V CS IS
2021-08-25

Pooja S Marakala

MODULE-1

Topics

Introduction: Unix Components/Architecture. Features of Unix. The UNIX Environment and UNIX Structure, Posix and Single Unix specification. General features of Unix commands/ command structure. Command arguments and options. Basic Unix commands such as echo, printf, ls, who, date, passwd, cal, Combining commands. Meaning of Internal and external commands. The type command: knowing the type of a command and locating it. The root login. Becoming the super user: su command.

Unix files: Naming files. Basic file types/categories. Organization of files. Hidden files. Standard directories. Parent child relationship. The home directory and the HOME variable. Reaching required files- the PATH variable, manipulating the PATH, Relative and absolute pathnames. Directory commands – pwd, cd, mkdir, rmdir commands. The dot (.) and double dots (..) notations to represent present and parent directories and their usage in relative path names. File related commands – cat, mv, rm, cp, wc and od commands.

CHAPTER 1 INTRODUCTION

1.1 WHAT IS AN OPERATING SYSTEM (OS)?

An operating system (OS) is an interface between hardware and user. It manages hardware and software resource. It takes the form of a set of software routines that allow users and application programs to access system resources (e.g. the CPU, memory, disks, modems, printers, network cards etc.) in a safe, efficient and abstract way.

For example, an OS ensures safe access to a printer by allowing only one application program to send data directly to the printer at any one time. An OS encourages efficient use of the CPU by suspending programs that are waiting for I/O operations to complete to make way for programs that can use the CPU more productively. An OS also provides convenient abstractions (such as files rather than disk locations) which isolate application programmers and users from the details of the underlying hardware. UNIX Operating system allows complex tasks to be performed with a few keystrokes. It doesn't tell or warn the user about the consequences of the command.

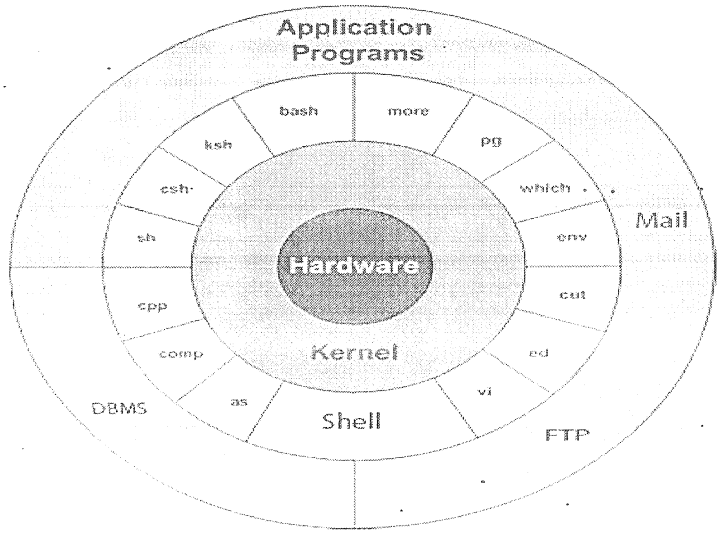
1.2 BRIEF HISTORY

- In the late 1960s, researchers from General Electric, MIT and Bell Labs launched a joint project to develop an ambitious multi-user, multi-tasking OS for mainframe computers known as MULTICS (Multiplexed Information and Computing System). MULTICS failed, but it did inspire Ken Thompson, who was a researcher at Bell Labs, to have a go at writing a simpler operating system himself.
- He wrote a simpler version of MULTICS on a PDP7 in assembler and called his attempt UNICS (Uniplexed Information and Computing System). Because memory and CPU power were at a premium in those days, UNICS (eventually shortened to UNIX) used short commands to minimize the space needed to store them and the time needed to decode them - hence the tradition of short.
- The limitation of UNICS was not portable. In order to overcome the limitation, Ken Thompson started to work on the development of system using higher level language called B Language. As B language did not yield expected results, Dennis Ritchie developed higher level language called C.

SRI SHIVA XEROX
#147, RPS Tower, Opp. JSS College,
Below UCO Bank, Dr. Vishnuvardhan Road,
Reflex Layout, Bangalore - 560 060.
Ph: 9620099557, 7676670591.

- Ken Thompson then teamed up with Dennis Ritchie, the author of the first C compiler in 1973.
- They rewrote the UNIX kernel in C - this was a big step forwards in terms of the system's portability - and released the fifth Edition of UNIX to universities in 1974.
- The Seventh Edition, released in 1978, marked a split in UNIX development into two main branches: SYSV (System 5) and BSD (Berkeley Software Distribution).
- SYSV was developed by AT&T and other commercial companies. UNIX flavors based on SYSV have traditionally been more conservative, but better supported than BSD-based flavors.
- Until recently, UNIX standards were nearly as numerous as its variants. In early days, AT&T published a document called System V Interface Definition (SVID). X/OPEN (now The Open Group), a consortium of vendors and users, had one too, in the X/Open Portability Guide (XPG).
- In the US, yet another set of standards, named Portable Operating System Interface for Computer Environments (POSIX), were developed at the Institution of Electrical and Electronics Engineers (IEEE).
- In 1998, X/OPEN and IEEE undertook an ambitious program of unifying the two standards. In 2001, this joint initiative resulted in a single specification called the Single UNIX Specification, Version 3 (SUSV3), that is also known as IEEE 1003.1:2001 (POSIX.1). In 2002, the International Organization for Standardization (ISO) approved SUSV3 and IEEE 1003.1:2001.(POSIX 2)

1.3 UNIX COMPONENTS/ARCHITECTURE



- **Kernel:** is the core of operating system. A collection of routines mostly written in C. It is loaded into memory when the system is booted and communicates directly with the hardware. User programs that need to access the hardware use the services of the kernel which performs the job on the user's behalf. These programs access the kernel through a set of functions called system calls. The kernel manages system memory, processes, decides priorities.
- **Shell:** interface between Kernel and User. It functions as **command interpreter** i.e it receives and interprets the command from user and interacts with the hardware. There is only one

kernel running on the system, there could be several shells in action - one for each user who is logged in. When user enters a command, shell thoroughly examines the keyboard input and simplifies to a command line, and communicates with kernel to see that the command is executed.

Eg. $echo Sun Solaris

Sun Solaris //ignores all spaces in the above command line.

- **Files and Process**: File is an array of bytes and it contain virtually anything. Unix considers even the directories and devices as members of file system. The dominant file type is text and behavior of system is mainly controlled by text files.
- The second entity is the process, which is the name given to a file when it is executed as a program. Process is simply a "time image" of an executable file.
- **System Calls**: Though there are thousands of commands in the Unix system, they all use a handful of functions called system calls. User programs that need to access the hardware use the services of the kernel, which performs the job on user's behalf. These programs access the kernel through a set of functions called system calls.
- Ex: open()-- system call to access both file and device. Write()—system call to write a file.

1.4 FEATURES OF UNIX

Several features of UNIX have made it popular. Some of them are:

- **Portable:** UNIX can be installed on many hardware platforms. Its widespread use can be traced to the decision to develop it using the C language. Because C programs are easily moved from one hardware environment to another, it is relatively simple to port it to different environments.

- **A Multiuser & Multitasking system:** It is a multiprogramming system. It permits multiple programs to run & compete for the attention of the CPU. This can happen in two ways:.
  - **Multiuser system**: Multiple users can run separate job. In unix, the resources are actually shared between all users. The computer breaks up a unit n terminate them time expires, the previous job is kept in waiting, & the next user's job is.
  - **Multitasking system**: A single user can run multiple job. It's usual for a user to edit a file, print another file, send email etc. without leaving any applications. The kernel is designed to handle a user's multiple needs. In this environment, a user sees one job running in the foreground, & the rest running at the background. We can switch jobs between background & foreground, suspend.

- **The Building-block approach:** Unix comes with hundreds of simple commands which perform one simple job. It's through pipes & filters unix implement small-is-beautiful philosophy. Using this feature, the output of one tool can be used as input of another tool.
  Ex: $ls | wc. Here the output of ls command is given as input to another command wc.

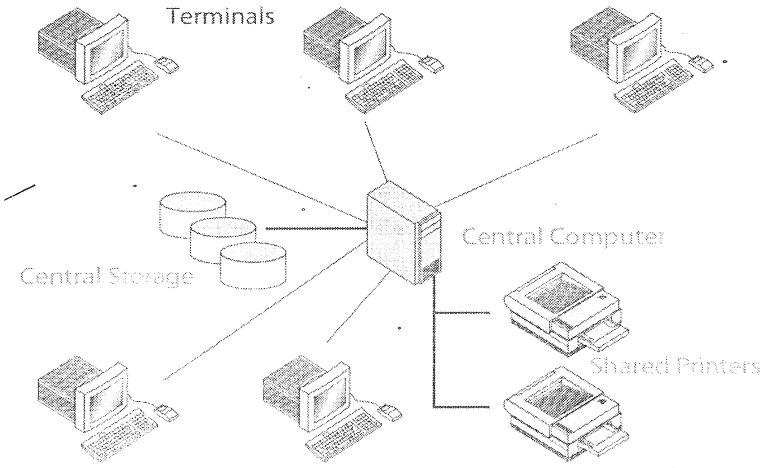
- **Unix toolkit:** Unix supplied with a set of applications such as compilers, interpreters, text-manipulation utilities, general purpose tools, & system administration tools. This set is constantly changing with every Unix release. The shell & utilities form part of POSIX specification. We must be able to download these tools & configure them to run on our machine.

- **Pattern Matching:** Unix has very strong pattern matching features. The * (known as metacharacter) is a special character used by the system to indicate that it can match a number of filenames. Ex: $ls *.c There are many more such special symbols in unix system. Some of the most advanced and useful tools use a special expression called regular expression that is framed with characters from this set.
- **Programming facility:** The Unix shell is a programming language. It has all the necessary features of a language like control structures, loops & variables. This makes it a powerful programming language. These features are used to design shell scripts, the programs that can also invoke the unix commands.
- **Documentation:** The principal online help facility is available is the man command in unix, which remains the most important reference for commands and their configuration files.

1.5 THE UNIX ENVIRONMENTS AND UNIX STRUCTURE

There 3 different environments in UNIX

- Personal environment
- Timesharing environment: Many users connected to one computer
- Client/server environment Computing split between a central computer (server) and users' computers (clients)

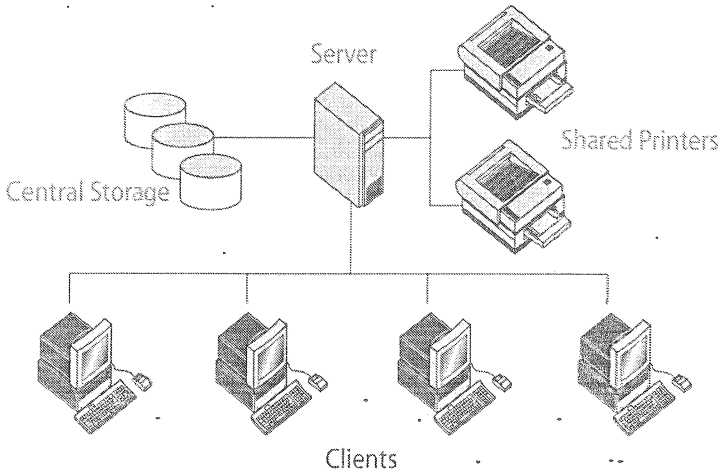


- **Personal environment:** Originally Unix designed as a multiuser environment, many users are installed UNIX on their personal computers this tends to personal Unix system environment.
- **Timesharing environment:** In time-sharing environment, many users are connected to one or more computers. The output devices (such as printers) and auxiliary storage devices (such as disks) are shared by all users. In this environment all of the computing must be done by the central computer. The central computer has several responsibilities
  - It must control the shared resources.
  - It must manage the shared data
  - It must manage the printing data and resource.
  - It must handle the computing all task

All of this work ten ds to keep the central computer busy, so user has to wait more time for get done their work so, it is non-productive because of slow response.

GANESH XEROX RNSIT

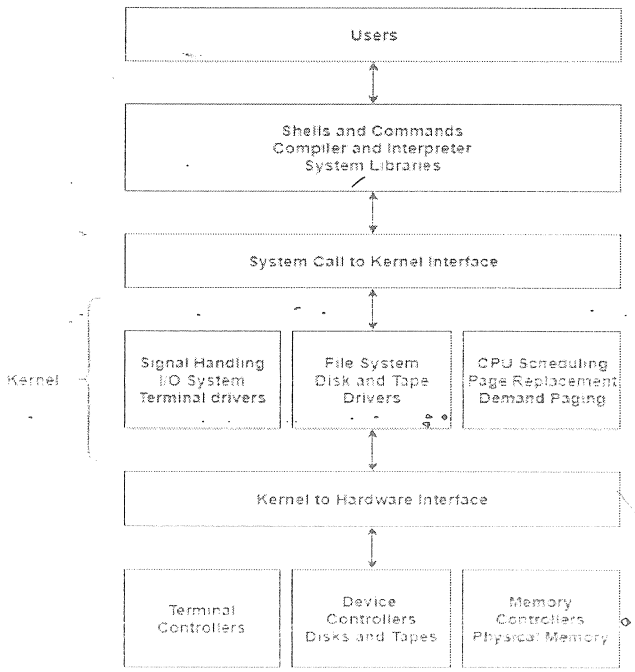
➢ **The Client/Server Environment**



- This splits the computing function between a central computer and user's computers.
- User computer are personal computer or workstation central computer assigned to the workstations. In client server environment the user's computer or workstation is called client and central computer is called server.
- The central computer, which may be a powerful microcomputer a minicomputer, a central mainframe system is known as server.
- Since work is shared between user's computer and the central computer, response time and monitor display are faster and users are more productive.

UNIX STRUCTURE

Unix consists of four major components: the kernel, the shell, a standard set of utilities and application programs. The below fig shows the Unix Structure.



SRI SHIVA XEROX
#147, KPS Tower, Opp. JSS College,
Below ICICI Bank, Mahalakshmi Road,
Rajiv Layout, Bangalore - 560 086.
Ph: 32205357, 7676710511.

- **Kernel:** is the heart of UNIX system. It contains two basic parts of the OS: process control and resource management. All other components of the system call on the kernel to perform these services for them.

- **Shell:** is an interface between Kernel and User. It functions as **command interpreter** i.e it receives and interprets the command from user and interacts with the hardware. There is only one kernel running on the system, there could be several shells in action- one for each user who is logged in. Shell has two major parts.
  a. Interpreter: reads user commands and works with the kernel to execute them.
  b. Shell Programming: is a programming capability that allows user to write a shell script.

A shell script is a file that contains the shell commands that perform a useful function. It is also known as shell program.

There are standard shells used in UNIX today.

1. The Bourne Shell

The Bourne shell (sh), written by Steve Bourne at AT&T Bell Labs, is the original UNIX shell. It is the preferred shell for shell programming because of its compactness and speed. A Bourne shell drawback is that it lacks features for interactive use, such as the ability to recall previous commands (history). The Bourne shell also lacks built-in arithmetic and logical expression handling.

The Bourne shell is the Solaris OS default shell. It is the standard shell for Solaris system administration scripts. For the Bourne shell the:

- Command full-path name is /bin/sh and /sbin/sh.
- Non-root user default prompt is $.
- Root user default prompt is #.

2. The C Shell

- Is a UNIX enhancement written by Bill Joy at the University of California at Berkeley.
- Incorporated features for interactive use, such as aliases and command history.
- Includes convenient programming features, such as built-in arithmetic and a C-like expression syntax.

For the C shell the:

- Command full-path name is /bin/csh.
- Non-root user default prompt is hostname %.
- Root user default prompt is hostname #.

3. The Korn Shell(ksh)

- Was written by David Korn at AT&T Bell Labs
- Is a superset of the Bourne shell.
- Supports everything in the Bourne shell.
- Has interactive features comparable to those in the C shell.
- Includes convenient programming features like built-in arithmetic and C-like arrays, functions, and string-manipulation facilities.
- Is faster than the C shell.
- Runs scripts written for the Bourne shell.

For the Korn shell the:

GANESH XEROX RNSIT

- Command full-path name is /bin/ksh.
- Non-root user default prompt is $.
- Root user default prompt is #.

4. The GNU Bourne-Again Shell(bash)

- Is compatible to the Bourne shell.
- Incorporates useful features from the Korn and C shells.
- Has arrow keys that are automatically mapped for command recall and editing.

For the GNU Bourne-Again shell the:

- Command full-path name is /bin/bash.
- Default prompt for a non-root user is bash-x.xx$. (Where x.xx indicates the shell version number. For example, bash-3.50$)
- Root user default prompt is bash-x.xx#. (Where x.xx indicates the shell version number. For example, bash-3.50$#)

➢ **Utilities:** A utility is a standard Unix program that provides a support for users. Three common utilities are text editors, search programs and sort programs.

➢ **Applications:** are programs that are not a standard part of UNIX. Written by system administrator's professional programmers or users they provide an extended capability to the system.

1.6 POSIX and The Single Unix Specification

- Unix fragmentation & the absence of a single standard adversely affected the development of portable applications. A group of standards , POSIX(Portable Operating System Interface for Computer Environments), were developed at IEEE. POSIX refers to operating systems in general, but was based on UNIX. Two of the most cited standards from the POSIX family are known as POSIX.1 & POSIX.2.
- POSIX.1 specifies the C application program interface – the system calls. POSIX.2 deals with the shell & utilities.
- In 2001, a joint initiative of X/Open & IEEE resulted in the Combining of two standards. This is the Single Unix specification, version 3(SUSV3). 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 ported to another POSIX-compliant Unix machine with minimum modifications.

Some of the commercial UNIX based on system V are:

- IBM's AIX
- Hewlett-Packard's HPUX
- SCO's Open Server Release 5
- DEC's Digital UNIX
- Sun Microsystems' Solaris 2

Some of the commercial UNIX based on BSD are:

- SunOS 4.1.X (now Solaris)
- DEC's Ultris
- BSD/OS, 4.4BSD

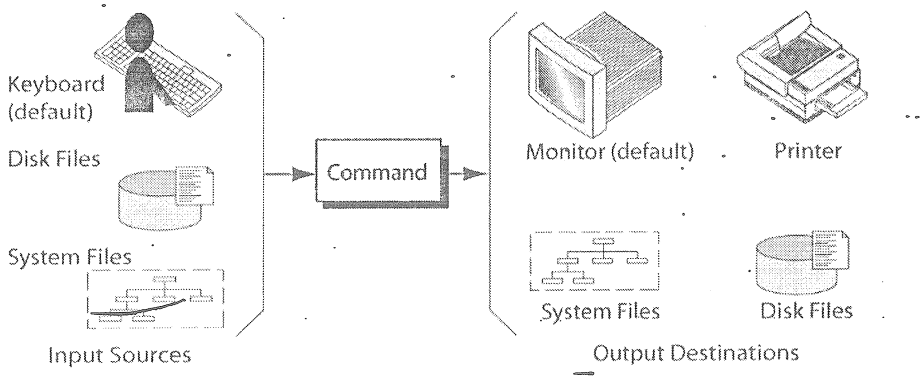
Some Free UNIX are:

- Linux, written by Linus Torvalds at University of Helsinki in Finland.
- sFreeBSD and NetBSD, a derivative of 4.4BSD

1.7 GENERAL FEATURES OF UNIX COMMANDS/ COMMAND STRUCTURE

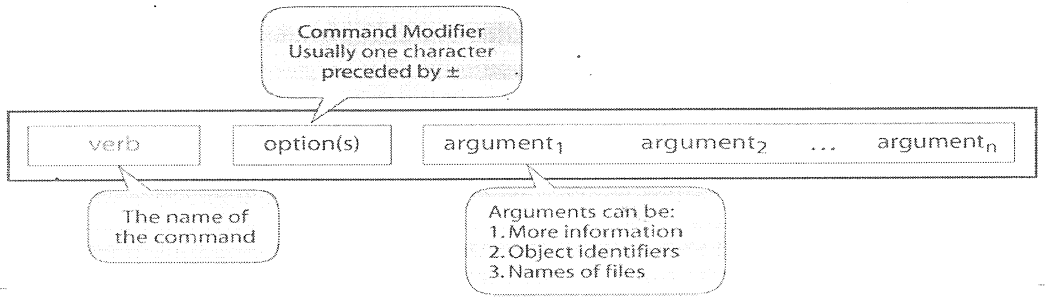
Commands are entered at shell prompt. The components of the command line are:

- the verb
- any options required by the command
- the command's arguments (if required)



For example, the general form of a UNIX command is:

$verb [-option(s)] [argument(s)]



- **Verb:** is the command name. The command indicates what action is to be taken. This action concept gives us the name verb for action.
- **Option:** modifies how the action is applied.
- **Argument:** provides additional information to the command.

