owner: Third field specifies owner of the file. In this example, this file is owned by username 'atria'.

Field 4 – Group: Fourth field specifies the group of the file. In this example, this file belongs to "atria" group.

Field 5 – Size: Fifth field specifies the size of file in bytes. In this example, '1176' indicates the file size in bytes.

Field 6 – Last modified date and time: Sixth field specifies the date and time of the last modification of the file. In this example, ‘Feb 16 00:19’ specifies the last modification time of the file.

Field 7 – File name: The last field is the name of the file. In this example, the file name is 1.c.

The -d Option: Listing Directory Attributes

The ls, when used with directory names, lists files in the directory rather than the directory itself. To force ls to list the attributes of a directory, rather than its contents, you need to use the -d (directory) option:

```
atria@atria-VirtualBox:~$ ls
2.c    capital.txt  examples.desktop  Music    state.txt    Videos
3.c    Desktop     geek.c           Pictures Templates
abc.txt Documents   ghi.txt          Public   Untitled Folder
bef.txt Downloads  IS2              Ranjitha USN
atria@atria-VirtualBox:~$ ls -d
.
atria@atria-VirtualBox:~$ ls -ld
drwxr-xr-x 21 atria atria 4096 Sep 28 14:41 .
atria@atria-VirtualBox:~$ ls -ld USN
drwxrwxr-x 3 atria atria 4096 Sep 28 14:42 USN
atria@atria-VirtualBox:~$ ls -ld Desktop
drwxr-xr-x 3 atria atria 4096 Sep 24 11:36 Desktop
atria@atria-VirtualBox:~$
```

Directories are easily identified in the listing by the first character of the first column, which here shows a ‘d’. for ordinary files, this slot always shows a ‘-’ (hyphen), and for device files, either a b or c. The significance of the attributes of a directory differ a good deal from an ordinary file.

File ownership

The privileges of the group are set by the owner of the file and not by the group member.

When the system administrator creates a user account he has to assign these parameters.

1. User id (UID)-both its name & numeric representation
2. Group id(GID)- both its name & numeric representation

```
atria@atria-VirtualBox:~$ id
uid=1000(atria) gid=1000(atria) groups=1000(atria),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
atria@atria-VirtualBox:~$
```

File permission

UNIX also provides a way to protect files based on users and groups.

- Three **types** of permissions:
 - read, process may read contents of file
 - write, process may write contents of file
 - execute, process may execute file
- Three **sets** of permissions:
 - permissions for owner
 - permissions for group
 - permissions for other

rwX r-x r - -

The first group has all three permission – rwX.

The second group has a ‘-’ in the middle slots which indicates the absence of write permission by the group owner of the file – r-x.

The third group has write and execute bits absent – r--.

Changing File Permission (Chmod)

They are two types of chmod permission.

1. Relative permissions
2. Absolute Permissions

Relative permission

In relative permission chmod only changes the permission specifying in the command line and leaves the other permission unchanged.

Syntax:

Chmod category operation permission filename(s)

Chmod takes as its argument an expression comprising some letters and symbols that completely describe the user category and the type of permission being assigned or removed.

The expression contains 3 components.

1. User category (user, group, others)
2. The operation to be performed (assign or remove a permission)
3. The type of permission (read, write, execute).

Example:

To assign execute permission to the user of the file 2.c, we need to frame a suitable expression by using appropriate characters from each of the three columns.

Table 6.1 Abbreviations Used by chmod

Category	Operation	Permission
u—User	+—Assigns permission	r—Read permission
g—Group	—Removes permission	w—Write permission
o—Others	—Assigns absolute permission	x—Execute permission
a—All (ugo)		

```

atria@atria-VirtualBox:~$ chmod u+x 2.c
atria@atria-VirtualBox:~$ ls -l 2.c
-rwxrw-r-- 1 atria atria 175 Sep 24 11:50 2.c
atria@atria-VirtualBox:~$ chmod ugo+x 2.c
atria@atria-VirtualBox:~$ ls -l 2.c
-rwxrwxr-x 1 atria atria 175 Sep 24 11:50 2.c
atria@atria-VirtualBox:~$ chmod go-r 2.c
atria@atria-VirtualBox:~$ ls -l 2.c
-rwx-wx--x 1 atria atria 175 Sep 24 11:50 2.c
atria@atria-VirtualBox:~$ chmod a-x,go+r 2.c
atria@atria-VirtualBox:~$ ls -l 2.c
-rw-rw-r-- 1 atria atria 175 Sep 24 11:50 2.c
atria@atria-VirtualBox:~$ chmod o+wx 2.c
    
```

Absolute Permission

The expression used by chmod here is a string of three octal numbers (base 8). Octal numbers use the base 8, and octal digits have the values 0 to 7. This means that a set of three bits can represent one octal digit. Represent the permission of each category by one octal digit -

1. Read permission – 4(octal – 100)
2. Write permission -2(octal – 010)
3. Execute permission -1(octal – 001)

Binary	Octal	Permissions	Significance
000	0	---	No permissions
001	1	--x	Executable only
010	2	-w-	Writable only
011	3	-wx	Writable and executable
100	4	r--	Readable only
101	5	r-x	Readable and executable
110	6	rw-	Readable and writable
111	7	rwX	Readable, writable and executable

Example:

```

atria@atria-VirtualBox:~$ chmod 666 2.c
atria@atria-VirtualBox:~$ ls -l 2.c
-rw-rw-rw- 1 atria atria 175 Sep 24 11:50 2.c
atria@atria-VirtualBox:~$ chmod 644 2.c
atria@atria-VirtualBox:~$ ls -l 2.c
-rw-r--r-- 1 atria atria 175 Sep 24 11:50 2.c
atria@atria-VirtualBox:~$ chmod 777 2.c
atria@atria-VirtualBox:~$ ls -l 2.c
-rwxrwxrwx 1 atria atria 175 Sep 24 11:50 2.c
atria@atria-VirtualBox:~$
    
```

Recursively changing file permissions

Chmod descent a directory and apply the expression to every file and sub directory it finds.

This is done with the -R option

1. Chmod -R 755 . (works on hidden files also)
2. Chmod -R a+x * (leaves out hidden files)

Example

```
ise@ise-VirtualBox: ~
ise@ise-VirtualBox:~$ ls -ld usn ise
drwxr-xr-x 2 ise ise 4096 Oct  5 12:27 ise
drwxr-xr-x 3 ise ise 4096 Oct  5 12:24 usn
ise@ise-VirtualBox:~$ chmod -R a+w usn
ise@ise-VirtualBox:~$ ls -ld usn
drwxrwxrwx 3 ise ise 4096 Oct  5 12:24 usn
ise@ise-VirtualBox:~$ ls -ld usn ise
drwxr-xr-x 2 ise ise 4096 Oct  5 12:27 ise
drwxrwxrwx 3 ise ise 4096 Oct  5 12:24 usn
ise@ise-VirtualBox:~$ chmod -R 755 . usn
chmod: changing permissions of './.dbus': Operation not permitted
chmod: cannot read directory './.dbus': Permission denied
chmod: changing permissions of './.gvfs': Operation not permitted
chmod: cannot read directory './.gvfs': Permission denied
chmod: changing permissions of './.cache/dconf': Operation not permitted
chmod: cannot read directory './.cache/dconf': Permission denied
ise@ise-VirtualBox:~$ chmod -R a+x usn
ise@ise-VirtualBox:~$ ls -ld usn
drwxr-xr-x 3 ise ise 4096 Oct  5 12:24 usn
ise@ise-VirtualBox:~$ ls -ld usn ise
drwxr-xr-x 2 ise ise 4096 Oct  5 12:27 ise
drwxr-xr-x 3 ise ise 4096 Oct  5 12:24 usn
ise@ise-VirtualBox:~$
```

Directory permissions

If the default directory permission are not altered, the **chmod** theory still applies.

mkdir command in Linux allows the user to create directories (also referred to as folders in some operating systems). This command can create multiple directories at once as well as set the permissions for the directories. It is important to note that the user executing this command must have enough permissions to create a directory in the parent directory, or he/she may receive a 'permission denied' error.

Example

```
$mkdir USN; ls -ld USN
```

```
drwxr-xr-x 2 atria atria 512 may 9 09:57 USN
```

the default permissions are different from those of ordinary files. The user has all the permission, and group and others have read and execute permissions only. The permissions of a directory also impact the security of its files.

```
ise@ise-VirtualBox: ~  
ise@ise-VirtualBox:~$ mkdir c_progs; ls -ld c_progs  
drwxrwxr-x 2 ise ise 4096 Oct  5 14:00 c_progs  
ise@ise-VirtualBox:~$
```

```
ise@ise-VirtualBox: ~/c_progs  
ise@ise-VirtualBox:~/c_progs$ ls -ld progs  
--w--w--- 1 ise ise 9 Oct  5 14:04 progs  
ise@ise-VirtualBox:~/c_progs$ ls -ld progs1  
drwxrwxr-x 2 ise ise 4096 Oct  5 14:07 progs1  
ise@ise-VirtualBox:~/c_progs$ chmod -r progs1 ; ls progs1  
ls: cannot open directory progs1: Permission denied  
ise@ise-VirtualBox:~/c_progs$
```

The shells interpretive cycle

Work on Wild cards

The metacharacters that are used to construct the generalized pattern for matching filenames belong to the category called wildcards.

<i>Wild-Card</i>	<i>Matches</i>
*	Any number of characters including none
?	A single character
[<i>ijk</i>]	A single character—either an <i>i</i> , <i>j</i> or <i>k</i>
[<i>x-z</i>]	A single character that is within the ASCII range of the characters <i>x</i> and <i>z</i>
[! <i>ijk</i>]	A single character that is not an <i>i</i> , <i>j</i> or <i>k</i> (Not in C shell)
[! <i>x-z</i>]	A single character that is not within the ASCII range of the characters <i>x</i> and <i>z</i> (Not in C shell)
{ <i>pat1,pat2...</i> }	<i>pat1, pat2</i> , etc. (Not in Bourne Shell)

A **wildcard** is a character that can be used as a substitute for any of a class of characters in a search, thereby greatly increasing the flexibility and efficiency of searches.

Work on removing the special meanings of wild cards

Work on Three standard files and redirection

Let's first understand what the term "terminal". In the context of redirection, the terminal is a generic name that represents the screen, display or keyboard. In command output and error messages on the terminal(keyboard). The shell associates three files with the terminal- two for many commands see as input and output. A stream is simply a sequence of bytes.

When a user logs in, the shell makes available three files representing three streams. Each stream is associated with a default device, and – generically speaking – this device is the terminal:

1. Standard Input – The file (or stream) representing input, which is connected to the keyboard.
2. Standard output – the file (or stream) representing output, which is connected to the display.
3. Standard error – the file (or stream) representing error messages that emanate from the command or shell. This is also connected to the display.

Work on Pipes

A pipe is a form of redirection (transfer of standard output to some other destination) that is used in Linux and other **Unix**-like operating systems to send the output of one command/program/process to another command/program/process for further processing. The Unix/Linux systems allow stdout of a command to be connected to stdin of another command. You can make it do so by using the pipe character '|'.

Pipe is used to combine two or more commands, and in this, the output of one command acts as input to another command, and this command's output may act as input to the next command and so on.

Basic and Extended Regular Expression

Based on the use of Meta characters, a regular expression can be divided in two categories; BRE (Basic Regular Expression) and ERE (Extended Regular Expression). A regular expression is a string that can be used to describe several sequences of characters. Regular expressions are used by several different Unix commands, including **ed**, **sed**, **awk**, **grep**, and to a more limited extent, **vi**. Here **SED** stands for **stream editor**. This stream-oriented editor was created exclusively for executing scripts. Thus, all the input you feed into it passes through and goes to STDOUT and it does not change the input file.

Like the shell's wild-cards which match similar filenames with a single expression, **grep** uses an expression of a different sort to match a group of similar patterns.

- []: Matches any one of a set characters
- [-] **with hyphen**: Matches any one of a range characters
- ^: The pattern following it must occur at the beginning of each line
- ^ **with []**: The pattern must not contain any character in the set specified
- \$: The pattern preceding it must occur at the end of each line
- . (**dot**): Matches any one character
- \ (**backslash**): Ignores the special meaning of the character following it
- *: zero or more occurrences of the previous character
- (**dot**).*: Nothing or any numbers of characters.

Extended Regular Expression

An expression which uses the later added Meta characters. To instruct grep command to use later added characters as Meta characters, an option -E is used. Let's take an example. In original implementation, the pipe sign (|) is defined as regular character while in new implementation, it is defined as a Meta character. If we use pipe sign without -E option, grep will treat it as a regular character. But if we use it with -E option, grep will treat it as a Meta character. As a Meta character, it is used to search multiple words. Let's search two users' information in file /etc/passwd with and without -E option.

1. The + and ?

The ERE set includes two special characters, + and?. They are often used in place of the * to restrict the matching scope. They signify the following:

+ - matches one or more occurrences of the previous character.

? - matches zero or one occurrences of the previous character.

In both cases, the emphasis is on the previous character. This means that b+ matches b, bb, bbb, etc. The expression b? matches either a single instance of b or nothing.

Example- for ? - Aggarwal and Agarwal note that the character g occurs only once or twice. So, gg? Now restricts the expansion to one or two gs only. The grep's -E option to use an ERE.

```
$ grep -E "[aA]gg?arwal" emp.lst
```

```
Aggarwal
```

agarwal

2. Matching multiple patterns (|, (and))

Without **-E** option, grep searched the pattern as a single word **sanjay|rick** in the file **/etc/passwd**. While with **-E** option, it separated the pattern in two words **sanjay** and **rick** and searched them individually.

```
#grep "sanjay|rick" /etc/passwd
#grep -E "sanjay|rick" /etc/passwd
```

```
File Edit View Search Terminal Help
[root@localhost ~]# grep "sanjay|rick" /etc/passwd
[root@localhost ~]# grep -E "sanjay|rick" /etc/passwd
sanjay:x:1000:1000:sanjay:/home/sanjay:/bin/bash
rick:x:1004:1010::/home/rick:/bin/bash
[root@localhost ~]#
```

The grep

Grep is an acronym that stands for **Global Regular Expression Print**. Grep is a Linux / Unix command-line tool used to search for a string of characters in a specified file. The text search pattern is called a regular expression. When it finds a match, it prints the line with the result. The grep command is handy when searching through large log files.

The grep command consists of three parts in its most basic form. The first part starts with grep, followed by the pattern that you are searching for. After the string comes the file name that the grep searches through.

The simplest grep command syntax looks like this:

```
test@test-VirtualBox: ~/Desktop/files
File Edit View Search Terminal Help
test@test-VirtualBox:~/Desktop/files$ grep phoenixNAP sample
```

The command can contain many options, pattern variations, and file names. Combine as many options as necessary to get the results you need. Below are the most common grep commands with examples.

To print any line from a file that contains a specific pattern of characters, in our case phoenix in the file sample2, run the command:

grep phoenix sample2

Grep will display every line where there is a match for the word phoenix. When executing this command, you do not get exact matches. Instead, the terminal prints the lines with words containing the string of characters you entered. Here is an example:



Work on Options used by grep

Table 13.1 Options Used by **grep**

<i>Option</i>	<i>Significance</i>
-i	Ignores case for matching
-v	Doesn't display lines matching expression
-n	Displays line numbers along with lines
-c	Displays count of number of occurrences
-l	Displays list of filenames only
-e <i>exp</i>	Specifies expression with this option. Can use multiple times. Also used for matching expression beginning with a hyphen.
-x	Matches pattern with entire line (doesn't match embedded patterns)
-f <i>file</i>	Takes patterns from <i>file</i> , one per line
-E	Treats pattern as an extended regular expression (ERE)
-F	Matches multiple fixed strings (in fgrep -style)

The Shell Programming

Ordinary and environment variables

An important Unix concept is the environment, which is defined by environment variables. Some are set by the system, others by you, yet others by the shell, or any program that loads another program. A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.

For example, first we set a variable TEST and then we access its value using the echo command

```
$TEST="Unix Programming"
```

```
$echo $TEST - It produces the following result.
```

```
Unix Programming
```

Note that the environment variables are set without using the \$ sign but while accessing them we use the \$ sign as prefix. These variables retain their values until we come out of the shell. When

you log in to the system, the shell undergoes a phase called initialization to set up the environment. This is usually a two-step process that involves the shell reading the following files

`/etc/profile`

`profile`

The process is as follows –

- 1) The shell checks to see whether the file `/etc/profile` exists.
- 2) If it exists, the shell reads it. Otherwise, this file is skipped. No error message is displayed.
- 3) The shell checks to see whether the file `.profile` exists in your home directory. Your home directory is the directory that you start out in after you log in.
- 4) If it exists, the shell reads it; otherwise, the shell skips it. No error message is displayed.
- 5) As soon as both of these files have been read, the shell displays a prompt – `$`

This is the prompt where you can enter commands in order to have them executed.

Note – The shell initialization process detailed here applies to all Bourne type shells

The .profile

The file `/etc/profile` is maintained by the system administrator of your Unix machine and contains shell initialization information required by all users on a system. The file `.profile` is under your control. You can add as much shell customization information as you want to this file. The minimum set of information that you need to configure includes –

- The type of terminal you are using.
- A list of directories in which to locate the commands.
- A list of variables affecting the look and feel of your terminal.

You can check your `.profile` available in your home directory. Open it using the vi editor and check all the variables set for your environment.

Work on read & readonly commands

Command line arguments

The Unix shell is used to run commands, and it allows users to pass run time arguments to these commands. These arguments, also known as command line parameters, that allows the users to either control the flow of the command or to specify the input data for the command.

`$*` - it stores the complete set of positional parameters as a single string.

`$#` - it is set to the number of arguments specified. This lets you design scripts that check whether the right number of arguments have been entered.

`$0` – holds the command name itself. You can link a shell script to be invoked by more than one name. The script logic can check `$0` to behave differently depending on the name by which it is invoked.

```
#!/bin/sh
# emp2.sh: Non-interactive version - uses command line arguments
#
echo "Program: $0"           # $0 contains the program name
echo "The number of arguments specified is $#"
```

The arguments are `$*` # All arguments stored in `$*`

```
grep "$1" $2
echo "\nJob Over"
```

The script, `emp2.sh`, runs `grep` with two positional parameters that are set by the script argument `director` and `emp.lst`:

```
$ emp2.sh director emp.lst
Program: emp2.sh
The number of arguments specified is 2
The arguments are director emp.lst
1006|chanchal singhvi |director |sales      |03/09/38|6700
6521|lalit chowdury  |director |marketing |26/09/45|8200

Job Over
```

When arguments are specified in this way, the first word (the command itself) is assigned to \$0, the second word (the first argument) to \$1, and the third word (the second argument) to \$2. You can use more positional parameters in this way up to \$9. All assignments to positional and special parameters are made by the shell.

Exit and exit status of command

C programs and shell scripts have a lot in common, and one of them is that they both use the same command(or function in C) to terminate a program. It has the name `exit` in the shell and `exit()` in C.

The command is generally run with a numeric argument:

- `Exit 0` - used when everything went fine
- `Exit 1` - used when something went wrong

These are two very common exit values. You don't need to place this statement at the end of every shell script because the shell understands when script execution is complete. The shell offers a variable (`$?`) and a command (`test`) that evaluates a command's exit status.

The parameters `$?` – the `$?` Stores the exit status of the last command. It has the value 0 if the command succeeds and a nonzero value if it fails. This parameter is set by `exit` 's argument.

If no exit status is specified, then `$?` Is set to zero(true).

```
$grep director emp.lst >/dev/null; echo $?
```

```
0          - success
```

```
$grep manager emp.lst >/dev/null; echo $?
```

```
1          - failure- in finding pattern
```

Work on Logical operators for conditional execution

The shell provides two operators that allow conditional execution- the && and ||

```
$ [-d ~/Myfolder] || mkdir ~/Myfolder
```

The test command and its shortcut

Test is used as part of the conditional execution of shell commands. Test exits with the status determined by EXPRESSION. Placing the EXPRESSION between square brackets ([and]) is

the same as testing the EXPRESSION with test. To see the exit status at the command prompt, echo the value "\$?". A value of 0 means the expression evaluated as true, and a value of 1 means the expression evaluated as false.

Syntax

```
test EXPRESSION  
  
[ EXPRESSION ]
```

The test works in 3 ways

- 1) Numeric Comparison - Compares two numbers
 - 2) String comparison - Compares two strings
 - 3) File tests - Checks a file attributes.
- 1) **Numeric Comparison** – the numeric comparison used by test have a form different from what you would have seen anywhere. They always begin with a – (hyphen), followed by a two-letter string, and enclosed on either side by whitespace. Here's a typical operator:

-ne not equal

- | | |
|------------------------------------|-------------------------------|
| 1) -eq is equal to | if ["\$a" -eq "\$b"] |
| 2) -ne is not equal to | if ["\$a" -ne "\$b"] |
| 3) -gt is greater than | if ["\$a" -gt "\$b"] |
| 4) -ge is greater than or equal to | if ["\$a" -ge "\$b"] |
| 5) -lt is less than | if ["\$a" -lt "\$b"] |
| 6) -le is less than or equal to | if ["\$a" -le "\$b"] |

Example:

```

ranjitha@ranjitha-VirtualBox:~$ x=5; y=7; z=7
ranjitha@ranjitha-VirtualBox:~$ test $x -eq $y ; echo $?
1
ranjitha@ranjitha-VirtualBox:~$ test $x -lt $y ; echo $?
0
ranjitha@ranjitha-VirtualBox:~$ test $z -gt $y ; echo $?
1
ranjitha@ranjitha-VirtualBox:~$ test $z -eq $y ; echo $?
0
ranjitha@ranjitha-VirtualBox:~$
    
```

2) String comparison

Test can be used to compare strings with yet another set of operators. Equality is performed with = and inequality with the c-type operator !=. others test checks can be negated by the ! too. Thus [! -z \$string] negates [-z \$string].

Table 14.3 String Tests Used by test

Test	True if
$s1 = s2$	String $s1 = s2$
$s1 != s2$	String $s1$ is not equal to $s2$
$-n stg$	String stg is not a null string
$-z stg$	String stg is a null string
stg	String stg is assigned and not null
$s1 == s2$	String $s1 = s2$ (Korn and Bash only)

Example:

```

ranjitha@ranjitha-VirtualBox:~$ test howto = forge && echo "same"
Same
ranjitha@ranjitha-VirtualBox:~$ test howto = howto && echo "same"
same
ranjitha@ranjitha-VirtualBox:~$
    
```

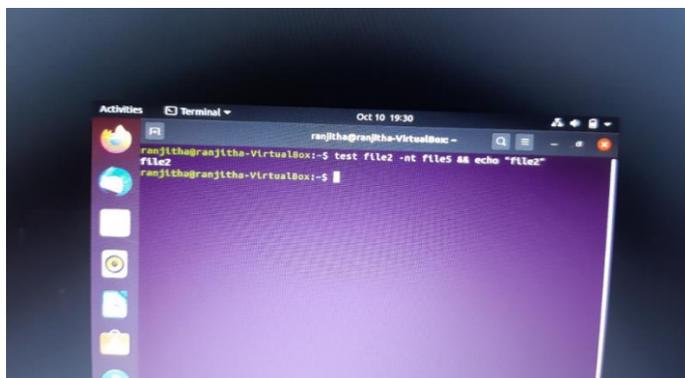
3) File tests - Checks a file attributes

Test can be used to test the various file attributes like its type (file, directory or symbolic link) or its permissions (read, write, execute).

Table 14.4 File-related Tests with `test`

Test	True if File
<code>-f file</code>	<code>file</code> exists and is a regular file
<code>-r file</code>	<code>file</code> exists and is readable
<code>-w file</code>	<code>file</code> exists and is writable
<code>-x file</code>	<code>file</code> exists and is executable
<code>-d file</code>	<code>file</code> exists and is a directory
<code>-s file</code>	<code>file</code> exists and has a size greater than zero
<code>-e file</code>	<code>file</code> exists (Korn and Bash only)
<code>-u file</code>	<code>file</code> exists and has SUID bit set
<code>-k file</code>	<code>file</code> exists and has sticky bit set
<code>-l file</code>	<code>file</code> exists and is a symbolic link (Korn and Bash only)
<code>f1 -nt f2</code>	<code>f1</code> is newer than <code>f2</code> (Korn and Bash only)
<code>f1 -ot f2</code>	<code>f1</code> is older than <code>f2</code> (Korn and Bash only)
<code>f1 -ef f2</code>	<code>f1</code> is linked to <code>f2</code> (Korn and Bash only)

Example



While looping

The while loop has a similar role to play; it repeatedly iterates the loop as long as the

The general syntax:

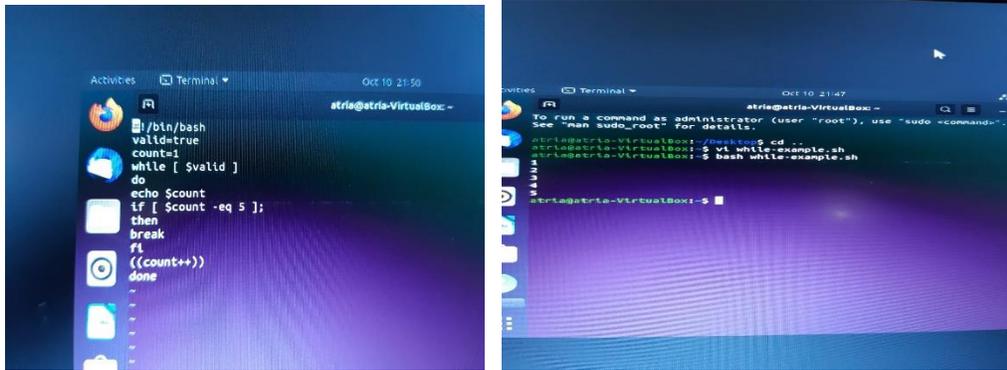
while (condition is true)

do

commands

done

The command enclosed by `do` and `done` are executed repeatedly as long as condition remains true.



if statement

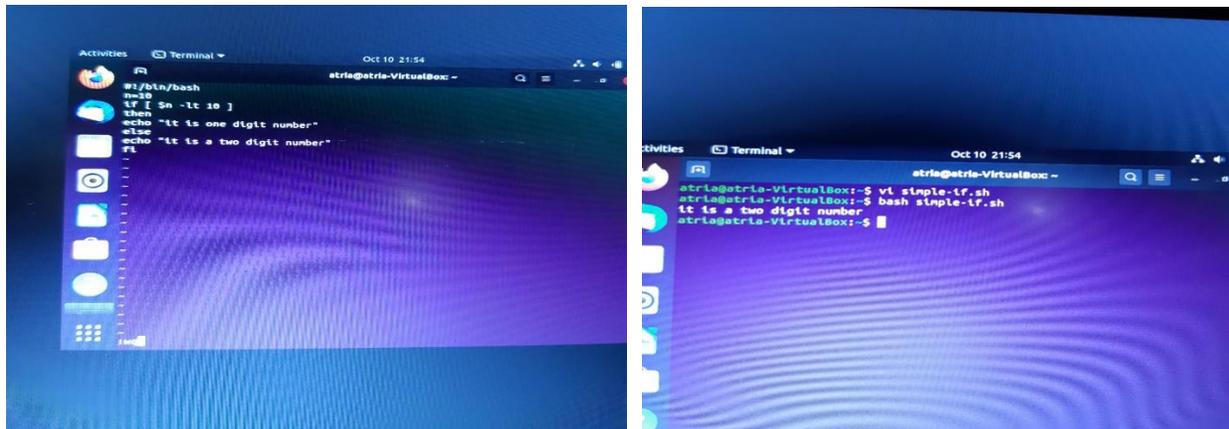
Like any programming language, awk supports conditional structures (the if statement) and loops (while and for). They all execute a body of statements as long as their control command evaluates to true. This control command is simply a condition that is specified in the first line of the construct.

The if statement permits two-way decision making, and its behavior is well known to all programmers. The construct has also been elaborated in Section 13.6 where it appears in three forms. The statement in awk takes this form:

syntax:

```
if command is successful
then
    execute commands
else
    execute commands
fi
```

if also requires a then. It evaluates the success or failure of the command that is specified in its “command line”. If command succeeds, the sequence of commands following it is executed. If command fails, then the else statement (if present) is executed. Every if is closed with a corresponding fi.



For looping

The shell's for loop differs in structure from the ones used in other programming languages. There is no three part structure as used in c, awk and perl.

syntax

for variable in list

do

commands

loop body

done

The loop body also uses the keywords do and done, but the additional parameters here are variable and list.