Note: Options MUST come after the command and before any command arguments. Options SHOULD NOT appear after the main argument(s). However, some options can have their own arguments

if **options** are enclosed within the [] then options are not mandatory else it is compulsory

if **arguments** are enclosed within the [] then options are not mandatory else it is compulsory.

Options

An option is preceded by a minus sign (-) to distinguish it from filenames.

Example: $ ls –l

GANESH XEROX RNSIT

There must not be any whitespaces between – and l. Options are also arguments, but given a special name because they are predetermined. Multiple options can be given with only one – sign. i.e., instead of using

$ ls –l –a –t

we can as well use,

$ ls –lat

Because UNIX was developed by people who had their own ideas as to what options should look like, there will be variations in the options. Some commands use + as an option prefix instead of -.

Filename arguments:

- Many unix commands use a filename as arguments so the command can take input from the file. If a command uses a filename as argument at all, it will generally be its last argument and after all options.
- Its also quite common to see many commands working with multiple filenames as arguments. ls -lat chap01, chap02, chap03
- The command with its arguments and options is known as the command line.
- This line can be considered complete only after the user hit[Enter]. The complete line is then fed to the shell as its input for interpretation and execution.

1.8 BASIC UNIX COMMANDS SUCH AS echo, printf, ls, who, date, passwd, cal, combining commands.

➢ **echo: Displaying a message**

- The **echo** command used to display a message.
- To display the diagnostic messages on the terminal or to issue prompts for taking user input (like echo Sun Solaris).
- To evaluate shell variable (like echo $SHELL)
  General Syntax: **$echo [Short-Option]... [String]...**
- Orginally, echo was an external command, but nowadays all shells have echo built-in.
- Most of the echo's behavior differences relates to the way echo's interprets certain strings known as escape sequences.
- An escape sequence is generally a two character – string beginning with \(backslash).

Options

-n	Do not output a trailing newline.
-e	Enable interpretation of backslash escape sequences (see below for a list of these).
-E	Disable interpretation of backslash escape sequences (this is the default).
--help	Display a help message and exit
version	Output version information and exit.

If you specify the -e option, the following escape sequences are recognized:

\\\	Literal backslash character ("\")
\a	An alert (The BELL character)
\b	Backspace

\c	Produce no further output after this
\e	The escape character; equivalent to pressing the escape key
\f	A form feed
\n	A newline
\r	A carriage return
\t	A horizontal tab
\v	A vertical tab

Execution of commands

```
chandana@chandana-VirtualBox: ~
chandana@chandana-VirtualBox:~$ echo $SHELL
/bin/bash
chandana@chandana-VirtualBox:~$ echo "Welcome to RNSIT"
Welcome to RNSIT
chandana@chandana-VirtualBox:~$ echo "Welcome \bto \bRNSIT"
Welcome \bto \bRNSIT
chandana@chandana-VirtualBox:~$ echo -e "Welcome \bto \bRNSIT"
WelcometoRNSIT
chandana@chandana-VirtualBox:~$ echo -e "Welcome \cto \cRNSIT"
Welcome chandana@chandana-VirtualBox:~$ echo -e "Welcome \nto \nRNSIT"
Welcome
to
RNSIT
chandana@chandana-VirtualBox:~$ echo -e "Welcome \tto \tRNSIT"
Welcome         to      RNSIT
chandana@chandana-VirtualBox:~$ echo -e "Welcome \rto RNSIT"
to RNSIT
chandana@chandana-VirtualBox:~$ echo -e "Welcome \vto \vRNSIT"
Welcome
        to
           RNSIT
chandana@chandana-VirtualBox:~$ echo -e "\aWelcome to RNSIT"
Welcome to RNSIT
chandana@chandana-VirtualBox:~$ echo -n "Welcome to RNSIT"
Welcome to RNSITchandana@chandana-VirtualBox:~$ echo *
combine.txt Desktop Documents Downloads Music myfolder out.txt Pictures Public snap Templates UNIX1 Videos
chandana@chandana-VirtualBox:~$
```

- **Printf: An alternative to echo command**
  - Like echo, it exists as an external command, its only the Bash shell that has printf built-in.
  - The command can be used as $printf "No file entered"
    Output: No file entered
  - Printf also accepts all escape sequences used by echo, but it doesn't automatically insert a newline unless the \n is used explicitly.

- While printf can do everything that echo does, some of its format strings can convert data from one form to another.

%s – String
%30s – As above but printed in a space 30 characters wide.
%f - Floating point number
%d - Decimal integer
%x - Hexadecimal integer
%o - octal integer

Execution of commands

```
chandana@chandana-VirtualBox: ~
chandana@chandana-VirtualBox:~$ printf "%s\n" "Welcomt to RNSIT\n"
Welcomt to RNSIT\n
chandana@chandana-VirtualBox:~$ printf "%b\n" "Welcomt to RNSIT\n" "From ISE\n"
Welcomt to RNSIT

From ISE

chandana@chandana-VirtualBox:~$ printf "%30b\n" "Welcomt to RNSIT\n" "From ISE\n"
        Welcomt to RNSIT

                  From ISE

chandana@chandana-VirtualBox:~$ printf "%d\n" "220" "420"
220
420
chandana@chandana-VirtualBox:~$ printf "%f\n" "220" "420.25"
220.000000
420.250000
chandana@chandana-VirtualBox:~$ printf "The value of 255 is %o in octal and %x in hexadecimal\n" 255 255
The value of 255 is 377 in octal and ff in hexadecimal
chandana@chandana-VirtualBox:~$ printf "%08x\n" "1024"
00000400
chandana@chandana-VirtualBox:~$
```

In the first line printf command \n will print, no special meaning is there for \n when it is used with %s.

- **ls: Listing Files**
  - The files are organized in separate folders called directories. We can list the names of the files available in this directory with the ls command. Following is the ls command without any option:

    $ ls

    Chap01 Chap02 Pgm.c F1.doc
  - Unix has a special symbol * to access the files with same pattern $ ls Chap*

    Chap01 Chap02

Execution of commands

```
chandana@chandana-VirtualBox:~$ ls
combine.txt  Desktop  Documents  Downloads  Music  myfolder  out.txt  Pictures  Public  snap  Templates  UNIX1  Videos
chandana@chandana-VirtualBox:~$ ls -l
total 52
-rw-rw-r-- 1 chandana chandana   85 Sep 10 10:05 combine.txt
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Desktop
drwxr-xr-x 3 chandana chandana 4096 Sep  4 10:18 Documents
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Downloads
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Music
drwxrwxr-x 3 chandana chandana 4096 Sep 10 10:11 myfolder
-rw-rw-r-- 1 chandana chandana  274 Sep 10 10:08 out.txt
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Pictures
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Public
drwxr-xr-x 3 chandana chandana 4096 Sep 14 10:31 snap
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Templates
drwxrwxr-x 2 chandana chandana 4096 Sep  9 19:12 UNIX1
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Videos
```

$ls – lists directory contents of files and directories.
$ls -l – Display all information about files/directories contents.
Here is the information about all the listed columns:

1. First Column: represents file type and permission given on the file. Above is the description of all type of files.
2. Second Column: represents the number of memory blocks taken by the file or directory.
3. Third Column: represents owner of the file. This is the Unix user who created this file.
4. Fourth Column: represents group of the owner. Every Unix user would have an associated group.
5. Fifth Column: represents file size in bytes.
6. Sixth Column: represents date and time when this file was created.
7. Seventh Column: represents filename.

In the ls -l listing example, every file line began with a d, -, or l. These characters indicate the type of file that's listed.

```
chandana@chandana-VirtualBox:~$ ls -lh
total 52K
-rw-rw-r-- 1 chandana chandana   85 Sep 10 10:05 combine.txt
drwxr-xr-x 2 chandana chandana 4.0K Sep  2 19:58 Desktop
drwxr-xr-x 3 chandana chandana 4.0K Sep  4 10:18 Documents
drwxr-xr-x 2 chandana chandana 4.0K Sep  2 19:58 Downloads
drwxr-xr-x 2 chandana chandana 4.0K Sep  2 19:58 Music
drwxrwxr-x 3 chandana chandana 4.0K Sep 10 10:11 myfolder
-rw-rw-r-- 1 chandana chandana  274 Sep 10 10:08 out.txt
drwxr-xr-x 2 chandana chandana 4.0K Sep  2 19:58 Pictures
drwxr-xr-x 2 chandana chandana 4.0K Sep  2 19:58 Public
drwxr-xr-x 3 chandana chandana 4.0K Sep 14 10:31 snap
drwxr-xr-x 2 chandana chandana 4.0K Sep  2 19:58 Templates
drwxrwxr-x 2 chandana chandana 4.0K Sep  9 19:12 UNIX1
drwxr-xr-x 2 chandana chandana 4.0K Sep  2 19:58 Videos
chandana@chandana-VirtualBox:~$ ls -ld
drwxr-xr-x 19 chandana chandana 4096 Sep 14 14:20 .
```

$ls -lh – Displays file size in easy to read format. h represents human readable form.

```
chandana@chandana-VirtualBox:~$ ls -a
.                .cache          Downloads  myfolder  snap                        .vboxclient-clipboard.pid
..               combine.txt     .gnupg     out.txt   .ssh                        .vboxclient-display-svga-x11.pid
.bash_history    .config         .local     Pictures  .sudo_as_admin_successful   .vboxclient-draganddrop.pid
.bash_logout     Desktop         .mozilla   .profile  Templates                   .vboxclient-seamless.pid
.bashrc          Documents       Music      Public    UNIX1                       Videos
chandana@chandana-VirtualBox:~$ ls -A
.bash_history  combine.txt  Downloads  Music     .profile  .sudo_as_admin_successful  .vboxclient-display-svga-x11.pid
.bash_logout   .config      .gnupg     myfolder  Public    Templates                  .vboxclient-draganddrop.pid
.bashrc        Desktop      .local     out.txt   snap      UNIX1                      .vboxclient-seamless.pid
.cache         Documents    .mozilla   Pictures  .ssh      .vboxclient-clipboard.pid  Videos
chandana@chandana-VirtualBox:~$ ls -1
combine.txt
Desktop
Documents
Downloads
Music
myfolder
out.txt
Pictures
Public
snap
Templates
UNIX1
Videos
```

$ls -a – Displays all the hidden files including current and parent directory.
$ls -A – Displays all the hidden files excluding current and parent directory.
$ls -1 – Displays one file per line.

```
chandana@chandana-VirtualBox:~$ ls -lt
total 52
drwxr-xr-x 3 chandana chandana 4096 Sep 14 10:31 snap
drwxrwxr-x 3 chandana chandana 4096 Sep 10 10:11 myfolder
-rw-rw-r-- 1 chandana chandana  274 Sep 10 10:08 out.txt
-rw-rw-r-- 1 chandana chandana   85 Sep 10 10:05 combine.txt
drwxrwxr-x 2 chandana chandana 4096 Sep  9 19:12 UNIX1
drwxr-xr-x 3 chandana chandana 4096 Sep  4 10:18 Documents
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Desktop
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Downloads
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Music
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Pictures
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Public
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Templates
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Videos
chandana@chandana-VirtualBox:~$ ls -ltr
total 52
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Videos
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Templates
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Public
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Pictures
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Music
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Downloads
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Desktop
drwxr-xr-x 3 chandana chandana 4096 Sep  4 10:18 Documents
drwxrwxr-x 2 chandana chandana 4096 Sep  9 19:12 UNIX1
-rw-rw-r-- 1 chandana chandana   85 Sep 10 10:05 combine.txt
-rw-rw-r-- 1 chandana chandana  274 Sep 10 10:08 out.txt
drwxrwxr-x 3 chandana chandana 4096 Sep 10 10:11 myfolder
```

$ls -lt – Displays all information about file/directories based on last modification time.
$ls -ltr – Displays all information about file/directories in the reverse order of last modification time.

```
chandana@chandana-VirtualBox:~$ ls -F
combine.txt  Documents/  Music/    out.txt   Public/  Templates/  Videos/
Desktop/     Downloads/  myfolder/ Pictures/ snap/    UNIX1/
chandana@chandana-VirtualBox:~$ ls --version
ls (GNU coreutils) 8.30
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <https://gnu.org/licenses/gpl.html>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Written by Richard M. Stallman and David MacKenzie.
```

$ls -F – will add the '/' Character at the end each directory.
$ls –version – checks version of ls command.

- **Who: Displays the users who are currently logged into the computer.**
  - General Syntax: $ who [options] [filename]
  - who command is used to find out the following information:
    1. Time of last system boot
    2. Current run level of the system
    3. List of logged in users and more.

Options:

1. The who command displays the following information for each user currently logged in to the system if no option is provided:
   - Login name of the users
   - Terminal line numbers
   - Login time of the users in to system
   - Remote host name of the user
2. To display host name and user associated with standard input such as keyboard user needs to use who -m -H command.
3. To show all active processes which are spawned by INIT process who -p -H command will be used.
4. To show status of the users message as +, – or ?, who -T -H command need to be used.
5. To show list of users logged in to system, who -u is used.
6. To show time of the system when it booted last time, who -b -H command will be used.
7. To show details of all dead processes, who -d -H
8. To show system login process details, who -T -H
9. To display user identification information, $id is the command to be used.
10. To count number of users logged on to system, who -q -H to be used.
11. To display current run level of the system, who -r is the command.
12. To display all details of current logged in user. Who -a command is used.
13. To display system's username, whoami command is used.
14. To display list of users and their activities $w is used.

Execution of commands

```
chandana@chandana-VirtualBox:~$ who
chandana :0          2020-09-14 14:19 (:0)
chandana@chandana-VirtualBox:~$ who -m -H
NAME     LINE         TIME            COMMENT
chandana@chandana-VirtualBox:~$ who -p -H
NAME     LINE         TIME                     PID COMMENT
chandana@chandana-VirtualBox:~$ who -T -H
NAME      LINE         TIME            COMMENT
chandana ? :0          2020-09-14 14:19 (:0)
chandana@chandana-VirtualBox:~$ who -u
chandana :0          2020-09-14 14:19   ?         1379 (:0)
chandana@chandana-VirtualBox:~$ who -b -H
NAME     LINE         TIME                     PID COMMENT
         system boot  2020-09-14 14:18
chandana@chandana-VirtualBox:~$ who -d -H
NAME     LINE         TIME            IDLE          PID COMMENT  EXIT
chandana@chandana-VirtualBox:~$ who -l -H
NAME     LINE         TIME            IDLE          PID COMMENT
chandana@chandana-VirtualBox:~$ id
uid=1000(chandana) gid=1000(chandana) groups=1000(chandana),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131
(lxd),132(sambashare)
chandana@chandana-VirtualBox:~$ who -q -H
chandana
# users=1
chandana@chandana-VirtualBox:~$ who -r
         run-level 5  2020-09-14 14:19
```

```
chandana@chandana-VirtualBox:~$ who -a
          system boot  2020-09-14 14:18
          run-level 5  2020-09-14 14:19
chandana ? :0          2020-09-14 14:19   ?         1379 (:0)
chandana@chandana-VirtualBox:~$ whoami
chandana
chandana@chandana-VirtualBox:~$ w
 15:35:28 up  1:17,  1 user,  load average: 0.27, 0.17, 0.11
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
chandana :0       :0               14:19   7xdm?  3:33   0.01s /usr/lib/gdm3/gdm-x-session --run-script env GNOME_SHELL_
chandana@chandana-VirtualBox:~$
```

- **date :** It is a command that instructs the machine to display the current date & time. This command does not allow the user to change the date or the time. Only administrator can do that.

date %a	Displays Weekday name in short (like Mon, Tue, Wed)
date +%A	Displays Weekday name in full short (like Monday, Tuesday)
date +%b	Displays Month name in short (like Jan, Feb, Mar)
date +%B	Displays Month name in full short (like January, February)
date +%d	Displays Day of month (e.g., 01)
date +%D	Displays Current Date; shown in MM/DD/YY
date +%F	Displays Date; shown in YYYY-MM-DD
date +%H	Displays hour in (00..23) format
date +%m	Displays month (01..12)

date +%M	Displays minute (00..59)
date +%S	Displays second (00..60)
date +%Y	Displays full year i.e. YYYY
date +%T	Displays time; shown as HH:MM:SS Note: Hours in 24 Format

Execution of commands

```
chandana@chandana-VirtualBox:~$ date
Tuesday 15 September 2020 10:30:51 AM IST
chandana@chandana-VirtualBox:~$ date -u
Tuesday 15 September 2020 05:00:55 AM UTC
chandana@chandana-VirtualBox:~$ date +"%m"
09
chandana@chandana-VirtualBox:~$ date +"%d"
15
chandana@chandana-VirtualBox:~$ date +"%y"
20
chandana@chandana-VirtualBox:~$ date +"%Y"
2020
chandana@chandana-VirtualBox:~$ date +"%T"
10:31:49
chandana@chandana-VirtualBox:~$ date +"%r"
10:31:54 AM IST
chandana@chandana-VirtualBox:~$ date +"%H-%M"
10-32
chandana@chandana-VirtualBox:~$ date +"%m-%d-%Y"
09-15-2020
```

```
chandana@chandana-VirtualBox:~$ date --date="2/02/2020"
Sunday 02 February 2020 12:00:00 AM IST
chandana@chandana-VirtualBox:~$ date --date="2 year ago"
Saturday 15 September 2018 10:33:12 AM IST
chandana@chandana-VirtualBox:~$ date --date="yesterday"
Monday 14 September 2020 10:33:24 AM IST
chandana@chandana-VirtualBox:~$ date --date="next tue"
Tuesday 22 September 2020 12:00:00 AM IST
chandana@chandana-VirtualBox:~$ date --date="2 day"
Thursday 17 September 2020 10:33:48 AM IST
chandana@chandana-VirtualBox:~$ date --date="1 year"
Wednesday 15 September 2021 10:33:59 AM IST
chandana@chandana-VirtualBox:~$ date --date="5 sec ago"
Tuesday 15 September 2020 10:04:14 AM IST
chandana@chandana-VirtualBox:~$
```

- **passwd: It is used to change the password**

  $ passwd

  old password: ****

  new password: ******

  re-enter new password: ******

Now the new password is encrypted & registered by the system. The encrypted password is stored in the file /etc/shadow.

Rules for giving passwords

GANESH XEROX RNSIT

a) Must be >=6 characters long,
b) Must contain 2 out of 3 of upper-case letters, lower-case letters, non-letters (digits, punct)
c) May not be a dictionary word or too similar to your name

- **cal:** It displays the calendar of any specific month, or a complete year.
  If a user wants a quick view of calendar in Unix terminal, cal is the command. By default, cal command shows current month calendar as output.
  General Syntax: cal [[month] year]
  cal : Shows current month calendar on the terminal.
  cal 01 1990 : Shows calendar of selected month and year.
  cal 2020 : Shows the whole calendar of the year.
  cal -3 : Shows calendar of previous, current and next month

```
chandana@chandana-VirtualBox:~$ cal
    September 2020
Su Mo Tu We Th Fr Sa
          1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30
```

```
chandana@chandana-VirtualBox:~$ cal 01 1990
    January 1990
Su Mo Tu We Th Fr Sa
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31

chandana@chandana-VirtualBox:~$ cal 09 2020
    September 2020
Su Mo Tu We Th Fr Sa
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30

chandana@chandana-VirtualBox:~$ cal 2020
                              2020
      January                February                 March
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
             1  2  3  4                 1   1  2  3  4  5  6  7
 5  6  7  8  9 10 11   2  3  4  5  6  7  8   8  9 10 11 12 13 14
12 13 14 15 16 17 18   9 10 11 12 13 14 15  15 16 17 18 19 20 21
19 20 21 22 23 24 25  16 17 18 19 20 21 22  22 23 24 25 26 27 28
26 27 28 29 30 31     23 24 25 26 27 28 29  29 30 31
                      30
```

```
chandana@chandana-VirtualBox:~$ cal -3
    August 2020          September 2020         October 2020
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
                   1         1  2  3  4  5                 1  2  3
 2  3  4  5  6  7  8   6  7  8  9 10 11 12   4  5  6  7  8  9 10
 9 10 11 12 13 14 15  13 14 15 16 17 18 19  11 12 13 14 15 16 17
16 17 18 19 20 21 22  20 21 22 23 24 25 26  18 19 20 21 22 23 24
23 24 25 26 27 28 29  27 28 29 30           25 26 27 28 29 30 31
30 31
```

1.9 COMBINING COMMANDS

UNIX allows you to specify more than one command in the command line. Each command has to be separated from the other by a ; (semicolon).

Example:

$wc note; ls -l note

The above command first it displays the line count, word cont and byte or character count along with this it also display the details of note file.

When you learn to redirect the output of these commands you may even like to group them together within parentheses .

Example :

$(wc note ; ls -l note) > newlist

The combined output of the two commands is now sent to the file newlist. Whitespace is provided here only for better readability. You might reduce a few keystrokes like this $wc note;ls -l note)>newlist

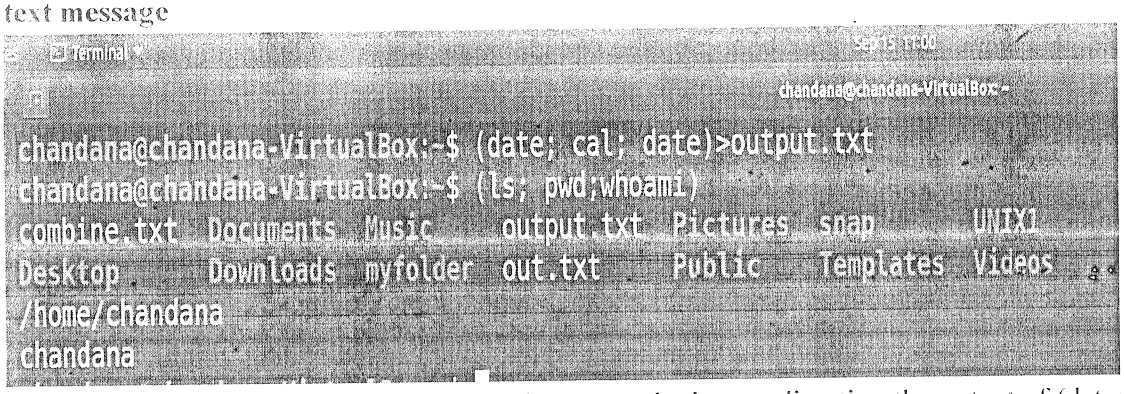
When a command line contains a semicolon, the shell understands that the command on each side of it needs to be processed separately. The ; here is known as a metacharacter, and you'll come across several metacharaters that have special meaning to the shell.

A command line can overflow ~~or be split~~ into multiple lines

A command is often keyed in. though the terminal width is restricted to 80 characters, that doesn't prevent you from entering a command, or a sequence of them, in one line even though the total width may exceed 80 characters. The command simply overflows to the next line though it is stil in a single logical line.

Sometimes, you'll find it necessary or desirable to split a long command line into multiple lines. In that case, the shell issues a **secondary prompt,** usually >, to indicate to you that the command line isn't complete. This is easily shown with the echo command:

$echo "this is
a three-line >text message"
output:
this is
a three-line
text message



For the first command output.txt text file has created, since redirecting the output of (date; cal; date) > output.txt file as shown below.

```
Text Editor
Open
1 Tuesday 15 September 2020 10:59:47 AM IST
2    September 2020
3 Su Mo Tu We Th Fr Sa
4        1  2  3  4  5
5  6  7  8  9 10 11 12
6 13 14 15 16 17 18 19
7 20 21 22 23 24 25 26
8 27 28 29 30
9
10 Tuesday 15 September 2020 10:59:47 AM IST
```

1.10 MEANING OF INTERNAL AND EXTERNAL COMMANDS

UNIX commands are classified into two types

Internal Commands - Ex: echo

External Commands - Ex: ls, cat

Internal Command: Internal commands are something which is built into the shell. For the shell built in commands, the execution speed is really high. It is because no process needs to be spawned for executing it.

For example, when using the "cd" command, no process is created. The current directory sim-ply gets changed on executing it.

External Command: External commands are not built into the shell. These are executable present in a separate file. When an external command has to be executed, a new process has to be spawned and the command gets executed.

For example, when you execute the "cat" command, which usually is at /usr/bin, the executable /usr/bin/cat gets executed.

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

***type* command:**

$ type cd

cd is a shell builtin

$ type cat cat

is /bin/cat

For the internal commands, the type command will clearly say its shell built-in, however for the external commands, it gives the path of the command from where it is executed.

1.11 THE TYPE COMMAND: knowing the type of a command and locating it.

type - Display information about command type.

The type command is a shell built-in that displays the kind of command the shell will execute, given a particular command name. It works like this: type *command*

where "command" is the name of the command you want to examine. Here are some examples:

$type type

Output: type is a shell built-in

$type ls

Output: ls is aliased to 'ls –color=tty'

Stype cp
Output: cp is /bin/cp

Here we see the results for three different commands. Notice that the one for ls (taken from a Fedora system) and how the ls command is actually an alias for the ls command with the "--color=tty" option added. Now we know why the output from ls is displayed in color!

1.12 THE ROOT LOGIN

Root: the system administrator's login. The unix system provides a special login name for the exclusive use of the administrator, it is called **root**. This account doesn't need to be separately created but comes with every system. Its password is generally set at the time of installation of the system and has to be used on logging in

Login: **root**

Password: ******

The prompt of the root is # other users(non privileged user) either $ or %

Once you login as root, you are place in root's home directory. Depending on the system, this could be / or **/root**

1.13 BECOMING THE SUPER USER: SU COMMAND

Any user can acquire super user status with the **su** command if user knows the root password. Example, the user RNSIT becomes a super user in this way

$su
Password:****
#pwd
/home/RNSIT

Though the current directory does not change the # prompt indicates the RNSIT now has powers of a super user. To be in root's home directory on superuser login, use **su –l.**

Creating a user's Environment: users often rush to the administrator with the complaint that a program has stopped running. The administrator first tries running it in a simulated environment.

su, when used with a -, recreates the user's environment without taking the login password route:

$su – RNSIT

This sequences executes RNSIT's .profile and temporarily creates RNSIT's environment. **Su** runs a separate sub shell, so this mode is terminated by hitting [Ctrl-d] or using **exit.**

CHAPTER 2 UNIX FILES

UNIX system has thousands of files. If user write a program, user add one more file to the system. When user compile it, user add some more. Files grow rapidly, and if they are not organized properly, user will find it difficult to locate them. So UNIX has a file system (UFS) to manage or organizes its own files in directory.

2.1 NAMING FILES

On a UNIX system, 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. Users are permitted to use control characters or other nonprintable characters in a filename. However, user should avoid using these characters while naming a file. It is recommended that only the following characters be used in filenames:

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

UNIX imposes no restrictions on the extension. In all cases, it is the application that imposes that restriction. Eg. A C-Compiler expects C program filenames to end with .c, Oracle requires SQL scripts to have .sql extension.

A file can have as many dots embedded in its name. A filename can also begin with or end with a dot. ex: .sample, .one

The file name may consist of ordinary characters, digits and special tokens like the underscore, except the forward slash (/).

It is permitted to use special tokens like the ampersand (&) or spaces in a filename. UNIX is case sensitive; chap01, Chap01 and CHAP01 are three different filenames that can coexist in the same directory.

2.2 BASIC FILE TYPES/CATEGORIES

A simple description of the UNIX system is this:

"On a UNIX system, everything is a file; if something is not a file, it is a process."

A UNIX system makes no difference between a file and a directory, since a directory is just a file containing names of other files. Programs, services, texts, images, and so forth, are all files. Input and output devices, and generally all devices, are considered to be files, according to the system.

Ordinary (Regular) File

This is the most common file type. An ordinary file can be either a text file or a binary file. A text file contains only printable characters and user can view and edit them. All C and Java program sources, shell scripts are text files. Every line of a text file is terminated with the newline character.

A binary file, on the other hand, contains both printable and nonprintable characters that cover the entire ASCII range. The object code and executables that you produce by compiling C programs are binary files. Sound and video files are also binary files.

Directory File

A directory contains no data, but keeps details of the files and subdirectories that it contains. A directory file contains one entry for every file and subdirectory that it houses. Each

entry has two components namely, the filename and a unique identification number of the file or directory (called the inode number). ls -i is the command to be used to know inode number.

When user create or remove a file, the kernel automatically updates its corresponding directory by adding or removing the entry (filename and inode number) associated with the file.

Device File

All the operations on the devices are performed by reading or writing the file representing the device. It is advantageous to treat devices as files as some of the commands used to access an ordinary file can be used with device files as well.

Device filenames are found in a single directory structure, /dev. A device file is not really a stream of characters. It is the attributes of the file that entirely govern the operation of the device. The kernel identifies a device from its attributes and uses them to operate the device.

2.3 ORGANIZATION OF FILES

All of the files in the UNIX file system are organized into a multi-leveled hierarchy called a directory tree. A family tree is an example of a hierarchical structure that represents how the UNIX file system is organized. The UNIX file system might also be envisioned as an inverted tree or the root system of plant. At the very top of the file system is single directory called "root" which is represented by a / (slash). All other files are "descendents" of root. The number of levels is largely arbitrary, although most UNIX systems share some organizational similarities. There is no single standard unix file structure. Most follow a general convention for filesystem organisation at the highest level.

- /: The slash / character alone denotes the root of the filesystem tree.
- /bin: Stands for "binaries" and contains certain fundamental utilities, such as ls or cp, which are generally needed by all users.
- /boot: Contains all the files that are required for successful booting process.
- /dev: Stands for "devices". Contains file representations of peripheral devices and pseudo-devices.
- /etc: Contains system-wide configuration files and system databases. Originally also contained "dangerous maintenance utilities" such as init, but these have typically been moved to /sbin or elsewhere.
- /home: Contains the home directories for the users.
- /lib: Contains system libraries, and some critical files such as kernel modules or device drivers.
- /root: The home directory for the superuser "root" – that is, the system administrator. This account's home directory is usually on the initial filesystem, and hence not in /home (which may be a mount point for another filesystem) in case specific maintenance needs to be performed, during which other filesystems are not available. Such a case could occur, for example, if a hard disk drive suffers physical failures and cannot be properly mounted.
- /tmp: A place for temporary files. Many systems clear this directory upon startup; it might have tmpfs mounted atop it, in which case its contents do not survive a reboot, or it might be explicitly cleared by a startup script at boot time.
- /usr: Originally the directory holding user home directories,its use has changed. It now holds executables, libraries, and shared resources that are not system critical, like the X Window

GANESH XEROX RNSIT

System, KDE, Perl, etc. However, on some Unix systems, some user accounts may still have a home directory that is a direct subdirectory of /usr, such as the default as in Minix. (on modern systems, these user accounts are often related to server or system use, and not directly used by a person).

- /usr/bin: This directory stores all binary programs distributed with the operating system not residing in /bin, /sbin or (rarely) /etc.
- /usr/include: Stores the development headers used throughout the system. Header files are mostly used by the #include directive in C/C++ programming language.
- /usr/lib: Stores the required libraries and data files for programs stored within /usr or elsewhere.
- /var: A short for "variable." A place for files that may change often – especially in size, for example e-mail sent to users on the system, or process-ID lock files.
- /var/log: Contains system log files.
- /var/mail: The place where all the incoming mails are stored. Users (other than root) can access their own mail only. Often, this directory is a symbolic link to /var/spool/mail.
- /var/spool: Spool directory. Contains print jobs, mail spools and other queued tasks.
- /var/tmp: A place for temporary files which should be preserved between system reboots.

2.4 HIDDEN FILES

An invisible file is one whose first character is the dot or period character (.)
UNIX programs (including the shell) use most of these files to store configuration information.
Some common examples of hidden files include the files:

.profile: the Bourne shell (sh) initialization script
.kshrc: the Korn shell (ksh) initialization script
.cshrc: the C shell (csh) initialization script
.rhosts: the remote shell configuration file

To list invisible files, specify the -a option to ls:

```
chandana@chandana-VirtualBox:~$ ls -a
.                .local          .ssh
..               .mozilla        .sudo_as_admin_successful
.bash_history    Music           Templates
.bash_logout     p1.c            .vboxclient-clipboard.pid
.bashrc          p2.c            .vboxclient-display-svga-x11.pid
.cache           p3.c            .vboxclient-draganddrop.pid
.config          p4.c            .vboxclient-hostversion.pid
Desktop          Pictures        .vboxclient-seamless.pid
Documents        .profile        Videos
Downloads        Public
.gnupg           snap
```

Single dot . : This represents current directory.
Double dot .. : This represents parent directory.

2.5 STANDARD DIRECTORIES

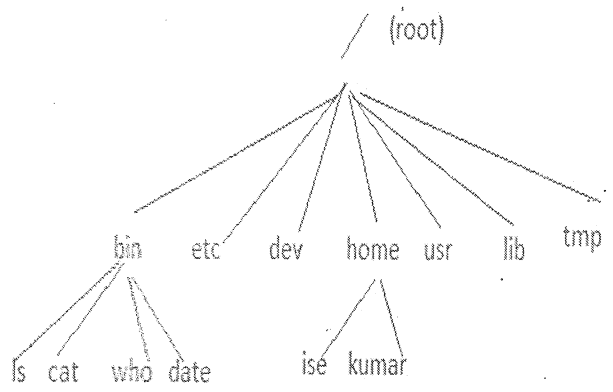
Although there may be a few exceptions, you will find a similar set of system directories on all UNIX and Linux operating systems. These system directories are located directly below the root directory, and are essential to the startup and continuous operation of the system. The following is a list of these key directories and their contents:

- /bin - The /bin directory contains programs needed for using and managing the system. Some of the frequently used commands in this directory are date (displays today's date), ls (lists the contents of a directory), and cp (makes a copy of a file). The name bin is used for this directory because executable programs in UNIX and Linux are called binary files.
- /dev - This directory contains system device files. A device file is a special object in the file system that provides an interface to a particular device. Examples of devices having device files in /dev are disk drives, tape drives, or CDROM drives.
- /etc - System specific configuration files, and files essential for system startup are located in the /etc directory. In the past, administrative executable programs were also stored in this directory but have been moved to the /sbin directory.
- /home - The /home directory is where the home directories for all users of the system are stored. For example, if John Doe's username is jdoe, the path to his home directory would be /home/jdoe. John would store his personal files and programs in this directory.
- /mnt - This directory is where temporary file systems are mounted. It may contain subdirectories like cdrom, floppy, and disk. Once a cdrom has been mounted on the system (made available for reading its contents), the path to the files located on the cdrom would be /mnt/cdrom.
- /opt - The /opt directory contains software files that are not installed when the operating system is installed. This directory usually contains products provided by third-party software vendors.
- /sbin - Programs for administering a system are located in the /sbin directory. Some examples are fdisk (used to partition a disk), fsck (used to check the integrity of a file system), and shutdown (used for stopping a system). An easy way to remember the contents of this directory is to equate sbin with "system binaries."
- /tmp - As the name implies, this directory is used for holding temporary files. This directory is commonly referred to as a scratch directory, and can be used by all system users. You should not store any important files in this directory because its contents may be deleted at any time. Your important files should be kept in your home directory.
- /usr - The /usr directory contains programs and files related to the users of a system. The data in /usr is typically read-only, and may be shared with other computer systems on a network. You will notice that the directory structure under /usr is somewhat similar to the directory structure under the root directory (/).
- /var - Files with varying content are stored in the /var directory. This includes system log files, mail system files, and print spooling system files.

2.6 THE PARENT-CHILD RELATIONSHIP

- A file is a set of data that has a name. The information can be an ordinary text, a user-written computer program, results of a computation, a picture, and so on.
- Unix organizes files in a tree-like hierarchical structure, with the root directory, indicated by a forward slash (/), at the top of the tree.
- The directories have specific purposes and generally hold the same types of information for easily locating files.
- Following are the directories that exist on the major versions of Unix–

GANESH XEROX RNSIT ISE



- The root directory / has a number of subdirectories under it. These subdirectories in turn have more subdirectories and other files under them.
- Every file, apart from root, must have a parent, & it should be possible to trace the ultimate parent of a file to root. Thus, home directory is the parent of ise & kumar subdirectories & / is the parent of home & it is the grandparent of ise & kumar. If we create a file called sample.c under ise, then ise is the parent of sample.c

2.7 THE HOME VARIABLE: THE HOME DIRECTORY

- When user logs on to the system, UNIX automatically places user in a directory called the **home directory.** It is created by the system when user account is opened.
- If you log in using the login name chandana, you will land up in a directory that could have the pathname **/home/chandana.**
- The shell variable HOME known's yours home directory **$echo $HOME /home/chandana.** //First / represents root directory
- The above is an absolute pathname, which is simply a sequence of directory names separated by slashes.
- An absolute pathname shows a file's location with reference to the top i.e root. These slashes act as delimiters to file and directory names except that the first slash is a synonym
- for root. The directory Chandana is placed two levels below root.
- It's often convenient to refer a file foo located in user home directory as $HOME/foo and it is same as ~/foo in these shells.
- The ~ symbol is a tricky to use because it can refer to any user's home directory and not just our own. If user **yuktha** has the file foo in her home directory, then Chandana can access it as ~yuktha/foo.
- The principle is this: A tilde followed by / (like ~/foo) refers to one's own home directory, but when followed by a string(~yuktha) refers to the home directory of that user represented by the string.

2.8 REACHING REQUIRED FILES - THE PATH VARIABLE

- A command runs in UNIX by executing a disk file. When user specify a command like date, the system will locate the associated file from a list of directories specified in the PATH variable and then executes it. The PATH variable normally includes the current directory also.

- When user enter any UNIX command, user is actually specifying the name of an executable file located somewhere on the system. The system goes through the following steps in order to determine which program to execute:
  - Built in commands (such as cd and echo) are executed within the shell.
  - If an absolute path name (such as /bin/ls) or a relative path name (such as ./myprog) is specified, then the system executes the program from the specified directory.
  - Otherwise the PATH variable is used.
- The PATH variable is a colon-delimited list of directories that the shell searches through when user enter a command. Program files (executables) are kept in many different places on the Unix system. User path tells the Unix shell where to look on the system when user request a particular program. To find out what user path is, at the Unix shell prompt echo $PATH, user path will look something like the following.
  /usr2/username/bin:/usr/local/bin:/usr/bin:.
- User can see their username in place of username. Using the above example path, if user enter the ls command, shell will look for the appropriate executable file in the following order: first, it would look through the directory /usr2/username/bin, then /usr/local/bin, then /usr/bin, and finally the local directory, indicated by the . (a period).