Example:

```
for file in ch1 ch2; do
```

```
> cp $file ${file}.bak
```

```
> echo $file copied to $file.bak
```

```
done
```

Output:

```
ch1 copied to ch1.bak
```

```
ch2 copied to ch2.bak
```

Sources of list -

List from variables: Series of variables are evaluated by the shell before executing the loop

Example:

```
$ for var in $PATH $HOME; do echo "$var" ; done
```

Output:

```
/bin:/usr/bin:/home/local/bin;
```

```
/home/user1
```

List from command substitution: Command substitution is used for creating a list. This is used when list is large.

Example:

```
$ for var in `cat clist`
```

List from wildcards: Here the shell interprets the wildcards as filenames.

Example:

```
for file in *.htm *.html ; do
```

```
sed 's/strong/STRONG/g
```

```
s/img src/IMG SRC/g' $file > $$
```

```
mv $$ $file
```

```
done
```

Case control statement

The case statement is the second conditional offered by the shell. It doesn't have a parallel either in C (Switch is similar) or perl. The statement matches an expression for more than one

alternative, and uses a compact construct to permit multiway branching. case also handles string tests, but in a more efficient manner than if.

Syntax:

case expression in

Pattern 1) commands1 ;;

Pattern 2) commands2 ;;

Pattern3) commands3 ;; ...

Esac

Case first matches expression with pattern1. if the match succeeds, then it executes commands1, which may be one or more commands. If the match fails, then pattern2 is matched and so forth. Each command list is terminated with a pair of semicolon and the entire construct is closed with esac (reverse of case).

Example:

```
#!/bin/sh
```

```
#echo " Menu\n
```

```
1. List of files\n2. Processes of user\n3. Today's Date
```

```
4. Users of system\n5.Quit\nEnter your option: \c"
```

```
read choice
```

```
case "$choice" in
```

```
1) ls -l;;
```

```
2) ps -f ;;
```

```
3) date ;;
```

4) who ;;

5) exit ;;

*) echo "Invalid option"

esac

Output

```
$ menu.sh
```

Menu

1. List of files
2. Processes of user
3. Today's Date
4. Users of system
5. Quit

Enter your option: 3

Mon Oct 8 08:02:45 IST 2007

Work on The set & shift commands & handling positional parameters

Module – 3: Unix File APIs

General File APIs

There are special API's to create these types of files. There is a set of Generic API's that can be used to manipulate and create more than one type of files. These API's are:

API	Use
open	Opens a file for data access
read	Reads data from a file
write	Writes data to a file
lseek	Allows random access of data in a file
close	Terminates the connection to a file
stat, fstat	Queries attributes of a file
chmod	Changes access permission of a file
chown	Changes UID and/or GID of a file
utime	Changes last modification and access time stamps of a file
link	Creates a hard link to a file
unlink	Deletes a hard link of a file
umask	Sets default file creation mask

These general APIs are explained as follows.

1. Open - This is used to establish a connection between a process and a file i.e. it is used to open an existing file for data transfer function or else it may be also be used to create a new file. The returned value of the open system call is the file descriptor (row number of the file table), which contains the inode information. The prototype of open function is

```
#include<sys/types.h>
#include<sys/fcntl.h>
int open(const char *pathname, int accessmode, mode_t permission);
```

If successful, open returns a nonnegative integer representing the open file descriptor. If unsuccessful, open returns -1.

The **first argument** is the name of the file to be created or opened. This may be an absolute pathname or relative pathname.

If the given pathname is symbolic link, the open function will resolve the symbolic link reference to a non symbolic link file to which it refers.

The **second argument** is access modes, which is an integer value that specifies how actually the file should be accessed by the calling process.

Generally, the access modes are specified in `<fcntl.h>`. Various access modes are:

`O_RDONLY` - open for reading file only

`O_WRONLY` - open for writing file only

`O_RDWR` - opens for reading and writing file.

There are other access modes, which are termed as access modifier flags, and one or more of the following can be specified by bitwise-ORing them with one of the above access mode flags to alter the access mechanism of the file.

<code>O_APPEND</code>	- Append data to the end of file.
<code>O_CREAT</code>	- Create the file if it doesn't exist
<code>O_EXCL</code>	- Generate an error if <code>O_CREAT</code> is also specified and the file already exists.
<code>O_TRUNC</code>	- If file exists discard the file content and set the file size to zero bytes.
<code>O_NONBLOCK</code>	- Specify subsequent read or write on the file should be non-blocking.
<code>O_NOCTTY</code>	- Specify not to use terminal device file as the calling process control terminal.

To illustrate the use of the above flags, the following example statement opens a file called `/usr/abc/usp` for read and write in append mode:

```
int fd=open("/usr/abc/usp",O_RDWR | O_APPEND,0);
```

If the file is opened in read only, then no other modifier flags can be used. If a file is opened in write only or read write, then we are allowed to use any modifier flags along with them. The third argument is used only when a new file is being created. The symbolic names for file permission are given in the table.

symbol	meaning
S_IRUSR	read by owner
S_IWUSR	write by owner
S_IXUSR	execute by owner
S_IRWXU	read, write, execute by owner
S_IRGRP	read by group
S_IWGRP	write by group
S_IXGRP	execute by group
S_IRWXG	read, write, execute by group
S_IROTH	read by others
S_IWOTH	write by others
S_IXOTH	execute by others
S_IRWXO	read, write, execute by others

2. **Creat** - This system call is used to create new regular files. The prototype of creat is

```
#include <sys/types.h>
#include<unistd.h>
int creat(const char *pathname, mode_t mode);
```

Returns: file descriptor opened for write-only if OK, -1 on error. The first argument pathname specifies name of the file to be created. The second argument mode_t, specifies permission of a file to be accessed by owner group and others. The creat function can be implemented using open function as:

```
#define creat(path_name, mode)
open (pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

3. **Read** - The read function fetches a fixed size of block of data from a file referenced by a given file descriptor. The prototype of read function is:

```
#include<sys/types.h>
#include<unistd.h>
size_t read(int fdesc, void *buf, size_t nbyte);
```

If successful, read returns the number of bytes actually read. If unsuccessful, read returns -1.

The first argument is an integer, fdesc that refers to an opened file. The second argument, buf is the address of a buffer holding any data read. The third argument specifies how many bytes of data are to be read from the file. The size_t data type is defined in the

<sys/types.h> header and should be the same as unsigned int. There are several cases in which the number of bytes actually read is less than the amount requested:

- When reading from a regular file, if the end of file is reached before the requested number of bytes has been read. For example, if 30 bytes remain until the end of file and we try to read 100 bytes, read returns 30. The next time we call read, it will return 0 (end of file).
 - When reading from a terminal device. Normally, up to one line is read at a time.
 - When reading from a network. Buffering within the network may cause less than the requested amount to be returned.
 - When reading from a pipe or FIFO. If the pipe contains fewer bytes than requested, read will return only what is available.
4. Write - The write system call is used to write data into a file. The write function puts data to a file in the form of fixed block size referred by a given file descriptor. The prototype of write is

```
#include<sys/types.h>
#include<unistd.h>
ssize_t write(int fdesc, const void *buf, size_t size);
```

If successful, write returns the number of bytes actually written.

If unsuccessful, write returns -1.

The first argument, fdesc is an integer that refers to an opened file.

The second argument, buf is the address of a buffer that contains data to be written.

The third argument, size specifies how many bytes of data are in the buf argument.

The return value is usually equal to the number of bytes of data successfully written to a file. (size value)

5. Close - The close system call is used to terminate the connection to a file from a process. The prototype of the close is

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

If successful, close returns 0. If unsuccessful, close returns -1.

The argument fdesc refers to an opened file.

Close function frees the unused file descriptors so that they can be reused to reference other files. This is important because a process may open up to OPEN_MAX files at any time and the close function allows a process to reuse file descriptors to access more than OPEN_MAX files in the course of its execution. The close function de-allocates system resources like file table entry and memory buffer allocated to hold the read/write.

- 6. Fcntl - The fcntl function helps a user to query or set flags and the close-on-exec flag of any file descriptor. The prototype of fcntl is

```
#include<fcntl.h>
int fcntl(int fdesc, int cmd, ...);
```

The first argument is the file descriptor.

The second argument cmd specifies what operation has to be performed.

The third argument is dependent on the actual cmd value.

The possible cmd values are defined in <fcntl.h> header.

cmd value	Use
F_GETFL	Returns the access control flags of a file descriptor fdesc
F_SETFL	Sets or clears access control flags that are specified in the third argument to fcntl. The allowed access control flags are O_APPEND & O_NONBLOCK
F_GETFD	Returns the close-on-exec flag of a file referenced by fdesc. If a return value is zero, the flag is off; otherwise on.
F_SETFD	Sets or clears the close-on-exec flag of a fdesc. The third argument to fcntl is an integer value, which is 0 to clear the flag, or 1 to set the flag
F_DUPFD	Duplicates file descriptor fdesc with another file descriptor. The third argument to fcntl is an integer value which specifies that the duplicated file descriptor must be greater than or equal to that value. The return value of fcntl is the duplicated file descriptor

The fcntl function is useful in changing the access control flag of a file descriptor.

For example: after a file is opened for blocking read-write access and the process needs to change the access to non-blocking and in write-append mode, it can call:

```
int cur_flags=fcntl(fdesc,F_GETFL);
int rc=fcntl(fdesc,F_SETFL,cur_flag | O_APPEND | O_NONBLOCK);
```

The following example reports the close-on-exec flag of fdesc, sets it to on afterwards:

```
cout<<fdesc<<"close-on-exec"<<fcntl(fdesc,F_GETFD)<<endl;
(void)fcntl(fdesc,F_SETFD,1); //turn on close-on-exec flag
```

The following statements change the standard input of a process to a file called FOO:

```
int fdesc=open("FOO",O_RDONLY);           //open FOO for read
close(0);                                 //close standard input
if(fcntl(fdesc,F_DUPFD,0)==-1)
perror("fcntl");                          //stdin from FOO now
char buf[256];
int rc=read(0,buf,256);                   //read data from FOO
```

The dup and dup2 functions in UNIX perform the same file duplication function as fcntl. They can be implemented using fcntl as:

```
#define dup(fdesc)    fcntl(fdesc, F_DUPFD,0)
#define dup2(fdesc1,fd2)    close(fd2),fcntl(fdesc,F_DUPFD,fd2)
```

7. Lseek - The lseek function is also used to change the file offset to a different value. Thus lseek allows a process to perform random access of data on any opened file. The prototype of lseek is

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

On success it returns new file offset, and -1 on error. The first argument fdesc, is an integer file descriptor that refer to an opened file. The second argument pos, specifies a byte offset to be added to a reference location in deriving the new file offset value. The third argument whence, is the reference location.

Whence value	Reference location
SEEK_CUR	Current file pointer address
SEEK_SET	The beginning of a file
SEEK_END	The end of a file

They are defined in the <unistd.h> header. If an lseek call will result in a new file offset that is beyond the current end-of-file, two outcomes possible are:

- If a file is opened for read-only, lseek will fail.

- If a file is opened for write access, lseek will succeed.
 - The data between the end-of-file and the new file offset address will be initialised with NULL characters.
8. Link - The link function creates a new link for the existing file. The prototype of the link function is

```
#include <unistd.h>
int link(const char *cur_link, const char *new_link);
```

If successful, the link function returns 0. If unsuccessful, link returns -1. The first argument cur_link, is the pathname of existing file. The second argument new_link is a new pathname to be assigned to the same file. If this call succeeds, the hard link count will be increased by 1. The UNIX ln command is implemented using the link API.

```
/*test_ln.c*/
#include<iostream.h>
#include<stdio.h>
#include<unistd.h>

int main(int argc, char* argv)
{
    if(argc!=3)
    {
        cerr<<"usage:"<<argv[0]<<"<<"<<"src_file"<<"dest_file"<<"\n";
        return 0;
    }
    if(link(argv[1],argv[2])==-1)
    {
        perror("\link");
        return 1;
    }
    return 0;
}
```

9. Unlink - The unlink function deletes a link of an existing file. This function decreases the hard link count attributes of the named file, and removes the file name entry of the link from directory file. A file is removed from the file system when its hard link count is zero and no process has any file descriptor referencing that file. The prototype of unlink is

```
#include <unistd.h>
int unlink(const char * cur_link);
```

If successful, the unlink function returns 0. If unsuccessful, unlink returns -1. The argument cur_link is a path name that references an existing file.

ANSI C defines the rename function which does the similar unlink operation. The prototype of the rename function is:

```
#include<stdio.h>
int rename(const char * old_path_name, const char * new_path_name);
```

The UNIX mv command can be implemented using the link and unlink APIs as shown:

```
#include <iostream.h>
#include <unistd.h>
#include<string.h>
int main ( int argc, char *argv[ ] )
{
    if (argc != 3 || strcmp(argv[1],argv[2]))
        cerr<<"usage:"<<argv[0]<<"<<"<old_link><new_link>\n";
    else if(link(argv[1],argv[2]) == 0)
        return unlink(argv[1]);
    return 1;
}
```

10. Stat, fstat - The stat and fstat function retrieves the file attributes of a given file. The only difference between stat and fstat is that the first argument of a stat is a file pathname, where as the first argument of fstat is file descriptor. The prototypes of these functions are

```
#include<sys/stat.h>
#include<unistd.h>

int stat(const char *pathname, struct stat *statv);
int fstat(const int fdesc, struct stat *statv);
```

The second argument to stat and fstat is the address of a struct stat-typed variable which is defined in the <sys/stat.h> header. Its declaration is as follows:

```
struct stat
{
    dev_t      st_dev;      /* file system ID */
    ino_t      st_ino;     /* file inode number */
    mode_t     st_mode;    /* contains file type and permission */
    nlink_t    st_nlink;   /* hard link count */
    uid_t      st_uid;     /* file user ID */
    gid_t      st_gid;     /* file group ID */
    dev_t      st_rdev;    /*contains major and minor device#*/
    off_t      st_size;    /* file size in bytes */
    time_t     st_atime;   /* last access time */
    time_t     st_mtime;   /* last modification time */
    time_t     st_ctime;   /* last status change time */
};
```

The return value of both functions is 0 if they succeed and -1 if they fail errno contains an error status code. The lstat function prototype is the same as that of stat:

```
int lstat(const char * path_name, struct stat* statv);
```

We can determine the file type with the macros as shown.

macro	Type of file
S_ISREG()	regular file
S_ISDIR()	directory file
S_ISCHR()	character special file
S_ISBLK()	block special file
S_ISFIFO()	pipe or FIFO
S_ISLNK()	symbolic link
S_ISSOCK()	socket

11. access - The access system call checks the existence and access permission of user to a named file. The prototype of access function is:

```
#include<unistd.h>
int access(const char *path_name, int flag);
```

On success access returns 0, on failure it returns -1. The first argument is the pathname of a file. The second argument flag, contains one or more of the following bit flag .

Bit flag	Uses
F_OK	Checks whether a named file exist
R_OK	Test for read permission
W_OK	Test for write permission
X_OK	Test for execute permission

The flag argument value to an access call is composed by bitwise-ORing one or more of the above bit flags as shown:

```
int rc=access("/usr/divya/usp.txt",R_OK | W_OK);
```

example to check whether a file exists:

```
if(access("/usr/divya/usp.txt", F_OK)==-1)
    printf("file does not exists");
else
    printf("file exists");
```

12. chmod, fchmod - The chmod and fchmod functions change file access permissions for owner, group & others as well as the set_UID, set_GID and sticky flags. A process must have the effective UID of either the super-user/owner of the file. The prototypes of these functions are

```
#include<sys/types.h>
#include<sys/stat.h>
#include<unistd.h>

int chmod(const char *pathname, mode_t flag);
int fchmod(int fdesc, mode_t flag);
```

The pathname argument of chmod is the path name of a file whereas the fdesc argument of fchmod is the file descriptor of a file. The chmod function operates on the specified file, whereas the fchmod function operates on a file that has already been opened. To change the permission bits of a file, the effective user ID of the process must be equal to

the owner ID of the file, or the process must have super-user permissions. The mode is specified as the bitwise OR of the constants shown below

<u>Mode</u>	<u>Description</u>
S_ISUID	set-user-ID on execution
S_ISGID	set-group-ID on execution
S_ISVTX	saved-text (sticky bit)
S_IRWXU	read, write, and execute by user (owner)
S_IRUSR	read by user (owner)
S_IWUSR	write by user (owner)
S_IXUSR	execute by user (owner)
S_IRWXG	read, write, and execute by group
S_IRGRP	read by group
S_IWGRP	write by group
S_IXGRP	execute by group
S_IRWXO	read, write, and execute by other (world)
S_IROTH	read by other (world)
S_IWOTH	write by other (world)
S_IXOTH	execute by other (world)

13. chown, fchown, lchown - The chown functions changes the user ID and group ID of files.

The prototypes of these functions are

```
#include<unistd.h>
#include<sys/types.h>

int chown(const char *path_name, uid_t uid, gid_t gid);
int fchown(int fdesc, uid_t uid, gid_t gid);
int lchown(const char *path_name, uid_t uid, gid_t gid);
```

The path_name argument is the path name of a file. The uid argument specifies the new user ID to be assigned to the file. The gid argument specifies the new group ID to be assigned to the file.

```
/* Program to illustrate chown function */
#include<iostream.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<unistd.h>
#include<pwd.h>

int main(int argc, char *argv[ ])
{
    if(argc>3)
    {
        cerr<<"usage: "<<argv[0]<<"<<"<<usr_name<<file>. .. \n";
        return 1;
    }

    struct passwd *pwd = getpwuid(argv[1]) ;
    uid_t      UID = pwd ? pwd -> pw uid : -1 ;
    struct      stat  statv;

    if (UID == (uid_t)-1)
        cerr <<"Invalid user name";
    else for (int i = 2; i < argc ; i++)
        if (stat(argv[i], &statv)==0)
        {
            if (chown(argv[i], UID, statv.st_gid))
                perror ("chown");
            else
                perror ("stat");
        }
    return 0;
}
```

The above program takes at least two command line arguments:

- The first one is the user name to be assigned to files
- The second and any subsequent arguments are file path names.

The program first converts a given user name to a user ID via getpwuid function. If that succeeds, the program processes each named file as follows: it calls stat to get the file group ID, then it calls chown to change the file user ID. If either the stat or chown fails, error is displayed.

14. Utime - The utime function modifies the access time and the modification time stamps of a file. The prototype of utime function is

```
#include<sys/types.h>
#include<unistd.h>
#include<utime.h>

int utime(const char *path_name, struct utimbuf *times);
```

On success it returns 0, on failure it returns -1. The path_name argument specifies the path name of a file. The times argument specifies the new access time and modification time for the file.

The struct utimbuf is defined in the <utime.h> header as:

```
struct utimbuf
{
    time_t      actime;      /* access time */
    time_t      modtime;    /* modification time */
}
```

The time_t datatype is an unsigned long and its data is the number of the seconds elapsed since the birthday of UNIX : 12 AM , Jan 1 of 1970. If the times (variable) is specified as NULL, the function will set the named file access and modification time to the current time. If the times (variable) is an address of the variable of the type struct utimbuf, the function will set the file access time and modification time to the value specified by the variable.

File and Record Locking

Multiple processes performs read and write operation on the same file concurrently. This provides a means for data sharing among processes, but it also renders difficulty for any process in determining when the other process can override data in a file. So, in order to overcome this drawback UNIX and POSIX standard support file locking mechanism. File locking is applicable for regular files. Only a process can impose a write lock or read lock on either a portion of a file or on the entire file. The differences between the read lock and the write lock is that when write lock is set, it prevents the other process from setting any over-lapping read or write lock on the locked file. Similarly when a read lock is set, it prevents other processes from setting any overlapping write locks on the locked region. The intension of the write lock is to prevent other processes from both reading and writing the locked region while the process that sets the lock is

modifying the region, so write lock is termed as “Exclusive lock”. The use of read lock is to prevent other processes from writing to the locked region while the process that sets the lock is reading data from the region. Other processes are allowed to lock and read data from the locked regions. Hence a read lock is also called as “shared lock “. File lock may be mandatory if they are enforced by an operating system kernel. If a mandatory exclusive lock is set on a file, no process can use the read or write system calls to access the data on the locked region. These mechanisms can be used to synchronize reading and writing of shared files by multiple processes. If a process locks up a file, other processes that attempt to write to the locked regions are blocked until the former process releases its lock. Problem with mandatory lock is – if a runaway process sets a mandatory exclusive lock on a file and never unlocks it, then, no other process can access the locked region of the file until the runaway process is killed or the system has to be rebooted. If locks are not mandatory, then it has to be advisory lock. A kernel at the system call level does not enforce advisory locks. This means that even though a lock may be set on a file, no other processes can still use the read and write functions to access the file. To make use of advisory locks, process that manipulate the same file must co-operate such that they follow the given below procedure for every read or write operation to the file.

- Try to set a lock at the region to be accesses. If this fails, a process can either wait for the lock request to become successful.
- After a lock is acquired successfully, read or write the locked region.
- Release the lock.

If a process sets a read lock on a file, for example from address 0 to 256, then sets a write lock on the file from address 0 to 512, the process will own only one write lock on the file from 0 to 512, the previous read lock from 0 to 256 is now covered by the write lock and the process does not own two locks on the region from 0 to 256. This process is called “Lock Promotion”.

Furthermore, if a process now unblocks the file from 128 to 480, it will own two write locks on the file: one from 0 to 127 and the other from 481 to 512. This process is called “Lock Splitting”. UNIX systems provide `fcntl` function to support file locking. By using `fcntl` it is possible to impose read or write locks on either a region or an entire file.

The prototype of `fcntl` is

```
#include<fcntl.h>
int fcntl(int fdesc, int cmd_flag, ... );
```

The first argument specifies the file descriptor. The second argument cmd_flag specifies what operation has to be performed. If fcntl is used for file locking then it can values as

- F_SETLK sets a file lock, do not block if this cannot succeed immediately.
- F_SETLKW sets a file lock and blocks the process until the lock is acquired.
- F_GETLK queries as to which process locked a specified region of file.

For file locking purpose, the third argument to fcntl is an address of a struct flock type variable. This variable specifies a region of a file where lock is to be set, unset or queried.

```
struct flock
{

short l_type; /* what lock to be set or to unlock file */
short l_whence; /* Reference address for the next field */
off_t l_start; /*offset from the l_whence reference addr*/
off_t l_len; /*how many bytes in the locked region */
pid_t l_pid; /*pid of a process which has locked the file */
};
```

The l_type field specifies the lock type to be set or unset. The possible values, which are defined in the <fcntl.h> header, and their uses are

L_type value	Use
F_RDLCK	Set a read lock on a specified region
F_WRLCK	Set a write lock on a specified region
F_UNLCK	Unlock a specified region

The l_whence, l_start & l_len define a region of a file to be locked or unlocked. The possible values of l_whence and their uses are

l_whence value	Use
SEEK_CUR	The l_start value is added to current file pointer address
SEEK_SET	The l_start value is added to byte 0 of the file
SEEK_END	The l_start value is added to the end of the file

□ A lock set by the fcntl API is an advisory lock but we can also use fcntl for mandatory locking purpose with the following attributes set before using fcntl

- Turn on the set-GID flag of the file.
- Turn off the group execute right permission of the file.

In the given example program we have performed a read lock on a file “divya” from the 10th byte to 25th byte.

Example Program

```
#include <unistd.h>
#include <fcntl.h>
int main ( )
{
    int fd;
    struct flock lock;
    fd=open("divya",O_RDONLY);
    lock.l_type=F_RDLCK;
    lock.l_whence=0;
    lock.l_start=10;
    lock.l_len=15;
    fcntl(fd,F_SETLK,&lock);
}
```

Directory File APIs

A Directory file is a record-oriented file, where each record stores a file name and the inode number of a file that resides in that directory. Directories are created with the mkdir API and deleted with the rmdir API. The prototype of mkdir is

```
#include<sys/stat.h>
#include<unistd.h>

int mkdir(const char *path_name, mode_t mode);
```

The first argument is the path name of a directory file to be created. The second argument mode, specifies the access permission for the owner, groups and others to be assigned to the file. This function creates a new empty directory. The entries for “.” and “..” are automatically created. The specified file access permission, mode, are modified by the file mode creation mask of the

process. To allow a process to scan directories in a file system independent manner, a directory record is defined as struct dirent in the <dirent.h> header for UNIX. Some of the functions that are defined for directory file operations in the above header are

```
#include<sys/types.h>
#if defined (BSD)&&!_POSIX_SOURCE
    #include<sys/dir.h>
    typedef struct direct Dirent;
#else
    #include<dirent.h>
    typedef struct direct Dirent;
#endif

DIR *opendir(const char *path_name);
Dirent *readdir(DIR *dir_fdsec);
int closedir(DIR *dir_fdsec); void
rewinddir(DIR *dir_fdsec);
```

The uses of these functions are

Function	Use
opendir	Opens a directory file for read-only. Returns a file handle dir * for future reference of the file.
readdir	Reads a record from a directory file referenced by dir-fdesc and returns that record information.
rewinddir	Resets the file pointer to the beginning of the directory file referenced by dir-fdesc. The next call to readdir will read the first record from the file.
closedir	closes a directory file referenced by dir-fdesc.

An empty directory is deleted with the rmdir API. The prototype of rmdir is

```
#include<unistd.h>
int rmdir (const char * path_name);
```

If the link count of the directory becomes 0, with the call and no other process has the directory open then the space occupied by the directory is freed. UNIX systems have defined additional functions for random access of directory file records.

Function	Use
telldir	Returns the file pointer of a given dir_fdsec
seekdir	Changes the file pointer of a given dir_fdsec to a specified address

Device File APIs

Device files are used to interface physical device with application programs. A process with superuser privileges to create a device file must call the `mknod` API. The user ID and group ID attributes of a device file are assigned in the same manner as for regular files. When a process reads or writes to a device file, the kernel uses the major and minor device numbers of a file to select a device driver function to carry out the actual data transfer. Device file support is implementation dependent. UNIX System defines the `mknod` API to create device files. The prototype of `mknod` is

```
#include<sys/stat.h>
#include<unistd.h>

int mknod(const char* path_name, mode_t mode, int device_id);
```

The first argument `pathname` is the pathname of a device file to be created. The second argument `mode` specifies the access permission, for the owner, group and others, also `S_IFCHR` or `S_IFBLK` flag to be assigned to the file. The third argument `device_id` contains the major and minor device number.

Example

```
mknod("SCSI5",S_IFBLK | S_IRWXU | S_IRWXG | S_IRWXO,(15<<8) | 3);
```

The above function creates a block device file “abc”, to which all the three i.e. read, write and execute permission is granted for user, group and others with major number as 8 and minor number 3. On success `mknod` API returns 0 , else it returns -1. The following `test_mknod.C` program illustrates the use of the `mknod`, `open`, `read`, `write` and `close` APIs on a block device file.