2.9 MANIPULATING THE PATH

- 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.
- Few shells also use export command to manipulate the path.

```
chandana@chandana-VirtualBox:~$ echo $HOME
/home/chandana
chandana@chandana-VirtualBox:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/loc
al/games:/snap/bin
chandana@chandana-VirtualBox:~$ export chandana=dir/path
chandana@chandana-VirtualBox:~$ echo $chandana
dir/path
chandana@chandana-VirtualBox:~$ PATH=$PATH:/opt/bin
chandana@chandana-VirtualBox:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/loc
al/games:/snap/bin:/opt/bin
chandana@chandana-VirtualBox:~$
```

2.10 RELATIVE AND ABSOLUTE PATHNAMES

Relative pathname: Pathnames that don't begin with / specify locations relative to the current working directory. It uses either the current or parent directory as reference and specifies path relative to it. A relative pathname uses one of these cryptic symbols.

cd progs

cat login.sql

Here both are presumed to exist in the current directory. Now, if progs contain a directory scripts under it, user won't need an absolute pathname to change to that directory. Just user can use, cd progs/scripts.

Here we have a pathname that has a /, but it is not absolute because it doesn't begin with a /.

Absolute: If the first character of a pathname is / the files location must be determined with respect to root(/) . Such a pathname is called absolute pathname.

cat /home/chandana

When user have more than one / in a pathname for such / have to descend one level in the file system. Thus chandana is one level below home and two levels below root.

When user specify a file by using frontslashes to demarcate the various levels, have a mechanism of identifying a file uniquely. No two files in a UNIX system can have identical absolute pathnames. User can have two files with the same name, but in different directories, their pathnames will also be different. Thus, the file /home/chandana/progs/p1.c can coexist with the file /home/chandana/scripts/p1.c.

When user specify the date command, the system has to locate the file date from a list of directories specified in the PATH variable and then execute it. However if user know the location of a command in prior, for example date is usually located in /bin or /usr/bin .

Use absolute pathname i,e precede its name with complete path $/bin/date. For example if you need to execute program less residing in /usr/local/bin you need to enter the absolute pathname $/usr/local/bin/less

2.11 DIRECTORY COMMANDS – PWD, CD, MKDIR, RMDIR COMMANDS

- **pwd: Print Working Directory (checking your current directory)**
  - Unix encourages us to believe that, like a file, a user is placed in a specific directory of the file system on logging in.
  - We can move around from one directory to another, but at any point of time, we are located in only one directory. This directory is known as current directory.
  - At any time, we can able to know what is our directory. The pwd command tells us that

    $pwd

    /home/csr
  - This command is built in shell command and is available on most of the shell – bash, Bourne shell, ksh,zsh, etc.
  - Like HOME, pwd displays the absolute pathname. As we navigate the file system with the cd command, we'll be using pwd to know our current directory.

Basic syntax

$pwd [option]

Options	Description
-L (logical)	Use PWD from environment, even if it contains symbolic links
-P (physical)	Avoid all symbolic links

. If both '-L' and '-P' options are used, option 'L' is taken into priority. If no option is specified at the prompt, pwd will avoid all symlinks, i.e., take option '-P' into account.

```
csr@csr-VirtualBox:~$ pwd
/home/csr
csr@csr-VirtualBox:~$ pwd -L
/home/csr
csr@csr-VirtualBox:~$ pwd -P
/home/csr
csr@csr-VirtualBox:~$ pwd --help
pwd: pwd [-LP]
    Print the name of the current working directory.

    Options:
      -L        print the value of $PWD if it names the current working
                directory
      -P        print the physical directory, without any symbolic links

    By default, `pwd' behaves as if `-L' were specified.

    Exit Status:
    Returns 0 unless an invalid option is given or the current directory
    cannot be read.
```

- **cd: Changing The Current Directory**
  - The **cd** command, which stands for "change directory", changes the shell's current working directory.
  - The **cd** command is one of the commands you will use the most at the command line in UNIX. It allows you to change your working directory. You use it to move around within the hierarchy of your file system.

Ex 1: When used with an argument it changes the current directory to the directory specified as argument. For example, **unix** is a directory under user directory **csr**. To change from csr directory to unix directory, issue the command as follows

$pwd
/home/csr
$cd unix
$pwd
/home/csr/unix

```
csr@csr-VirtualBox:~$ pwd
/home/csr
csr@csr-VirtualBox:~$ cd unix
bash: cd: unix: No such file or directory
csr@csr-VirtualBox:~$ mkdir unix
csr@csr-VirtualBox:~$ cd unix
csr@csr-VirtualBox:~/unix$ pwd
/home/csr/unix
```

Ex 2: When cd used without arguments: cd when used without arguments reverts to home directory

$pwd
/home/csr/unix
$cd
cd without argument will change directory from unix to its home directory csr
$pwd
/home/csr

Ex 3: If your present working directory is /home/csr/unix and you need to switch to /bin directory directly, use absolute pathname i.e /bin with cd command

$pwd
/home/csr/unix
$cd /bin
$pwd
/bin

```
csr@csr-VirtualBox:~/unix$ pwd
/home/csr/unix
csr@csr-VirtualBox:~/unix$ cd /bin
csr@csr-VirtualBox:/bin$ pwd
/bin
csr@csr-VirtualBox:/bin$
```

- **mkdir: "making directory". mkdir** is used to create directories on a file system.

If the specified directory does not already exist, **mkdir** creates it. More than one directory may be specified when calling **mkdir**.

mkdir syntax

mkdir [OPTION ...] DIRECTORY ...

Ex 1: To create a directory named rnsit, issue the following command.

$mkdir rnsit

rnsit directory is created under present working directory.

Assume that pwd is /home/csr, then rnsit directory is created under csr directory.

Ex 2: To create three directories at a time, named ise cse mca pass directory names as arguments.

$mkdir ise cse mca

Ex 3: To create a directory tree:

To create a directory named Faculty and create two subdirectories named Teaching and NonTeaching under Faculty, issue the command. Faculty is a parent directory.

$mkdir	parent directory	sub-directories	
$mkdir	**Faculty**	**Faculty/Teaching**	**Faculty/NonTeaching**

Ex 4: Error while creating a directory tree

$mkdir Faculty/Teaching Faculty/NonTeaching

mkdir: Failed to make a directory "Faculty/Teaching": no such file or directory

mkdir: Failed to make a directory "Faculty/NonTeaching": no such file or directory

Error is due to the fact that the parent directory named Faculty is not created before creating sub directories Teaching and NonTeaching.

Ex 5: **$mkdir test**

mkdir: Failed to make directory "test"; Permission denied.

This can happen due to:

a. The directory named test may already exist

b. There may be an ordinary file by the same name in the current directory.

c. The permissions set for the current directory do not permit the creation of files and directories by the user.

- **rmdir: Removing Directories**

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

Ex 1: **$rmdir rnsit**

removes the directory named rnsit

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

The reverse logic of mkdir is applied.

$rmdir	**subdirectories**	**parent directory**
$rmdir	**Faculty/NonTeaching Faculty/Teaching**	**_Faculty**

You cant delete a directory with rmdir unless it is empty. In this example Faculty directory cannot be removed until the sub directories Faculty/NonTeaching and Faculty/Teaching are removed.

You cant remove a sub directory unless you are place in a directory which is hierarchically above the one you have chosen to remove.

2.12 THE DOT(.) AND DOUBLE DOTS(..) NOTATIONS TO REPRESENT PRESENT AND PARENT DIRECTORIES AND THEIR USAGE IN RELATIVE PATH NAMES.

- User can move from working directory /home/csr/unix/module1 to home directory /home/unix using cd command like

  $pwd

  /home/csr/unix/module1

  $cd /home/csr

  $pwd

  /home/csr

- Navigation becomes easy by using common ancestor.

. (a single dot) - This represents the current directory

.. (two dots) - This represents the parent directory

- Assume user is currently placed in /home/csr/unix/module1

  $pwd

  /home/csr/unix/module1

  $cd ..

  $pwd

  /home/csr/unix

- This method is compact and easy when ascending the directory hierarchy. The command

GANESH XEROX

cd .. Translates to this —change your current directory to parent of current directory.

- The relative paths can also be used as:

$pwd
/home/csr/unix
$cd ../..
$pwd
/home

2.13 FILE RELATED COMMANDS – CAT, MV, RM, CP, WC, AND OD COMMANDS.

➢ **cat: Create, view, concatenate files**

cat command is used to display the contents of a small file on the terminal.

```
$ cat cprogram.c
#include<stdio.h>
void main()
{
        printf("hello");
}
```

–As like other files cat accepts more than one filename as arguments

$ cat a.txt b.txt
It contains the contents of a.txt
It contains the contents of b.txt

In this the contents of the second files are shown immediately after the first file without any header information. So, cat concatenates two files - hence its name.

cat Options

- To view contents of a file preceding with line numbers **-n** is the numbering option helps programmer in debugging programs.
- To create a file **$cat >newfile** This is a new file which contains some text, just to add some contents to the file new [ctrl-d] $_
  When the command line is terminated with [Enter], the prompt vanishes. Cat now waits to take input from the user. Enter few lines; press [ctrl-d] to signify the end of input to the system To display the file contents of new use file name with cat command. **$ cat new** This is a new file which contains some text, just to Add some contents to the file new.
- is the command and can input the content of newfile also.
- To copy the contents of one file to another file **$cat [filename whose contents is to be copied] > [destination filename]** is the command.
- Cat command can append the contents of one file to the end of another file by using the command **$cat file1 >> file2.**
- Cat command can display content in reverse order using tac command. **$tac filename**
- Cat command can highlight the end of line **$cat -E "filename"**
- Cat command to merge the contents of multiple file **$cat "file1" "file2" "file3" > "newfile"**

```
csr@csr-VirtualBox:~/Desktop$ cat a.txt
Hi Hello
csr@csr-VirtualBox:~/Desktop$ cat a.txt b.txt
Hi Hello
RNSIT, Bengaluru
csr@csr-VirtualBox:~/Desktop$ cat -n a.txt
     1  Hi Hello
csr@csr-VirtualBox:~/Desktop$ cat >newfile
Welcome to unix programming
^C
csr@csr-VirtualBox:~/Desktop$ cat newfile
Welcome to unix programming
csr@csr-VirtualBox:~/Desktop$ cat a.txt > b.txt
csr@csr-VirtualBox:~/Desktop$ cat a.txt
Hi Hello
csr@csr-VirtualBox:~/Desktop$ cat b.txt
Hi Hello
csr@csr-VirtualBox:~/Desktop$ cat a.txt >> b.txt
csr@csr-VirtualBox:~/Desktop$ cat b.txt
Hi Hello
Hi Hello
```

```
csr@csr-VirtualBox:~/Desktop$ cat c.txt
Welcome to RNSIT
ISE Department
5th sem A
csr@csr-VirtualBox:~/Desktop$ tac c.txt
5th sem A
ISE Department
Welcome to RNSIT
csr@csr-VirtualBox:~/Desktop$ cat -E "c.txt"
Welcome to RNSIT$
ISE Department $
5th sem A$
```

```
csr@csr-VirtualBox:~/Desktop$ cat *.txt
Hi Hello
Hi Hello
Hi Hello
Welcome to RNSIT
ISE Department
5th sem A
Hi, Hello
csr@csr-VirtualBox:~/Desktop$ cat "a.txt" "b.txt" "c.txt" > "d.txt"
csr@csr-VirtualBox:~/Desktop$ cat d.txt
Hi Hello
Hi Hello
Hi Hello
Welcome to RNSIT
ISE Department
5th sem A
```

GANESH XEROX RNSIT ISE

- **mv: Renaming Files**

The mv command renames (moves) files. The main two functions are:

1. It renames a file(or directory)
2. It moves a group of files to different directory

It doesn't create a copy of the file; it merely renames it. No additional space is consumed on disk during renaming.

Syntax: mv [option] source destination

Ex: To rename the file a.txt as movefile.txt we can use the following command

$ mv a.txt movefile.txt

- If the destination file doesn't exist in the current directory, it will be created. Or else it will just rename the specified file in mv command.

```
csr@csr-VirtualBox:~/Desktop$ ls
a.txt  b.txt
csr@csr-VirtualBox:~/Desktop$ cat a.txt
RNSIT, Bengaluru
csr@csr-VirtualBox:~/Desktop$ cat b.txt
Hi Hello
Welcome to RNSIT
ISE Department
csr@csr-VirtualBox:~/Desktop$ mv a.txt movefile.txt
csr@csr-VirtualBox:~/Desktop$ ls
b.txt  movefile.txt
csr@csr-VirtualBox:~/Desktop$ cat movefile.txt
RNSIT, Bengaluru
csr@csr-VirtualBox:~/Desktop$ mv movefile.txt b.txt
csr@csr-VirtualBox:~/Desktop$ ls
b.txt
```

- A group of files can be moved to a directory. Ex: Moves three files a.txt, b.txt, to the directory Module

$ mv a.txt b.txt Module

```
Terminal                                Sep 30 13:46
                                        csr@csr-VirtualBox: ~/Desktop
csr@csr-VirtualBox:~/Desktop$ ls
a.txt  b.txt
csr@csr-VirtualBox:~/Desktop$ mv a.txt b.txt Module/
mv: target 'Module/' is not a directory
csr@csr-VirtualBox:~/Desktop$ mkdir Module/
csr@csr-VirtualBox:~/Desktop$ mv a.txt b.txt Module/
csr@csr-VirtualBox:~/Desktop$ cat Module/
cat: Module/: Is a directory
csr@csr-VirtualBox:~/Desktop$ ls Module/
a.txt  b.txt
```

- Can also used to rename directory

$ mv nameofthedirectory newname

- mv replaces the filename in the existing directory entry with the new name. It doesn't create a copy of the file; it renames it.

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.

```
csr@csr-VirtualBox:~/Desktop$ ls
a.txt  b.txt
csr@csr-VirtualBox:~/Desktop$ cat a.txt b.txt
RNSIT, Bengaluru
Hi Hello
Welcome to RNSIT
ISE Department
csr@csr-VirtualBox:~/Desktop$ mv -i a.txt b.txt
mv: overwrite 'b.txt'? y
csr@csr-VirtualBox:~/Desktop$ ls
b.txt
csr@csr-VirtualBox:~/Desktop$ cat b.txt
RNSIT, Bengaluru
```

2. **-f (Force): mv** prompts for confirmation overwriting the destination file if a file is write protected. The **-f** option overrides this minor protection and overwrite the destination file forcefully and delete the source file.

```
csr@csr-VirtualBox:~/Desktop$ cat b.txt
RNSIT, Bengaluru
csr@csr-VirtualBox:~/Desktop$ ls -l b.txt
-r--r--r-- 1 csr csr 17 Sep 30 12:16 b.txt
csr@csr-VirtualBox:~/Desktop$ mv a.txt b.txt
mv: replace 'b.txt', overriding mode 0444 (r--r--r--)? ls
csr@csr-VirtualBox:~/Desktop$ ls
a.txt  b.txt
csr@csr-VirtualBox:~/Desktop$ mv -f a.txt b.txt
csr@csr-VirtualBox:~/Desktop$ ls
b.txt
csr@csr-VirtualBox:~/Desktop$ cat b.txt
RNSIT, Bengaluru
```

3. **-n (no-clobber):** With **-n** option, **mv** prevent an existing file from being overwritten.
4. **-b(backup):** With this option it is easier to take a backup of an existing file that will be overwritten as a result of the **mv** command. This will create a backup file with the tilde character(~) appended to it.
5. **-version:** This option is used to display the version of mv which is currently running on your system.

```
csr@csr-VirtualBox:~/Desktop$ ls
a.txt  b.txt
csr@csr-VirtualBox:~/Desktop$ mv -b a.txt b.txt
csr@csr-VirtualBox:~/Desktop$ ls
b.txt  b.txt~
csr@csr-VirtualBox:~/Desktop$ mv --version
mv (GNU coreutils) 8.30
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <https://gnu.org/licenses/gpl.ht
ml>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Written by Mike Parker, David MacKenzie, and Jim Meyering.
```


GANESH XEROX RNSIT

- **rm: Deleting Files**

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] [filename]

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.

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

3. -r (Recursive Deletion): With **-r(or -R)** option rm command performs a tree-walk and will delete all the files and sub-directories recursively of the parent directory. At each stage it deletes everything it finds. Normally, **rm** wouldn't delete the directories but when used with this option, it will delete.

4. –version: This option is used to display the version of **rm** which is currently running on your system.

```
csr@csr-VirtualBox:~/Desktop$ ls
a.txt  b.txt  c.txt  d.txt  e.txt
csr@csr-VirtualBox:~/Desktop$ rm a.txt
csr@csr-VirtualBox:~/Desktop$ ls
b.txt  c.txt  d.txt  e.txt
csr@csr-VirtualBox:~/Desktop$ rm b.txt c.txt
csr@csr-VirtualBox:~/Desktop$ ls
d.txt  e.txt
csr@csr-VirtualBox:~/Desktop$ rm -i d.txt
rm: remove regular file 'd.txt'? n
csr@csr-VirtualBox:~/Desktop$ ls
d.txt  e.txt
csr@csr-VirtualBox:~/Desktop$ ls -l
total 8
-rw-rw-r-- 1 csr csr 26 Sep 30 14:07 d.txt
-rw-rw-r-- 1 csr csr 16 Sep 30 14:08 e.txt
csr@csr-VirtualBox:~/Desktop$ chmod a-w *.txt
csr@csr-VirtualBox:~/Desktop$ ls -l
total 8
-r--r--r-- 1 csr csr 26 Sep 30 14:07 d.txt
-r--r--r-- 1 csr csr 16 Sep 30 14:08 e.txt
```

```
csr@csr-VirtualBox:~/Desktop$ rm d.txt
rm: remove write-protected regular file 'd.txt'? y
csr@csr-VirtualBox:~/Desktop$ ls
e.txt
csr@csr-VirtualBox:~/Desktop$ rm -f e.txt
csr@csr-VirtualBox:~/Desktop$ ls
csr@csr-VirtualBox:~/Desktop$ mkdir A A/B A/C
csr@csr-VirtualBox:~/Desktop$ ls
A
csr@csr-VirtualBox:~/Desktop$ cd A
csr@csr-VirtualBox:~/Desktop/A$ ls
B  C
csr@csr-VirtualBox:~/Desktop/A$ ls B
a.txt  b.txt
csr@csr-VirtualBox:~/Desktop/A$ ls C
c.txt  d.txt
csr@csr-VirtualBox:~/Desktop/A$ rm *
rm: cannot remove 'B': Is a directory
rm: cannot remove 'C': Is a directory
```

```
csr@csr-VirtualBox:~/Desktop/A$ rm -r *
csr@csr-VirtualBox:~/Desktop/A$ ls
csr@csr-VirtualBox:~/Desktop/A$ rm --version
rm (GNU coreutils) 8.30
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <https://gnu.org/licenses/gpl.ht
ml>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Written by Paul Rubin, David MacKenzie, Richard M. Stallman,
and Jim Meyering.
```

- **cp: Copying A File**

The cp command copies a file or a group of files. It creates an exact image of the file on the disk with a different name. The syntax takes two filename to be specified in the command line.

Syntax:

cp [option] source destination

cp [option] source directory

cp [option] source-1 source-2 source-3 source-n directory

1. Two file names: If the command contains two file names, then it copies the contents of 1st file to the 2nd file. If the 2nd file doesn't exist, then first it creates one and content is copied to it. But if it existed then it is simply overwritten without any warning. So be careful when you choose destination file name.

2. One or more arguments: If the command has one or more arguments, specifying file names and following those arguments, an argument specifying directory name then this command copies each source file to the destination directory with the same name, created if not existed but if already existed then it will be overwritten.

Suppose there is a directory named **A** having a text file a.txt, b.txt and a directory name **B** in which we are going to copy all files.

```
csr@csr-VirtualBox:~/Desktop$ ls
A  a.txt  b.txt  c.txt  d.txt  e.txt
csr@csr-VirtualBox:~/Desktop$ cp a.txt new.txt
csr@csr-VirtualBox:~/Desktop$ ls
A  a.txt  b.txt  c.txt  d.txt  e.txt  new.txt
csr@csr-VirtualBox:~/Desktop$ ls A
csr@csr-VirtualBox:~/Desktop$ cp a.txt b.txt A
csr@csr-VirtualBox:~/Desktop$ ls A
a.txt  b.txt
```

3. Two directory names: If the command contains two directory names, cp copies all files of the source directory to the destination directory, creating any files or directories needed. This mode of operation requires an additional option, typically R, to indicate the recursive copying of directories.

cp behavior depends upon whether Dest_directory is exist or not. If the Dest_directory doesn't exist, cp creates it and copies content of Src_directory recursively as it is. But if Dest_directory exists then copy of Src_directory becomes sub-directory under Dest_directory.

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.

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.

```
csr@csr-VirtualBox:~/Desktop$ ls
A  a.txt  b.txt  c.txt  d.txt  e.txt  new.txt
csr@csr-VirtualBox:~/Desktop$ cat a.txt
RNSIT, Bengaluru
csr@csr-VirtualBox:~/Desktop$ cat b.txt
Hi Hello
Welcome to RNSIT
ISE Department
csr@csr-VirtualBox:~/Desktop$ cp -i a.txt b.txt
cp: overwrite 'b.txt'? y
csr@csr-VirtualBox:~/Desktop$ cat a.txt
RNSIT, Bengaluru
csr@csr-VirtualBox:~/Desktop$ cat b.txt
RNSIT, Bengaluru
csr@csr-VirtualBox:~/Desktop$ cp -b a.txt b.txt
csr@csr-VirtualBox:~/Desktop$ ls
A  a.txt  b.txt  b.txt~  c.txt  d.txt  e.txt  new.txt
```

3. -f(force): If the system is unable to open destination file for writing operation because the user doesn't have writing permission for this file then by using **-f** option with **cp** command, destination file is deleted first and then copying of content is done from source to destination file.

4. -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 A directory containing many files, directories into **unix** directory(not exist).

```
csr@csr-VirtualBox:~/Desktop$ ls -l a.txt
-r--r--r-- 1 csr csr 17 Sep 30 12:16 a.txt
csr@csr-VirtualBox:~/Desktop$ cp b.txt a.txt
cp: cannot create regular file 'a.txt': Permission denied
csr@csr-VirtualBox:~/Desktop$ cp -f b.txt a.txt
csr@csr-VirtualBox:~/Desktop$ ls A/
a.txt  b.txt
csr@csr-VirtualBox:~/Desktop$ cp A unix
cp: -r not specified; omitting directory 'A'
csr@csr-VirtualBox:~/Desktop$ cp -r A unix
csr@csr-VirtualBox:~/Desktop$ ls unix
a.txt  b.txt
```

5. -p(preserve): With **-p** option **cp** preserves the following characteristics of each source file in the corresponding destination file: the time of the last data modification and the time of the last access, the ownership (only if it has permissions to do this), and the file permission-bits.

```
csr@csr-VirtualBox:~/Desktop$ ls -l a.txt
-rw-rw-r-- 1 csr csr 17 Sep 30 14:42 a.txt
csr@csr-VirtualBox:~/Desktop$ cp a.txt b.txt
csr@csr-VirtualBox:~/Desktop$ ls -l b.txt
-rw-rw-r-- 1 csr csr 17 Sep 30 14:45 b.txt
csr@csr-VirtualBox:~/Desktop$ ls -l a.txt
-rw-rw-r-- 1 csr csr 17 Sep 30 14:42 a.txt
csr@csr-VirtualBox:~/Desktop$ ls -l b.txt
-rw-rw-r-- 1 csr csr 17 Sep 30 14:45 b.txt
csr@csr-VirtualBox:~/Desktop$ cp -p a.txt b.txt
csr@csr-VirtualBox:~/Desktop$ ls -l b.txt
-rw-rw-r-- 1 csr csr 17 Sep 30 14:42 b.txt
```

Copying using * wildcard: The star wildcard represents anything i.e. all files and directories. Suppose we have many text document in a directory and wants to copy it another directory, it takes lots of time if we copy files 1 by 1 or command becomes too long if specify all these file names as the argument, but by using * wildcard it becomes simple.

```
csr@csr-VirtualBox:~/Desktop$ ls A
csr@csr-VirtualBox:~/Desktop$ ls
A  a.txt  b.txt  b.txt~  c.txt  d.txt  e.txt  new.txt
csr@csr-VirtualBox:~/Desktop$ cp *.txt A
csr@csr-VirtualBox:~/Desktop$ ls A
a.txt  b.txt  c.txt  d.txt  e.txt  new.txt
```

- **wc: Counting Lines, Words and Characters**
  - wc command performs Word-counting including counting of lines and characters in a specified file. It takes one or more filename as arguments and displays a four columnar output.

    Syntax: $ wc filename
  - Line: Any group of characters not containing a newline
  - Word: group of characters not containing a space, tab or newline
  - Character: smallest unit of information, and includes a space, tab and newline

    wc offers 3 options to make a specific count. –l option counts only number of lines, - w and –c options count words and characters, respectively.

```
csr@csr-VirtualBox:~/Desktop$ ls
a.txt  b.txt  b.txt~  c.txt  d.txt  e.txt
csr@csr-VirtualBox:~/Desktop$ cat b.txt
Hi Hello
Welcome to RNSIT
ISE Department
csr@csr-VirtualBox:~/Desktop$ wc -l b.txt
3 b.txt
csr@csr-VirtualBox:~/Desktop$ wc -w b.txt
7 b.txt
csr@csr-VirtualBox:~/Desktop$ wc -c b.txt
41 b.txt
csr@csr-VirtualBox:~/Desktop$ wc -m b.txt
41 b.txt
csr@csr-VirtualBox:~/Desktop$ wc -L b.txt
16 b.txt
```

- Multiple filenames, wc produces a line for each file, as well as a total count.

```
csr@csr-VirtualBox:~/Desktop$ wc *.txt
  1   2  17 a.txt
  3   7  41 b.txt
  0   0   0 chap11.txt
  0   0   0 chap12.txt
  1   2  10 c.txt
  1   4  26 d.txt
  1   3  16 e.txt
  7  18 110 total
```

➢ **od: Displaying Data in Octal**

od command 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]

Options:

1. **-b:** It displays the contents of input in octal format. The first column in the output of od represents the byte offset in file.
2. **-c:** It displays the contents of input in character format.
3. **-An:** It displays the contents of input in character format but with no offset information.
4. **-A:** It displays the contents of input in different format by concatenation some special character with -A.
   - **-Ax for Hexadecimal format (we concatenate x with -A)**
   - **-Ao for Hexadecimal format (we concatenate o with -A)**
   - **-Ad for Hexadecimal format(we concatenate d with -A)**
5. **–:** Accept input from command line.

Ex 1: $ more odfile this file is an example for od command created manually with content below:

White space includes a
The ^G character rings a bell
The ^L character rings a page

```
csr@csr-VirtualBox:~/Desktop$ gedit odfile
csr@csr-VirtualBox:~/Desktop$ more odfile
White space includes a
The ^G character rings a bell
The ^L character skips a page
csr@csr-VirtualBox:~/Desktop$ od -b odfile
0000000 127 150 151 164 145 040 163 160 141 143 145 040 151 156 143 154
0000020 165 144 145 163 040 141 012 124 150 145 040 136 107 040 143 150
0000040 141 162 141 143 164 145 162 040 162 151 156 147 163 040 141 040
0000060 142 145 154 154 012 124 150 145 040 136 114 040 143 150 141 162
0000100 141 143 164 145 162 040 163 153 151 160 163 040 141 040 160 141
0000120 147 145 012
0000123
```

```
csr@csr-VirtualBox:~/Desktop$ od -bc odfile
0000000 127 150 151 164 145 040 163 160 141 143 145 040 151 156 143 154
          W   h   i   t   e       s   p   a   c   e       i   n   c   l
0000020 165 144 145 163 040 141 012 124 150 145 040 136 107 040 143 150
          u   d   e   s       a  \n   T   h   e       ^   G       c   h
0000040 141 162 141 143 164 145 162 040 162 151 156 147 163 040 141 040
          a   r   a   c   t   e   r       r   i   n   g   s       a
0000060 142 145 154 154 012 124 150 145 040 136 114 040 143 150 141 162
          b   e   l   l  \n   T   h   e       ^   L       c   h   a   r
0000100 141 143 164 145 162 040 163 153 151 160 163 040 141 040 160 141
          a   c   t   e   r       s   k   i   p   s       a       p   a
0000120 147 145 012
          g   e  \n
0000123
```

Ex 2: input.txt is the input file for od command which contains the digits from 101 to 110 as input.

```
csr@csr-VirtualBox:~/Desktop$ cat input.txt
101
102
103
104
105
106
107
108
109
110
csr@csr-VirtualBox:~/Desktop$ od -b input.txt
0000000 061 060 061 012 061 060 062 012 061 060 063 012 061 060 064 012
0000020 061 060 065 012 061 060 066 012 061 060 067 012 061 060 070 012
0000040 061 060 071 012 061 061 060 012
0000050
csr@csr-VirtualBox:~/Desktop$ od -c input.txt
0000000   1   0   1  \n   1   0   2  \n   1   0   3  \n   1   0   4  \n
0000020   1   0   5  \n   1   0   6  \n   1   0   7  \n   1   0   8  \n
0000040   1   0   9  \n   1   1   0  \n
0000050
```

```
csr@csr-VirtualBox:~/Desktop$ od -An -c input.txt
   1   0   1  \n   1   0   2  \n   1   0   3  \n   1   0   4  \n
   1   0   5  \n   1   0   6  \n   1   0   7  \n   1   0   8  \n
   1   0   9  \n   1   1   0  \n
csr@csr-VirtualBox:~/Desktop$ od -Ax -c input.txt
000000   1   0   1  \n   1   0   2  \n   1   0   3  \n   1   0   4  \n
000010   1   0   5  \n   1   0   6  \n   1   0   7  \n   1   0   8  \n
000020   1   0   9  \n   1   1   0  \n
000028
csr@csr-VirtualBox:~/Desktop$ od -Ao -c input.txt
0000000   1   0   1  \n   1   0   2  \n   1   0   3  \n   1   0   4  \n
0000020   1   0   5  \n   1   0   6  \n   1   0   7  \n   1   0   8  \n
0000040   1   0   9  \n   1   1   0  \n
0000050
csr@csr-VirtualBox:~/Desktop$ od -Ad -c input.txt
0000000   1   0   1  \n   1   0   2  \n   1   0   3  \n   1   0   4  \n
0000016   1   0   5  \n   1   0   6  \n   1   0   7  \n   1   0   8  \n
0000032   1   0   9  \n   1   1   0  \n
0000040
```

```
csr@csr-VirtualBox:~/Desktop$ od -c -
welcome to rnsit
0000000   w   e   l   c   o   m   e       t   o       r   n   s   i   t
0000020  \n
0000021
csr@csr-VirtualBox:~/Desktop$ od -bc input.txt
0000000 061 060 061 012 061 060 062 012 061 060 063 012 061 060 064 012
          1   0   1  \n   1   0   2  \n   1   0   3  \n   1   0   4  \n
0000020 061 060 065 012 061 060 066 012 061 060 067 012 061 060 070 012
          1   0   5  \n   1   0   6  \n   1   0   7  \n   1   0   8  \n
0000040 061 060 071 012 061 061 060 012
          1   0   9  \n   1   1   0  \n
0000050
csr@csr-VirtualBox:~/Desktop$
```

- Some of the representation:
- The tab character, [ctrl-i], is shown as \t and the octal vlaue 011
- The bell character , [ctrl-g] is shown as 007, some system show it as \a
- The form feed character,[ctrl-l], is shown as \f and 014
- The LF character, [ctrl-j], is shown as \n and 012
- Od makes the newline character visible too.

MODULE-2

Topics

File attributes and permissions: The ls command with options. Changing file permissions: the relative and absolute permissions changing methods. Recursively changing file permissions. Directory permissions.
The shells interpretive cycle: Wild cards. Removing the special meanings of wild cards. Three standard files and redirection. **Connecting commands:** Pipe. Basic and Extended regular expressions. The grep, egrep. Typical examples involving different regular expressions.
Shell programming: Ordinary and environment variables. The. profile. Read and readonly commands. Command line arguments. exit and exit status of a command. Logical operators for conditional execution. The test command and its shortcut. The if, while, for and case control statements. The set and shift commands and handling positional parameters. The here (<<) document and trap command. Simple shell program examples.

Chapter 3 File Attributes and Permissions

3.1 Listing File Attributes: ls -l

ls command is used to obtain a list of all filenames in the current directory. The output in UNIX lingo is often referred to as the listing. Sometimes we combine this option with other options for displaying other attributes, or ordering the list in a different sequence.

ls look up the file's inode to fetch its attributes. The list is preceded by the words total 72, which indicates that a total of 72 blocks are occupied by these files on disk, each block consisting of 512 bytes (1024 in linux).

It lists seven attributes of all files in the current directory and they are:

- File type and Permissions: The first column shows the type and permissions associated with each file. The first character in this column is mostly a -, which indicates that the file is an ordinary one. This is, however, not so for the directories myfolder, documents etc where you see a d at the same position. You then see a series of characters that can take the values r, w, x and -. In the unix system, a file can have three types of permissions – read, write and execute.
- Links: The second column indicates the number of links associated with the file. This is actually the number of filenames maintained by the system of that file. Unix lets a file have as many names as you want it to have, even though there is a single file on disk.
- Ownership: When you create a file, you automatically become its owner. The third column shows Chandana as the owner of all of these files. The owner has full authority to tamper with a file's contents and permissions.
- Group ownership: When opening a user account, the system administrator also assigns the user to same group. The fourth column represents the group owner of the file. Its desirable that the group have a set of privileges distinct from others as well as the owner.
- File size: The fifth column shows the size of the file in bytes i.e the amount of data it contains. It is only a character count of the file and not a measure of disk space that it occupies.

- Last modification date and time: The sixth, seventh and eight columns indicate the last modification time of the file, which is stored to the nearest second. A file is said to be modified only if its contents have changed not for the changing of permissions, ownership of the file.
- File name: The last column displays the filenames arranged in ASCII collating sequence.

```
chandana@chandana-VirtualBox:~$ ls
combine.txt  Desktop  Documents  Downloads  Music  myfolder  out.txt  Pictures  Public  snap  Templates  UNIX1  Videos
chandana@chandana-VirtualBox:~$ ls -l
total 52
-rw-rw-r-- 1 chandana chandana   85 Sep 10 10:05 combine.txt
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Desktop
drwxr-xr-x 3 chandana chandana 4096 Sep  4 10:18 Documents
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Downloads
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Music
drwxrwxr-x 3 chandana chandana 4096 Sep 10 10:11 myfolder
-rw-rw-r-- 1 chandana chandana  274 Sep 10 10:08 out.txt
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Pictures
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Public
drwxr-xr-x 3 chandana chandana 4096 Sep 14 10:31 snap
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Templates
drwxrwxr-x 2 chandana chandana 4096 Sep  9 19:12 UNIX1
drwxr-xr-x 2 chandana chandana 4096 Sep  2 19:58 Videos
```

Listing Directory Attributes

Directories are easily identified in the listing by the first character of the first column, which here shows a d. The significance of the attributes of a directory differs a good deal from an ordinary file. To see the attributes of a directory rather than the files contained in it, use ls –ld with the directory name. Note that simply using ls –d will not list all subdirectories in the current directory. Strange though it may seem, ls has no option to list only directories.

```
csr@csr-VirtualBox:~/Desktop$ ls -ld
drwxr-xr-x 5 csr csr 4096 Oct  1 13:35 .
csr@csr-VirtualBox:~/Desktop$ ls -ld A B
drwxrwxr-x 2 csr csr 4096 Oct  1 10:41 A
drwxrwxr-x 2 csr csr 4096 Oct  1 10:45 B
csr@csr-VirtualBox:~/Desktop$ id
uid=1000(csr) gid=1000(csr) groups=1000(csr),4(adm),24(cdrom),27(sudo),30(d
ip),46(plugdev),120(lpadmin),131(lxd),132(sambashare)
```

3.2 File Ownership

When you create a file, you become its owner. Every owner is attached to a group owner. Several users may belong to a single group, but the privileges of the group are set by the owner of the file and not by the group members. When the system administrator creates a user account, he has to assign these parameters to the user:

The user-id (UID) – both its name and numeric representation
The group-id (GID) – both its name and numeric representation

File Permissions

UNIX follows a three-tiered file protection system that determines a file's access rights. It is displayed in the following format:

Filetype owner (rwx) groupowner (rwx) others (rwx)

For Example:

-rwxr-xr-- 1 csr education 20500 may 10 19:21 chap02

r w x	r - x	r - -
owner/user	group owner	others

The first group has all three permissions. The file is readable, writable and executable by the owner of the file.

The second group has a hyphen in the middle slot, which indicates the absence of write permission by the group owner of the file.

The third group has the write and execute bits absent. This set of permissions is applicable to others.

You can set different permissions for the three categories of users – owner, group and others. It's important that you understand them because a little learning here can be a dangerous thing. Faulty file permission is a sure recipe for disaster.

Changing File Permissions: chmod

A file or a directory is created with a default set of permissions, which can be determined by umask. Let us assume that the file permission for the created file is -rw-r--r--. Using **chmod** command, we can change the file permissions and allow the owner to execute his file. The command can be used in two ways:

- In a relative manner by specifying the changes to the current permissions.
- In an absolute manner by specifying the final permissions.

Relative Permissions (symbolic mode)

chmod only changes the permissions specified in the command line and leaves the other permissions unchanged. Its syntax is:

chmod category operation permission filename(s)

chmod takes an expression as its argument which contains:

- user category (user, group, others)
- operation to be performed (assign or remove a permission)
- type of permission (read, write, execute)

Category	operation	permission
u - user	+ assign	r - read
g - group	- remove	w - write

GANESH KOREKI RNSIT

o - others = absolute x - execute

a - all (ugo)

Let us discuss some examples:

1. Initially,

-rw-r--r-- 1 csr education 1906 sep 23:38 xstart

chmod u+x xstart

-rwxr--r-- 1 csr education 1906 sep 23:38 xstart

The command assigns (+) execute (x) permission to the user (u), other permissions remain unchanged.

chmod ugo+x xstart or

chmod a+x xstart or

chmod +x xstart

-rwxr-xr-x 1 csr education 1906 sep 23:38 xstart

chmod accepts multiple file names in command line

chmod u+x note note1 note3

2. Let initially,

-rwxr-xr-x 1 csr education 1906 sep 23:38 xstart

chmod go-r xstart

Then, it becomes

-rwx--x--x 1 csr education 1906 sep 23:38 xstart

Absolute Permissions

Here, we need not to know the current file permissions. We can set all nine permissions explicitly. A string of three octal digits is used as an expression. The permission can be represented by one octal digit for each category. For each category, we add octal digits. If we represent the permissions of each category by one octal digit, this is how the permission can be represented:

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

Octal	Permissions	Significance
0	- - -	no permissions
1	- - x	execute only
2	- w -	write only
3	- w x	write and execute

SRI SHETTY & ... BOOK STALL, 2nd Floor, OPP. Jain College, Below ICICI Bank, Dr. B.R. Ambedkar Road, Ashok Layout, Bangalore - 560 050. Ph: 9620099337, 7676070931.