```
#include<iostream.h>
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<unistd.h>
#include<fcntl.h>
#include<sys/stat.h>

int main(int argc, char* argv[])
{
    if(argc!=4)
    {
        cout<<"usage:"<<argv[0]<<"<<"<<"file"<<"<<"major_no"<<"<<"minor_no">";
        return 0;
    }
    int major=atoi(argv[2],minor=atoi(argv[3]);
    (void) mknod(argv[1], S_IFCHR | S_IRWXU | S_IRWXG | S_IRWXO, (major<<8) | minor);

    int rc=1,fd=open(argv[1],O_RDWR | O_NONBLOCK | O_NOCTTY);
    char buf[256];
    while(rc && fd!=-1)
    if((rc=read(fd,buf,sizeof(buf)))<0)
        perror("read");

    else if(rc)
        cout<<buf<<endl;
    close(fd);
}
```

FIFO File APIs

FIFO files are sometimes called named pipes. Pipes can be used only between related processes when a common ancestor has created the pipe. Creating a FIFO is similar to creating a file. Indeed the pathname for a FIFO exists in the file system. The prototype of `mkfifo` is

```
#include<sys/types.h>
#include<sys/stat.h>
#include<unistd.h>

int mkfifo(const char *path_name, mode_t mode);
```

The first argument `pathname` is the `pathname(filename)` of a FIFO file to be created. The second argument `mode` specifies the access permission for user, group and others and as well as the `S_IFIFO` flag to indicate that it is a FIFO file. On success it returns 0 and on failure it returns `-1`.

Example

```
mkfifo("FIFO5",S_IFIFO | S_IRWXU | S_IRGRP | S_ROTH);
```

The above statement creates a FIFO file “divya” with read-write-execute permission for user and only read permission for group and others. Once we have created a FIFO using `mkfifo`, we open it using `open`. Indeed, the normal file I/O functions (`read`, `write`, `unlink` etc) all work with FIFOs. When a process opens a FIFO file for reading, the kernel will block the process until there is another process that opens the same file for writing. Similarly whenever a process opens a FIFO file write, the kernel will block the process until another process opens the same FIFO for reading. This provides a means for synchronization in order to undergo inter-process communication. If a particular process tries to write something to a FIFO file that is full, then that process will be blocked until another process has read data from the FIFO to make space for the process to write. Similarly, if a process attempts to read data from an empty FIFO, the process will be blocked until another process writes data to the FIFO. From any of the above condition if the process doesn't want to get blocked then we should specify `O_NONBLOCK` in the `open` call to the FIFO file. If the data is not ready for read/write then `open` returns `-1` instead of process getting blocked. If a process writes to a FIFO file that has no other process attached to it for read, the kernel will send `SIGPIPE` signal to the process to notify that it is an illegal operation. Another method to create FIFO files (not exactly) for inter-process communication is to use the pipe system call. The prototype of pipe is

```
#include <unistd.h>
int pipe(int fds[2]);
```

Returns 0 on success and `-1` on failure. If the pipe call executes successfully, the process can read from `fd[0]` and write to `fd[1]`. A single process with a pipe is not very useful. Usually a parent process uses pipes to communicate with its children.

The following `test_fifo.C` example illustrates the use of `mkfifo`, `open`, `read`, `write` and `close` APIs for a FIFO file:

```
#include<iostream.h>
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<fcntl.h>
#include<sys/stat.h>
#include<string.h>
#include<errno.h>

int main(int argc, char* argv[])
{
    if(argc!=2 && argc!=3)
    {
        cout<<"usage:"<<argv[0]<<"<<"<file> [<arg>]";
        return 0;
    }
    int fd;
    char buf[256];
    (void) mkfifo(argv[1], S_IFIFO | S_IRWXU | S_IRWXG | S_IRWXO );
    if(argc==2)
    {
        fd=open(argv[1], O_RDONLY | O_NONBLOCK);
        while(read(fd, buf, sizeof(buf))==-1 && errno==EAGAIN)
            sleep(1);
        while(read(fd, buf, sizeof(buf))>0)
            cout<<buf<<endl;
    }
    else
    {
        fd=open(argv[1], O_WRONLY);
        write(fd, argv[2], strlen(argv[2]));
    }
    close(fd);
}
```

Symbolic Link File APIs

A symbolic link is an indirect pointer to a file, unlike the hard links which pointed directly to the inode of the file.

Symbolic links are developed to get around the limitations of hard links:

- 1) Symbolic links can link files across file systems.
- 2) Symbolic links can link directory files
- 3) Symbolic links always reference the latest version of the files to which they link

- 4) There are no file system limitations on a symbolic link and what it points to and anyone can create a symbolic link to a directory.
- 5) Symbolic links are typically used to move a file or an entire directory hierarchy to some other location on a system.
- 6) A symbolic link is created with the `symlink`.

The prototype is

```
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>

int symlink(const char *org_link, const char *sym_link);
int readlink(const char* sym_link, char* buf, int size);
int lstat(const char * sym_link, struct stat* statv);
```

The `org_link` and `sym_link` arguments to a `symlink` call specify the original file path name and the symbolic link path name to be created.

```
/* Program to illustrate symlink function */
#include<unistd.h>
#include<sys/types.h>
#include<string.h>

int main(int argc, char *argv[])
{
    char *buf [256], tname [256];
    if (argc ==4)
        return symlink(argv[2], argv[3]); /* create a symbolic link */
    else
        return link(argv[1], argv[2]); /* creates a hard link */
}
```

The Environment of a Unix Process

Introduction

A Process is a program under execution in a UNIX or POSIX system.

main Function

A C program starts execution with a function called main. The prototype for the main function is

```
int main(int argc, char *argv[]);
```

where argc is the number of command-line arguments, and argv is an array of pointers to the arguments. When a C program is executed by the kernel by one of the exec functions, a special start-up routine is called before the main function is called. The executable program file specifies this routine as the starting address for the program; this is set up by the link editor when it is invoked by the C compiler. This start-up routine takes values from the kernel, the command-line arguments and the environment and sets things up so that the main function is called.

Process Termination

There are eight ways for a process to terminate. Normal termination occurs in five ways:

- 1) Return from main
- 2) Calling exit
- 3) Calling _exit or _Exit
- 4) Return of the last thread from its start routine
- 5) Calling pthread_exit from the last thread

Abnormal termination occurs in three ways:

- 6) Calling abort
- 7) Receipt of a signal
- 8) Response of the last thread to a cancellation request

Exit Functions

Three functions terminate a program normally: _exit and _Exit, which return to the kernel immediately, and exit, which performs certain cleanup processing and then returns to the kernel.

```
#include <stdlib.h>
void exit(int status);
void _Exit(int status);

#include <unistd.h>
```

All three exit functions expect a single integer argument, called the exit status. Returning an integer value from the main function is equivalent to calling exit with the same value. Thus `exit(0);` is the same as `return(0);` from the main function.

In the following situations the exit status of the process is undefined.

- any of these functions is called without an exit status.
- main does a return without a return value
- main “falls off the end”, i.e if the exit status of the process is undefined.

Example:

```
$ cc hello.c

$ ./a.out hello, world           //print the exit status

$ echo $? 13
```

atexit Function

With ISO C, a process can register up to 32 functions that are automatically called by exit. These are called exit handlers and are registered by calling the `atexit` function.

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

Returns: 0 if OK, nonzero on error. This declaration says that we pass the address of a function as the argument to `atexit`. When this function is called, it is not passed any arguments and is not expected to return a value. The exit function calls these functions in reverse order of their registration. Each function is called as many times as it was registered.

Example of exit handlers

```
#include "apue.h"

static void my_exit1(void);
static void my_exit2(void);

int main(void)
{
    if (atexit(my_exit2) != 0)
        err_sys("can't register my_exit2");

    if (atexit(my_exit1) != 0)
        err_sys("can't register my_exit1");

    if (atexit(my_exit1) != 0)
        err_sys("can't register my_exit1");

    printf("main is done\n");
    return(0);
}

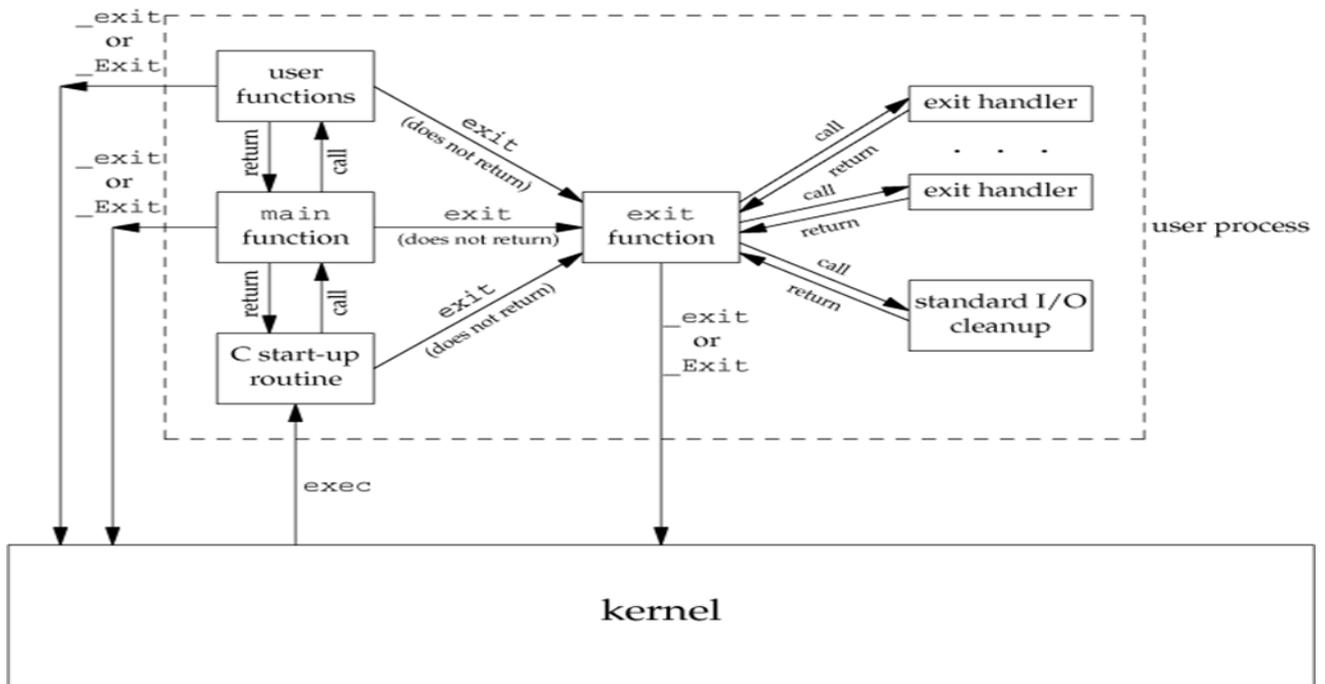
static void
my_exit1(void)
{
    printf("first exit handler\n");
}

static void
my_exit2(void)
{
    printf("second exit handler\n");
}
```

Output:

```
$ ./a.out
main is done
first exit handler
first exit handler
second exit handler
```

The below figure summarizes how a C program is started and the various ways it can terminate.



Command-Line Arguments

When a program is executed, the process that does the exec can pass command-line arguments to the new program.

Example: Echo all command-line arguments to standard output

```

#include "apue.h"

int main(int argc, char *argv[])
{
    int    i;

    for (i = 0; i < argc; i++) /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);
    exit(0);
}
    
```

Output:

```

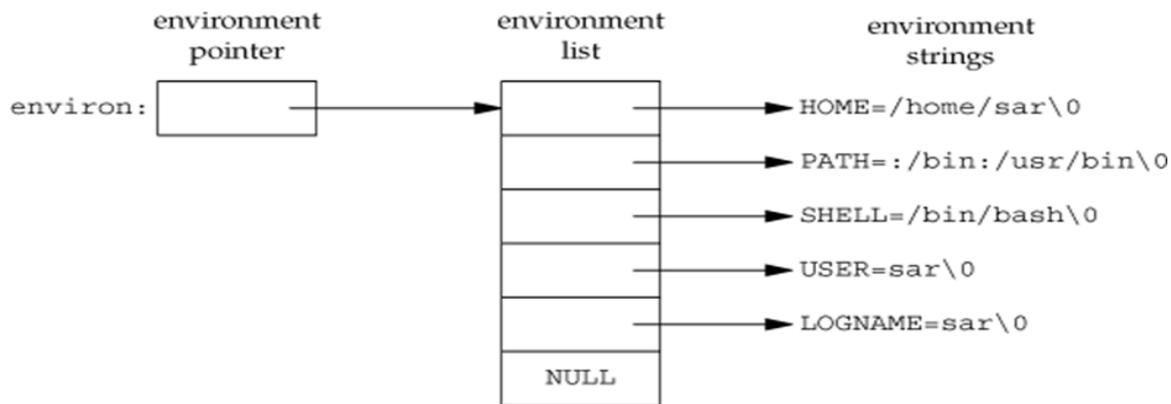
$ ./echoarg arg1 TEST foo
argv[0]: ./echoarg
argv[1]: arg1
argv[2]: TEST
argv[3]: foo
    
```

Environment List

Each program is also passed an environment list. Like the argument list, the environment list is an array of character pointers, with each pointer containing the address of a null-terminated C string. The address of the array of pointers is contained in the global variable environ:

```
extern char **environ;
```

Figure: Environment consisting of five C character strings



Generally, any environmental variable is of the form: name=value.

Memory Layout of a C Program

Historically, a C program has been composed of the following pieces:

- 1) **Text segment**, the machine instructions that the CPU executes. Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.
- 2) **Initialized data segment**, usually called simply the data segment, containing variables that are specifically initialized in the program. For example, the C declaration


```
int maxcount =99;
```

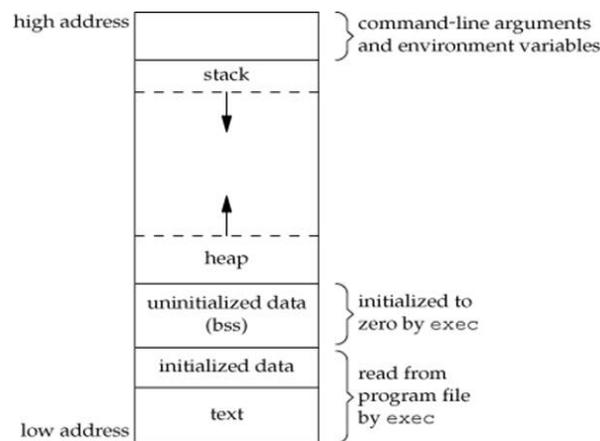
 appearing outside any function causes this variable to be stored in the initialized data segment with its initial value.
- 3) **Uninitialized data segment**, often called the "bss" segment, named after an ancient assembler operator that stood for "block started by symbol." Data in this segment is

initialized by the kernel to arithmetic 0 or null pointers before the program starts executing. The C declaration

```
long sum[1000];
```

appearing outside any function causes this variable to be stored in the uninitialized data segment.

- 4) **Stack**, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller's environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn't interfere with the variables from another instance of the function.
- 5) **Heap**, where dynamic memory allocation usually takes place. Historically, the heap has been located between the uninitialized data and the stack.



Shared Libraries

Nowadays most UNIX systems support shared libraries. Shared libraries remove the common library routines from the executable file, instead maintaining a single copy of the library routine somewhere in memory that all processes reference. This reduces the size of each executable file but may add some runtime overhead, either when the program is first executed or the first time

each shared library function is called. Another advantage of shared libraries is that, library functions can be replaced with new versions without having to re-link, edit every program that uses the library. With cc compiler we can use the option `-g` to indicate that we are using shared library.

Memory Allocation

ISO C specifies three functions for memory allocation:

- 1) `malloc`, which allocates a specified number of bytes of memory. The initial value of the memory is indeterminate.
- 2) `calloc`, which allocates space for a specified number of objects of a specified size. The space is initialized to all 0 bits.
- 3) `realloc`, which increases or decreases the size of a previously allocated area. When the size increases, it may involve moving the previously allocated area somewhere else, to provide the additional room at the end. Also, when the size increases, the initial value of the space between the old contents and the end of the new area is indeterminate.

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t nobj, size_t size);
void *realloc(void *ptr, size_t newsize);
```

All three return: non-null pointer if OK, NULL on error

```
void free(void *ptr);
```

The pointer returned by the three allocation functions is guaranteed to be suitably aligned so that it can be used for any data object. Because the three alloc functions return a generic `void *` pointer, if we `#include<stdlib.h>` (to obtain the function prototypes), we do not explicitly have to cast the pointer returned by these functions when we assign it to a pointer of a different type. The function `free` causes the space pointed to by `ptr` to be deallocated. This freed space is usually put into a pool of available memory and can be allocated in a later call to one of the three alloc functions. The `realloc` function lets us increase or decrease the size of a previously allocated area.

For example, if we allocate room for 512 elements in an array that we fill in at runtime but find that we need room for more than 512 elements, we can call `realloc`. If there is room beyond the end of the existing region for the requested space, then `realloc` doesn't have to move anything; it simply allocates the additional area at the end and returns the same pointer that is large enough, copies the existing 512-element array to the new area, frees the old area, and returns the pointer to the new area.

The allocation routines are usually implemented with the `sbrk(2)` system call. Although `sbrk` can expand or contract the memory of a process, most versions of `malloc` and `free` never decrease their memory size. The space that we free is available for a later allocation, but the freed space is not usually returned to the kernel; that space is kept in the `malloc` pool.

It is important to realize that most implementations allocate a little more space than is requested and use the additional space for record keeping the size of the allocated block, a pointer to the next allocated block, and the like. This means that writing past the end of an allocated area could overwrite this record-keeping information in a later block. These types of errors are often catastrophic, but difficult to find, because the error may not show up until much later. Also, it is possible to overwrite this record keeping by writing before the start of the allocated area.

Because memory allocation errors are difficult to track down, some systems provide versions of these functions that do additional error checking every time one of the three `alloc` functions or `free` is called. These versions of the functions are often specified by including a special library for the link editor. There are also publicly available sources that you can compile with special flags to enable additional runtime checking.

Alternate Memory Allocators

Many replacements for `malloc` and `free` are available.

- 1) `libmalloc`

SVR4-based systems, such as Solaris, include the `libmalloc` library, which provides a set of interfaces matching the ISO C memory allocation functions. The `libmalloc` library includes `mallopt`, a function that allows a process to set certain variables that control the operation of the

storage allocator. A function called mallinfo is also available to provide statistics on the memory allocator.

2) vmalloc

Vo describes a memory allocator that allows processes to allocate memory using different techniques for different regions of memory. In addition to the functions specific to vmalloc, the library also provides emulations of the ISO C memory allocation functions.

3) quick-fit

Historically, the standard malloc algorithm used either a best-fit or a first-fit memory allocation strategy. Quick-fit is faster than either, but tends to use more memory. Free implementations of malloc and free based on quick-fit are readily available from several FTP sites.

4) alloca Function

The function alloca has the same calling sequence as malloc; however, instead of allocating memory from the heap, the memory is allocated from the stack frame of the current function. The advantage is that we don't have to free the space; it goes away automatically when the function returns. The alloca function increases the size of the stack frame. The disadvantage is that some systems can't support alloca, if it's impossible to increase the size of the stack frame after the function has been called.

Environment Variables

The environment strings are usually of the form: *name=value*. The UNIX kernel never looks at these strings; their interpretation is up to the various applications. The shells, for example, use numerous environment variables. Some, such as HOME and USER, are set automatically at login, and others are for us to set. We normally set environment variables in a shell start-up file to control the shell's actions. The functions that we can use to set and fetch values from the variables are setenv, putenv, and getenv functions. The prototype of these functions are

```
#include <stdlib.h>
char *getenv(const char *name);
```

Returns: pointer to value associated with name, NULL if not found.

Note that this function returns a pointer to the value of a name=value string. We should always use `getenv` to fetch a specific value from the environment, instead of accessing `environ` directly. In addition to fetching the value of an environment variable, sometimes we may want to set an environment variable. We may want to change the value of an existing variable or add a new variable to the environment. The prototypes of these functions are

```
#include <stdlib.h>
int putenv(char *str);

int setenv(const char *name, const char *value, int rewrite);
int unsetenv(const char *name);
```

All return: 0 if OK, nonzero on error.

- The `putenv` function takes a string of the form name=value and places it in the environment list. If name already exists, its old definition is first removed.
- The `setenv` function sets name to value. If name already exists in the environment, then
 - a) if `rewrite` is nonzero, the existing definition for name is first removed;
 - b) if `rewrite` is 0, an existing definition for name is not removed, name is not set to the new value, and no error occurs.
- The `unsetenv` function removes any definition of name. It is not an error if such a definition does not exist. Note the difference between `putenv` and `setenv`. Whereas `setenv` must allocate memory to create the

name=value string from its arguments, `putenv` is free to place the string passed to it directly into the environment.

Environment variables defined in the Single UNIX Specification	
Variable	Description
COLUMNS	terminal width
DATMSK	getdate(3) template file pathname
HOME	home directory
LANG	name of locale
LC_ALL	name of locale
LC_COLLATE	name of locale for collation
LC_CTYPE	name of locale for character classification
LC_MESSAGES	name of locale for messages
LC_MONETARY	name of locale for monetary editing
LC_NUMERIC	name of locale for numeric editing
LC_TIME	name of locale for date/time formatting
LINES	terminal height
LOGNAME	login name
MSGVERB	fmtmsg(3) message components to process
NLSPATH	sequence of templates for message catalogs
PATH	list of path prefixes to search for executable file
PWD	absolute pathname of current working directory
SHELL	name of user's preferred shell
TERM	terminal type
TMPDIR	pathname of directory for creating temporary files
TZ	time zone information

NOTE:

- If we're modifying an existing name:
 - a) If the size of the new value is less than or equal to the size of the existing value, we can just copy the new string over the old string.
 - b) If the size of the new value is larger than the old one, however, we must malloc to obtain room for the new string, copy the new string to this area, and then replace the old pointer in the environment list for name with the pointer to this allocated area.
- If we're adding a new name, it's more complicated. First, we have to call malloc to allocate room for the name=value string and copy the string to this area.
 - a) Then, if it's the first time we've added a new name, we have to call malloc to obtain room for a new list of pointers. We copy the old environment list to this new area and

store a pointer to the name=value string at the end of this list of pointers. We also store a null pointer at the end of this list, of course. Finally, we set environ to point to this new list of pointers.

- b) If this isn't the first time, we've added new strings to the environment list, then we know that we've already allocated room for the list on the heap, so we just call realloc to allocate room for one more pointer. The pointer to the new name=value string is stored at the end of the list (on top of the previous null pointer), followed by a null pointer.

setjmp and longjmp Functions

In C, we can't goto a label that's in another function. Instead, we must use the setjmp and longjmp functions to perform this type of branching. As we'll see, these two functions are useful for handling error conditions that occur in a deeply nested function call.

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

Returns: 0 if called directly, nonzero if returning from a call to longjmp

```
void longjmp(jmp_buf env, int val);
```

The setjmp function records or marks a location in a program code so that later when the longjmp function is called from some other function, the execution continues from the location onwards. The env variable (the first argument) records the necessary information needed to continue execution. The env is of the jmp_buf defined in <setjmp.h> file, it contains the task.

Example of setjmp and longjmp

```
#include "apue.h"
#include <setjmp.h>

#define TOK_ADD    5
jmp_buf jmpbuffer;

int main(void)
{
    char    line[MAXLINE];

    if (setjmp(jmpbuffer) != 0)
        printf("error");
    while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
    exit(0);
}

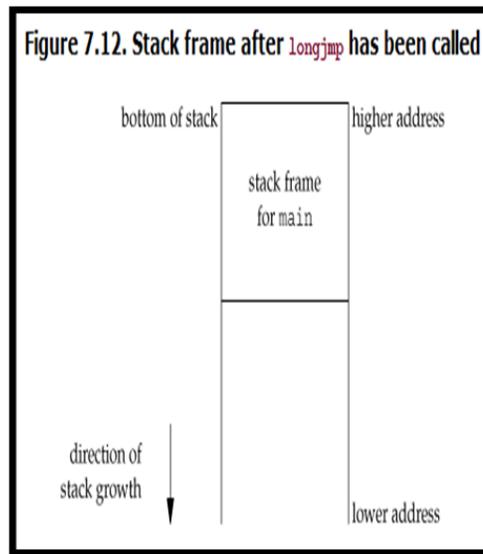
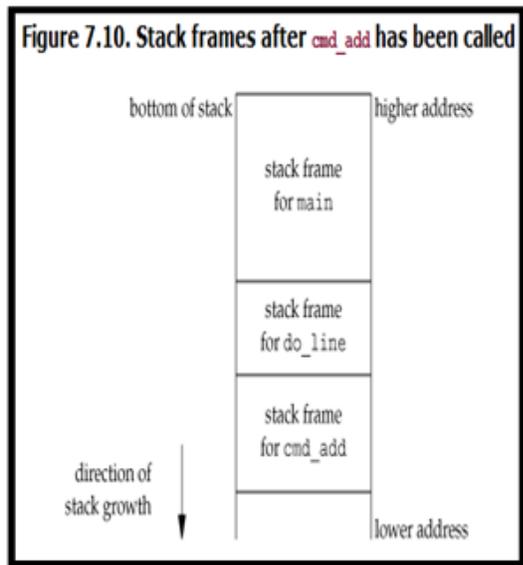
...

void cmd_add(void)
{
    int    token;
    token = get_token();
    if (token < 0)    /* an error has occurred */
        longjmp(jmpbuffer, 1);

    /* rest of processing for this command */
}
```

- The setjmp function always returns '0' on its success when it is called directly in a process (for the first time).
- The longjmp function is called to transfer a program flow to a location that was stored in the env argument.
- The program code marked by the env must be in a function that is among the callers of the current function.
- When the process is jumping to the target function, all the stack space used in the current function and its callers, upto the target function are discarded by the longjmp function.
- The process resumes execution by re-executing the setjmp statement in the target function that is marked by env. The return value of setjmp function is the value(val), as specified in the longjmp function call.
- The 'val' should be nonzero, so that it can be used to indicate where and why the longjmp function was invoked and process can do error handling accordingly.

Note: The values of automatic and register variables are indeterminate when the longjmp is called but static and global variable are unaltered. The variables that we don't want to roll back after longjmp are declared with keyword 'volatile'.



getrlimit and setrlimit Function

Every process has a set of resource limits, some of which can be queried and changed by the `getrlimit` and `setrlimit` functions.

```
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlptr);
int setrlimit(int resource, const struct rlimit *rlptr);
```

Both return: 0 if OK, nonzero on error

Each call to these two functions specifies a single resource and a pointer to the following structure:

```
struct rlimit
{
    rlim_t rlim_cur; /* soft limit: current limit */
    rlim_t rlim_max; /* hard limit: maximum value for rlim_cur */
};
```

Three rules govern the changing of the resource limits.

- A process can change its soft limit to a value less than or equal to its hard limit.
- A process can lower its hard limit to a value greater than or equal to its soft limit. This lowering of the hard limit is irreversible for normal users.
- Only a superuser process can raise a hard limit.

An infinite limit is specified by the constant `RLIM_INFINITY`.

RLIMIT_AS	The maximum size in bytes of a process's total available memory.
RLIMIT_CORE	The maximum size in bytes of a core file. A limit of 0 prevents the creation of a core file.
RLIMIT_CPU	The maximum amount of CPU time in seconds. When the soft limit is exceeded, the SIGXCPU signal is sent to the process.
RLIMIT_DATA	The maximum size in bytes of the data segment: the sum of the initialized data, uninitialized data, and heap.
RLIMIT_FSIZE	The maximum size in bytes of a file that may be created. When the soft limit is exceeded, the process is sent the SIGXFSZ signal.
RLIMIT_LOCKS	The maximum number of file locks a process can hold.
RLIMIT_MEMLOCK	The maximum amount of memory in bytes that a process can lock into memory using <code>mlock(2)</code> .
RLIMIT_NOFILE	The maximum number of open files per process. Changing this limit affects the value returned by the <code>sysconf</code> function for its <code>_SC_OPEN_MAX</code> argument
RLIMIT_NPROC	The maximum number of child processes per real user ID. Changing this limit affects the value returned for <code>_SC_CHILD_MAX</code> by the <code>sysconf</code> function
RLIMIT_RSS	Maximum resident set size (RSS) in bytes. If available physical memory is low, the kernel takes memory from processes that exceed their RSS.
RLIMIT_SBSIZE	The maximum size in bytes of socket buffers that a user can consume at any given time.
RLIMIT_STACK	The maximum size in bytes of the stack.
RLIMIT_VMEM	This is a synonym for <code>RLIMIT_AS</code> .

The resource limits affect the calling process and are inherited by any of its children. This means that the setting of resource limits needs to be built into the shells to affect all our future processes.