4	r - -	read only
5	r - x	read and execute
6	r w -	read and write
7	r w x	read, write and execute

We have three categories and three permissions for each category, so three octal digits can describe a file's permissions completely. The most significant digit represents user and the least one represents others. chmod can use this three-digit string as the expression.

Using relative permission, we have,

chmod a+rw xstart

Using absolute permission, we have,

chmod 666 xstart

chmod 644 xstart

chmod 761 xstart

will assign all permissions to the owner, read and write permissions for the group and only execute permission to the others.

777 signify all permissions for all categories, but still we can prevent a file from being deleted. 000 signifies absence of all permissions for all categories, but still we can delete a file. It is the directory permissions that determine whether a file can be deleted or not. Only owner can change the file permissions. User cannot change other user's file's permissions. But the system administrator can do anything.

The Security Implications

Let the default permission for the file xstart is

-rw-r--r--

chmod u-rw, go-r xstart or

chmod 000 xstart

This is simply useless but still the user can delete this file

On the other hand,

chmod a+rwx xstart

.chmod 777 xstart

-rwxrwxrwx

The UNIX system by default, never allows this situation as you can never have a secure system. Hence, directory permissions also play a very vital role here

We can use chmod Recursively.

chmod -R a+x shell_scripts

This makes all the files and subdirectories found in the shell_scripts directory, executable by all users. When you know the shell meta characters well, you will appreciate that the * doesn't match filenames beginning with a dot. The dot is generally a safer but note that both commands change the permissions of directories also.

3.3 Recursively Changing File Permissions

It is possible to make chmod descend a directory hierarchy & apply the expression to every file & subdirectory it finds. This is done with –R (recursive) option:

$chmod –R a+x ise

This makes all files & subdirectories found in the tree-walk that commences from the directory ise, executable by all users. (a represents all)

$chmod –R 755 .	applies to hidden files also.
$chmod –R a+x *	Leaving hidden files ,it is applied on all other files (because the wildcard * doesnot match filenames beginning with a dot)

3.4 Directory permissions

Directories also have their own permissions & the significance of these permissions differ a great deal from those of ordinary files. We must be aware that access rights to an ordinary file are also influenced by the permissions of the directory under which they appear. It's possible that a file can't be accessed even though it has read permission, can be removed even when it's write-protected.

The usual default permissions of a directory on our system are rwrr-xr-x (or 755). A directory must never be writable by group or others.

Example:

mkdir c_progs

ls –ld c_progs

drwxr-xr-x 2 csr education 512 may 9 09:57 c_progs

If a directory has written permission for group and others also, be assured that every user can remove every file in the directory. As a rule, you must not make directories universally writable unless you have definite reasons to do so.

Chapter 4 The Shells Interpretive Cycle

What is a shell?

The shell is the utility that processes user requests. It is an agency that sits between the user and the UNIX system. When the user type in a command at his terminal, the shell interprets the command and calls the program that it is needed. It is the one which understands all user directives and carries them out. It processes the commands issued by the user.

The shell and its interpretive cycle

The shell sits between the user and the operating system, acting as a command interpreter. It reads user terminal input and translates the commands into actions taken by the system. When the user log into the system, is given a default shell. When the shell starts up, it reads its startup files and may set environment variables, command search paths, and command aliases, and executes any commands specified in these files. The original shell was the Bourne shell, ***sh***. Every Unix platform will either have the Bourne shell, or a Bourne compatible shell available.

Some of the well known Unix shells are:

- The Korn shell, ***ksh***, by David Korn,
- The Bourne Again SHell, ***bash***, from the Free Software Foundations GNU project, both based on Bourne shell.
- C shell, ***csh***, by Bill Joy and its derivatives are
- The T-C shell, ***tcsh***, and the extended C shell, ***cshe***.

Even though the shell appears not to be doing anything meaningful when there is no activity at the terminal, it swings into action the moment we key in something.

The following activities are typically performed by the shell in its interpretive cycle:

- The shell issues the prompt and waits for the user to enter a command.
- After a command is entered, the shell scans the command line for metacharacters (*,?) and expands abbreviations (like the * in rm *) to recreate a simplified command line.
- It then passes on the command line to the kernel for execution.
- The shell waits for the command to complete and normally can't do any work while the command is running.
- After the command execution is complete, the prompt reappears and the shell returns to its waiting role to start the next cycle. Now the user is free to enter another command.

4.1 Wild Cards

A pattern is framed using ordinary characters and a metacharacters.(like *,?) using well-defined rules. The pattern can then be used as an argument to the command, and the shell will expand it suitably before the command is executed.

The metacharacters that are used to construct the generalized pattern for matching filenames belong to a category called wild-cards. The following table lists the shell's wild-cards:

Wild-Card	Matches
*	Any number of characters including none
?	A single character
[ijk]	A single character – either an i, j or k
[x-z]	A single character that is within the ASCII range of characters x and x
[!ijk]	A single character that is not an i,j or k
[!x-z]	A single character that is not within the ASCII range of the characters x and x (Not in C Shell)
{pat1,pat2...}	Pat1, pat2, etc. (Not in Bourne shell)

The * and ?

The metacharacter *, is one of the shell's wild-card set. It matches any number of characters(including none). It thus matches all filenames specified in the previous command line which can now be shortened in this way.

To list all files that begin with *chap*, use

```
$ ls chap*
chap chap01 chap02 chap03 chap04 chap15 chap16 chap17 chapx chapy chapz
```

Observe that chap* also matches the string chap. When the shell encounters this command line, it identifies the * immediately as a wild-card. It then looks in the current directory and recreates the command line as below from the filenames that match the pattern chap*:

```
ls chap chap01 chap02 chap03 chap04 chap15 chap16 chap17 chapx chapy chapz
```

The shell now hands over this command line to the kernel which uses its process creation facilities to run the command. Be careful when you use the * with rm to remove files, this deletes all the files.

The next wild-card is the ?, which matches a single character. When used with chap (chap?), the shell matches all five-character filenames beginning with chap. Appending another creates the pattern chap??, which matches six-character filenames.

```
$ls chap?
chapx chapy chapz

$ls chap??
Chap01 chap02 chap03 chap04 chap15 chap16 chap17.
```

Note: Both * and ? operate with some restrictions. The * doesn't match all files beginning with a . (dot) or the / of a pathname. If you wish to list all hidden filenames in your directory having at least three characters after the dot, the dot must be matched explicitly.

```
$ ls .???*
```

However, if the filename contains a dot anywhere else, it need not be matched explicitly. Similarly, these characters don't match the / in a pathname. So, you cannot use $ cd /usr?local to change to /usr/local.

The character class

It compromises a set of characters enclosed by the rectangular brackets, [and], but it matches a single character in the class. The pattern [abcd] is a character class, and it matches a single character – an a, b, c or d. This can be combined with any string or another wild-card expression, so selecting chap01, chap02 and chap04 now becomes simpler.

$ls chap0[124]
Chap01 chap02 chap04

Range specification is also possible inside the class with a -; the two characters on either side of it form the range of characters to be matched. Here are two examples:

ls chap0[1-4]	Lists chap01, chap02, chap03 and chap04
ls chap[x-z]	Lists chapx, chapy and chapz

A valid range specification requires that the character on the left have a lower ASCII value than the one on the right.

Negating the character class(!)

We can use the ! as the first character in the class to negate the class. The two examples shown below:

- *.[!co] Matches all filenames with a single-character extension but not the .c or .o files.
- [!a-zA-Z]* Matches all filenames that don't begin with an alphabetic character.

Matching totally dissimilar patterns

Delimit the patterns with a comma, and then put curly braces around them (no spaces).

cp $HOME/prog_sources/*.c{c,java} This copies C and Java source programs from another directory.

Using the curly brace form, you can also access multiple directories.

cp /home/csr/{project,html,scripts}/* This copies all files from three directories(project, html and scripts) to the current directory.

Rounding up

Some of the wild-card characters have different meanings depending on where they are placed in the pattern. The * and ? lose their meaning when used inside the class, and are matched literally. Similarly, - and ! also lose their significance when placed outside the class.

Some examples shown below

ls *.c	Lists all files with extension .c
mv * ../bin	Moves all files to bin subdirectory of parent directory
cp foo foo*	Copies foo to foo*(* loses meaning here)
cp ?????? progs	Copies to progs directory all files with six-character names.

GANESH XEROX PRINTING & PRESENTATIONS

lp note[0-1][0-9]	Prints file note00, note01, ...through note19
rm *.[!l][!o][!g]	Removes all files with three-character extensions except the ones with the .log extension.
cp -r /home/csr/{include,lib,bin}	Copies recursively the three directories, include, lib ad bin from /home/csr to the current directory.

4.2 Removing the Special Meaning of Metacharacters

Some files may contain special character in its filename. Ex. Chap* is the filename and if we try to remove this file by using rm chap* would be dangerous. It would remove the other filenames beginning with chap also. We must be able to protect all special characters (including wildcards) so the shell is not able to interpret them. The shell provides two solutions to prevent it's own interference.

Escaping: providing a \ (backlash) before the wild-card to remove its special meaning.

Quoting: enclosing the wild-card, or even the entire pattern, within quotes (like 'chap*'). Anything within these quotes are left alone by the shell and not interpreted.

In most cases we can use either mechanism but some situations prefer the use of quotes, sometimes we also need to make a distinction between single and double quotes.

Escaping: Placing a \ immediately before a metacharacter turns off its special meaning. For instance in the pattern *, the backslash \ tell the shell that the asterisk has to be matched literally instead of being interpreted as a metacharacter. This means that we can remove the file Chap* without affecting the other file names that also begin with Chap by using

rm chap * Doen't remove chap1, chap2

The \ suppresses the wild-card nature of the *, thus preventing the shell from performing filename expansion on it. This feature is known as escaping.

To consider another example, if you have files chap01, chap02 and chap03 in your current directory, and still dare to create a file chap0[1-3] by using

echo > chap0[1-3] creates a file chap0[1-3]

then you should escape the two rectangular brackets when accessing the file.

$ls chap0\[1-3\] must escape the [and]
$rm chap0\[1-3\]
$ls chap0\[1-3\]
Chap0[1-3] not found

Escaping the space: Apart from metacharacters, there are other characters that are special – like the space character. The shell uses it to delimit command line arguments. So to remove the file My Document.doc which has a space embedded, a similar reasoning should be followed.

rm My\ Document.doc

Escaping the \ itself: Sometimes we may need to interpret the \ itself literally. We need another \ before it, that's all

**Secho **

\

$echo The newline character is \\n

The newline character is \\n

Escaping the newline character: The newline character is also special, it marks the end of the command line. Some command lines that use several arguments can be long enough to overflow to the next line.

**$find /usr/local/bin /usr/bin -name "*.pl" -mtime +7 -size +1024 ** [Enter]

>-size -2048 -atime +25 -print

This is the find command at work, a command often used with several arguments. The \ here escapes the meaning of the newline character generated by [Enter]. It also produces the second prompt (which could be a > or a ?), which indicates that the command line is incomplete. For better readability, you should split them into multiple lines wherever possible.

Quoting: When a command argument is enclosed in quotes, the meanings of all enclosed special characters are turned off.

echo '\'	**Displays a **
rm 'chap*'	**Removes file chap***
rm "My Document.doc"	**Removes file My Document.doc**

Quoting is often a better solution. The below example shows the protection of four special characters using single quotes.

$echo 'The characters |, <, > and $ are also special'

The characters |, <, > and $ are also special.

We could have used escaping here, but then we would need to use four \s in front of each of these four metacharacters. We used single quotes because they protect all special characters (except the single quote). Double quotes are more permissive, they don't protect (apart from the double quotes itself) the $ and the `(backquote)

$echo "command substitution uses `` while TERM is evaluated using $TERM"

command substitution uses while TERM is evaluated using vt100

Observe that the pair of backquote characters(``) and the variable $TERM have been interpreted by the shell inside double quotes. $TERM is vt100 for this terminal, and `` evaluated to a null command. Now try out the same example using single quotes.

$echo 'command substitution uses `` while TERM is evaluated using $TERM'

command substitution uses `` while TERM is evaluated using $TERM

4.3 Redirection: The three standard files

The shell associates three files with the terminal – two for display and one for the keyboard. These files are streams of characters which many commands see as input and output. When a user logs in, the shell makes available three files representing three streams. Each stream is associated with a default device:

Standard input: The file (stream) representing input, connected to the keyboard.

Standard output: The file (stream) representing output, connected to the display.

Standard error: The file (stream) representing error messages that generates from the command or shell, connected to the display.

GANESH XEROX RNSIT ISE

Standard input:

The standard input can represent three input sources:

- The keyboard- the default source,
- A file using redirection symbol(<) and
- Another program using a pipeline(|).

When you use wc without an argument and have no special symbols like the < and | in the command line, wc obtains its input from the default source. You have to provide this input from the keyboard and mark the end of input with [ctrl-d].

```
$wc
UNIX is awesome.
[ctrl-d]
1    3    15
```

This situation is similar to the one encountered when creating a file with cat. Enter the text and signify the end of input with [ctrl-d]. wc, which takes the stream from standard input immediately counts lines, words and characters.

```
$wc < file
1    3    15
```

It can reassign the standard input file to a disk file. This means it can redirect the standard input to originate from a file on disk. This reassignment or redirection requires the < symbol as shown above.

Standard output:

All command outputs are displayed on the terminal are actually written to standard output file as stream of characters and not directly to the terminal

The standard output can represent three possible destinations:

- The terminal- the default destination,
- A file using the redirection symbols > and >> and
- As input to another program using a pipeline.

By default, the shell directs standard output from a command to the screen. The shell can effect redirection of this stream when it sees the > or >> symbols in the command line. You can replace the default destination with any file using > (right chevron) operator, followed by the filename. Ex: $wc sample.txt >temp.txt

```
$cat temp.txt
2    2    10          //output of wc stored in temp.txt
```

The 1st command sends the word count of sample.txt to temp.txt.

The shell also provides the >> symbol (used twice) to append a file.

```
$wc sample.txt >> temp.txt
```

2 2 10

2 2 10

Standard errors:

Each of the three standard files is represented by a number called file descriptor. A file is opened by referring to its pathname, but subsequent read and write operations identify the file by a unique number called a file descriptor. The kernel maintains a table of file descriptors for every process running in the system. The first three slots are generally allocated to the three standard streams as,

0 – Standard input
1 – Standard output
2 – Standard error

These descriptors are implicitly prefixed to the redirection symbols.

> and 1>

< and 0< mean the same

When we enter an incorrect command or when the input file to a command doesn't exist, error messages show up on the screen.

Trying to "cat " a non-existent file produces the error stream.
Ex: $ cat foo
cat: cannot open foo

Cat files to open the file and writes to standard error. If you are not using the c shell you can redirect this stream to a file.

The redirect output symbol (>) instructs the shell to redirect the error messages of a command to the specified file instead of to the screen.
Ex: $cat foo>errorfile
cat:cannot open foo //error stream can't be captured with>

Redirecting the standard error requires use of 2>symbols.
Ex: cat foo2>errorfile
$cat errorfile
cat: cannot open foo

Filters: Using Both Standard input and output:

UNIX commands can be grouped into four categories:
1) Directory-oriented commands like mkdir, rmdir and cd, and basic file handling commands like cp, mv and rm use neither standard input nor standard output.
2) Commands like ls, pwd, who etc don't read standard input but they write to standard output.
3) Commands like lp that read standard input but don't write to standard output.
4) Commands like cat, wc, cmp etc. that use both standard input and standard output. Commands like cat, wc, cmp etc. that use both standard input and standard output are called filters.
Let's use bc command as a filter this time, consider the file containing some Unix arithmetic expressions;
Ex: $cat calc.txt
2^32
25*50
30*25 + 15^2

GANESH XEROX RNSIT

You can redirect bc's standard input to come from this file and save output in yet another,

```
$ bc <calc.txt> result.txt
$ cat result.txt
4294967296   //this is 2^32
1250         //this is 25^50
975          //this is 30*25+15^2
```

Connecting Commands

4.4 Pipes

You can connect two commands together so that the output from one program becomes the input of the next program. Two or more commands connected in this way form a pipe(|).With piping, the output of a command can be used as input (piped) to a subsequent command.

```
$ command1 | command2
```

Output from command1 is piped into input for command2.

This is equivalent to, but more efficient than:

```
$ command1 > temp
$ command2 < temp
$ rm temp
```

Ex:

```
$ ls -al | more
$ who | sort | lpr
```

When a command needs to be ignorant of its source

If we wish to find total size of all C programs contained in the working directory, we can use the command,

```
$ wc –c *.c
```

However, it also shows the usage for each file(size of each file) . We are not interested in individual statistics, but a single figure representing the total size. To be able to do that, we must make wc ignorant of its input source. We can do that by feeding the concatenated output stream of all the .c files to wc –c as its input:

```
$.cat *.c | wc –c
```

4.5 grep: Searching for a pattern

We often need to search a file for a pattern, either to see the lines containing(or not containing) it or to have it replaced with something else. grep & egrep are two most popular filters of UNIX used to search for patterns.

The grep Command: The name "grep" derives from the ed (a UNIX line editor) command g/re/p which means "globally search for a regular expression and print all lines containing it."

It scans the file / input for a pattern and displays lines containing the pattern, the line numbers or filenames where the pattern occurs. It's a command from a special family in UNIX for handling search requirements.

grep *options pattern filename(s)*

A pattern or regular expression is either some plain text (a word, for example) and/or special characters used for pattern matching.

$grep "sales" emp.lst

This will display lines containing sales from the file emp.lst. Patterns with and without quotes is possible. It's generally safe to quote the pattern. Quote is mandatory when pattern involves more than one word. It returns the prompt in case the pattern can't be located.
When grep is used with multiple filenames, it displays the filenames along with the output.