Example: Print the current resource limits

```
#include "apue.h"
#if defined(BSD) || defined(MACOS)
#include <sys/time.h>
#define FMT "%10lld "
#else
#define FMT "%10ld "
#endif
#include <sys/resource.h>

#define doit(name) pr_limits(#name, name)
```

```
static void pr_limits(char *, int);

int main(void)
{
#ifdef RLIMIT_AS
    doit(RLIMIT_AS);
#endif
    doit(RLIMIT_CORE);
    doit(RLIMIT_CPU);
    doit(RLIMIT_DATA);
    doit(RLIMIT_FSIZE);
#ifdef RLIMIT_LOCKS
    doit(RLIMIT_LOCKS);
#endif
#ifdef RLIMIT_MEMLOCK
    doit(RLIMIT_MEMLOCK);
#endif
    doit(RLIMIT_NOFILE);
#ifdef RLIMIT_NPROC
    doit(RLIMIT_NPROC);
#endif
#ifdef RLIMIT_RSS
    doit(RLIMIT_RSS);
#endif
#ifdef RLIMIT_SBSIZE
    doit(RLIMIT_SBSIZE);
#endif
    doit(RLIMIT_STACK);
#ifdef RLIMIT_VMEM
    doit(RLIMIT_VMEM);
#endif
    exit(0);
}

static void pr_limits(char *name, int resource)
{
    struct rlimit limit;

    if (getrlimit(resource, &limit) < 0)
        err_sys("getrlimit error for %s", name);
    printf("%-14s ", name);
    if (limit.rlim_cur == RLIM_INFINITY)
        printf("(infinite) ");
    else
        printf(FMT, limit.rlim_cur);
    if (limit.rlim_max == RLIM_INFINITY)
        printf("(infinite)");
    else
        printf(FMT, limit.rlim_max);
    putchar((int)'\n');
}
```

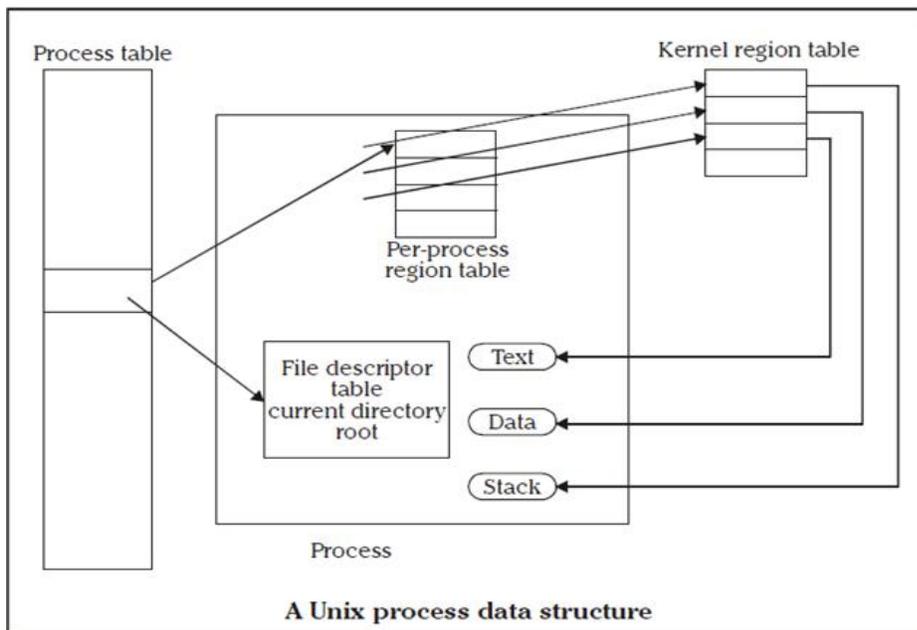
Unix Kernel Support for Process

The data structure and execution of processes are dependent on operating system implementation.

A UNIX process consists minimally of a text segment, a data segment and a stack segment. A segment is an area of memory that is managed by the system as a unit.

- A text segment consists of the program text in machine executable instruction code format.
- The data segment contains static and global variables and their corresponding data.
- A stack segment contains runtime variables and the return addresses of all active functions for a process.

UNIX kernel has a process table that keeps track of all active process present in the system. Some of these processes belongs to the kernel and are called as “system process”. Every entry in the process table contains pointers to the text, data and the stack segments and also to U-area of a process. U-area of a process is an extension of the process table entry and contains other process specific data such as the file descriptor table, current root and working directory inode numbers and set of system-imposed process limits.



All processes in UNIX system except the process that is created by the system boot code, are created by the fork system call. After the fork system call, once the child process is created, both the parent and child processes resumes execution. When a process is created by fork, it contains duplicated copies of the text, data and stack segments of its parent as shown in the Figure below. also it has a file descriptor table, which contains reference to the same opened files as the parent, such that they both share the same file pointer to each opened files.

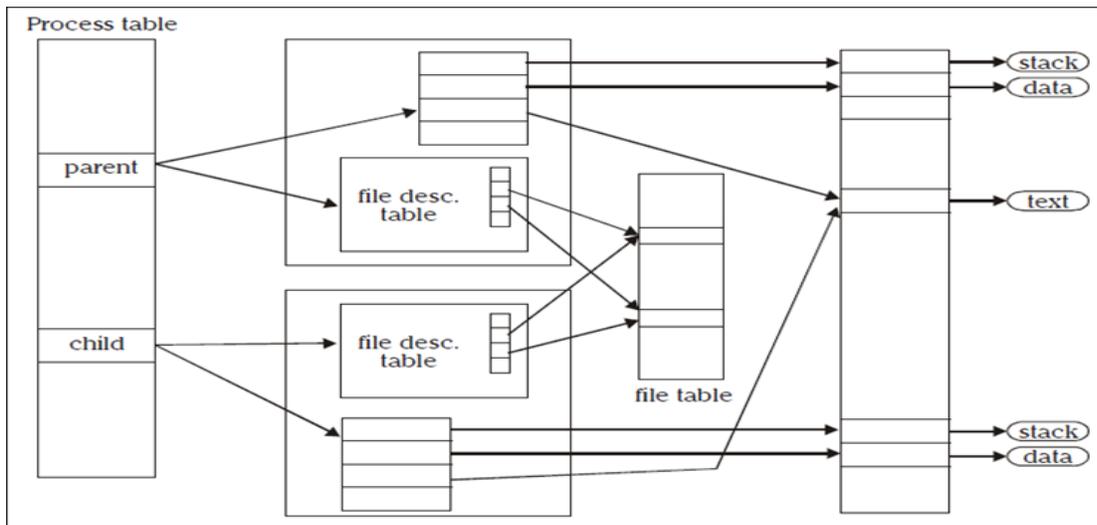


Figure: Parent & child relationship after fork

The process will be assigned with attributes, which are either inherited from its parent or will be set by the kernel.

- A real user identification number (rUID): the user ID of a user who created the parent process.
- A real group identification number (rGID): the group ID of a user who created that parent process.
- An effective user identification number (eUID): this allows the process to access and create files with the same privileges as the program file owner.
- An effective group identification number (eGID): this allows the process to access and create files with the same privileges as the group to which the program file belongs.
- Saved set-UID and saved set-GID: these are the assigned eUID and eGID of the process respectively.
- Process group identification number (PGID) and session identification number (SID): these identify the process group and session of which the process is member.
- Supplementary group identification numbers: this is a set of additional group IDs for a user who created the process.
- Current directory: this is the reference (inode number) to a working directory file.
- Root directory: this is the reference to a root directory.
- Signal handling: the signal handling settings.
- Signal mask: a signal mask that specifies which signals are to be blocked.
- Unmask: a file mode mask that is used in creation of files to specify which accession rights should be taken out.
- Nice value: the process scheduling priority value.

- Controlling terminal: the controlling terminal of the process.

In addition to the above attributes, the following attributes are different between the parent and child processes:

- Process identification number (PID): an integer identification number that is unique per process in an entire operating system.
- Parent process identification number (PPID): the parent process PID.
- Pending signals: the set of signals that are pending delivery to the parent process.
- Alarm clock time: the process alarm clock time is reset to zero in the child process.
- File locks: the set of file locks owned by the parent process is not inherited by the child process.
- fork and exec are commonly used together to spawn a sub-process to execute a different program. The **advantages** of this method are:

A process can create multiple processes to execute multiple programs concurrently.

Because each child process executes in its own virtual address space, the parent process is not affected by the execution status of its child process.

Process Control

Introduction

Process control is concerned about creation of new processes, program execution, and process termination.

Process Identifiers

```
#include <unistd.h>
```

```
pid_t getpid(void);
```

Returns: process ID of calling process

```
pid_t getppid(void);
```

Returns: parent process ID of calling process

```
uid_t getuid(void);
```

Returns: real user ID of calling process

```
uid_t geteuid(void);
```

Returns: effective user ID of calling process

```
gid_t getgid(void);
```

Returns: real group ID of calling process

```
gid_t getegid(void);
```

Returns: effective group ID of calling process

fork Function

An existing process can create a new one by calling the fork function.

```
#include <unistd.h>  
pid_t fork(void);
```

Returns: 0 in child, process ID of child in parent, 1 on error.

The new process created by fork is called the child process. This function is called once but returns twice. The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child. The reason the child's process ID is returned to the parent is that a process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children. The reason fork returns 0 to the child is that a process can have only a single parent, and the child can always call getppid to

obtain the process ID of its parent. (Process ID 0 is reserved for use by the kernel, so it's not possible for 0 to be the process ID of a child.) Both the child and the parent continue executing with the instruction that follows the call to fork. The child is a copy of the parent. For example, the child gets a copy of the parent's data space, heap, and stack. Note that this is a copy for the child; the parent and the child do not share these portions of memory. The parent and the child share the text segment.

Example programs:

Program 1

```
/* Program to demonstrate fork function Program name – fork1.c */
#include<sys/types.h>
#include<unistd.h>
int main( )
{
    fork( );
    printf("\n hello USP");
}
```

Output :

```
$ cc fork1.c
$ ./a.out
hello USP
hello USP
```

Note: The statement hello USP is executed twice as both the child and parent have executed that instruction.

Program 2

```

/* Program name – fork2.c */
#include<sys/types.h>
#include<unistd.h>
int main( )
{
    printf("\n 6 sem ");
    fork( );
    printf("\n hello USP");
}

```

Output :

```

$ cc fork1.c
$ ./a.out
6 sem
hello USP
hello USP

```

Note: The statement 6 sem is executed only once by the parent because it is called before fork and statement hello USP is executed twice by child and parent.

File Sharing

Consider a process that has three different files opened for standard input, standard output, and standard error. On return from fork, we have the arrangement shown in Figure 8.2.

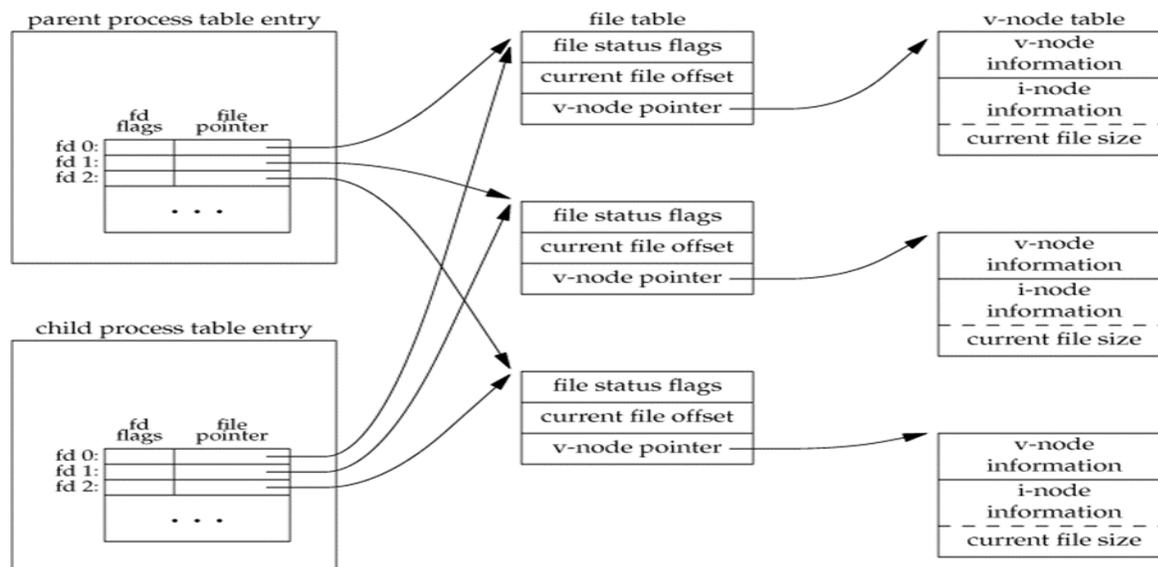


Figure 8.2 Sharing of open files between parent and child after fork

It is important that the parent and the child share the same file offset. Consider a process that forks a child, then waits for the child to complete. Assume that both processes write to standard output

as part of their normal processing. If the parent has its standard output redirected (by a shell, perhaps) it is essential that the parent's file offset be updated by the child when the child writes to standard output. In this case, the child can write to standard output while the parent is waiting for it; on completion of the child, the parent can continue writing to standard output, knowing that its output will be appended to whatever the child wrote. If the parent and the child did not share the same file offset, this type of interaction would be more difficult to accomplish and would require explicit actions by the parent.

There are two normal cases for handling the descriptors after a fork.

- The parent waits for the child to complete. In this case, the parent does not need to do anything with its descriptors. When the child terminates, any of the shared descriptors that the child read from or wrote to will have their file offsets updated accordingly.
- Both the parent and the child go their own ways. Here, after the fork, the parent closes the descriptors that it doesn't need, and the child does the same thing. This way, neither interferes with the other's open descriptors. This scenario is often the case with network servers.

There are numerous other properties of the parent that are inherited by the child:

- Real user ID, real group ID, effective user ID, effective group ID
- Supplementary group IDs
- Process group ID
- Session ID
- Controlling terminal
- The set-user-ID and set-group-ID flags
- Current working directory
- Root directory
- File mode creation mask
- Signal mask and dispositions
- The close-on-exec flag for any open file descriptors
- Environment
- Attached shared memory segments
- Memory mappings
- Resource limits

The differences between the parent and child are

- a) The return value from fork
- b) The process IDs are different
- c) The two processes have different parent process IDs: the parent process ID of the child is the parent; the parent process ID of the parent doesn't change
- d) The child's `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime` values are set to 0 File locks set by the parent are not inherited by the child
- e) Pending alarms are cleared for the child
- f) The set of pending signals for the child is set to the empty set

The two main reasons for fork to fail are

- if too many processes are already in the system, which usually means that something else is wrong, or
- if the total number of processes for this real user ID exceeds the system's limit.

There are two uses for fork:

- a) When a process wants to duplicate itself so that the parent and child can each execute different sections of code at the same time. This is common for network servers, the parent waits for a service request from a client. When the request arrives, the parent calls `fork` and lets the child handle the request. The parent goes back to waiting for the next service request to arrive.
- b) When a process wants to execute a different program. This is common for shells. In this case, the child does an `exec` right after it returns from the `fork`.

vfork Function

The function `vfork` has the same calling sequence and same return values as `fork`. The `vfork` function is intended to create a new process when the purpose of the new process is to exec a new program. The `vfork` function creates the new process, just like `fork`, without copying the address space of the parent into the child, as the child won't reference that address space; the child simply

calls `exec` (or `exit`) right after the `vfork`. Instead, while the child is running and until it calls either `exec` or `exit`, the child runs in the address space of the parent. This optimization provides an efficiency gain on some paged virtual-memory implementations of the UNIX System. Another difference between the two functions is that `vfork` guarantees that the child runs first, until the child calls `exec` or `exit`. When the child calls either of these functions, the parent resumes.

Example of `vfork` function

```
#include "apue.h"
int    glob = 6;      /* external variable in initialized data */

int main(void)
{
    int    var;        /* automatic variable on the stack */
    pid_t  pid;

    var = 88;
    printf("before vfork\n"); /* we don't flush stdio */
    if ((pid = vfork()) < 0) {
        err_sys("vfork error");
    } else if (pid == 0) { /* child */
        glob++;           /* modify parent's variables */
        var++;
        _exit(0);        /* child terminates */
    }
    /*
     * Parent continues here.
     */
    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

Output:

```
$ ./a.out
before vfork
pid = 29039, glob = 7, var = 89
```

exit Functions

A process can terminate normally in five ways:

1. Executing a return from the main function.
2. Calling the `exit` function.
3. Calling the `_exit` or `_Exit` function.

In most UNIX system implementations, `exit(3)` is a function in the standard C library, whereas `_exit(2)` is a system call.

4. Executing a return from the start routine of the last thread in the process. When the last thread returns from its start routine, the process exits with a termination status of 0.
5. Calling the `pthread_exit` function from the last thread in the process.

The three forms of abnormal termination are as follows:

- 1) Calling abort. This is a special case of the next item, as it generates the SIGABRT signal.
- 2) When the process receives certain signals. Examples of signals generated by the kernel include the process referencing a memory location not within its address space or trying to divide by 0.
- 3) The last thread responds to a cancellation request. By default, cancellation occurs in a deferred manner: one thread requests that another be canceled, and sometime later, the target thread terminates.

wait and waitpid Functions

When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the SIGCHLD signal to the parent. Because the termination of a child is an asynchronous event - it can happen at any time while the parent is running - this signal is the asynchronous notification from the kernel to the parent. The parent can choose to ignore this signal, or it can provide a function that is called when the signal occurs: a signal handler.

A process that calls wait or waitpid can:

- 1) Block, if all of its children are still running
- 2) Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched
- 3) Return immediately with an error, if it doesn't have any child processes.

```
#include <sys/wait.h>

pid_t wait(int *statloc);

pid_t waitpid(pid_t pid, int *statloc, int options);
```

Both return: process ID if OK, 0 (see later), or 1 on error.

The differences between these two functions are as follows.

- 1) The wait function can block the caller until a child process terminates, whereas waitpid has an option that prevents it from blocking.
- 2) The waitpid function doesn't wait for the child that terminates first; it has a number of options that control which process it waits for.

If a child has already terminated and is a zombie, wait returns immediately with that child's status. Otherwise, it blocks the caller until a child terminates. If the caller blocks and has multiple children, wait returns when one terminates.

For both functions, the argument `statloc` is a pointer to an integer. If this argument is not a null pointer, the termination status of the terminated process is stored in the location pointed to by the argument.

Print a description of the exit status

```
#include "apue.h"
#include <sys/wait.h>

Void pr_exit(int status)
{
    if (WIFEXITED(status))
        printf("normal termination, exit status = %d\n",
            WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("abnormal termination, signal number = %d%s\n",
            WTERMSIG(status),
#ifdef WCOREDUMP
            WCOREDUMP(status) ? " (core file generated)" : "");
#else
            "");
#endif
    else if (WIFSTOPPED(status))
        printf("child stopped, signal number = %d\n",
            WSTOPSIG(status));
}
```

Program to Demonstrate various exit statuses

```
#include "apue.h"
#include <sys/wait.h>

Int main(void)
{
    pid_t  pid;
    int    status;

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)          /* child */
        exit(7);

    if (wait(&status) != pid) /* wait for child */
        err_sys("wait error");
    pr_exit(status);          /* and print its status */
}
```

```

if ((pid = fork()) < 0)
    err_sys("fork error");
else if (pid == 0)
    abort();
    /* child */
    /* generates SIGABRT */

if (wait(&status) != pid)
    err_sys("wait error");
pr_exit(status);
    /* wait for child */
    /* and print its status */

if ((pid = fork()) < 0)
    err_sys("fork error");
else if (pid == 0)
    status /= 0;
    /* child */
    /* divide by 0 generates SIGFPE */

if (wait(&status) != pid)
    err_sys("wait error");
pr_exit(status);
    /* wait for child */
    /* and print its status */

exit(0);
}

```

The interpretation of the pid argument for waitpid depends on its value:

- pid == 1 Waits for any child process. In this respect, waitpid is equivalent to wait.
- pid > 0 Waits for the child whose process ID equals pid.
- pid == 0 Waits for any child whose process group ID equals that of the calling process.
- pid < 1 Waits for any child whose process group ID equals the absolute value of pid.

Macro	Description
WIFEXITED(status)	True if status was returned for a child that terminated normally. In this case, we can execute WEXITSTATUS(status) to fetch the low-order 8 bits of the argument that the child passed to exit, _exit, or _Exit.
WIFSIGNALED(status)	True if status was returned for a child that terminated abnormally, by receipt of a signal that it didn't catch. In this case, we can execute WTERMSIG(status) to fetch the signal number that caused the termination. Additionally, some implementations (but not the Single UNIX Specification) define the macro WCOREDUMP(status) that returns true if a core file of the terminated process was generated.
WIFSTOPPED(status)	True if status was returned for a child that is currently stopped. In this case, we can execute WSTOPSIG(status) to fetch the signal number that caused the child to stop.
WIFCONTINUED(status)	True if status was returned for a child that has been continued after a job control stop.

The options constants for <code>waitpid</code>	
Constant	Description
<code>WCONTINUED</code>	If the implementation supports job control, the status of any child specified by <code>pid</code> that has been continued after being stopped, but whose status has not yet been reported, is returned.
<code>WNOHANG</code>	The <code>waitpid</code> function will not block if a child specified by <code>pid</code> is not immediately available. In this case, the return value is 0.
<code>WUNTRACED</code>	If the implementation supports job control, the status of any child specified by <code>pid</code> that has stopped, and whose status has not been reported since it has stopped, is returned. The <code>WIFSTOPPED</code> macro determines whether the return value corresponds to a stopped child process.

The `waitpid` function provides three features that aren't provided by the `wait` function.

- 1) The `waitpid` function lets us wait for one particular process, whereas the `wait` function returns the status of any terminated child. We'll return to this feature when we discuss the `waitpid` function.
- 2) The `waitpid` function provides a nonblocking version of `wait`. There are times when we want to fetch a child's status, but we don't want to block.
- 3) The `waitpid` function provides support for job control with the `WUNTRACED` and `WCONTINUED` options.

Program to Avoid zombie processes by calling `fork` twice

```
#include "apue.h"
#include <sys/wait.h>

Int main(void)
{
    pid_t  pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) { /* first child */
        if ((pid = fork()) < 0)
            err_sys("fork error");
        else if (pid > 0)
            exit(0); /* parent from second fork == first child */
        /*
         * We're the second child; our parent becomes init as soon
         * as our real parent calls exit() in the statement above.
         * Here's where we'd continue executing, knowing that when
         * we're done, init will reap our status.
         */
        sleep(2);
        printf("second child, parent pid = %d\n", getppid());
        exit(0);
    }

    if (waitpid(pid, NULL, 0) != pid) /* wait for first child */
        err_sys("waitpid error");

    /*
     * We're the parent (the original process); we continue executing,
     * knowing that we're not the parent of the second child.
     */
    exit(0);
}
```

Output:

```
$ ./a.out
$ second child, parent pid = 1
```

waitid Function

The waitid function is similar to waitpid, but provides extra flexibility.

```
#include <sys/wait.h>

Int waited(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

Returns: 0 if OK, -1 on error

The idtype constants for waited are as follows:

Constant	Description
P_PID	Wait for a particular process: id contains the process ID of the child to wait for.
P_PGID	Wait for any child process in a particular process group: id contains the process group ID of the children to wait for.
P_ALL	Wait for any child process: id is ignored.

The options argument is a bitwise OR of the flags as shown below: these flags indicate which state changes the caller is interested in.

Constant	Description
WCONTINUED	Wait for a process that has previously stopped and has been continued, and whose status has not yet been reported.
WEXITED	Wait for processes that have exited.
WNOHANG	Return immediately instead of blocking if there is no child exit status available.
WNOWAIT	Don't destroy the child exit status. The child's exit status can be retrieved by a subsequent call to <code>wait</code> , <code>waitid</code> , or <code>waitpid</code> .
WSTOPPED	Wait for a process that has stopped and whose status has not yet been reported.

wait3 and wait4 Functions

The only feature provided by these two functions that isn't provided by the `wait`, `waitid`, and `waitpid` functions is an additional argument that allows the kernel to return a summary of the resources used by the terminated process and all its child processes.

The prototypes of these functions are:

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>

pid_t wait3(int *statloc, int options, struct rusage *rusage);
pid_t wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);
```

Both return: process ID if OK, -1 on error

The resource information includes such statistics as the amount of user CPU time, the amount of system CPU time, number of page faults, number of signals received etc. the resource information is available only for terminated child process not for the process that were stopped due to job control.

Race Conditions

A race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run.

Example: The program below outputs two strings: one from the child and one from the parent. The program contains a race condition because the output depends on the order in which the processes are run by the kernel and for how long each process runs.

```

#include "apue.h"

static void charatime(char *);

int main(void)
{
    pid_t  pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {
        charatime("output from child\n");
    } else {
        charatime("output from parent\n");
    }
    exit(0);
}

static void
charatime(char *str)
{
    char   *ptr;
    int    c;

    setbuf(stdout, NULL);          /* set unbuffered */
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}

```

Output:

```

$ ./a.out
output from child
output from parent
$ ./a.out
output from child
output from parent
$ ./a.out
output from child
output from parent

```

program modification to avoid race condition

```

#include "apue.h"

static void charatime(char *);

int main(void)
{
    pid_t  pid;

+   TELL_WAIT();
+
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {
+       WAIT_PARENT();          /* parent goes first */
        charatime("output from child\n");
    } else {
        charatime("output from parent\n");
+       TELL_CHILD(pid);
    }
    exit(0);
}
static void

```

```
charatotime(char *str)
{
    char    *ptr;
    int     c;

    setbuf(stdout, NULL);          /* set unbuffered */
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}
```

When we run this program, the output is as we expect; there is no intermixing of output from the two processes.

exec Functions

When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function. The process ID does not change across an exec, because a new process is not created; exec merely replaces the current process - its text, data, heap, and stack segments - with a brand new program from disk.

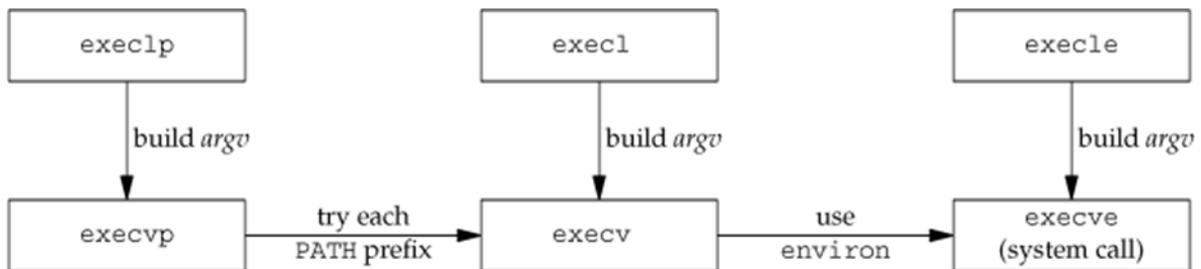
There are 6 exec functions:

```
#include <unistd.h>
int  execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );
int  execv(const char *pathname, char *const argv []);
int  execlp(const char *pathname, const char *arg0, ... /*(char *)0, char
*const envp */ );
int  execve(const char *pathname, char *const argv[], char *const envp[]);
int  execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );
int  execvp(const char *filename, char *const argv []);
```

All six return: -1 on error, no return on success.

- 1) The first difference in these functions is that the first four take a pathname argument, whereas the last two take a filename argument. When a filename argument is specified
 - If filename contains a slash, it is taken as a pathname.
 - Otherwise, the executable file is searched for in the directories specified by the PATH environment variable.
- 2) The next difference concerns the passing of the argument list (l stands for list and v stands for vector). The functions execl, execlp, and execlp require each of the command-line arguments to the new program to be specified as separate arguments. For the other three functions (execv, execvp, and execve), we have to build an array of pointers to the arguments, and the address of this array is the argument to these three functions.

- 3) The final difference is the passing of the environment list to the new program. The two functions whose names end in an e (execle and execve) allow us to pass a pointer to an array of pointers to the environment strings. The other four functions, however, use the environ variable in the calling process to copy the existing environment for the new program.



Example of exec functions

```

#include "apue.h"
#include <sys/wait.h>

char    *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int main(void)
{
    pid_t  pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) { /* specify pathname, specify environment */
        if (execle("/home/sar/bin/echoall", "echoall", "myarg1",
                  "MY ARG2", (char *)0, env_init) < 0)
            err_sys("execle error");
    }

    if (waitpid(pid, NULL, 0) < 0)
        err_sys("wait error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) { /* specify filename, inherit environment */
        if (execlp("echoall", "echoall", "only 1 arg", (char *)0) < 0)
            err_sys("execlp error");
    }

    exit(0);
}
  
```

Output:

```
$ ./a.out
argv[0]: echoall
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
```

```
PATH=/tmp
$ argv[0]: echoall argv[1]: only 1 arg USER=sar LOGNAME=sar
SHELL=/bin/bash

HOME=/home/sar
```

Note that the shell prompt appeared before the printing of argv[0] from the second exec. This is because the parent did not wait for this child process to finish.

Module – 4

Changing User IDs and Group IDs

When our programs need additional privileges or need to gain access to resources that they currently aren't allowed to access, they need to change their user or group ID to an ID that has the appropriate privilege or access. Similarly, when our programs need to lower their privileges or prevent access to certain resources, they do so by changing either their user ID or group ID to an ID without the privilege or ability access to the resource.

```
#include <unistd.h>

int setuid(uid_t uid);
int setgid(gid_t gid);
```

Both return: 0 if OK, 1 on error

There are rules for who can change the IDs. Let's consider only the user ID for now. (Everything we describe for the user ID also applies to the group ID.)

- If the process has superuser privileges, the setuid function sets the real user ID, effective user ID, and saved set-user-ID to uid.
- If the process does not have superuser privileges, but uid equals either the real user ID or the saved set-user-ID, setuid sets only the effective user ID to uid. The real user ID and the saved set-user-ID are not changed.
- If neither of these two conditions is true, errno is set to EPERM, and 1 is returned.

We can make a few statements about the three user IDs that the kernel maintains.

1. Only a superuser process can change the real user ID. Normally, the real user ID is set by the login(1) program when we log in and never changes. Because login is a superuser process, it sets all three user IDs when it calls setuid.
2. The effective user ID is set by the exec functions only if the set-user-ID bit is set for the program file. If the set-user-ID bit is not set, the exec functions leave the effective user ID

3. as its current value. We can call setuid at any time to set the effective user ID to either the real user ID or the saved set-user-ID. Naturally, we can't set the effective user ID to any random value.
4. The saved set-user-ID is copied from the effective user ID by exec. If the file's set-user-ID bit is set, this copy is saved after exec stores the effective user ID from the file's user ID.

The above figure summarises the various ways these three user IDs can be changed

ID	exec		setuid(uid)	
	set-user-ID bit off	set-user-ID bit on	superuser	unprivileged
				user
real user ID	unchanged	unchanged	set to uid	unchanged
effective user ID	unchanged	set from user ID of program file	set to uid	set to uid
saved set-user ID	copied from effective user ID	copied from effective user ID	set to uid	unchanged

The above figure summarises the various ways these three user IDs can be changed

setreuid and setregid Functions

Swapping of the real user ID and the effective user ID with the setreuid function.

```
#include <unistd.h>
int setreuid(uid_t ruid, uid_t euid);
int setregid(gid_t rgid, gid_t egid);
```

Both return : 0 if OK, -1 on error

We can supply a value of 1 for any of the arguments to indicate that the corresponding ID should remain unchanged. The rule is simple: an unprivileged user can always swap between the real user ID and the effective user ID. This allows a set-user-ID program to swap to the user's normal permissions and swap back again later for set-user- ID operations.

seteuid and setegid functions

POSIX.1 includes the two functions seteuid and setegid. These functions are similar to setuid and setgid, but only the effective user ID or effective group ID is changed.

```
#include <unistd.h>
int seteuid(uid_t uid);
int setegid(gid_t gid);
```

Both return : 0 if OK, 1 on error

An unprivileged user can set its effective user ID to either its real user ID or its saved set-user-ID. For a privileged user, only the effective user ID is set to uid. (This differs from the setuid function, which changes all three user IDs.)

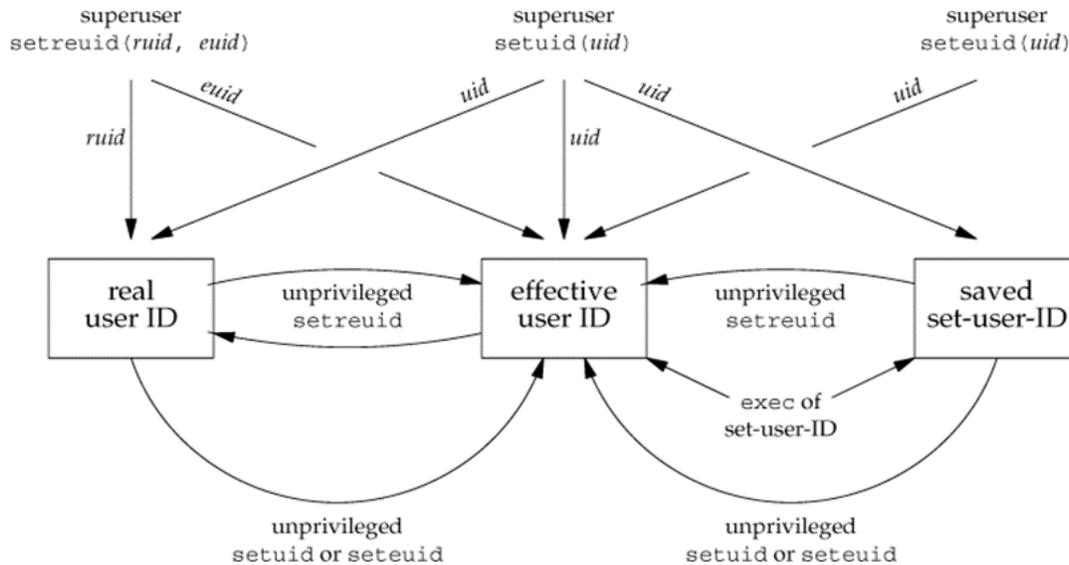


Figure: Summary of all the functions that set the various user IDs

Interpreter Files

These files are text files that begin with a line of the form

```
#! pathname [ optional-argument ]
```

The space between the exclamation point and the pathname is optional. The most common of these interpreter files begin with the line

```
#!/bin/sh
```

The pathname is normally an absolute pathname, since no special operations are performed on it (i.e., PATH is not used). The recognition of these files is done within the kernel as part of processing the exec system call. The actual file that gets executed by the kernel is not the interpreter file, but the file specified by the pathname on the first line of the interpreter file. Be sure to differentiate between the interpreter file a text file that begins with #! and the interpreter, which is specified by the pathname on the first line of the interpreter file.

Be aware that systems place a size limit on the first line of an interpreter file. This limit includes the #!, the pathname, the optional argument, the terminating newline, and any spaces.

A program that execs an interpreter file

```
#include "apue.h"
#include <sys/wait.h>

int main(void)
{
    pid_t pid;
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) { /* child */
        if (execl("/home/sar/bin/testinterp",
                "testinterp", "myarg1", "MY ARG2", (char *)0) < 0)
            err_sys("execl error");
    }
    if (waitpid(pid, NULL, 0) < 0) /* parent */
        err_sys("waitpid error");
    exit(0);
}
```

Output:

```
$ cat /home/sar/bin/testinterp
#!/home/sar/bin/echoarg foo
$ ./a.out
argv[0]: /home/sar/bin/echoarg
argv[1]: foo
argv[2]: /home/sar/bin/testinterp
argv[3]: myarg1
argv[4]: MY ARG2
```

System Function

```
#include <stdlib.h>

int system(const char *cmdstring);
```

If cmdstring is a null pointer, system returns nonzero only if a command processor is available. This feature

determines whether the system function is supported on a given operating system. Under the UNIX System,

system is always available.

Because system is implemented by calling fork, exec, and waitpid, there are three types of return values.

- If either the fork fails or waitpid returns an error other than EINTR, system returns 1 with errno set to indicate the error.
- If the exec fails, implying that the shell can't be executed, the return value is as if the shell had executed exit(127).
- Otherwise, all three functions fork, exec, and waitpid succeed, and the return value from system is the termination status of the shell, in the format specified for waitpid.

Program: The system function, without signal handling

```
#include <sys/wait.h>
#include <errno.h>
#include <unistd.h>

Int system(const char *cmdstring) /* version without signal handling */
{
    pid_t pid;
    int status;

    if (cmdstring == NULL)
        return(1); /* always a command processor with UNIX */

    if ((pid = fork()) < 0) {
        status = -1; /* probably out of processes */
    } else if (pid == 0) { /* child */
        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127); /* execl error */
    } else { /* parent */
        while (waitpid(pid, &status, 0) < 0) {
            if (errno != EINTR) {
                status = -1; /* error other than EINTR from waitpid() */
                break;
            }
        }
    }

    return(status);
}
```

Program: Calling the system function

```
#include "apue.h"
#include <sys/wait.h>

Int main(void)
{
    int status;

    if ((status = system("date")) < 0)
        err_sys("system() error");
    pr_exit(status);

    if ((status = system("nosuchcommand")) < 0)
        err_sys("system() error");
    pr_exit(status);

    if ((status = system("who; exit 44")) < 0)
        err_sys("system() error");
    pr_exit(status);

    exit(0);
}
```

Program: Execute the command-line argument using system

```
#include "apue.h"

int main(int argc, char *argv[])
{
    int    status;

    if (argc < 2)
        err_quit("command-line argument required");

    if ((status = system(argv[1])) < 0)
        err_sys("system() error");
    pr_exit(status);

    exit(0);
}
```

Program: Print real and effective user IDs

```
#include "apue.h"

int
main(void)
{
    printf("real uid = %d, effective uid = %d\n", getuid(), geteuid());

    exit(0);
}
```

Process Accounting

Most UNIX systems provide an option to do process accounting. When enabled, the kernel writes an accounting record each time a process terminates. These accounting records are typically a small amount of binary data with the name of the command, the amount of CPU time used, the user ID and group ID, the starting time, and so on. A superuser executes `accton` with a pathname argument to enable accounting. The accounting records are written to the specified file, which is usually `/var/account/acct`. Accounting is turned off by executing `accton` without any arguments. The data required for the accounting record, such as CPU times and number of characters transferred, is kept by the kernel in the process table and initialized whenever a new process is created, as in the child after a fork. Each accounting record is written when the process terminates. This means that the order of the records in the accounting file corresponds to the termination order of the processes, not the order in which they were started. The accounting records correspond to processes, not programs. A new record is initialized by the kernel for the child after a fork, not

when a new program is executed. The structure of the accounting records is defined in the header <sys/acct.h> and looks something like

```
typedef u_short comp_t; /* 3-bit base 8 exponent; 13-bit fraction */

struct acct
{
    char ac_flag; /* flag */
    char ac_stat; /* termination status (signal & core flag only) */
                /* (Solaris only) */
    uid_t ac_uid; /* real user ID */
    gid_t ac_gid; /* real group ID */
    dev_t ac_tty; /* controlling terminal */
    time_t ac_btime; /* starting calendar time */
    comp_t ac_utime; /* user CPU time (clock ticks) */
    comp_t ac_stime; /* system CPU time (clock ticks) */
    comp_t ac_etime; /* elapsed time (clock ticks) */
    comp_t ac_mem; /* average memory usage */
    comp_t ac_io; /* bytes transferred (by read and write) */
                /* "blocks" on BSD systems */
    comp_t ac_rw; /* blocks read or written */
                /* (not present on BSD systems) */
    char ac_comm[8]; /* command name: [8] for Solaris, */
                    /* [10] for Mac OS X, [16] for FreeBSD, and */
                    /* [17] for Linux */
};
```

Values for ac_flag from accounting record

ac_flag	Description
AFORK	process is the result of fork, but never called exec
ASU	process used superuser privileges
ACOMPAT	process used compatibility mode
ACORE	process dumped core
AXSIG	process was killed by a signal
AEXPND	expanded accounting entry

Program to generate accounting data

```
#include "apue.h"

Int main(void)
{
```

```

pid_t  pid;

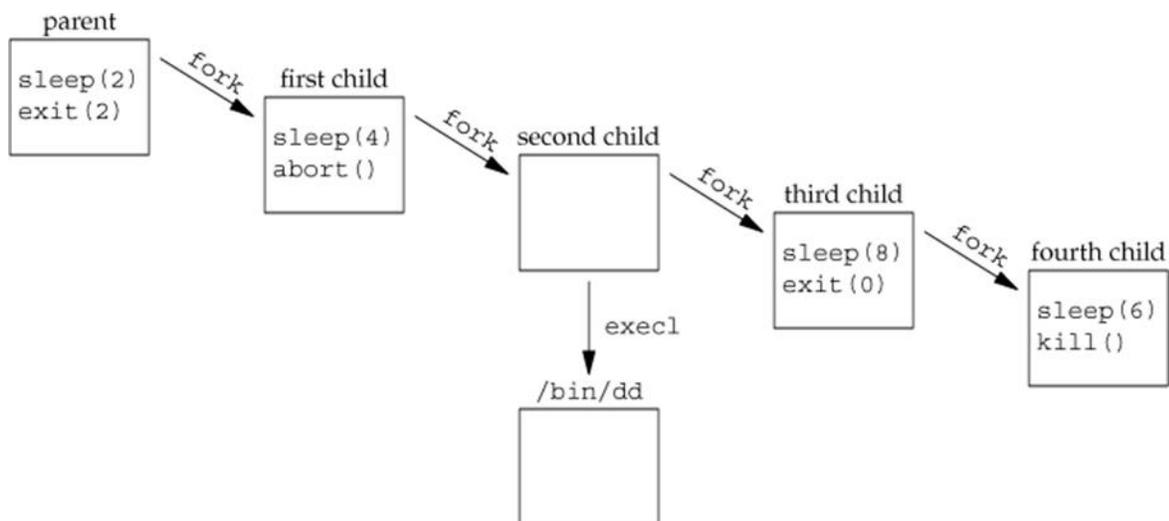
if ((pid = fork()) < 0)
    err_sys("fork error");
else if (pid != 0) {      /* parent */
    sleep(2);
    exit(2);              /* terminate with exit status 2 */
}

                          /* first child */
if ((pid = fork()) < 0)
    err_sys("fork error");
else if (pid != 0) {
    sleep(4);
    abort();              /* terminate with core dump */
}

                          /* second child */
if ((pid = fork()) < 0)
    err_sys("fork error");
else if (pid != 0) {
    execl("/bin/dd", "dd", "if=/etc/termcap", "of=/dev/null", NULL);
    exit(7);              /* shouldn't get here */
}

                          /* third child */
if ((pid = fork()) < 0)
    err_sys("fork error");
else if (pid != 0) {
    sleep(8);
    exit(0);              /* normal exit */
}

                          /* fourth child */
sleep(6);
kill(getpid(), SIGKILL); /* terminate w/signal, no core dump */
exit(6);                 /* shouldn't get here */
}
    
```



Process structure for accounting example

User Identification

Any process can find out its real and effective user ID and group ID. Sometimes, however, we want to find out the login name of the user who's running the program. We could call `getpwuid(getuid())`, but what if a single user has multiple login names, each with the same user ID? (A person might have multiple entries in the password file with the same user ID to have a different login shell for each entry.) The system normally keeps track of the name we log in and the `getlogin` function provides a way to fetch that login name.

```
#include <unistd.h>

char *getlogin(void);
```

Returns : pointer to string giving login name if OK, NULL on error

This function can fail if the process is not attached to a terminal that a user logged in to.

Process Times

We describe three times that we can measure: wall clock time, user CPU time, and system CPU time. Any process can call the `times` function to obtain these values for itself and any terminated children.

```
#include <sys/times.h>

clock_t times(struct tms *buf);
```

Returns: elapsed wall clock time in clock ticks if OK, 1 on error

This function fills in the `tms` structure pointed to by `buf`:

```
struct tms {
    clock_t tms_ftime; /* user CPU time */
    clock_t tms_stime; /* system CPU time */
    clock_t tms_cutime; /* user CPU time, terminated children */
    clock_t tms_cstime; /* system CPU time, terminated children */
};
```

Note that the structure does not contain any measurement for the wall clock time. Instead, the function returns the wall clock time as the value of the function, each time it's called. This value is measured from some arbitrary point in the past, so we can't use its absolute value; instead, we use its relative value.

I/O Redirection

The Shell input/output redirections. Most Unix system commands take input from your terminal and send the resulting output back to your terminal. A command normally reads its input from the standard input, which happens to be your terminal by default. Similarly, a command normally writes its output to standard output, which is again your terminal by default.

Output Redirection

The output from a command normally intended for standard output can be easily diverted to a file instead. This capability is known as output redirection. If the notation `> file` is appended to any command that normally writes its output to standard output, the output of that command will be written to file instead of your terminal. Check the following who command which redirects the complete output of the command in the users file.

```
$ who > users
```

Notice that no output appears at the terminal. This is because the output has been redirected from the default standard output device (the terminal) into the specified file. You can check the users file for the complete content –

```
$ cat users
oko      tty01   Sep 12 07:30
ai       tty15   Sep 12 13:32
ruth     tty21   Sep 12 10:10
pat      tty24   Sep 12 13:07
steve    tty25   Sep 12 13:03
```

If a command has its output redirected to a file and the file already contains some data, that data will be lost. Consider the following example –

```
$ echo line 1 > users
$ cat users
line 1
$
```

You can use `>>` operator to append the output in an existing file as follows –

```
$ echo line 2 >> users
$ cat users
line 1
line 2
$
```

Input Redirection

Just as the output of a command can be redirected to a file, so can the input of a command be redirected from a file. As the greater-than character > is used for output redirection, the less-than character < is used to redirect the input of a command.

The commands that normally take their input from the standard input can have their input redirected from a file in this manner. For example, to count the number of lines in the file users generated above, you can execute the command as follows –

```
$ wc -l users
2 users
$
```

Upon execution, you will receive the following output. You can count the number of lines in the file by redirecting the standard input of the wc command from the file users –

```
$ wc -l < users
2
$
```

Note that there is a difference in the output produced by the two forms of the wc command. In the first case, the name of the file users is listed with the line count; in the second case, it is not. In the first case, wc knows that it is reading its input from the file users. In the second case, it only knows that it is reading its input from standard input so it does not display file name.

Overview of IPC Methods

Pipes

Pipes are the oldest form of UNIX System IPC. Pipes have two limitations.

1. Historically, they have been half duplex (i.e., data flows in only one direction).
2. Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child.

A pipe is created by calling the pipe function.

```
#include <unistd.h>
int pipe(int filedes[2]);
```

Returns: 0 if OK, 1 on error.

Two file descriptors are returned through the filedes argument: filedes[0] is open for reading, and filedes[1] is open for writing. The output of filedes[1] is the input for filedes[0].

Two ways to picture a half-duplex pipe are shown in Figure-1. The left half of the figure shows the two ends of the pipe connected in a single process. The right half of the figure emphasizes that the data in the pipe flows through the kernel.

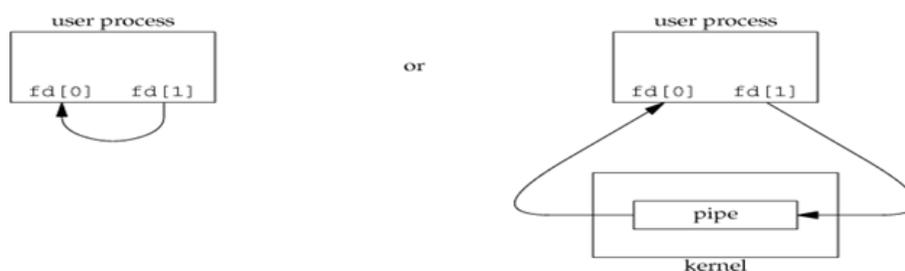


Fig-1: Two ways to view a half-duplex pipe

A pipe in a single process is next to useless. Normally, the process that calls pipe then calls fork, creating an IPC channel from the parent to the child or vice versa. Figure-2 shows this scenario.

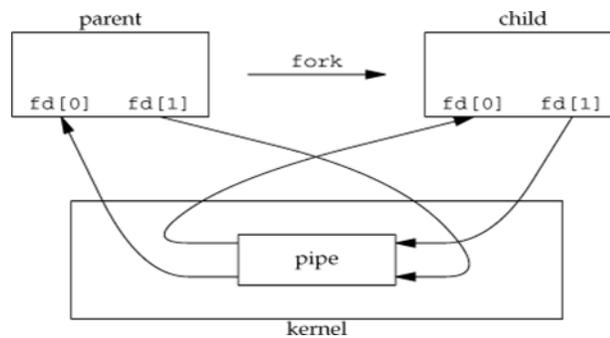


Fig-2: Half-duplex pipe after a fork

What happens after the fork depends on which direction of data flow we want. For a pipe from the parent to the child, the parent closes the read end of the pipe (fd[0]), and the child closes the write end (fd[1]). Figure-3 shows the resulting arrangement of descriptors.

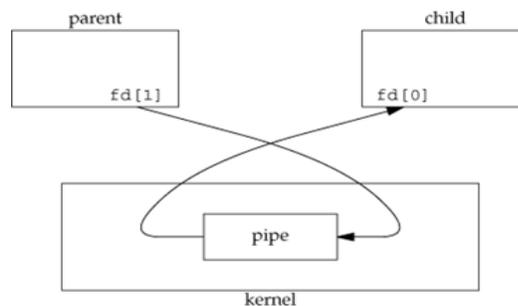


Fig-3: Pipe from parent to child

For a pipe from the child to the parent, the parent closes fd[1], and the child closes fd[0]. When one end of a pipe is closed, the following two rules apply.

- If we read from a pipe whose write end has been closed, read returns 0 to indicate an end of file after all the data has been read.
- If we write to a pipe whose read end has been closed, the signal SIGPIPE is generated. If we either ignore the signal or catch it and return from the signal handler, write returns 1 with errno set to EPIPE.

Program: shows the code to create a pipe between a parent and its child and to send data down the pipe.

```
#include "apue.h"

int
main(void)
{
    int    n;
    int    fd[2];
    pid_t  pid;
    char   line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("pipe error");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) { /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else { /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

Popen and Pclose Function

Since a common operation is to create a pipe to another process, to either read its output or send it input, the standard I/O library has historically provided the popen and pclose functions. These two functions handle all the dirty work that we've been doing ourselves: creating a pipe, forking a child, closing the unused ends of the pipe, executing a shell to run the command, and waiting for the command to terminate.

```
#include <stdio.h>

FILE *popen(const char *cmdstring, const char *type);
```

Returns: file pointer if OK, NULL on error

```
int pclose(FILE *fp);
```

Returns: termination status of cmdstring, or 1 on error

The function `popen` does a `fork` and `exec` to execute the `cmdstring`, and returns a standard I/O file pointer. If type is "r", the file pointer is connected to the standard output of `cmdstring`.



Fig-1: Result of `fp = popen(cmdstring, "r")`

If type is "w", the file pointer is connected to the standard input of `cmdstring`, as shown:



Coprocesses

A UNIX system filter is a program that reads from standard input and writes to standard output. Filters are normally connected linearly in shell pipelines. A filter becomes a coprocess when the same program generates the filter's input and reads the filter's output. A coprocess normally runs in the background from a shell, and its standard input and standard output are connected to another program using a pipe.

The process creates two pipes: one is the standard input of the coprocess, and the other is the standard output of the coprocess. Figure shows this arrangement.

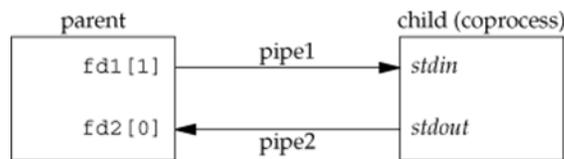


Figure- Driving a coprocess by writing its standard input and reading its standard output

Program: Simple filter to add two numbers

```
#include "apue.h"

int main(void)
{
    int    n, int1, int2;
    char   line[MAXLINE];

    while ((n = read(STDIN_FILENO, line, MAXLINE)) > 0) {
        line[n] = 0;          /* null terminate */

        if (sscanf(line, "%d%d", &int1, &int2) == 2) {
            sprintf(line, "%d\n", int1 + int2);
            n = strlen(line);
            if (write(STDOUT_FILENO, line, n) != n)
                err_sys("write error");
        } else {
            if (write(STDOUT_FILENO, "invalid args\n", 13) != 13)
                err_sys("write error");
        }
    }
    exit(0);
}
```

FIFOs

FIFOs are sometimes called named pipes. Pipes can be used only between related processes when a common ancestor has created the pipe.

```
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);
```

Returns: 0 if OK, 1 on error

Once we have used `mkfifo` to create a FIFO, we open it using `open`. When we open a FIFO, the nonblocking flag (`O_NONBLOCK`) affects what happens.

In the normal case (`O_NONBLOCK` not specified), an open for read-only blocks until some other process opens the FIFO for writing. Similarly, an open for write-only blocks until some other process opens the FIFO for reading.

If `O_NONBLOCK` is specified, an open for read-only returns immediately. But an open for write-only returns 1 with `errno` set to `ENXIO` if no process has the FIFO open for reading.

There are two uses for FIFOs.

1. FIFOs are used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files.
2. FIFOs are used as rendezvous points in client-server applications to pass data between the clients and the servers.

Example Using FIFOs to Duplicate Output Streams

FIFOs can be used to duplicate an output stream in a series of shell commands. This prevents writing the data to an intermediate disk file. Consider a procedure that needs to process a filtered input stream twice. Figure shows this arrangement.

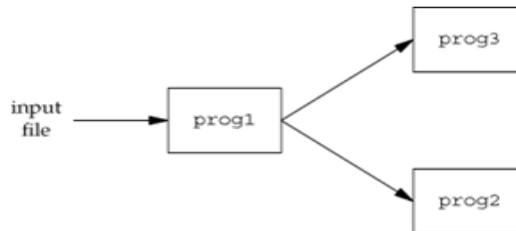


Figure- Procedure that processes a filtered input stream twice

With a FIFO and the UNIX program tee(1), we can accomplish this procedure without using a temporary file. (The tee program copies its standard input to both its standard output and to the file named on its command line.)

```

mkfifo fifo1
prog3 < fifo1 &
prog1 < infile | tee fifo1 | prog2
  
```

We create the FIFO and then start prog3 in the background, reading from the FIFO. We then start prog1 and use tee to send its input to both the FIFO and prog2. Figure shows the process arrangement.

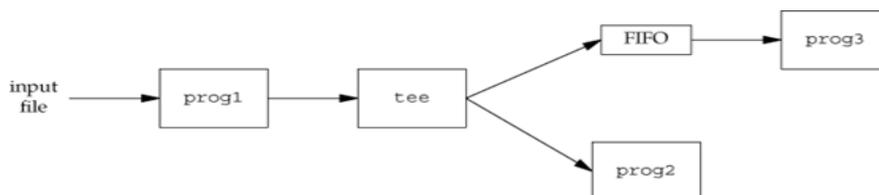


FIGURE: Using a FIFO and tee to send a stream to two different processes

Example Client-Server Communication Using a FIFO

1. FIFO's can be used to send data between a client and a server. If we have a server that is contacted by numerous clients, each client can write its request to a well-known FIFO that the server creates. Since there are multiple writers for the FIFO, the requests sent by the clients to the server need to be less than PIPE_BUF bytes in size.
2. This prevents any interleaving of the client writes. The problem in using FIFOs for this type of client server communication is how to send replies back from the server to each client.

3. A single FIFO can't be used, as the clients would never know when to read their response versus responses for other clients. One solution is for each client to send its process ID with the request. The server then creates a unique FIFO for each client, using a pathname based on the client's process ID.
4. For example, the server can create a FIFO with the name /vtu/ ser.XXXXX, where XXXXX is replaced with the client's process ID. This arrangement works, although it is impossible for the server to tell whether a client crashes. This causes the client-specific FIFOs to be left in the file system.
5. The server also must catch SIGPIPE, since it's possible for a client to send a request and terminate before reading the response, leaving the client-specific FIFO with one writer (the server) and no reader.

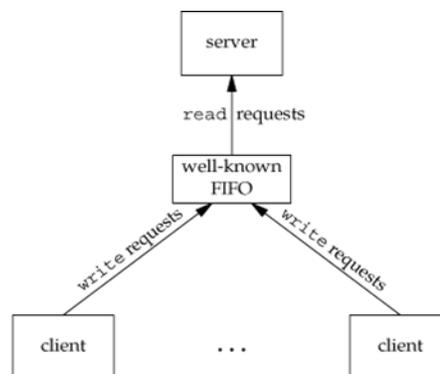


Figure Clients sending requests to a server using a FIFO

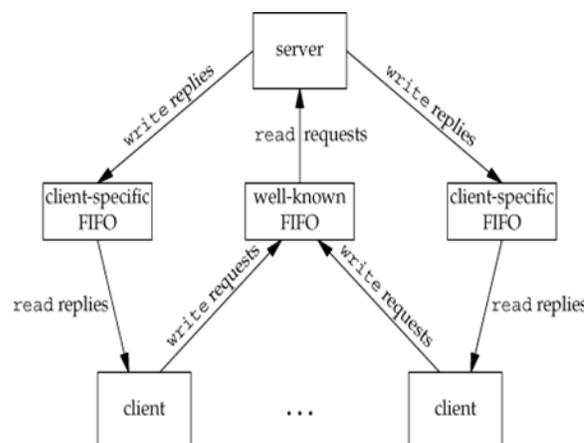


Figure 15.23. Client-server communication using FIFOs

XSI IPC

1. Identifiers and Keys
2. Permission Structure

3. Configuration Limits

1) Identifiers and Keys

Each IPC structure (message queue, semaphore, or shared memory segment) in the kernel is referred to by a non- negative integer identifier. The identifier is an internal name for an IPC object. Cooperating processes need an external naming scheme to be able to rendezvous using the same IPC object. For this purpose, an IPC object is associated with a key that acts as an external name.

Whenever an IPC structure is being created, a key must be specified. The data type of this key is the primitive system data type `key_t`, which is often defined as a long integer in the header `<sys/types.h>`. This key is converted into an identifier by the kernel.