$grep "director" emp1.lst emp2.lst

Where it shows filename followed by the contents

The simplest use of grep is to look for a pattern consisting of a single word. It can be used in a pipe so that only those lines of the input files containing a given string are sent to the standard output. If you don't give grep a filename to read, it reads its standard input; that's the way all filter programs work:

```
$ ls -l | grep "Aug"
-rw-rw-rw-  1 john doc     11008 Aug 6 14:10 ch02
-rw-rw-rw-  1 john doc      8515 Aug 6 15:30 ch07
-rw-rw-r--  1 john  doc     2488 Aug 15 10:51 intro
-rw-rw-r--  1 Amrit doc     1605 Aug 23 07:35 macros
$
```

There are various options which you can use along with grep command:

-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 exp	specifies expression with this option
-x	matches pattern with entire line
-f file	takes patterns from file, one per line
-E	treats pattern as an extended RE
-F	matches multiple fixed strings

Examples:

$grep -i 'agarwal' emp.lst

$grep -v 'director' emp.lst > otherlist

$wc -l otherlist will display 7 otherlist

$grep –n 'marketing' emp.lst

$grep –c 'director' emp.lst

$grep –c 'director' emp*.lst

will print filenames prefixed to the line count

$grep –l 'manager' *.lst

will display filenames only

$grep –e 'Agarwal' –e 'aggarwal' –e 'agrawal' emp.lst

will print matching multiple patterns

```
csr@csr-VirtualBox:~/Desktop$ cat emp.lst
2233|a.k.shukla        |g.m.          |sales         |12/12/52      |6000
2365|barun sengupta    |director      |personnel     |05/11/47      |7800
2476|anil aggarwal     |manager       |sales         |05/01/59      |5000
2345|j.b.saxena        |g.m.          |marketing     |03/12/2020    |8000
9876|jai sharma        |director      |production    |03/12/50      |7000
3564|sudhir agarwal    |executive     |personnel     |07/06/47      |7500
1265|s.n.dasgupta      |manager       |sales         |12/09/2020    |5600

csr@csr-VirtualBox:~/Desktop$ grep -i 'agarwal' emp.lst
3564|sudhir agarwal    |executive     |personnel     |07/06/47      |7500
csr@csr-VirtualBox:~/Desktop$ grep -i 'Agarwal' emp.lst
3564|sudhir agarwal    |executive     |personnel     |07/06/47      |7500
csr@csr-VirtualBox:~/Desktop$ grep -v 'Agarwal' emp.lst
2233|a.k.shukla        |g.m.          |sales         |12/12/52      |6000
2365|barun sengupta    |director      |personnel     |05/11/47      |7800
2476|anil aggarwal     |manager       |sales         |05/01/59      |5000
2345|j.b.saxena        |g.m.          |marketing     |03/12/2020    |8000
9876|jai sharma        |director      |production    |03/12/50      |7000
3564|sudhir agarwal    |executive     |personnel     |07/06/47      |7500
1265|s.n.dasgupta      |manager       |sales         |12/09/2020    |5600

csr@csr-VirtualBox:~/Desktop$ grep -v 'agarwal' emp.lst
2233|a.k.shukla        |g.m.          |sales         |12/12/52      |6000
2365|barun sengupta    |director      |personnel     |05/11/47      |7800
2476|anil aggarwal     |manager       |sales         |05/01/59      |5000
2345|j.b.saxena        |g.m.          |marketing     |03/12/2020    |8000
9876|jai sharma        |director      |production    |03/12/50      |7000
1265|s.n.dasgupta      |manager       |sales         |12/09/2020    |5600

csr@csr-VirtualBox:~/Desktop$ grep -v 'agarwal' emp.lst>otherlist
csr@csr-VirtualBox:~/Desktop$ cat otherlist
2233|a.k.shukla        |g.m.          |sales         |12/12/52      |6000
2365|barun sengupta    |director      |personnel     |05/11/47      |7800
2476|anil aggarwal     |manager       |sales         |05/01/59      |5000
2345|j.b.saxena        |g.m.          |marketing     |03/12/2020    |8000
9876|jai sharma        |director      |production    |03/12/50      |7000
1265|s.n.dasgupta      |manager       |sales         |12/09/2020    |5600
```

```
csr@csr-VirtualBox:~/Desktop$ wc -l otherlist
7 otherlist
csr@csr-VirtualBox:~/Desktop$ grep -n 'director' emp.lst
2:2365|barun sengupta    |director      |personnel     |05/11/47      |7800
5:9876|jai sharma        |director      |production    |03/12/50      |7000
csr@csr-VirtualBox:~/Desktop$ grep -c 'director' emp.lst
2
csr@csr-VirtualBox:~/Desktop$ grep -l 'director' emp.lst
emp.lst
```

4.6 Basic Regular Expressions (BRE) – An Introduction

It is tedious to specify each pattern separately with the -e option. grep uses an expression of a different type to match a group of similar patterns. If an expression uses meta characters, it is termed a regular expression. Some of the characters used by regular expression are also meaningful to the shell.

BRE character subset

The basic regular expression character subset uses an elaborate meta character set, overshadowing the shell's wild-cards, and can perform amazing matches.

The character class

grep supports basic regular expressions (BRE) by default and extended regular expressions (ERE) with the –e option. A regular expression allows a group of characters enclosed within a pair of [ ], in which the match is performed for a single character in the group.

$grep "[aA]g[ar][ar]wal" emp.lst

A single pattern has matched two similar strings. The pattern [a-zA-Z0-9] matches a single alphanumeric character. When we use range, make sure that the character on the left of the hyphen has a lower ASCII value than the one on the right. Negating a class (^) (caret) can be used to negate the character class. When the character class begins with this character, all characters other than the ones grouped in the class are matched.

The *

The asterisk refers to the immediately preceding character. * indicates zero or more occurrences of the previous character.
g* nothing or g, gg, ggg, etc.
grep "[aA]gg*[ar][ar]wal" emp.lst
Notice that we don't require to use –e option three times to get the same output.

The dot

A dot matches a single character. The shell uses ? Character to indicate that. The pattern 7.... Matches a four-character pattern beginning with a 7. The shells equivalent pattern is 7????.

The regular expression .* signifies any number of characters or none.

$grep "j.*saxena" emp.lst
2345 | j.b. saxena | g.m. |marketing |03/12/45 | 8000

Specifying Pattern Locations (^ and $)

Most of the regular expression characters are used for matching patterns, but there are two that can match a pattern at the beginning or end of a line. Anchoring a pattern is often necessary when it can occur in more than one place in a line, and we are interested in its occurrence only at a particular location.

^ for matching at the beginning of a line

$ for matching at the end of a line

$grep "^2" emp.lst

GANESH XEROX RNSIT

Selects lines where emp_id starting with 2

Sgrep "7...$" emp.lst

Selects lines where emp_salary ranges between 7000 to 7999

Sgrep "^[^2]".emp.lst

Selects lines where emp_id doesn't start with 2

Unix has no command that lists only directories. However we can use a pipeline to "grep" those lines from the listing that begin with a d:

ls -l | grep "^d"

```
csr@csr-VirtualBox:~/Desktop$ grep "^2" emp.lst
2233|a.k.shukla        |g.m.          |sales         |12/12/52      |6000
2365|barun sengupta    |director      |personnel     |05/11/47      |7800
2476|anil aggarwal     |manager       |sales         |05/01/59      |5000
2345|j.b.saxena        |g.m.          |marketing     |03/12/2020    |8000
csr@csr-VirtualBox:~/Desktop$ grep "7...$" emp.lst
2365|barun sengupta    |director      |personnel     |05/11/47      |7800
9876|jai sharma        |director      |production    |03/12/50      |7000
3564|sudhir agarwal    |executive     |personnel     |07/06/47      |7500
csr@csr-VirtualBox:~/Desktop$ grep "^[^2]" emp.lst
9876|jai sharma        |director      |production    |03/12/50      |7000
3564|sudhir agarwal    |executive     |personnel     |07/06/47      |7500
1265|s.n.dasgupta      |manager       |sales         |12/09/2020    |5600
```

When meta characters lose their meaning

It is possible that some of these special characters actually exist as part of the text. Sometimes, we need to escape these characters. For example, when looking for a pattern g*, we have to use \ for escaping. Similarly to look for a [, we use \[and to look for the pattern .* we should use \.*.

Extended Regular Expression (ERE) and egrep

ERE make it possible to match dissimilar patterns with a single expression. If current version of grep doesn't support ERE, then use egrep but without the –E option. -E option treats pattern as an ERE.

+	matches one or more occurrences of the previous character
?	Matches zero or one occurrence of the previous character

b+ matches b, bb, bbb,etc.

b? matches either a single instance of b or nothing

These characters restrict the scope of match as compared to the *

The ERE set

ch+	matches one or more occurrences of character ch
ch?	Matches zero or one occurrence of character ch
exp1|exp2	matches exp1 or exp2
(x1|x2)x3	matches x1x3 or x2x3 Matching multiple patterns (|, (and))

Examples: $grep –E "[aA]gg?arwal" emp.lst

 # ?include +<stdio.h>

$grep –E 'sengupta|dasgupta' emp.lst

We can locate both without using –e option twice, or

$grep –E '(sen|das)gupta' emp.lst

```
csr@csr-VirtualBox:~/Desktop$ cat emp.lst
2233|a.k.shukla        |g.m.        |sales        |12/12/52        |6000
2365|barun sengupta    |director    |personnel    |05/11/47        |7800
2476|anil aggarwal     |manager     |sales        |05/01/59        |5000
2345|j.b.saxena        |g.m.        |marketing    |03/12/2020      |8000
9876|jai sharma        |director    |production   |03/12/50        |7000
3564|sudhir agarwal    |executive   |personnel    |07/06/47        |7500
1265|s.n.dasgupta      |manager     |sales        |12/09/2020      |5600

csr@csr-VirtualBox:~/Desktop$ grep -E "[aA]gg?arwal" emp.lst
2476|anil aggarwal     |manager     |sales        |05/01/59        |5000
3564|sudhir agarwal    |executive   |personnel    |07/06/47        |7500
csr@csr-VirtualBox:~/Desktop$ grep -E 'sengupta|dasgupta' emp.lst
2365|barun sengupta    |director    |personnel    |05/11/47        |7800
1265|s.n.dasgupta      |manager     |sales        |12/09/2020      |5600
csr@csr-VirtualBox:~/Desktop$ grep -E '(sen|das)gupta' emp.lst
2365|barun sengupta    |director    |personnel    |05/11/47        |7800
1265|s.n.dasgupta      |manager     |sales        |12/09/2020      |5600
```


GANESH XEROX RNSIT

Chapter 5 Shell Programming

- A shell script contains a list of commands which have to be executed regularly.
- Shell script is also known as shell program.
- The user can execute the shell script itself to execute commands in it.
- A shell script runs in interpretive mode..i.e. the entire script is compiled internally in memory and then executed.
- Hence, shell scripts run slower than the high-level language programs.
- ".sh" is used as an extension for shell scripts.
- Example: A shell script (script.sh) to execute few commands.

```
1#!/bin/sh
2#script.sh: Sample shell script
3echo "Today's date: `date`"
4echo "This month's calendar:"
5cal `date "+%m 20%y"`
6echo "My shell: $SHELL"
```

- The hash symbol # indicates the comments in the script.
- The shell ignores all the characters that follow the # symbol. However, this does not apply to the first line.
- The first line

  "#! /bin/sh" indicates the path where the shell script is available.

There are 2 ways to execute a shell script:

1. Execute Shell Script Using File Name

- By default, script is not executable. So, the chmod command can be used to make the script executable.
- The scripts are executed in a separate child shell process.
- The child shell reads and executes each statement in interpretive mode.

Run:

$ chmod +x script.sh // add executable permission

$ ./script1.sh //execute the script script.sh

Output:

```
csr@csr-VirtualBox:~/Desktop$ gedit script.sh
csr@csr-VirtualBox:~/Desktop$ ./script.sh
Today's date: Thursday 05 November 2020 01:20:48 PM IST
This month's calendar:
   November 2020
Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30

My shell: /bin/bash
```

2. Execute Shell Script by Specifying the Interpreter

- The user can also execute a shell script by specifying the interpreter in the command line.
- Here, the script neither requires an executable permission nor an interpreter line.

Run:

```
$ sh script.sh          //Execute using sh interpreter
$ bash script.sh        //Execute using bash interpreter
```

5.1 Ordinary and Environment Variables

- Shell variables are of 2 types: 1) Environment and 2) Ordinary

Environment Variable

- Environmental variables are used to provide information to the programs you use.
- These variables control the behaviour of the system.
- They determine the environment in which the user works.
- If environment variables are not set properly, the users may not be able to use some commands.
- Environment variables are so called because they are available in the user's total environment i.e. the sub-shells that run shell scripts and mail commands and editors.
- Some variables are set by the system, others by the users, others by the shell programs.
- env command can be used to display environment variables.

For example: $ env

```
HOME=home/kumar
IFS=''
LOGNAME=kumar
MAIL= /var/mail/kumar
MAILCHECK=60
PATH=/bin:/usr/bin PS1='$'
PS2='>'
SHELL=/usr/bin/bash TERM= tty1
```

1. **HOME**
   - This variable indicates the home directory of the current user.
   - This variable is set for a user by the system admin in /etc/passwd.
2. **IFS**
   - This variable contains a string of characters that are used as word separator in the command line.
   - The string normally consists of the space, tab and newline characters.

GANESH XEROX PRINTING

3. **LOGNAME**
   - This variable shows the username
4. **MAIL**
   - This variable specifies the path to user's mailbox.
5. **MAILCHECK**
   - This variable determines how often the shell checks the file for the arrival of new mail.
6. **PATH**
   - This variable specifies the locations in which the shell should look for commands.
   - Usually, the PATH variable can be set as follows:

     $PATH=/bin:/usr/bin
7. **PS1 and PS2**
   - The shell has 2 prompts:
     - The primary prompt $ is the one the user normally sees on the monitor. $ is stored in PS1.
     - The user can change the primary prompt as follows:

       $ PS1="C>"

       C> //similar to windows
     - The secondary prompt > is stored in PS2.
8. **SHELL**
   - This variable specifies the current shell being used by the users.
   - Different types of shells are:

     1)Bourne shell /bin/sh

     2) C-shell /bin/csh

     3) Korn shell /bin/ksh
   - This variable is set for a user by the system admin in /etc/passwd.
9. **TERM**
   - This variable indicates the terminal type that is used.
   - Every terminal has certain characteristics that are defined in a separate control file in the term info directory.
   - If TERM is not set correctly, vi will not work and the display will be faulty.

Ordinary (or Local) Variable

- A variable is a character string to which the user assigns a value.
- The value assigned can be a number, text, filename, device, or any other type of data.

Syntax:

variable = value // variable definition

- The value of variables are stored in the ASCII format.
- For example:
  $ x=50
  echo $x //displays 50
- In command line, all words that are preceded by a $ are identified and evaluated as variables.
- A variable can be removed with unset and protected from reassignment by readonly. Both are shell internal commands.
  $ set count=5
  $ readonly size = 10
- The variables exist only for a short time during the execution of a shell script.
- The variables are local to the user's shell environment.
- The variables are not available for the other scripts or processes.
- As the variables are defined and used by specific users, they are also called user-defined variables.

Uses of Local variables

- Setting pathnames: If a pathname is used several times in a script, we can assign it to a variable and use it as an argument to any command.
- Using command substitution: We can assign the result of execution of a command to a variable. The command to be executed must be enclosed in backquotes.
- Concatenating variables and strings: Two variables can be concatenated to form a new variable.
- Example:
  $ base=foo ; ext=.c
  $ file=$base$ext
  $ echo $file

5.2 The .profile

When we login to the system, the shell undergoes a phase called initialization to set-up various environment. This is usually a two-step process that involves the shell reading the following files:

/etc/profile

- The shell checks to see whether the file **/etc/profile** exists.
- If it exists, the shell reads it. Otherwise, this file is skipped. No error message is displayed.
- The file **/etc/profile** is maintained by the system administrator of our UNIX machine and contains shell initialization information required by all users on a system.

GANESH XEROX PRINT

.profile

- The shell checks to see whether the file **.profile** exists in our home directory. Our home directory is the directory that we start out in after we log in.
- If it exists, the shell reads it; otherwise, the shell skips it. No error message is displayed.
- The file **.profile** is under our control. We can add as much shell customization information as we want to this file. The minimum set of information that we need to configure includes
  - ✓ The type of terminal we are using
  - ✓ A list of directories in which to locate commands
  - ✓ A list of variables effecting look and feel of our terminal.
- We can check our **.profile** which is available in our home directory. Open it using **vi** editor and check all the variables set for our environment.
- As soon as both of these files have been read, the shell displays a prompt $
- The user can view his ".profile" as follows:

```
. $ cat .profile
MAIL= /var/mail/kumar
PATH=/bin:/usr/bin
PS1='$'
PS2='>'
SHELL=/usr/bin/bash
TERM= tty1
```

5.3 read and readonly Commands

read Command

The read statement is the shell's internal tool for making scripts interactive (i.e. taking input from the user). It is used with one or more variables. Inputs supplied with the standard input are read into these variables. For instance, the use of statement like

read name

causes the script to pause at that point to take input from the keyboard. Whatever is entered by you will be stored in the variable **name.**

Ex: A shell script that uses read to take a search string and filename from the terminal.

```
1#!/bin/sh
2#script1.sh
3#
4echo "Enter the pattern to be searched:\c"
5read pname
6echo "Enter the filename to be used:\c"
7read flname
8echo "Searching for the $pname from $flname"
9grep "$pname" $flname
10echo "pattern is found"
```

Output

```
csr@csr-VirtualBox:~/Desktop$ gedit script1.sh
csr@csr-VirtualBox:~/Desktop$ chmod +x script1.sh
csr@csr-VirtualBox:~/Desktop$ ./script1.sh
Enter the pattern to be searched:director
Enter the filename to be used:emp.lst
Searching for the director from emp.lst
2365|barun sengupta     |director      |personnel     |05/11/47      |7800
9876|jai sharma         |director      |production    |03/12/50      |7000
pattern is found
```

readonly Command

- readonly command can be used to make variables readonly i.e. the user cannot change the value of variables.
- During shell scripting, we may need a few variables, which cannot be modified.
- This may be needed for security reasons.
- Syntax:

  variable=value
- **For example:**

```
$ readonly PI=3.14
$ echo $PI                    //displays 3.14
$ PI=6.12                     // this will result in error
```

5.4 Command Line Arguments

- Shell scripts can accept arguments from the command line.
- Shell scripts can be run non-interactively and be used with redirection and pipelines.
- The arguments are assigned to special shell variables called shell parameters.
- The shell parameters are reserved for specific functions.
- Different shell parameters:
  a. $#: Stores the number of command-line arguments.
  b. $0, $1, $2, $3: These are called positional parameters which represent command line arguments.

     $0: Stores the filename of the current script.

     $1: Stores the first argument.

     $2: Stores the second argument

     $3: Stores the third argument

  c. $*: Stores all the arguments entered on the command line ($1 $2 ...).

  d. "$@": Stores all arguments entered on the command line, individually quoted ("$1" "$2")

  e. $?: Stores the exit status of the last command that was executed.

  f. $$: Stores Pid of the current shell.

  g. $!: Stores PID of the last background job.

Ex: Following is a simple shell script:

```
#!/bin/sh
# Illustrates using command-line arguments
# Execute with
# sh args1.sh sailing on the sea
echo "First command-line argument is: $1"
echo "Third argument is: $3"
echo "Number of arguments is: $#"
echo "The entire list of arguments is: $*"
```

Execute the following command:

```
$ simple.sh sailing on the sea
```

First command-line argument is: sailing
Third argument is: the
Number of arguments is: 4

Another example:

```
1#!/bin/sh
2#emp.lst
3#
4echo "Program:$0
5The no. of arguments specified is $#
6The arguments are $*"
7grep "$1" $2
8echo "\n job over"
```

```
csr@csr-VirtualBox:~/Desktop$ ./cmd.sh director emp.lst
Program:./cmd.sh
The no. of arguments specified is 2
The arguments are director emp.lst
2365|barun sengupta     |director      |personnel     |05/11/47      |7800
9876|jai sharma         |director      |production    |03/12/50      |7000

job over
```

5.5 exit and exit status of a Command

- exit command can be used to terminate a program (or script).
- This command returns value which will be available to the script's parent process.
- The $? variable contains exit status of the last command executed.
- Exit status is a numerical value returned by every command upon its completion.
- A command returns an exit status of
  - (0) upon successful execution and
  - (1) upon unsuccessful execution i.e. an error condition.
  - (2) failure in opening file
- Exit status can be used to devise program-logic that branches into different paths depending on success or failure of a command.

```
csr@csr-VirtualBox:~/Desktop$ grep director emp.lst >/dev/null; echo $?
0
csr@csr-VirtualBox:~/Desktop$ grep actor emp.lst >/dev/null; echo $?
1
csr@csr-VirtualBox:~/Desktop$ grep actor unix.lst >/dev/null; echo $?
grep: unix.lst: No such file or directory
2
```

5.6 Logical Operators for Conditional Execution

Two logical operators can be used for conditional execution: 1) && and 2) ||

1) && Operator

Syntax:

cmd1 && cmd2

Here, cmd2 gets executed only when cmd1 succeeds.