There are various ways for a client and a server to rendezvous at the same IPC structure.

- The server can create a new IPC structure by specifying a key of `IPC_PRIVATE` and store the returned identifier somewhere (such as a file) for the client to obtain. The key `IPC_PRIVATE` guarantees that the server creates a new IPC structure. The disadvantage to this technique is that file system operations are required for the server to write the integer identifier to a file, and then for the clients to retrieve this identifier later.
- The `IPC_PRIVATE` key is also used in a parent-child relationship. The parent creates a new IPC structure specifying `IPC_PRIVATE`, and the resulting identifier is then available to the child after the fork. The child can pass the identifier to a new program as an argument to one of the `exec` functions.
- The client and the server can agree on a key by defining the key in a common header, for example. The server then creates a new IPC structure specifying this key. The problem with this approach is that it's possible for the key to already be associated with an IPC structure, in which case the `get` function (`msgget`, `semget`, or `shmget`) returns an error. The server must handle this error, deleting the existing IPC structure, and try to create it again.
- The client and the server can agree on a pathname and project ID (the project ID is a character value between 0 and 255) and call the function `ftok` to convert these two values into a key. This key is then used in step 2. The only service provided by `ftok` is a way of generating a key from a pathname and project ID.

```
#include <sys/ipc.h>

key_t ftok(const char *path, int id);
```

Returns: key if OK, `(key_t)-1` on error

The path argument must refer to an existing file. Only the lower 8 bits of `id` are used when generating the key.

The key created by `ftok` is usually formed by taking parts of the `st_dev` and `st_ino` fields in the `stat` structure corresponding to the given pathname and combining them with the project ID. If two pathnames refer to two different files, then `ftok` usually returns two different keys for the two pathnames. However, because both i-node numbers and keys are often stored in long integers, there can be information loss creating a key. This means that two different pathnames to different files can generate the same key if the same project ID is used.

2) Permission Structure

XSI IPC associates an `ipc_perm` structure with each IPC structure. This structure defines the permissions and owner and includes at least the following members:

```

struct ipc_perm
{
    uid_t uid; /* owner's effective user id */
    gid_t gid; /* owner's effective group id */
    uid_t cuid; /* creator's effective user id */
    gid_t cgid; /* creator's effective group id */
    mode_t mode; /* access modes */
    .
    .
}
    
```

All the fields are initialized when the IPC structure is created. At a later time, we can modify the `uid`, `gid`, and `mode` fields by calling `msgctl`, `semctl`, or `shmctl`. To change these values, the calling process must be either the creator of the IPC structure or the superuser. Changing these fields is similar to calling `chown` or `chmod` for a file.

Permission	Bit
user-read	0400
user-write (alter)	0200
group-read	0040
group-write (alter)	0020
other-read	0004
other-write (alter)	0002

Figure XSI IPC permissions

3) Configuration Limits

All three forms of XSI IPC have built-in limits that we may encounter. Most of these limits can be changed by reconfiguring the kernel. We describe the limits when we describe each of the three forms of IPC.

Advantages and Disadvantages

1. A fundamental problem with XSI IPC is that the IPC structures are systemwide and do not have a reference count. For example, if we create a message queue, place some messages on the queue, and then terminate, the message queue and its contents are not deleted. They remain in the system until specifically read or deleted by some process calling `msgrcv` or `msgctl`, by someone executing the `ipcrm(1)` command, or by the system being rebooted. Compare this with a pipe, which is completely removed when the last process to reference

it terminates. With a FIFO, although the name stays in the file system until explicitly removed, any data left in a FIFO is removed when the last process to reference the FIFO terminates.

2. Another problem with XSI IPC is that these IPC structures are not known by names in the file system. We can't access them and modify their properties with the functions. Almost a dozen new system calls (msgget, semop, shmat, and so on) were added to the kernel to support these IPC objects. We can't see the IPC objects with an ls command, we can't remove them with the rm command, and we can't change their permissions with the chmod command. Instead, two new commands ipcs(1) and ipcrm(1) were added.
3. Since these forms of IPC don't use file descriptors, we can't use the multiplexed I/O functions (select and poll) with them. This makes it harder to use more than one of these IPC structures at a time or to use any of these IPC structures with file or device I/O. For example, we can't have a server wait for a message to be placed on one of two message queues without some form of busywait loop.

Message Queues

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. We'll call the message queue just a queue and its identifier a queue ID.

A new queue is created or an existing queue opened by msgget. New messages are added to the end of a queue by msgsnd. Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to msgsnd when the message is added to a queue. Messages are fetched from a queue by msgrcv. We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

Each queue has the following msqid_ds structure associated with it:

```

struct msqid_ds
{
    struct ipc_perm  msg_perm;      /* see Section 15.6.2 */
    msgqnum_t        msg_qnum;      /* # of messages on queue */
    msglen_t         msg_qbytes;    /* max # of bytes on queue */
    pid_t            msg_lspid;     /* pid of last msgsnd() */
    pid_t            msg_lrpid;     /* pid of last msgrcv() */

    time_t           msg_stime;     /* last-msgsnd() time */
    time_t           msg_rtime;     /* last-msgrcv() time */
    time_t           msg_ctime;     /* last-change time */
    .
    .
    .
}
    
```

This structure defines the current status of the queue.

The first function normally called is `msgget` to either open an existing queue or create a new queue.

```
#include <sys/msg.h>

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

Returns: message queue ID if OK, 1 on error

When a new queue is created, the following members of the `msgid_ds` structure are initialized.

1. The `ipc_perm` structure is initialized. The `mode` member of this structure is set to the corresponding permission bits of `flag`.
2. `msg_qnum`, `msg_lspid`, `msg_lrpid`, `msg_stime`, and `msg_rtime` are all set to 0.
3. `msg_ctime` is set to the current time.
4. `msg_qbytes` is set to the system limit.

On success, `msgget` returns the non-negative queue ID. This value is then used with the other three message queue functions.

The `msgctl` function performs various operations on a queue.

```
#include <sys/msg.h>

int msgctl(int msgid, int cmd, struct msgid_ds *buf );
```

Returns: 0 if OK, 1 on error.

The `cmd` argument specifies the command to be performed on the queue specified by `msgid`.

Table 9.7.2 POSIX:XSI values for the cmd parameter of msgctl.	
cmd	description
IPC_RMID	remove the message queue msqid and destroy the corresponding msqid_ds
IPC_SET	set members of the msqid_ds data structure from buf
IPC_STAT	copy members of the msqid_ds data structure into buf

Data is placed onto a message queue by calling msgsnd.

```
#include <sys/msg.h>

int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
```

Returns: 0 if OK, 1 on error.

Each message is composed of a positive long integer type field, a non-negative length (nbytes), and the actual data bytes (corresponding to the length). Messages are always placed at the end of the queue.

The ptr argument points to a long integer that contains the positive integer message type, and it is immediately followed by the message data. (There is no message data if nbytes is 0.) If the largest message we send is 512 bytes, we can define the following structure:

```
struct mymesg
{
    long mtype;      /* positive message type */
    char mtext[512]; /* message data, of length nbytes */
};
```

The ptr argument is then a pointer to a mymesg structure. The message type can be used by the receiver to fetch messages in an order other than first in, first out.

Messages are retrieved from a queue by msgrcv.

```
#include <sys/msg.h>

ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
```

Messages are retrieved from a queue by msgrcv.

Returns: size of data portion of message if OK, 1 on error.

The type argument lets us specify which message we want.

type == 0	The first message on the queue is returned.
type > 0	The first message on the queue whose message type equals type is returned.
type < 0	The first message on the queue whose message type is the lowest value less than or equal to the absolute value of type is returned.

Semaphores

A semaphore is a counter used to provide access to a shared data object for multiple processes.

To obtain a shared resource, a process needs to do the following:

1. Test the semaphore that controls the resource.
2. If the value of the semaphore is positive, the process can use the resource. In this case, the process decrements the semaphore value by 1, indicating that it has used one unit of the resource.
3. Otherwise, if the value of the semaphore is 0, the process goes to sleep until the semaphore value is greater than 0. When the process wakes up, it returns to step 1.

When a process is done with a shared resource that is controlled by a semaphore, the semaphore value is incremented by 1. If any other processes are asleep, waiting for the semaphore, they are awakened.

A common form of semaphore is called a binary semaphore. It controls a single resource, and its value is initialized to 1. In general, however, a semaphore can be initialized to any positive value, with the value indicating how many units of the shared resource are available for sharing.

XSI semaphores are, unfortunately, more complicated than this. Three features contribute to this unnecessary complication.

1. A semaphore is not simply a single non-negative value. Instead, we have to define a semaphore as a set of one or more semaphore values. When we create a semaphore, we specify the number of values in the set.
2. The creation of a semaphore (`semget`) is independent of its initialization (`semctl`). This is a fatal flaw, since we cannot atomically create a new semaphore set and initialize all the values in the set.
3. Since all forms of XSI IPC remain in existence even when no process is using them, we have to worry about a program that terminates without releasing the semaphores it has been allocated. The undo feature that we describe later is supposed to handle this.

The kernel maintains a `semid_ds` structure for each semaphore set:

```

struct semid_ds {
    struct ipc_perm  sem_perm; /* see Section 15.6.2 */
    unsigned short  sem_nsems; /* # of semaphores in set */
    time_t          sem_otime; /* last-semop() time */
    time_t          sem_ctime; /* last-change time */
    .
    .
};

```

Each semaphore is represented by an anonymous structure containing at least the following members:

```

struct {

    unsigned short  semval; /* semaphore value, always >= 0 */
    pid_t          sempid; /* pid for last operation */
    unsigned short  semncnt; /* # processes awaiting semval>curval */
    unsigned short  semzcnt; /* # processes awaiting semval==0 */
    .
    .
};

```

The first function to call is `semget` to obtain a semaphore ID.

```

#include <sys/sem.h>

int semget(key_t key, int nsems, int flag);

```

Returns: semaphore ID if OK, 1 on error

When a new set is created, the following members of the `semid_ds` structure are initialized.

- The `ipc_perm` structure is initialized. The mode member of this structure is set to the corresponding permission bits of flag.
- `sem_otime` is set to 0.
- `sem_ctime` is set to the current time.
- `sem_nsems` is set to `nsems`.

The number of semaphores in the set is `nsems`. If a new set is being created (typically in the server), we must specify `nsems`. If we are referencing an existing set (a client), we can specify `nsems` as 0.

The `semctl` function is the catchall for various semaphore operations.

```

#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, ... /* union semun arg */);

```

The fourth argument is optional, depending on the command requested, and if present, is of type `semun`, a union of various command-specific arguments:

```
union semun
{
  int val; /* for SETVAL */
  struct semid_ds *buf; /* for IPC_STAT and IPC_SET */
  unsigned short *array; /* for GETALL and SETALL */
};
```

Table 9.8.1 POSIX:XSI values for the cmd parameter of semctl.	
cmd	description
GETALL	return values of the semaphore set in arg.array
GETVAL	return value of a specific semaphore element
GETPID	return process ID of last process to manipulate element
GETNCNT	return number of processes waiting for element to increment
GETZCNT	return number of processes waiting for element to become 0
IPC_RMID	remove semaphore set identified by semid
IPC_SET	set permissions of the semaphore set from arg.buf
IPC_STAT	copy members of semid_ds of semaphore set semid into arg.buf
SETALL	set values of semaphore set from arg.array
SETVAL	set value of a specific semaphore element to arg.val

The cmd argument specifies one of the above ten commands to be performed on the set specified by semid. The function semop atomically performs an array of operations on a semaphore set.

```
#include <sys/sem.h>
```

```
int semop(int semid, struct sembuf semoparray[], size_t nops);
```

Returns: 0 if OK, 1 on error.

The semoparray argument is a pointer to an array of semaphore operations, represented by sembuf structures:

```
struct sembuf {
  unsigned short sem_num; /* member # in set (0, 1, ..., nsems-1) */
  short sem_op; /* operation (negative, 0, or positive) */
  short sem_flg; /* IPC_NOWAIT, SEM_UNDO */
};
```

The nops argument specifies the number of operations (elements) in the array.

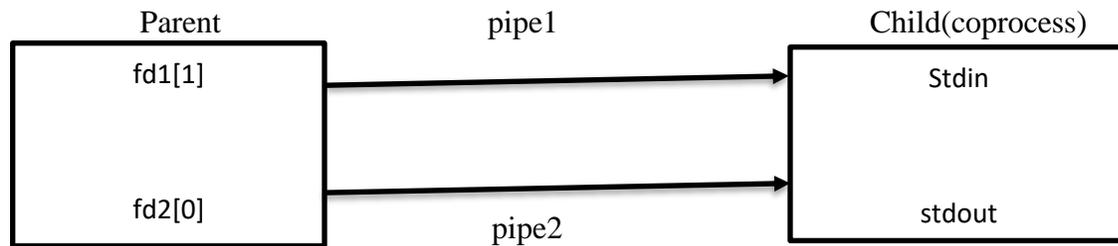
The sem_op element operations are values specifying the amount by which the semaphore value is to be changed.

1. If `sem_op` is an integer greater than zero, `semop` adds the value to the corresponding semaphore element value and awakens all processes that are waiting for the element to increase.
2. If `sem_op` is 0 and the semaphore element value is not 0, `semop` blocks the calling process (waiting for 0) and increments the count of processes waiting for a zero value of that element.
3. If `sem_op` is a negative number, `semop` adds the `sem_op` value to the corresponding semaphore element value provided that the result would not be negative. If the operation would make the element value negative, `semop` blocks the process on the event that the semaphore element value increases. If the resulting value is 0, `semop` wakes the processes waiting for 0.

Shared Memory

Client-Server Properties

The properties of clients and servers that are affected by the various types of IPC used between them. The simplest type of relationship is to have the client fork and exec the desired server. Two half-duplex pipes can be created before the fork to allow data to be transferred in both directions. Figure is an example of this. The server that is executed can be a set-user-ID program, giving it special privileges. Also, the server can determine the real identity of the client by looking at its real user ID.



With this arrangement, we can build an open server. It opens files for the client instead of the client calling the open function. This way, additional permission checking can be added, above and beyond the normal UNIX system user/group/other permissions. Then assume that the server is a set-user-ID program, giving it additional permissions (root permission, perhaps). The server uses the real user ID of the client to determine whether to give it access to the requested file. This way, can build a server that allows certain users permissions that they don't normally have.

In this example, since the server is a child of the parent, all the server can do is pass back the contents of the file to the parent. Although this works fine for regular files, it can't be used for special device files, for example. We would like to be able to have the server open the requested file and pass back the file descriptor. Whereas a parent can pass a child an open descriptor, a child cannot pass a descriptor back to the parent.

The next type of server is a daemon process that is contacted using some form of IPC by all clients. We can't use pipes for this type of clientserver. A form of named IPC is required, such as FIFOs or message queues. With FIFOs, we saw that an individual per client FIFO is also required if the server is to send data back to the client. If the clientserver application sends data only from the client to the server, a single well-known FIFO suffices.

Multiple possibilities exist with message queues.

1. A single queue can be used between the server and all the clients, using the type field of each message to indicate the message recipient. For example, the clients can send their requests with a type field of 1. Included in the request must be the client's process ID. The server then sends the response with the type field set to the client's process ID. The server receives only the messages with a type field of 1 (the fourth argument for msgrcv), and the clients receive only the messages with a type field equal to their process IDs.
2. Alternatively, an individual message queue can be used for each client. Before sending the first request to a server, each client creates its own message queue with a key of IPC_PRIVATE. The server also has its own queue, with a key or identifier known to all clients. The client sends its first request to the server's well-known queue, and this request

must contain the message queue ID of the client's queue. The server sends its first response to the client's queue, and all future requests and responses are exchanged on this queue.

One problem with this technique is that each client-specific queue usually has only a single message on it: a request for the server or a response for a client. This seems wasteful of a limited systemwide resource (a message queue), and a FIFO can be used instead. Another problem is that the server has to read messages from multiple queues. Neither select nor poll works with message queues.

Either of these two techniques using message queues can be implemented using shared memory segments and a synchronization method (a semaphore or record locking).

Stream Pipes

A STREAMS-based pipe ("STREAMS pipe," for short) is a bidirectional (full-duplex) pipe. To obtain bidirectional data flow between a parent and a child, only a single STREAMS pipe is required. STREAMS pipes are supported by Solaris and are available in an optional add-on package with Linux. Figure1 shows the two ways to view a STREAMS pipe.

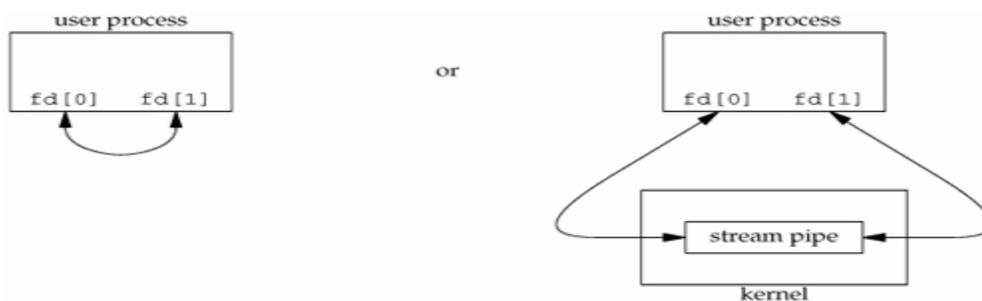


Figure1- Two ways to view a Streams pipe

If we look inside a STREAMS pipe Figure2, we see that it is simply two stream heads, with each write queue (WQ) pointing at the other's read queue (RQ). Data written to one end of the pipe is placed in messages on the other's read queue.

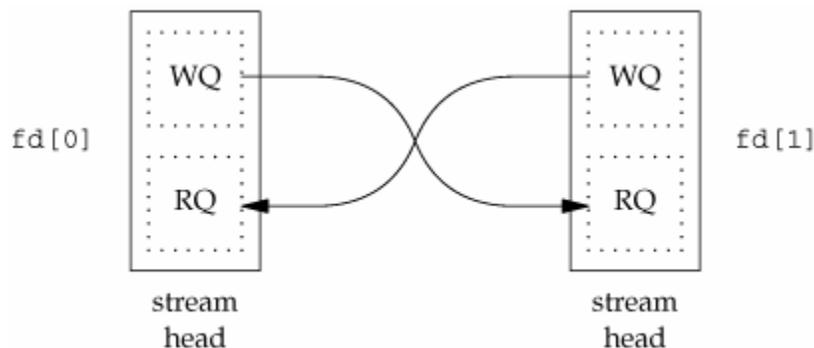


Figure2- Inside a streams pipe

Since a STREAMS pipe is a stream, we can push a STREAMS module onto either end of the pipe to process data written to the pipe Figure3 But if we push a module on one end, we can't pop it off the other end. If we want to remove it, we need to remove it from the same end on which it was pushed.

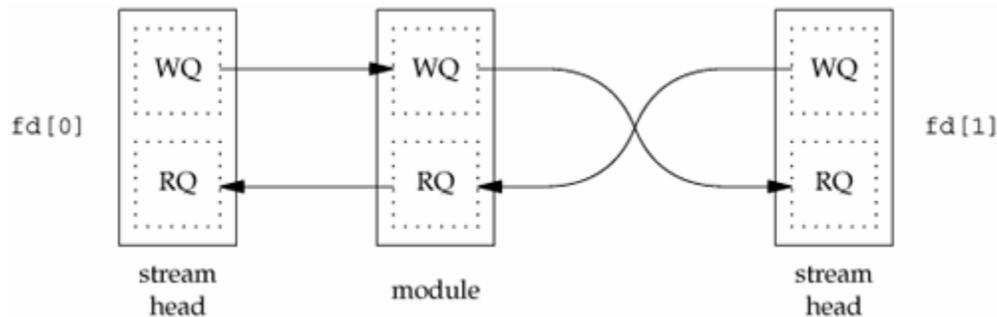


Figure3- Inside a streams pipe with a module

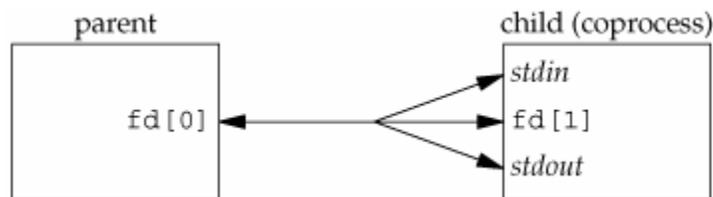


Figure4- Arrangement of descriptors for coprocess

The parent uses only fd[0], and the child uses only fd[1]. Since each end of the STREAMS pipe is full duplex, the parent reads and writes fd[0], and the child duplicates fd[1] to both standard input and standard output. Figure4- shows the resulting descriptors. Note that this example also works with full-duplex pipes that are not based on Streams, because it doesn't make use of any STREAMS features other than the full-duplex nature of STREAMS- based pipes.

Example- STREAMS-Based s_pipe Function

This example shows the STREAMS-based version of the s_pipe function. This version simply calls the standard pipe function, which creates a full-duplex pipe.

```
#include "apue.h"
/*
 * Returns a STREAMS-based pipe, with the two file descriptors
 * returned in fd[0] and fd[1].
 */
int
s_pipe(int fd[2])
{
    return(pipe(fd));
}
```

1. Naming STREAMS Pipes.
2. Unique Connection.

Naming STREAMS Pipes

Normally, pipes can be used only between related processes: child processes inheriting pipes from their parent processes. So that unrelated processes can communicate using FIFOs, but this provides only a one-way communication path. The STREAMS mechanism provides a way for processes to give a pipe a name in the file system. This bypasses the problem of dealing with unidirectional FIFOs. We can use the `fattach` function to give a STREAMS pipe a name in the file system.

```
#include <stropts.h>

int fattach(int filedes, const char *path);
```

Returns: 0 if OK, 1 on error

The path argument must refer to an existing file, and the calling process must either own the file and have write permissions to it or be running with superuser privileges.

Once a STREAMS pipe is attached to the file system namespace, the underlying file is inaccessible. Any process that opens the name will gain access to the pipe, not the underlying file. Any processes that had the underlying file open before `fattach` was called, however, can continue to access the underlying file. Indeed, these processes generally will be unaware that the name now refers to a different file.

Figure- shows a pipe attached to the pathname `/tmp/pipe`. Only one end of the pipe is attached to a name in the file system. The other end is used to communicate with processes that open the attached filename. Even though it can attach any kind of STREAMS file descriptor to a name in the file system, the `fattach` function is most commonly used to give a name to a STREAMS pipe.

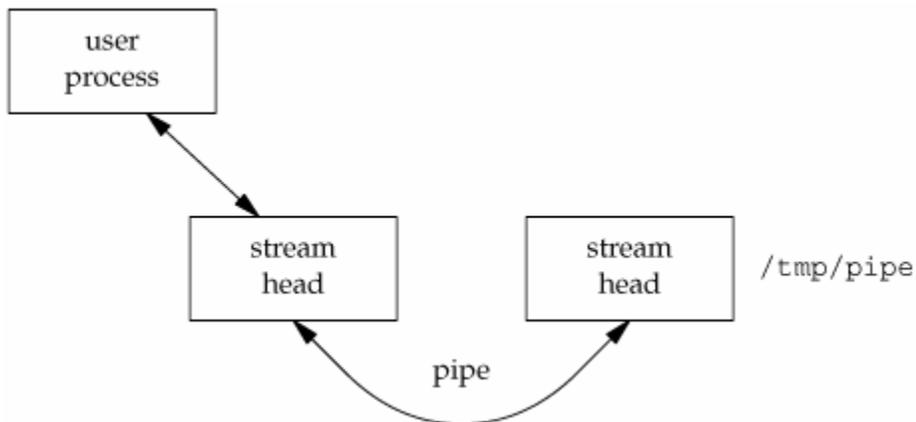


Figure- A pipe mounted on a name in the file system

A process can call `fdetach` to undo the association between a STREAMS file and the name in the file system.

```
#include <stropts.h>

int fdetach(const char *path);

Returns: 0 if OK, 1 on error
```

After `fdetach` is called, any processes that had accessed the STREAMS pipe by opening the path will still continue to access the stream, but subsequent opens of the path will access the original file residing in the file system.

Unique Connections

Although we can attach one end of a STREAMS pipe to the file system namespace, we still have problems if multiple processes want to communicate with a server using the named STREAMS pipe. Data from one client will be interleaved with data from the other clients writing to the pipe. Even if we guarantee that the clients write less than `PIPE_BUF` bytes so that the writes are atomic, we have no way to write back to an individual client and guarantee that the intended client will read the message. With multiple clients reading from the same pipe, we cannot control which one will be scheduled and actually read what we send.

The `connld` STREAMS module solves this problem. Before attaching a STREAMS pipe to a name in the file system, a server process can push the `connld` module on the end of the pipe that is to be attached. This results in the configuration shown in Figure 1.

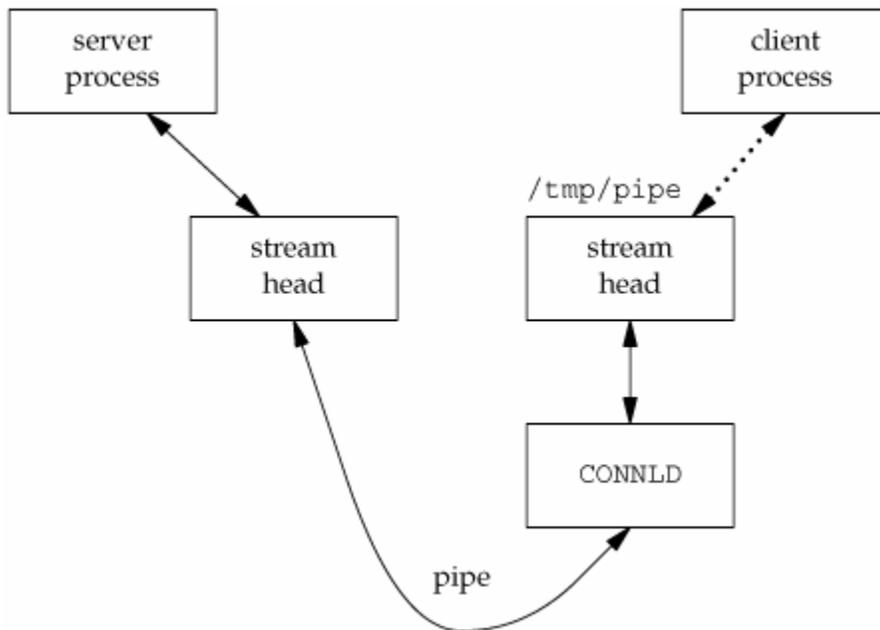


Figure1- Setting up connld for unique connections.

In Figure1, the server process has attached one end of its pipe to the path /tmp/pipe. We show a dotted line to indicate a client process in the middle of opening the attached STREAMS pipe. Once the open completes, we have the configuration shown in Figure2.

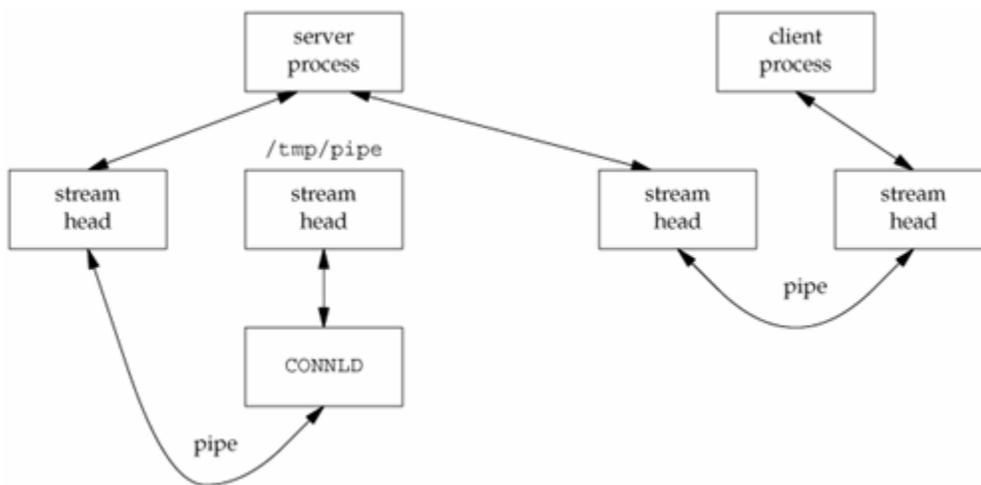


Figure2- Using connld to make unique connections

The three functions that can be used to create unique connections between unrelated processes.

```
#include "apue.h"
int serv_listen(const char *name);
```

Returns: file descriptor to listen on if OK, negative value on error
<pre>int serv_accept(int listenfd, uid_t *uidptr);</pre>
Returns: new file descriptor if OK, negative value on error
<pre>int cli_conn(const char *name);</pre>
Returns: file descriptor if OK, negative value on error

The `serv_listen` function can be used by a server to announce its willingness to listen for client connect requests on a well-known name (some pathname in the file system). Clients will use this name when they want to connect to the server. The return value is the server's end of the STREAMS pipe.

The `serv_accept` function is used by a server to wait for a client's connect request to arrive. When one arrives, the system automatically creates a new STREAMS pipe, and the function returns one end to the server. Additionally, the effective user ID of the client is stored in the memory to which `uidptr` points.

A client calls `cli_conn` to connect to a server. The name argument specified by the client must be the same name that was advertised by the server's call to `serv_listen`. On return, the client gets a file descriptor connected to the server.

Passing File Descriptors

The ability to pass an open file descriptor between processes is powerful. It can lead to different ways of designing clientserver applications. It allows one process (typically a server) to do everything that is required to open a file (involving such details as translating a network name to a network address, dialing a modem, negotiating locks for the file, etc.) and simply pass back to the calling process a descriptor that can be used with all the I/O functions. All the details involved in opening the file or device are hidden from the client.

When we pass an open file descriptor from one process to another, we want the passing process and the receiving process to share the same file table entry. Figure 1 shows the desired arrangement.

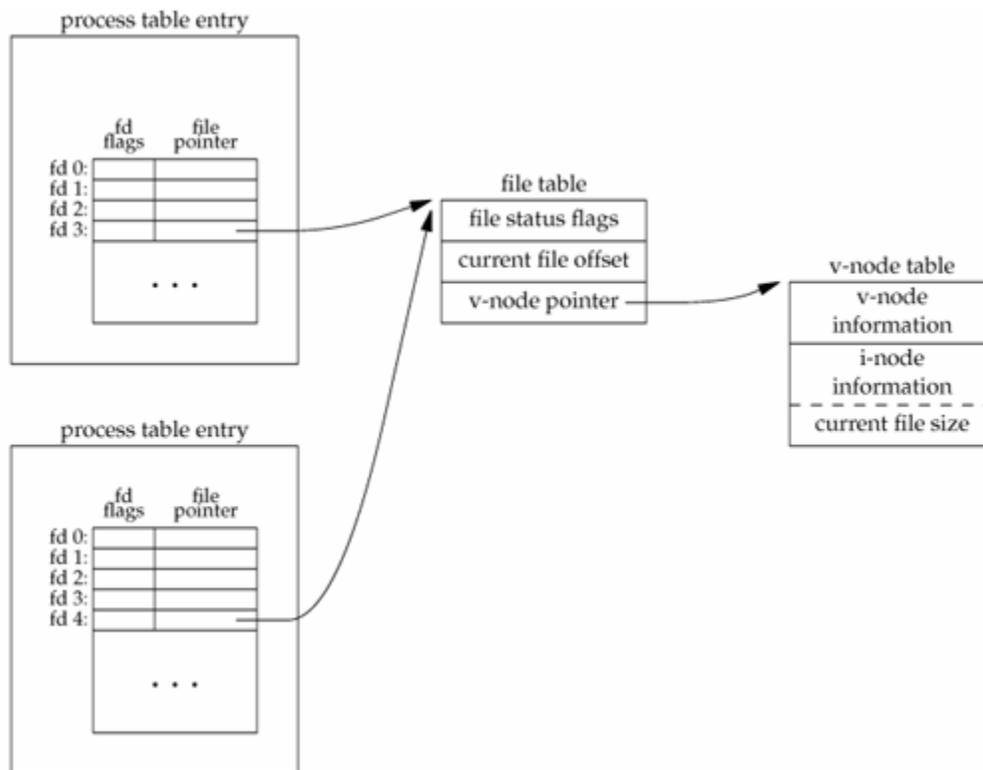


Figure 1- Passing an open file from the top process to the bottom process.

Technically, we are passing a pointer to an open file table entry from one process to another. This pointer is assigned the first available descriptor in the receiving process. (Saying that we are passing an open descriptor mistakenly gives the impression that the descriptor number in the receiving process is the same as in the sending process, which usually isn't true.)

What normally happens when a descriptor is passed from one process to another is that the sending process, after passing the descriptor, then closes the descriptor. Closing the descriptor by the sender doesn't really close the file or device, since the descriptor is still considered open by the receiving process (even if the receiver hasn't specifically received the descriptor yet).

```
#include "apue.h"

int send_fd(int fd, int fd_to_send);
int send_err(int fd, int status, const char *errmsg);

Both return: 0 if OK, 1 on error

int rcv_fd(int fd, ssize_t (*userfunc)(int, const
void *, size_t));

Returns: file descriptor if OK, negative value on error
```

A process (normally a server) that wants to pass a descriptor to another process calls either `send_fd` or `send_err`. The process waiting to receive the descriptor (the client) calls `recv_fd`.

The `send_fd` function sends the descriptor `fd_to_send` across using the STREAMS pipe or UNIX domain socket represented by `fd`. We'll use the term *s-pipe* to refer to a bidirectional communication channel that could be implemented as either a STREAMS pipe or a UNIX domain stream socket.

The `send_err` function sends the `errmsg` using `fd`, followed by the status byte. The value of status must be in the range 1 through 255.

Clients call `recv_fd` to receive a descriptor. If all is OK (the sender called `send_fd`), the non-negative descriptor is returned as the value of the function. Otherwise, the value returned is the status that was sent by `send_err` (a negative value in the range 1 through -255). Additionally, if an error message was sent by the server, the client's `userfunc` is called to process the message. The first argument to `userfunc` is the constant `STDERR_FILENO`, followed by a pointer to the error message and its length. The return value from `userfunc` is the number of bytes written or a negative number on error. Often, the client specifies the normal write function as the `userfunc`.

We implement our own protocol that is used by these three functions. To send a descriptor, `send_fd` sends two bytes of 0, followed by the actual descriptor. To send an error, `send_err` sends the `errmsg`, followed by a byte of 0, followed by the absolute value of the status byte (1 through 255). The `recv_fd` function reads everything on the *s-pipe* until it encounters a null byte. Any characters read up to this point are passed to the caller's `userfunc`. The next byte read by `recv_fd` is the status byte. If the status byte is 0, a descriptor was passed; otherwise, there is no descriptor to receive.

The function `send_err` calls the `send_fd` function after writing the error message to the *s-pipe*. This is shown in Figure 2.

```
#include "apue.h"
/*
 * Used when we had planned to send an fd using send_fd(),
 * but encountered an error instead. We send the error back
 * using the send_fd()/recv_fd() protocol.
 */
int
send_err(int fd, int errcode, const char *msg)
{
    int    n;

    if ((n = strlen(msg)) > 0)
        if (written(fd, msg, n) != n)    /* send the error message */
            return(-1);

    if (errcode >= 0)
        errcode = -1;    /* must be negative */
}
```

```

        if (send_fd(fd, errcode) < 0)
            return(-1);

        return(0);
}

```

Figure-2 The send_err function.

1. Passing File Descriptors over STREAMS-Based Pipes
2. Passing File Descriptors over UNIX Domain Sockets

Passing File Descriptors over STREAMS-Based Pipes

With STREAMS pipes, file descriptors are exchanged using two `ioctl` commands: `I_SENDFD` and `I_RECVFD`. To send a descriptor, we set the third argument for `ioctl` to the actual descriptor.

Figure-3 The send_fd function for STREAMS pipes

```

#include "apue.h"
#include <stropts.h>

/*
 * Pass a file descriptor to another process.
 * If fd<0, then -fd is sent back instead as the error status.
 */
int
send_fd(int fd, int fd_to_send)
{
    char    buf[2];        /* send_fd()/recv_fd() 2-byte protocol */

    buf[0] = 0;           /* null byte flag to recv_fd() */
    if (fd_to_send < 0) {
        buf[1] = -fd_to_send; /* nonzero status means error */
        if (buf[1] == 0)
            buf[1] = 1; /* -256, etc. would screw up protocol */
    } else {
        buf[1] = 0;        /* zero status means OK */
    }

    if (write(fd, buf, 2) != 2)
        return(-1);
    if (fd_to_send >= 0)
        if (ioctl(fd, I_SENDFD, fd_to_send) < 0)
            return(-1);
    return(0);
}

```

When we receive a descriptor, the third argument for `ioctl` is a pointer to a `strrecvfd` structure:

```

struct strrecvfd {
    int    fd;            /* new descriptor */

```

```
uid_t  uid;      /* effective user ID of sender */
gid_t  gid;      /* effective group ID of sender */
char   fill[8];
};
```

Passing File Descriptors over UNIX Domain Sockets

To exchange file descriptors using UNIX domain sockets, we call the `sendmsg(2)` and `recvmsg(2)` functions. Both functions take a pointer to a `msg_hdr` structure that contains all the information on what to send or receive. The structure on your system might look similar to the following:

```
struct msg_hdr {
    void          *msg_name;      /* optional address */
    socklen_t     msg_namelen;    /* address size in bytes */
    struct iovec  *msg_iov;       /* array of I/O buffers */
    int           msg_iovlen;     /* number of elements in array */
    void          *msg_control;   /* ancillary data */
    socklen_t     msg_controllen; /* number of ancillary bytes */
    int           msg_flags;      /* flags for received message */
};
```

The first two elements are normally used for sending datagrams on a network connection, where the destination address can be specified with each datagram.

Two elements deal with the passing or receiving of control information. The `msg_control` field points to a `cmsghdr` (control message header) structure, and the `msg_controllen` field contains the number of bytes of control information.

```
struct cmsghdr {
    socklen_t     cmsg_len;       /* data byte count, including header */
    int           cmsg_level;     /* originating protocol */
    int           cmsg_type;      /* protocol-specific type */
    /* followed by the actual control message data */
};
```

To send a file descriptor, we set `cmsg_len` to the size of the `cmsghdr` structure, plus the size of an integer (the descriptor). The `cmsg_level` field is set to `SOL_SOCKET`, and `cmsg_type` is set to `SCM_RIGHTS`, to indicate that we are passing access rights. (SCM stands for socket-level control message.) Access rights can be passed only across a UNIX domain socket. The descriptor is stored right after the `cmsg_type` field, using the macro `MSG_DATA` to obtain the pointer to this integer.

Three macros are used to access the control data, and one macro is used to help calculate the value to be used for `cmsg_len`.

<pre>#include <sys/socket.h></pre>
<pre>unsigned char *MSG_DATA(struct cmsghdr *cp);</pre>
Returns: pointer to data associated with <code>cmsghdr</code> structure
<pre>struct cmsghdr *MSG_FIRSTHDR(struct msghdr *mp);</pre>
Returns: pointer to first <code>cmsghdr</code> structure associated with the <code>msghdr</code> structure, or <code>NULL</code> if none exists
<pre>struct cmsghdr *MSG_NXTHDR(struct msghdr *mp, struct cmsghdr *cp);</pre>
Returns: pointer to next <code>cmsghdr</code> structure associated with the <code>msghdr</code> structure given the current <code>cmsghdr</code> structure, or <code>NULL</code> if we're at the last one
<pre>unsigned int MSG_LEN(unsigned int nbytes);</pre>
Returns: size to allocate for data object <code>nbytes</code> large

The Single UNIX Specification defines the first three macros, but omits `MSG_LEN`. The `MSG_LEN` macro returns the number of bytes needed to store a data object of size `nbytes`, after adding the size of the `cmsghdr` structure, adjusting for any alignment constraints required by the processor architecture, and rounding up.

An Open Server-Version 1

Using file descriptor passing, we now develop an open server—a program that is executed by a process to open one or more files. Instead of sending the contents of the file back to the calling process, however, this server sends back an open file descriptor. As a result, the open server can work with any type of file (such as a device or a socket) and not simply regular files. The client and server exchange a minimum amount of information using IPC: the filename and open mode sent by the client, and the descriptor returned by the server.

- There are several advantages in designing the server to be a separate executable program. The server can easily be contacted by any client, similar to the client calling a library function. We are not hard-coding a particular service into the application, but designing a general facility that others can reuse.
- If we need to change the server, only a single program is affected. Conversely, updating a library function can require that all programs that call the function be updated (i.e., relinked with the link editor).
- The server can be a set-user-ID program, providing it with additional permissions that the client does not have. Note that a library function (or shared library function) can't provide this capability.

The client process creates an fd-pipe and then calls fork and exec to invoke the server. The client sends requests across the fd-pipe using one end, and the server sends back responses over the fd-pipe using the other end. We define the following application protocol between the client and the server.

1. The client sends a request of the form “open <pathname> <openmode>\0” across the fd-pipe to the server. The <openmode> is the numeric value, in ASCII decimal, of the second argument to the open function. This request string is terminated by a null byte.
2. The server sends back an open descriptor or an error by calling either send_fd or send_err.

We first have the header, open.h (Figure-1), which includes the standard headers and defines the function prototypes.

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

#define CL_OPEN "open"          /* client's request for server */
int    csopen(char *, int);
```

Figure-1- The open.h header

The main function Figure-2 is a loop that reads a pathname from standard input and copies the file to standard output. The function calls csopen to contact the open server and return an open descriptor.

```
#include "open.h"
#include <fcntl.h>

#define BUFFSIZE 8192

int
main(int argc, char *argv[])
{
    int n, fd;
    char buf[BUFFSIZE];
    char line[MAXLINE];

    /* read filename to cat from stdin */
    while (fgets(line, MAXLINE, stdin) != NULL) {
        if (line[strlen(line) - 1] == '\n')
            line[strlen(line) - 1] = 0; /* replace newline with null */

        /* open the file */
        if ((fd = csopen(line, O_RDONLY)) < 0)
            continue; /* csopen() prints error from server */

        /* and cat to stdout */
        while ((n = read(fd, buf, BUFFSIZE)) > 0)
            if (write(STDOUT_FILENO, buf, n) != n)
                err_sys("write error");
        if (n < 0)
            err_sys("read error");
        close(fd);
    }
    exit(0);
}
```

Figure-2- The client main function, version-1

Client-Server Connection Functions.

The Client-Server architecture, which refers to two processes or two applications that communicate with each other to exchange some information. One of the two processes acts as a client process, and another process acts as a server.

Client Process- This is the process, which typically makes a request for information. After getting the response, this process may terminate or may do some other processing.

Server Process- This is the process which takes a request from the clients. After getting a request from the client, this process will perform the required processing, gather the requested information,

and send it to the requestor client. Once done, it becomes ready to serve another client. Server processes are always alert and ready to serve incoming requests.

There are two types of servers you can have –

Iterative Server – This is the simplest form of server where a server process serves one client and after completing the first request, it takes request from another client. Meanwhile, another client keeps waiting.

Concurrent Servers – This type of server runs multiple concurrent processes to serve many requests at a time because one process may take longer and another client cannot wait for so long. The simplest way to write a concurrent server under Unix is to fork a child process to handle each client separately.

The `opend.h` header (figure-1), which includes the standard headers and declares the global variables and function prototypes.

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

#define CL_OPEN "open"          /* client's request for server */

extern char  errmsg[]; /* error message string to return to client */
extern int   oflag;    /* open() flag: O_XXX ... */
extern char *pathname; /* of file to open() for client */

int  cli_args(int, char **);
void handle_request(char *, int, int);
```

Figure-1 - The `opend.h` header, version 1

The main function (Figure-2) reads the requests from the client on the fd-pipe (its standard input) and calls the function `handle_request`.

```
#include    "opend.h"

char      errmsg[MAXLINE];
int       oflag;
char      *pathname;

int
main(void)
{
    int     nread;
    char    buf[MAXLINE];

    for ( ; ; ) { /* read arg buffer from client, process request */
        if ((nread = read(STDIN_FILENO, buf, MAXLINE)) < 0)
            err_sys("read error on stream pipe");
        else if (nread == 0)
            break; /* client has closed the stream pipe */
        handle_request(buf, nread, STDOUT_FILENO);
    }
    exit(0);
}
```

Figure-2 The server main function, version 1

The function `handle_request` in Figure-3 does all the work. It calls the function `buf_args` to break up the client's request into a standard `argv`-style argument list and calls the function `cli_args` to process the client's arguments. If all is OK, `open` is called to open the file, and then `send_fd` sends the descriptor back to the client across the `fd`-pipe (its standard output). If an error is encountered, `send_err` is called to send back an error message, using the client-server protocol.

```
#include "opend.h"
#include <fcntl.h>

void
handle_request(char *buf, int nread, int fd)
{
    int newfd;

    if (buf[nread-1] != 0) {
        snprintf(errmsg, MAXLINE-1,
            "request not null terminated: %*.*s\n", nread, nread, buf);
        send_err(fd, -1, errmsg);
        return;
    }
    if (buf_args(buf, cli_args) < 0) { /* parse args & set options */
        send_err(fd, -1, errmsg);
        return;
    }
    if ((newfd = open(pathname, oflag)) < 0) {
        snprintf(errmsg, MAXLINE-1, "can't open %s: %s\n", pathname,
            strerror(errno));
        send_err(fd, -1, errmsg);
        return;
    }
    if (send_fd(fd, newfd) < 0) /* send the descriptor */
        err_sys("send_fd error");
    close(newfd); /* we're done with descriptor */
}
```

Figure-3 The handle_request function, version 1

Module – 5:Signal

Introduction

Signals are software interrupts. Signals provide a way of handling asynchronous events: a user at a terminal typing the interrupt key to stop a program or the next program in a pipeline terminating prematurely.

When a signal is sent to a process, it is pending on the process to handle it. The process can react to pending signals in one of three ways:

1. Accept the default action of the signal, which for most signals will terminate the process.
2. Ignore the signal. The signal will be discarded and it has no affect whatsoever on the recipient process.
3. Invoke a user-defined function. The function is known as a signal handler routine and the signal is said to be caught when this function is called.

Unix kernel support for signals

In Unix system version, each entry in the kernel process table slot has an array of signal flags, one for each defined in the system.

When a signal is generated for a process, the kernel will set the corresponding signal flag in the process table slot of the recipient process. If the recipient process is asleep, the kernel will awaken the process by scheduling it. When the recipient process runs, the kernel will check the process U-area that contains an array of signal handling specifications.

If array entry contains a zero value, the process will accept the default action of the signal.

If array entry contains a 1 value, the process will ignore the signal and kernel will discard it.

If array entry contains any other value, it is used as the function pointer for a user-defined signal handler routine.

The kernel will setup the process to execute the function immediately and the process will return to its current point of execution (or to some other place if signal handler does a long jmp), if the signal handler does not terminate the process. If there are different signals pending on a process, the order in which they are sent to a recipient process is undefined. If multiple instances of a signal are pending on a process, it is implementation-dependent on whether a signal instance or multiple instances of the signal is pending, but not how many of them are present.

Signal Concepts

All UNIX systems and ANSI – C support the signal API, which can be used to define the per-signal handling method.

The function prototype of the signal is:

```
#include <signal.h>
void (*signal (int signal_num, void (*handler)(int)))(int);
```

signal_num is the signal identifier like SIGINT or SIGTERM defined in the <signal.h>. handler is the function pointer of a user defined signal handler function. This function should take an integer formal argument and does not return any value.

Example below attempts to catch the SIGTERM, ignores the SIGINT, and accepts the default action of the SIGSEGV signal. The pause API suspends the calling process until it is interrupted by a signal and the corresponding signal handler does a return:

```
#include <iostream.h>
#include <signal.h>

void catch_sig(int sig_num)
{
    signal(sig_num, catch_sig);
    cout << "catch_sig:" << sig_num << endl;
}

int main()
{
    signal(SIGTERM, catch_sig);
    signal(SIGINT, SIG_IGN);
    signal(SIGSEGV, SIG_DFL);
    pause();           // wait for signal interruption
}
```

The SIG_IGN and SIG_DFL are manifest constants defined in <signal.h>

```
#define SIG_DFL    void (*)(int)0 // Default action
#define SIG_IGN    void (*)(int)1 // Ignore the signal
```

The return value of signal API is the previous signal handler for the signal. UNIX system V.3 and V.4 support the sigset API, which has the same prototype and similar use a signal.

```
#include <signal.h>

void (*sigset (int signal_num, void (*handler)(int))(int);
```

the sigset arguments and return value is the same as that of signal. Both the functions set signal handling methods for any named signal; but, signal API is unreliable and sigset is reliable.

This means that when a signal is set to be caught by a signal handler via sigset, when multiple instances of the signal arrive one of them is handled while other instances are blocked. Further, the signal handler is not reset to SIG_DFL when it is invoked.

Signal Mask

Each process in UNIX or POSIX.1 system has signal mask that defines which signals are blocked when generated to a process. A blocked signal depends on the recipient process to unblock it and

handle it accordingly. If a signal is specified to be ignored and blocked, it is implementation dependent on whether the signal will be discarded or left pending when it is sent to the process. A process initially inherits the parent’s signal mask when it is created, but any pending signals for the parent process are not passed on. A process may query or set its signal mask via the sigprocmask API:

```
#include <signal.h>
int sigprocmask(int cmd, const sigset_t *new_mask, sigset_t *old_mask);
```

new_mask defines a set of to be set or reset in a calling process signal mask. cmd specifies how the new_mask value is to be used by the API. The possible values cmd are:

cmd value	Meaning
SIG_SETMASK	Overrides the calling process signal mask with the value specified in the new_mask argument
SIG_BLOCK	Adds the signals specified in the new_mask argument to the calling process signal mask
SIG_UNBLOCK	Removes the signals specified in the new_mask argument to the calling process signal mask

If the actual argument to new_mask argument is a NULL pointer, the cmd argument will be ignored, and the current process signal mask will not be altered. The old_mask argument is the address of a sigset_t variable that will be assigned the calling process’s original signal mask prior to a sigprocmask call. If the actual argument to old_mask is a NULL pointer, no previous signal mask will be returned. The return value of sigprocmask call is zero if it succeeds or -1 if it fails.

The sigset_t is a data type defined in <signal.h>. It contains a collection of bit flags, with each bit flag representing one signal defined in the system. The BSD UNIX and POSIX.1 define a set of API known as sigsetops functions, which set, reset, and query the presence of signals in a sigset_t typed variable.

```
#include <signal.h>
int sigemptyset(sigset_t *sigmask);
int sigaddset(sigset_t *sigmask, const int signal_num);
int sigdelset(sigset_t *sigmask, const int signal_num);
int sigfillset(sigset_t sigmask);
int sigismember(const sigset_t *sigmask, const int signal_num);
```

The sigemptyset API clears all signal flags in the sigmask argument. The sigaddset API sets the flag corresponding to the signal_num signal in sigmask. The sigdelset API clears the flag corresponding to the signal_num signal in sigmask. The sigfillset API sets all the flags in the sigmask. The return value of the sigemptyset, sigaddset, sigdelset, and sigfillset calls is zero if the call succeed or -1 if they fail. The sigismember API returns 1 if the flag corresponding to the signal_num signal in the sigmask is set, zero if not set, and -1 if the call fails. The following example checks whether the SIGINT signal is present in a process signal mask and adds it to the mask if it is not there. Then clears the SIGSEGV signal from the process signal mask.

```
#include <stdio.h>
#include <signal.h>
int main()
{
    sigset_t sigmask;
    sigemptyset(&sigmask);           /*initialize set*/
    if (sigprocmask(0, 0, &mask) == -1) { /*get current signal mask*/
        perror("sigprocmask");
        exit(1);
    } else
        sigaddset(&sigmask, SIGINT); /*set SIGINT flag*/
    sigdelset(&sigmask, SIGSEGV); /*clear SIGSEGV flag*/
    if (sigprocmask(SIG_SETMASK, &sigmask, 0) == -1)
        perror("sigprocmask"); /*set a new signal mask*/
}
```

If there are multiple instances of the same signal pending for the process, it is implementation dependent whether one or all of those instances will be delivered to the process. A process can query which signals are pending for it via the sigpending API

```
#include <signal.h>
int sigpending(sigset_t *sigmask);
```

sigmask is assigned the set of signals pending for the calling process by the API. sigpending returns a zero if it succeeds and a -1 value if it fails. UNIX system V.3 and V.4 support the following APIs as simplified means for signal mask manipulation.

```
#include <signal.h>
int sighold(int signal_num);
int sigrelse(int signal_num);
int sigignore(int signal_num);
int sigpause(int signal_num);
```

The sighold API adds the named signal signal_num to the calling process signal mask. The sigrelse API removes the named signal signal_num to the calling process signal mask. The sigignore API sets the signal handling method for the named signal signal_num to SIG_DFT. The sigpause API removes the named signal signal_num from the calling process signal mask and suspends the process until it is interrupted by a signal.

Sigaction

The sigaction API is a replacement for the signal API in the latest UNIX and POSIX systems. The sigaction API is called by a process to set up a signal handling method for each signal it wants to deal with. sigaction API returns the previous signal handling method for a given signal. The sigaction API prototype is:

```
#include <signal.h>
int sigaction(int signal_num, struct sigaction *action, struct sigaction *old_action);
```

The struct sigaction data type is defined in the <signal.h> header as:

```
struct sigaction
{
    void          (*sa_handler)(int);
    sigset_t      sa_mask;
    int          sa_flag;
};
```

The `sa_handler` field can be set to `SIG_IGN`, `SIG_DFL`, or a user defined signal handler function. The `sa_mask` field specifies additional signals that process wishes to block when it is handling `signal_num` signal. The `signal_num` argument designates which signal handling action is defined in the `action` argument. The previous signal handling method for `signal_num` will be returned via the `old_action` argument if it is not a `NULL` pointer. If `action` argument is a `NULL` pointer, the calling process's existing signal handling method for `signal_num` will be unchanged. The following C program illustrates the use of `sigaction`:

```
#include <iostream.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void callme ( int sig_num )
{
    cout << "catch signal:" << sig_num << endl;
}

int main ( int argc, char *argv[] )
{
    sigset_t sigmask;
    struct sigaction action, old_action;

    sigemptyset(&sigmask);

    if ( sigaddset( &sigmask, SIGTERM) == -1 ||
        sigprocmask( SIG_SETMASK, &sigmask, 0) == -1)
        perror("Set signal mask");

    sigemptyset( &action.sa_mask);

    sigaddset( &action.sa_mask, SIGSEGV);
    action.sa_handler = callme;
    action.sa_flags = 0;
    if (sigaction (SIGINT, &action, &old_action) == -1)
        perror("sigaction");
    pause();          /* wait for signal interruption*/
    cout << argv[0] << "exits\n";
}
}
```

In the above example, the process signal mask is set with `SIGTERM` signal. The process then defines a signal handler for the `SIGINT` signal and also specifies that the `SIGSEGV` signal is to be blocked when the process is handling the `SIGINT` signal. The process then terminates its execution

via the pause API. The output of the above program would be as:

```
% cc sigaction.c -o sigaction
% ./sigaction &
[1] 495
% kill -INT 495
catch signal: 2
sigaction exits
[1] Done sigaction
```

The sa_flag field of the struct sigaction is used to specify special handling for certain signals. POSIX.1 defines only two values for the sa_flag: zero or SA_NOCHLDSTOP. The SA_NOCHLDSTOP flag is an integer literal defined in the <signal.h> header and can be used when signal_num is SIGCHLD. The effect of the SA_NOCHLDSTOP flag is that the kernel will generate the SIGCHLD signal to a process when its child process has terminated, but not when the child process has been stopped. If the sa_flag value is set to zero in sigaction call for SIGCHLD, the kernel will send the SIGCHLD signal to the calling process whenever its child process is either terminated or stopped. UNIX System V.4 defines additional flags for the sa_flags field. These flags can be used to specify the UNIX System V.3 style of signal handling method:

sa_flags value	Effect on handling signal_num
SA_RESETHAND	If signal_num is caught, the sa_handler is set to SIG_DFL before the signal handler function is called, and signal_num will not be added to the process signal mask when the signal handler function is executed.
SA_RESTART	If a signal is caught while a process is executing a system call, the kernel will restart the system call after the signal handler returns. If this flag is not set in the sa_flags, after the signal handler returns, the system call will be aborted with a return value of -1 and will set errno to EINTR.

The SIGCHLD Signal and the waitpid API

When a child process terminates or stops, the kernel will generate a SIGCHLD signal to its parent process. Depending upon how the parent sets up signal handling of the SIGCHLD signal, different events may occur:

1. Parent accepts the default action of the SIGCHLD signal: The SIGCHLD signal does not terminate the parent process. It affects only the parent process if it arrives at the same time the parent process is suspended by the waitpid system call. In this case, the parent process is awakened, the API will return child's exit status and process ID to the parent, and the

kernel will clear up the process table slot allocated for the child process. Thus, with this setup, a parent process can call waitpid API repeatedly to wait for each child it created.

2. Parent ignores the SIGCHLD signal: The SIGCHLD signal will be discarded, and the parent will not be disturbed, even if it is executing the waitpid system call. The effect of this setup is that if the parent calls waitpid API, the API will suspend the parent until all its child processes have terminated. Furthermore, the child process table slots will be cleared by the kernel, and the API will return -1 value to the parent process.
3. Process catches the SIGCHLD signal: The signal handler function will be called in the parent whenever the child process terminates. Furthermore, if the SIGCHLD signal arrives while the parent process is executing the waitpid system call, after the signal handler returns, the waitpid API may be restarted to collect the child exit status and clear its process table slot. On the other hand, the API may be aborted and the child process table slot not freed, depending upon the parent setup of the signal action for the SIGCHLD signal.

The sigsetjmp and siglongjmp APIs

The sigsetjmp and siglongjmp APIs have similar functions as their corresponding setjmp and longjmp APIs. The sigsetjmp and siglongjmp APIs are defined in POSIX.1 and on most UNIX systems that support signal mask. The function prototypes of the APIs are:

```
#include <setjmp.h>
int sigsetjmp ( sigjmpbuf env, int save_sigmask );
int siglongjmp ( sigjmpbuf env, int ret_val );
```

The sigsetjmp and siglongjmp are created to support signal mask processing. Specifically, it is implementation dependent on whether a process signal mask is saved and restored when it invokes the setjmp and longjmp APIs respectively. The sigsetjmp API behaves similarly to the setjmp API, except that it has a second argument, save_sigmask, which allows a user to specify whether a calling process signal mask should be saved to the provided env argument. If the save_sigmask argument is nonzero, the caller's signal mask is saved, else signal mask is not saved. The siglongjmp API does all operations as the longjmp API, but it also restores a calling process signal mask if the mask was saved in its env argument. The ret_val argument specifies the return value of the corresponding sigsetjmp API when called by siglongjmp API. Its value should be nonzero

number, and if it is zero the siglongjmp API will reset it to 1. The siglongjmp API is usually called from user-defined signal handling functions. This is because a process signal mask is modified when a signal handler is called, and siglongjmp should be called to ensure that the process signal mask is restored properly when “jumping out” from a signal handling function. The following C program illustrates the use of sigsetjmp and siglongjmp APIs.

```
#include <iostream.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <setjmp.h>

sigjmp_buf env;

void callme ( int sig_num )
{
    cout << "catch signal:" << sig_num << endl;
    siglongjmp ( env, 2 );
}

int main ( int argc, char *argv[] )
{
    sigset_t sigmask;
    struct sigaction action, old_action;
    sigemptyset(&sigmask);

    if ( sigaddset( &sigmask, SIGTERM) == -1 ||
        sigprocmask( SIG_SETMASK, &sigmask, 0) == -1)
        perror("Set signal mask");

    sigemptyset( &action.sa_mask);
    sigaddset( &action.sa_mask, SIGSEGV);

    action.sa_handler = (void (*)())callme;
    action.sa_flags = 0;

    if(sigaction (SIGINT, &action, &old_action) == -1)
        perror("sigaction");

    if(sigsetjmp( env, 1) != 0 ) {
        cerr << "Return from signal interruption\n";
        return 0;
    } else
        cerr << "Return from first time sigsetjmp is called\n";

    pause();    /* wait for signal interruption*/
}
```

The program sets its signal mask to contain SIGTERM, and then sets up a signal trap for the SIGINT signal. The program then calls sigsetjmp to store its code location in the env global variable. Note the sigsetjmp call returns a zero value when directly called in user program and not via siglongjmp. The program suspends its execution via the pause API. When ever the user interrupts the process from the keyboard, the callme function is called. The callme function calls siglongjmp API to transfer flow back to the sigsetjmp function in main, which now returns a 2 value. The sample output of the above program is:

```
% cc sigsetjmp.c
% ./a.out &

[1] 377
Return from first time sigsetjmp is called
% kill -INT 377
catch signal: 2
Return from signal interruption
[1] Done a.out
%
```

kill

A process can send signal to a related process via the kill API. This is a simple means of IPC or control. The sender and recipient processes must be related such that either sender process real or effective user ID matches that of the recipient process, or the sender has su privileges. For example, a parent and child process can send signals to each other via the kill API. The kill API is defined in most UNIX system and is a POSIX.1 standard. The function prototype is as:

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

The sig_num argument is the integer value of a signal to be sent to one or more processes designated by pid. The possible values of pid and its use by the kill API are:

pid value	Effects on the kill API
A positive value	pid is a process ID. Sends signal_num to that process.
0	Sends signal_num to all processes whose process group ID is the same as the calling process.
-1	Sends signal_num to all processes whose real user ID is the same as the effective user ID of the calling process. If the calling process effective user ID is su user ID, signal_num will be sent to all processes in the system (except processes - 0 and 1). The later case is used when the system is shutting down - kernel calls the kill API to terminate all processes except 0 and 1. Note: POSIX.1 does not specify the behavior of the kill API when the pid value is -1. This effect is for UNIX systems only.
A negative value	Sends signal_num to all processes whose process group ID matches the absolute value of pid.

The return value of kill is zero if it succeeds or -1 if it fails.

alarm

The alarm API can be called by a process to request the kernel to send the SIGALRM signal after a certain number of real clock seconds. The alarm API is defined in most UNIX systems and is a POSIX.1 standard. The function prototype of the API is as:

```
#include <signal.h>
unsigned int alarm ( unsigned int time_interval );
```

The time_interval argument is the number of CPU seconds elapse time, after which the kernel will send the SIGALRM signal to the calling process. If a time_interval value is zero, it turns off the alarm clock. The return value of the alarm API is the number of CPU seconds left in the process timer, as set by a previous alarm system call. The effect of the previous alarm API call is canceled, and the with new alarm call. process timer is reset. A process alarm clock is not passed on to its forked child process, but an exec'ed process retains the same alarm clock value as was prior to the exec API call. The alarm API can be used to implement the sleep API.

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void wakeup() {}

unsigned int sleep ( unsigned int timer )
{
    struct sigaction action;

    action.sa_handler = wakeup;
    action.sa_flags = 0;

    sigemptyset ( &action.sa_mask );

    if ( sigaction (SIGALRM, &action, 0) == -1 ) {
        perror("sigaction");
        return -1;
    }
    (void)alarm( timer );
    (void)pause( );
}
```

The sleep function above sets up a signal handler for the SIGALRM, calls the alarm API to request the kernel to send the SIGALRM signal after the timer interval, and finally, suspends its execution via the pause system call. The wakeup signal handler function is called when the SIGALRM signal is sent to the process. When it returns, the pause system call will be aborted, and the calling process will return from the sleep function. BSD UNIX defines the ualarm function, which is the same as the alarm API except that the argument and return value of the ualarm function are in microsecond units.

Interval Timers

The use of the alarm API is to set up a interval timer in a process. The interval timer can be used to schedule a process to do some tasks at fixed time interval, to time the execution of some operations, or limit the time allowed for the execution of some tasks. The following program illustrates how to set up a real-time clock interval timer using the alarm API.

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

#define INTERVAL 5
void callme ( int sig_no )
{
    alarm( INTERVAL );
    /* do scheduled tasks */
}
```

```
int main ( )
{
    struct sigaction action;

    sigemptyset( &action.sa_mask);
    action.sa_handler = (void (*)( ))callme;
    action.sa_flags = SA_RESTART;

    if ( sigaction( SIGALRM, &action, 0) == -1) {
        perror("sigaction");
        return 1;
    }
    if ( alarm( INTERVAL ) == -1 )
        perror("alarm");
    else
        while (1) {
            /* do normal operation */
        }
    return 0;
}
```

The sigaction API is called to set up callme as the signal handling function for the SIGALRM signal. The program then invokes real clock seconds. het alarm API to send itself het SIGALRM signal after 5. The program then goes off to perform its normal operation in an infinite loop. When the timer expires, the callme function is invoked, which restarts the alarm clock for another 5 seconds and then does the scheduled tasks. When the callme function returns, the program continues its “normal” operation until another timer expiration. BSD UNIX invented the setitimer API, which provides capabilities additional to those of the alarm API. The setitimer resolution time is in microseconds, whereas the resolution time for alarm is in seconds. The alarm API can be used to set up real-time clock timer per process. The setitimer API can be used to define up to three different types of timers in a process:

1. Real time clock timer.
2. Timer based on the user time spent by a process
3. Timer based on the total user and system times spent by a process.

The getitimer API is also defined in BSD and System V UNIX for users to query the timer values that are set by the setitimer API. The setitimer and getitimer function prototypes are:

```
#include <sys/time.h>
int setitimer( int which, const struct itimerval *val, struct itimerval *old );
int getitimer( int which, struct itimerval *old );
```

The which argument specify which timer to process, the possible values are:

which argument value	Timer type
ITIMER_REAL	Timer based on real-time clock. Generates a SIGALRM signal when expires
ITIMER_VIRTUAL	Timer based on user-time spent by a process. Generates a SIGVTALRM signal when it expires
ITIMER_PROF	Timer based on total user and system times spent by a process. Generates a SIGPROF signal when it expires

The struct itimerval data type is defined in the <sys/time.h> header as:

```
struct itimerval
{
    struct timerval it_interval;    //timer interval
    struct timerval it_value;      //current value
}
```

For setitimer API, the val.it_value is the time to set the named timer, and the val.it_interval is the time to reload the timer when it expires. The val.it_interval may be set to zero if the timer is to run only once and if the val.it_value is set to zero, it stops the named timer if it is running. For getitimer API, old.it_value and the old.it_interval return the named timer’s remaining time and the reload time, respectively. The old argument of the setitimer API is like the old argument of the getitimer API. If this is an address of a struct itimerval typed variable, it returns the previous timer value, if set to NULL the old timer value will not be returned. The ITIMER_VIRTUAL and ITIMER_PROF timers are primary useful in timing the total execution time of selected user functions, as the timer runs only while the user process is running or the kernel is executing system functions on behalf of the user process for the ITIMER_PROF timer. Both the APIs return zero on success or -1 value if they fail.

POSIX.1b Timers

POSIX.1b defines a set of APIs for interval timer manipulation. The POSIX.1b timers are more flexible and powerful than UNIX timers in following ways:

1. Users may define multiple independent timers per system clock.
2. The timer resolution is in nanoseconds.
3. Users may specify, on a timer basis, the signal to be raised when a timer expires.
4. The timer interval may be specified as either an absolute or a relative time

There is a limit on how many POSIX timers can be created per process, this is `TIMER_MAX` constant defined in `<limits.h>` header.

POSIX timers created by a process are not inherited by its child process, but are retained across the `exec` system call. A POSIX.1 timer does not use the `SIGALRM` signal when it expires, it can be used safely with the `sleep` API in the same program. The POSIX.1b APIs for timer manipulation are:

```
#include <signal.h>
#include <time.h>

int timer_create(clockid_t clock, struct sigevent *spec, timer_t *timer_hdrp);
int timer_settime(timer_t timer_hdr, int flag, struct itimerspec *val, struct itimerspec *old);
int timer_gettime(timer_t timer_hdr, struct itimerspec *old);
int timer_getoverrun(timer_t timer_hdr);
int timer_delete(timer_t timer_hdr);
```

The `timer_create` API is used to dynamically create a timer and returns its handler. The `clock` argument specifies which system clock would be the new timer based on, its value may be `CLOCK_REALTIME` for creating a real time clock timer – this defined by POSIX.1b – other values are system dependent. The `spec` argument defines what action to take when the timer expires. The `struct sigevent` data type is defined as:

```
struct sigevent
{
    int          sigev_notify;
    int          sigev_signo;
    union sigval sigev_value;
};
```

The `sigev_signo` field specifies a signal number to be raised at the timer expiration. Its valid only when the `sigev_notify` field is set to `SIGEV_SIGNAL`. If `sigev_notify` field is set to `SIGEV_NONE`, no signal is raised by the timer when it expires. Because multiple timers may generate the same signal, the `sigev_value` field is used to contain any user defined data to identify that a signal is raised by a specific timer.

Daemon Process

Introduction

Daemons are processes that live for a long time. They are often started when the system is bootstrapped and terminate only when the system is shut down. They do not have a controlling terminal; so, we say that they run in the background. UNIX systems have numerous daemons that perform day-to-day activities. Here we look at the process structure of daemons and how to write

a daemon. Since a daemon does not have a controlling terminal, we need to see how a daemon can report error conditions when something goes wrong.

We look at some common system daemons and how they relate to the concepts of process groups, controlling terminals, and sessions. The `ps` command prints the status of various processes in the system. We will execute: `ps -axj` under BSD UNIX. The `-a` option shows the status of processes owned by others, and `-x` shows processes that do not have a controlling terminal. The `-j` option displays the job-related information: the session ID, process group ID, controlling terminal, and terminal process group ID. Under System V based systems, a similar command is `ps -efjc`. The output from `ps` looks like-

PPID	PID	PGID	SID	TTY	TPGID	UID	COMMAND
0	1	0	0	?	-1	0	init
1	2	1	1	?	-1	0	[keventd]
1	3	1	1	?	-1	0	[kapmd]
0	5	1	1	?	-1	0	[kswapd]
0	6	1	1	?	-1	0	[bdf flush]
0	7	1	1	?	-1	0	[kupdated]
1	1009	1009	1009	?	-1	32	portmap
1	1048	1048	1048	?	-1	0	syslogd -m 0
1	1335	1335	1335	?	-1	0	xinetd -pidfile /var/run/xinetd.pid
1	1403	1	1	?	-1	0	[nfsd]
1	1405	1	1	?	-1	0	[lockd]
1405	1406	1	1	?	-1	0	[rpciod]
1	1853	1853	1853	?	-1	0	crond
1	2182	2182	2182	?	-1	0	/usr/sbin/cupsd

The system processes depend on the operating system implementation. Anything with a parent process ID of 0 is usually a kernel process started as part of the system bootstrap procedure. (An exception to this is `init`, since it is a user-level command started by the kernel at boot time.). Kernel processes are special and generally exist for the entire lifetime of the system. They run with superuser privileges and have no controlling terminal and no command line. Process 1 is usually `init`, is a system daemon responsible for, among other things, starting system services specific to various run levels. These services are usually implemented with the help of their own daemons. On Linux, the `kevenTD` daemon provides process context for running scheduled functions in the kernel. The `kapmd` daemon provides support for the advanced power management features

available with various computer systems. The kswapd daemon is also known as the pageout daemon. It supports the virtual memory subsystem by writing dirty pages to disk slowly over time. The Linux kernel flushes cached data to disk using two additional daemons: bdflood and kupdated. The syslogd daemon is available to any program to log system messages for an operator. The messages may be printed on a console device and also written to a file. The inetd daemon (xinetd) listens on the system's network interfaces for incoming requests for various network servers. The nfsd, lockd, and rpciod daemons provide support for the Network File System (NFS). The cron daemon (crond) executes commands at specified dates and times. Numerous system administration tasks are handled by having programs executed regularly by cron. The cupsd daemon is a print spooler; it handles print requests on the system. The kernel daemons are started without a controlling terminal. The lack of a controlling terminal in the user-level daemons is probably the result of the daemons having called setsid. All the user-level daemons are process group leaders and session leaders and are the only processes in their process group and session. Finally, note that the parent of most of these daemons is the init process.

Coding Rules

Some basic rules to coding a daemon prevent unwanted interactions from happening. We state these rules and then show a function, `daemonize`, that implements them.

1. The first thing to do is call `umask` to set the file mode creation mask to 0. The file mode creation mask that is inherited could be set to deny certain permissions.
2. Call `fork` and have the parent exit. This does several things.
First, if the daemon was started as a simple shell command, having the parent terminate makes the shell think that the command is done.
Second, the child inherits the process group ID of the parent but gets a new process ID, so we are guaranteed that the child is not a process group leader. This is a prerequisite for the call to `setsid` that is done next.
3. Call `setsid` to create a new session. Three steps occur. The process becomes a session leader of a new session, becomes the process group leader of a new process group has no controlling terminal.

4. Change the current working directory to the root directory. The current working directory inherited from the parent could be on a mounted file system. Since daemons normally exist until the system is rebooted, if the daemon stays on a mounted file system, that file system cannot be unmounted.
5. Unneeded file descriptors should be closed. This prevents the daemon from holding open any descriptors that it may have inherited from its parent.
6. Some daemons open file descriptors 0, 1, and 2 to /dev/null so that any library routines that try to read from standard input or write to standard output or standard error will have no effect. Since the daemon is not associated with a terminal device, there is nowhere for output to be displayed; nor is there anywhere to receive input from an interactive user. Even if the daemon was started from an interactive session, the daemon runs in the background, and the login session can terminate without affecting the daemon. If other users log in on the same terminal device, we wouldn't want output from the daemon showing up on the terminal, and the users wouldn't expect their input to be read by the daemon.

```
#include "apue.h"
#include <syslog.h>
#include <fcntl.h>
#include <sys/resource.h>

void
daemonize(const char *cmd)
{
    int i, fd0, fd1, fd2;
    pid_t pid;
    struct rlimit rl;
    struct sigaction sa;

    /*
     * Clear file creation mask.
     */
```

```
umask(0);

/*
 * Get maximum number of file descriptors.
 */
if (getrlimit(RLIMIT_NOFILE, &rl) < 0)
    err_quit("%s: can't get file limit", cmd);

/*
 * Become a session leader to lose controlling TTY.
 */
if ((pid = fork()) < 0)
    err_quit("%s: can't fork", cmd);
else if (pid != 0) /* parent */
    exit(0);
setsid();

/*
 * Ensure future opens won't allocate controlling TTYS.
 */
sa.sa_handler = SIG_IGN;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
if (sigaction(SIGHUP, &sa, NULL) < 0)
    err_quit("%s: can't ignore SIGHUP", cmd);
if ((pid = fork()) < 0)
    err_quit("%s: can't fork", cmd);
else if (pid != 0) /* parent */
    exit(0);

/*
```

```
* Change the current working directory to the root so
* we won't prevent file systems from being unmounted.
*/
if (chdir("/") < 0)
    err_quit("%s: can't change directory to /", cmd);

/*
* Close all open file descriptors.
*/
if (rl.rlim_max == RLIM_INFINITY)
    rl.rlim_max = 1024;
for (i = 0; i < rl.rlim_max; i++)
    close(i);

/*
* Attach file descriptors 0, 1, and 2 to /dev/null.
*/
fd0 = open("/dev/null", O_RDWR);
fd1 = dup(0);
fd2 = dup(0);

/*
* Initialize the log file.
*/
openlog(cmd, LOG_CONS, LOG_DAEMON);
if (fd0 != 0 || fd1 != 1 || fd2 != 2) {
    syslog(LOG_ERR, "unexpected file descriptors %d %d %d",
        fd0, fd1, fd2);
    exit(1);
}
}
```

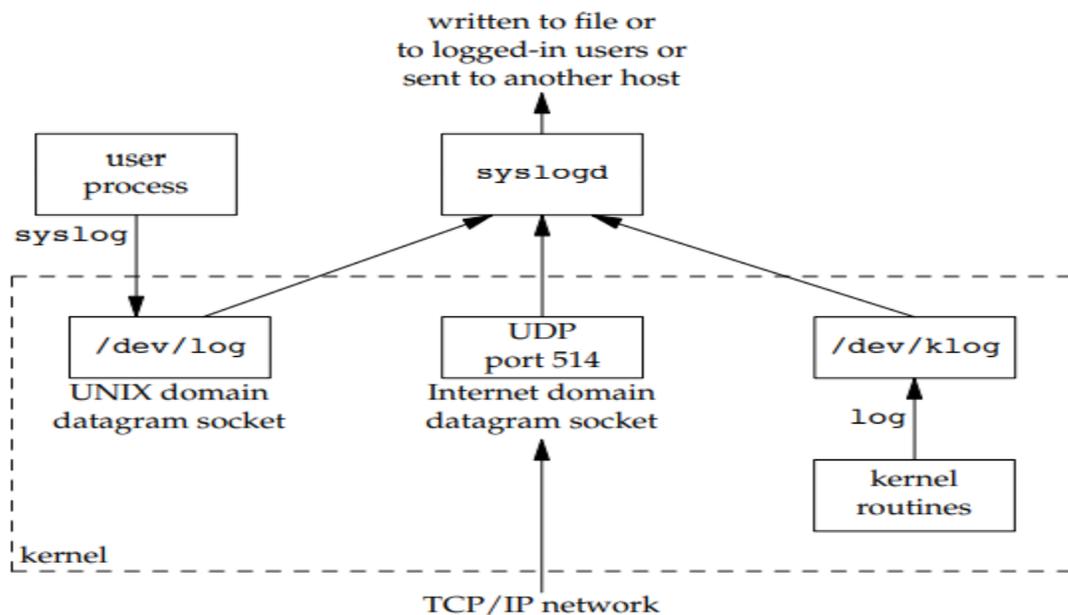
If the daemonize function is called from a main program that then goes to sleep, we can check the status of the daemon with the ps command:

```

$ ./a.out
$ ps -efj
UID PID PPID PGID SID TTY CMD
sar 13800 1 13799 13799 ? ./a.out
$ ps -efj | grep 13799
sar 13800 1 13799 13799 ? ./a.out
    
```

Error Logging

One problem a daemon has is how to handle error messages. It can not simply write to standard error, since it should not have a controlling terminal. The BSD syslog facility is in 4.2BSD and most systems derived from BSD support syslog. The syslog function is included as an XSI extension in the Single UNIX Specification. The BSD syslog facility is used by most daemons. Figure below illustrates its structure.



There are three ways to generate log messages:

1. Kernel routines can call the log function. These messages can be read by any user process that opens and reads the /dev/klog device.
2. Most user processes (daemons) call the syslog function to generate log messages. This causes the message to be sent to the UNIX domain datagram socket /dev/log.
3. A user process on this host, or on some other host that is connected to this host by a TCP/IP network, can send log messages to UDP port 514. Note that the syslog function never generates these UDP datagrams: they require explicit network programming by the process generating the log message.

Normally, the syslogd daemon reads all three forms of log messages. On start-up, this daemon reads a configuration file, usually /etc/syslog.conf, which determines where different classes of messages are to be sent. For example, urgent messages can be sent to the system administrator (if logged in) and printed on the console, whereas warnings may be logged to a file. Our interface to this facility is through the syslog function.

```
#include <syslog.h>
void openlog(const char *ident, int option, int facility);
void syslog(int priority, const char *format, ...);
void closelog(void);
int setlogmask(int maskpri);
```

Calling openlog is optional. If it's not called, the first time syslog is called, openlog is called automatically. Calling closelog is also optional—it just closes the descriptor that was being used to communicate with the syslogd daemon. Calling openlog lets us specify an ident that is added to each log message. This is normally the name of the program (cron, inetd, etc.). The option argument is a bitmask specifying various options. The available options are as follows:

option	Description
LOG_CONS	If the log message can't be sent to syslogd via the UNIX domain datagram, the message is written to the console instead.
LOG_NDELAY	Open the UNIX domain datagram socket to the syslogd daemon immediately; don't wait until the first message is logged. Normally, the
	socket is not opened until the first message is logged.
LOG_NOWAIT	Do not wait for child processes that might have been created in the process of logging the message. This prevents conflicts with applications that catch SIGCHLD, since the application might have retrieved the child's status by the time that syslog calls wait.

Daemon Conventions

Several common conventions are followed by daemons in the UNIX System.

1. If the daemon uses a lock file, the file is usually stored in `/var/run`. Note, however, that the daemon might need superuser permissions to create a file here. The name of the file is usually `name.pid`, where `name` is the name of the daemon or the service. For example, the name of the cron daemon's lock file is `/var/run/crond.pid`.
2. If the daemon supports configuration options, they are usually stored in `/etc`. The configuration file is named `name.conf`, where `name` is the name of the daemon or the name of the service. For example, the configuration for the `syslogd` daemon is `/etc/syslog.conf`.
3. Daemons can be started from the command line, but they are usually started from one of the system initialization scripts (`/etc/rc*` or `/etc/init.d/*`). If the daemon should be restarted automatically when it exits, we can arrange for `init` to restart it if we include a `respawn` entry for it in `/etc/inittab`.
4. If a daemon has a configuration file, the daemon reads it when it starts, but usually won't look at it again. If an administrator changes the configuration, the daemon would need to be stopped and restarted to account for the configuration changes. To avoid this, some daemons will catch `SIGHUP` and reread their configuration files when they receive the signal. Since they aren't associated with terminals and are either session leaders without

controlling terminals or members of orphaned process groups, daemons have no reason to expect to receive SIGHUP. Thus, they can safely reuse it.

ClientServer Model

A common use for a daemon process is as a server process. We can call the syslogd process a server that has messages sent to it by user processes (clients) using a UNIX domain datagram socket. In general, a server is a process that waits for a client to contact it, requesting some type of service. The service being provided by the syslogd server is the logging of an error message.

Question Bank

Module1

1. Briefly explain the concept of Unix architecture.
2. Explain features of Unix.
3. Describe the command arguments and options.
4. Explain the basic Unix commands such as – echo, printf, ls, who, date, passwd, cal.
5. Explain the combining commands.
6. Briefly explain the concept of ‘type’ command.
7. List and explain the file types?
8. Describe the parent child relationship in unix files
9. Describe the relative and absolute pathname
10. Explain the basic directory commands such as - pwd, cd, mkdir, rmdir.
11. Explain the file related commands such as – cat, mv, rm, cp,wc and od.
12. Explain the posix and single unix specification.
13. Write a short notes on internal and external commands
14. Explain the structure of organization of files.

Module2

1. Explain the ls command and options with examples
2. Describe the relative and absolute file permissions changing methods
3. Give the syntax and example for recursively changing file permissions
4. Explain the directory permissions
5. What is wildcard? List and explain the metacharacters in wildcard with examples
6. Explain the two solution for removing the special meanings of wildcards
7. Name and explain in three standard files and redirection
8. Explain the concept of Pipes with an example
9. Describe the BRE and ERE concepts with example
10. Write a short notes on: a) grep b) egrep
11. Give an explanation on ordinary and environment variable
12. Discuss the following concepts in shell programming-: a) The .Profile b) Read cmd
c) Read-only cmd
13. Describe the command line arguments
14. Write a short note on exit and exit status of command
15. Explain the logical operators for conditional execution
16. Discuss the concept the test command and its shortcut
17. Explain the control statements in detail
18. Write a short notes on a) set and shift commands b) handling positional parameters
c) trap command

Module3

1. List and explain the general file APIs
2. Describe the file and record locking
3. Explain the directory file APIs with an example
4. Explain the device file APIs with an example
5. Explain the FIFO files APIs with example
6. Describe the symbolic link file APIs
7. Explain the unix process data structure with a neat diagram
8. Discuss the following concept in process APIs a) fork, vfork b) _ exit c)wait,waitpid d)exec e)pipe f) I/O direction g) wait3 wait4 Functions h) Race functions e) exec functions
9. Explain the concept of process termination.
10. Write a short notes: a) command-line arguments b) Environment list
11. Explain the memory layout of c program with a block diagram
12. What is memory allocation? Explain the three functions of memory allocation
13. List and explain the alternate memory allocators
14. Explain setjmp and longjmp function with examples

Module4

1. Explain the concept of changing user id and group id
2. Explain the interpreter files in detail
3. What is process accounting? Explain in detail
4. Describe user identification
5. Explain the pipes with neat diagram and example
6. Describe the popen and pclose function with example
7. Describe co-process
8. What is FIFO? Explain with example\
9. Explain the 3 types of IPC
10. Write a short note message queues?
11. What are semaphores? Explain in detail
12. Describe the client-server properties
13. Write a short note on stream pipes
14. Discuss the concept of passing file descriptors
15. Explain client server connection function with an example

Module5

1. Explain the signal concepts
2. Describe the unix system signals
3. Explain the signal features of the unix system
4. Write a short on: a) kill function b) alarm function c) sigmask d) sigaction e) sigsetjmp and siglongjmp
5. Write the daemon characteristics
6. List and state the coding rules?
7. Write a function that can be called from a program that wants to initialize itself as a daemon
8. Describe the error logging with a neat diagram
9. Write a short note on client server model

Thankyou

ALL THE BEST FOR EXAM

Ranjitha J

Assistant Professor

Dept of ISE

ATRIA, Bangalore