2) || Operator

Syntax:

cmd1 || cmd2

Here, cmd2 gets executed only when cmd1 fails.

Example: A script to illustrate the usage of && and ||.

&& delimits two commands. cmd2 executed only when cmd1 succeeds, then the following instructions are executed and in case of || , following statements are executed of cmd1 is true or cmd2 is true or both.

```
csr@csr-VirtualBox:~/Desktop$ grep 'director' emp.lst && echo "pattern found in file"
2365|barun sengupta     |director        |personnel       |05/11/47        |7800
9876|jai sharma         |director        |production      |03/12/50        |7000
pattern found in file
csr@csr-VirtualBox:~/Desktop$ grep 'actor' emp.lst || echo "pattern not found in file"
pattern not found in file
```

5.7 test Command and its Shortcut

- Usually, if-construct cannot directly handle the true or false value returned by evaluation of an expression.
- So, test command can be used to handle the true or false value returned by evaluation of an expression.
- Test command

  → uses certain operators to evaluate the condition on its right and

  → returns either a true or false exit status.

- Then, if-construct uses the exit status for making decisions.
- Test command

  → does not display any output

  → sets the parameter $? (exit status).

- Test command works in 3 ways:
  - Compare two numbers.
  - Compares two strings or a single one for a null value.
  - Checks files attributes.

1. Numeric Comparison

Operator	Meaning
-eq	Equal to
-ne	Not equal to
-gt	Greater than
-ge	Greater than or equal to
-lt	Less than
-le	Less than or equal

Syntax:

test $op1 -operator $op2

Operators always begin with a – (Hyphen) followed by a two characters word.

Numeric comparison can be done on integer values only. (The decimal values are truncated).

Shorthand for test

[and] can be used instead of test.

GANESH XEROX RNSIT ISE

Test $x –eq $y is equivalent to [$x –eq $y]

Example: A shell script to find relationship between 2 numbers.

```
#!/bin/usr
$x=5; y=4
$test $x –eq $y; echo $?
1                          #Not equal
$test $x -gt $y
0                          #true
```

```
csr@csr-VirtualBox:~/Desktop$ x=5;y=7;z=7.2
csr@csr-VirtualBox:~/Desktop$ test $x -eq $y; echo $?
1
csr@csr-VirtualBox:~/Desktop$ test $x -lt $y; echo $?
0
csr@csr-VirtualBox:~/Desktop$ test $x -gt $y; echo $?
1
csr@csr-VirtualBox:~/Desktop$ test $z -eq $y; echo $?
bash: test: 7.2: integer expression expected
2
```

Ex. Using test in if

```
#!/bin/sh
if test $# -ne 2
then
        echo "µ have not entered 2 arguments"
        exit 3
else
        if grep "$1" $2
        then
                echo "Pattern Found"
        else
                echo "Pattern not Found"
        fi
fi
```

A pair of rectangular brackets can replace the word test as shown:

```
if [$# -ne 2]
```

Also if-then-else-if can be replaced by if-then-elif

Ex: Following script uses [ ] to evaluate the shell parameter $#.It displays 'pattern file' if no arguments, runs grep if two arguments, otherwise error message is displayed.

```
1#!/bin/sh
2#numeric.sh
3#
4if test $# -eq 0; then
5echo "usage: $0 pattern file"  >/dev/tty
6elif test $# -eq 2; then
7grep  $1  $2 || echo  $1 not found in $2" >/dev/tty
8else echo "you didnt enter two arguments">/dev/tty
9fi
```

The echo output is redirected to /dev/tty to make the script work both with or without redirection. In either case, the output of the echo statements must appear only on the terminal. This is one of the importance of /dev/tty file. The above script works even without redirecting it.

```
csr@csr-VirtualBox:~/Desktop$ ./numeric.sh > foo
usage: ./numeric.sh pattern file
csr@csr-VirtualBox:~/Desktop$ ./numeric.sh ftp > foo
you didnt enter two arguments
csr@csr-VirtualBox:~/Desktop$ ./numeric.sh director emp.lst  > foo
csr@csr-VirtualBox:~/Desktop$ cat foo
2365|barun sengupta     |director       |personnel      |05/11/47       |7800
9876|jai sharma         |director       |production     |03/12/50       |7000
```

2. String Comparison

Test command is also used for testing strings.

Operator	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

Example: A shell script to check if 2 strings are equal or not.

This script behaves both interactively and noninteractively. When run without arguments, it turns interactive and takes input from you. It then runs numeric.sh, the script developed previously, with the supplied input as arguments. However, when this string.sh itself is run with at least one argument, it runs numeric.sh with the same arguments.

String.sh

```
#!/bin/sh
#string.sh
#
if [ $# -eq 0 ]; then
echo "Enter the string to be searched:\c"
read pname
if [ -z "$pname" ]; then
echo "you have not entered the string"; exit 1
fi
echo "Enter the filename to be used: \c"
read flname
if [ -z "$flname" ]; then
echo "you have not entered filename"; exit 2
fi
./numeric.sh $pname $flname
else
./numeric.sh $*
fi
```

Output

```
csr@csr-VirtualBox:~/Desktop$ gedit string.sh
csr@csr-VirtualBox:~/Desktop$ ./string.sh
Enter the string to be searched:
you have not entered the string
csr@csr-VirtualBox:~/Desktop$ ./string.sh
Enter the string to be searched:director
Enter the filename to be used: emp.lst
2365|barun sengupta     |director       |personnel      |05/11/47       |7800
9876|jai sharma         |director       |production     |03/12/50       |7000
csr@csr-VirtualBox:~/Desktop$ ./string.sh director
you didnt enter two arguments
csr@csr-VirtualBox:~/Desktop$ ./string.sh director emp.lst
2365|barun sengupta     |director       |personnel      |05/11/47       |7800
9876|jai sharma         |director       |production     |03/12/50       |7000
```

3. File Tests

Test command can be used to check various file attributes such as file type (-, d or l) & file permission (r, w, x).

Test	True if
-e file	File exists
-f file	File exists and is a regular file
-d file	File exists and is a directory
-L file	File exists and is a symbolic link
-r file	File exists and readable
-w file	File exists and is writable
-x file	File exists and is executable
-s file	File exists and has a size greater than zero
f1 –nt f2	File f1 is newer than f2
f1 –ot f2	File f1 is older than f2
f1 –ef f2	File f1 is linked to f2

Example: A shell script (program8.sh) to check whether a file has permission for read, write and execute.

```
#!/bin/sh
#filetest.sh
#
if [ ! -e $1 ] ; then
echo "File does not exist"
elif [ ! -r $1 ]; then
echo "File is not readable"
elif [ ! -w $1 ] ; then
echo "File is not writable"
else
echo "File is both readable and writable"
fi
```

SRI SEIVA YEBOK
#147, HPS Tower, CTP, JSS College,
Below UCO Bank, Dr Vishnuvardan Road,
Rajiex Layout, Bangalore - 560 068.
Ph: 9620099557, 7676670591.

```
csr@csr-VirtualBox:~/Desktop$ ls -l emp.lst
-rwxrwxrwx 1 csr csr 378 Oct 15 11:33 emp.lst
csr@csr-VirtualBox:~/Desktop$ [ -x emp.lst ]; echo $?
0
csr@csr-VirtualBox:~/Desktop$ [ -f emp.lst ]; echo $?
0
csr@csr-VirtualBox:~/Desktop$ [ ! -w emp.lst ] || echo "False that file is not writable"
False that file is not writable
csr@csr-VirtualBox:~/Desktop$ ./filetest.sh emp.lst
File is both readable and writable
```

5.8 Control Statements

1. if Statement

- if statement is basically a "two-way" decision statement.
- This is used when we must choose between two alternatives.
- Three forms of if..:else statement:
  1) if...fi statement
  2) if...else...fi statement
  3) if...elif...else...fi statement

Syntax 1:

```
if command is successful
then
        execute statements
fi
```

Syntax 2:

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

Syntax 3:

```
if command is successful
then
        execute statements
elif command is successful
        then
        else
fi
```

Here is how it works:

- If the command succeeds, the statements within then-block are executed.
- If the command fails, the statements within else-block are executed.

```
1#!/bin/sh
2#ifelse.sh
3#
4if grep "^$1" /etc/passwd 2> /dev/null
5then
6echo "pattern found"
7else
8echo " pattern not found"
9fi
```

Example: A script to check whether an integer is positive or negative.

```
#!/bin/sh
echo "Enter any non zero integer: \n" read num
if [$num -gt 0]; then
echo "Number is positive number"
else
```

echo "Number is negative number"
fi

2. case Statement

- case statement is basically a "multi-way" decision statement.
- This is used when we must choose among many alternatives.
- This also handles string tests, but in a more efficient manner than if statement.
- Syntax:

```
case expression in
    pattern1) statement1 ;;
    pattern2) statement2 ;;
    pattern3) statement3 ;;
    ...
esac
```

Here is how it works:

- Firstly, the expression is matched with pattern1.
- If the match succeeds, then statement1 will be executed.
- If the match fails, then the expression is matched with pattern2 and this process continues.
- Each statement is terminated with a pair of semicolon (;;).
- This can match only strings but cannot handle numeric and file tests. However, this can also handle numbers but treating them as strings.
- This is very effective when the string is fetched by command substitution.

```
1 #!/bin/sh
2 #
3 echo "                Menu\n
4 1. List of files
5 2. Processes of user
6 3. Today's date
7 4. Users of system
8 5. Quit
9 "
10 read choice
11 case "$choice" in
12 1) ls -l;;
13 2) ps -f;;
14 3) date ;;
15 4) who ;;
16 5) exit ;;
17 *) echo "Invalid option"
18 esac
19
```

```
csr@csr-VirtualBox:~/Desktop$ ./menu.sh
                Menu

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

1
total 28
-rw-rw-r-- 1 csr csr   17 Sep 30 14:39  b.txt~
-rwxrwxr-x 1 csr csr  378 Nov  5 10:59  emp.lst.bak
-rwxrwxr-x 1 csr csr  239 Oct 29 11:16  menu.sh
-rwxrwxr-x 1 csr csr  220 Oct 29 09:59  numeric.sh
-rwxrwxr-x 1 csr csr  105 Oct 22 09:44  script.sh
drwxrwxr-x 2 csr csr 4096 Nov  5 15:06 'Shell Scripts'
-rwxrwxr-x 1 csr csr  126 Oct 29 10:18  string1.sh
csr@csr-VirtualBox:~/Desktop$ ./menu.sh
                Menu

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

2
UID          PID    PPID  C STIME TTY          TIME CMD
csr         7753    7742  0 14:06 pts/0    00:00:00 bash
csr         8665    7753  0 15:08 pts/0    00:00:00 /bin/sh ./menu.sh
csr         8666    8665  0 15:08 pts/0    00:00:00 ps -f
```

Matching Multiple Patterns

- case statement can also specify the same action for more than one pattern.
- Example: Try this: A script to test a user response for both y and Y (or n and N).

```
#!/bin/sh
echo "Do you wish to continue? [y/n]:"
read ans
case "$ans" in
Y | y ) ;;
N | n ) exit ;;
esac
```

Wild-Cards

- case statement has a superb string-matching feature that uses wild-cards.
- case statement uses

  → filename matching meta-characters * and ?

  → string matching character.

Example: Try this: A script to test a user response for YES, yes, Yes, yEs (or no, NO, No, nO).

```
#!/bin/sh
echo "Do you wish to continue? [y/n]:"
read ans
case "$ans" in
[Yy] [eE]* ) ;;          # Matches YES, yes, Yes, yEs, etc
[Nn] [oO]  ) exit ;;     # Matches no, NO, No, nO
* ) echo "Invalid Response"
esac
```

3. while Statement

while loop can be used to execute a set of statements repeatedly as long as a given condition is true.

Syntax:

```
while condition is true
do
        execute statements
done
```

The statements enclosed between do and done are executed repeatedly as long as condition is true.

Examples: Try this: A script to display a message 3 times using while loop.

```
#!/bin/sh
num=1
while [$num -le 3]
        do
                echo " Welcome to Shell Programming "
                expr $num = $num +1;
        done
```

```
#!/bin/sh
# whileloop.sh:
#
answer=y
while [ "$answer" = "y" ]
do
        echo "Enter the code and description:\c" >/dev/tty
        read code description
        echo "$code|$description">>newlist
        echo "Enter any more (y/n)? \c">/dev/tty
        read anymore
        case $anymore in
        y*|Y*) answer=y;;
        n*|N*) answer=n;;
        *) answer=y;;
        esac
done
```

```
csr@csr-VirtualBox:~/Desktop$ gedit whileloop.sh
csr@csr-VirtualBox:~/Desktop$ gedit whileloop.sh
csr@csr-VirtualBox:~/Desktop$ ./whileloop.sh
Enter the code and description:18CS31 Maths
Enter any more (y/n)? y
Enter the code and description:18CS32 Data Structures
Enter any more (y/n)? n
csr@csr-VirtualBox:~/Desktop$ cat newlist
18CS31|Maths
18CS32|Data Structures
```

4. for Statement

for loop can be used to iterate over all items (or strings) within a list.

Syntax:

```
for variable in list
do
        statements
done
```

Here, list consists of a set of items (or strings).
Each item of the list is picked up and assigned to the "variable".
The iteration continues until all items are picked from the array.

Example: A script to display elements of an array.

```
1#!/bin/sh
2#for1.sh
3#
4printf "Here are the numbers in the list:\n"
5for var in 10 20 30 40 50 60
6do
7        echo "$var \t"
8done
```

```
csr@csr-VirtualBox:~/Desktop$ gedit for1.sh
csr@csr-VirtualBox:~/Desktop$ ./for1.sh
Here are the numbers in the list:
10
20
30
40
50
60
```

Possible Sources of List

Possible sources of list are

- List from variables
- List from command substitution
- List from wildcards and
- List from positional parameters

1) List from Variables

Example: A script to evaluate & display a set of variables using for-loop.

```
1#!/bin/sh
2x="Dream"
3y="Believe"
4z="Achieve"
5for var in $x $y $z;
6do
7        echo  $var \t
8done
```

```
csr@csr-VirtualBox:~/Desktop$ chmod +x list.sh
csr@csr-VirtualBox:~/Desktop$ ./list.sh
Dream
Believe
Achieve
```

2) List from Command Substitution

Command substitution can be used for creating a list.
Useful: when list is large.
Example: A script to display current date using for-loop.

```
1#!/bin/sh
2for var in `date`
3do
4echo "$var\t"
5done
```

```
csr@csr-VirtualBox:~/Desktop$ gedit com.sh
csr@csr-VirtualBox:~/Desktop$ chmod +x com.sh
csr@csr-VirtualBox:~/Desktop$ ./com.sh
Thursday
05
November
2020
09:56:18
PM
IST
```

3) List from Wildcards

The shell can use wildcards for matching filenames.
Example: A script to print all files with pdf extension.

```
#! /bin/sh
for file in *.pdf
do
        echo "Printing $file \n"
        lp $file
done
```

4) List from Positional Parameters

for is also used to process positional arguments thar are assigned from command line arguments. The example shown below, scans a file repeatedly for each argument. It uses the shell parameter "$@" to represent all command line arguments.

Example:

```
1#!/bin/sh
2for pattern in "$@" ;
3do
4grep "$pattern" emp.lst || echo "pattern $pattern not found"
5done
```

```
csr@csr-VirtualBox:~/Desktop$ ./pattern.sh 2345 1265 "jai sharma" 4379
2345|j.b.saxena      |g.m.        |marketing    |03/12/2020    |8000
1265|s.n.dasgupta    |manager     |sales        |12/09/2020    |5600
9876|jai sharma      |director    |production   |03/12/50      |7000
pattern 4379 not found
```

Example: Try this: A script (program4.sh) to read & display a positional parameter using for-loop.

```
#!/bin/sh
for var in "$*"          # even "$@" can be used
do
echo "$var \t"
done
```

GANESH XEROX RNSIT ISE

5.9 set and shift Commands and Handling Positional Parameters

set

set command can be used to assign positional parameters ($1, $2 and $3) to command line arguments.

This command can be used for picking up individual fields from the output of a program.

Example:

$ set 101 102 103

Here, above line assigns

→ 101 to $1

→ 102 to $2

→ 103 to $3.

This command can also be used to assign the other parameters $# and $*.

Example:

```
csr@csr-VirtualBox:~/Desktop$ set 101 102 103
csr@csr-VirtualBox:~/Desktop$ echo "$@"
101 102 103
csr@csr-VirtualBox:~/Desktop$ echo "The $# arguments are $*"
The 3 arguments are 101 102 103
csr@csr-VirtualBox:~/Desktop$ set `date`
csr@csr-VirtualBox:~/Desktop$ echo $*
Thursday 05 November 2020 10:17:17 PM IST
csr@csr-VirtualBox:~/Desktop$ echo "The date today is $2 $3, $6"
The date today is 05 November, PM
```

shift

shift command is a shell built-in that operates on the positional parameters.

Each time shift command is called, it shifts/transfers all the positional parameters down by one.

For example: $2 becomes $1

$3 becomes $2

$4 becomes $3, and so on.

Example:

```
csr@csr-VirtualBox:~/Desktop$ echo "$@"
Thursday 05 November 2020 10:17:17 PM IST
csr@csr-VirtualBox:~/Desktop$ echo $1 $2 $3
Thursday 05 November
csr@csr-VirtualBox:~/Desktop$ shift
csr@csr-VirtualBox:~/Desktop$ echo $1 $2 $3
05 November 2020
csr@csr-VirtualBox:~/Desktop$ shift 2
csr@csr-VirtualBox:~/Desktop$ echo $1 $2 $3
2020 10:17:17 PM
```

Set -- : Helps Command Substitution

Problem with set command:

When set command is used with command substitution, the output of the command may begin with a -(hypen). In this case, set command interprets -(hypen) as an option and does not work correctly.

For example:

$set 'ls -l student.lst'
-rwxr-xr--: bad option

Solution: Use --(double hypen) immediately after set command.

$set -- 'ls -l student.lst'
-rwxr-xr-- 2 kumar group 163 Jul 13 21:36 student.lst

5.10 The Here (<<) Document

- The << symbol can be used to read data from the same file containing the script. This file is called as a here document.
- The term 'here' signifies that the data is here rather than in the file.
- Any command using standard input can also take input from a here document.

Syntax:

```
command << delimiter
document
delimiter
```

For example:

Following shows the mail command along with the message to be mailed:

```
mail kumar << MARK
Your program for printing the invoices has been executed on `date`.
Check the print queue.
The updated file is $fname
MARK
```

The string (MARK) is delimiter. We can use any string as delimiter. The shell treats every line following the command and delimited by MARK as input to the command. Kumar at the other end will see three lines of message text with the date inserted by command. The word MARK itself doesn't showup.

Example: This shows the here document with an interactive script script1.sh and also noninteractively by supplying the inputs through a here document.

```
csr@csr-VirtualBox:~/Desktop$ gedit script1.sh
csr@csr-VirtualBox:~/Desktop$ ./script1.sh
Enter the pattern to be searched:director
Enter the filename to be used:emp.lst
Searching for the director from emp.lst
2365|barun sengupta     |director       |personnel      |05/11/47       |7800
9876|jai sharma         |director       |production     |03/12/50       |7000
pattern is found
csr@csr-VirtualBox:~/Desktop$ ./script1.sh << END
director
emp.lst
END
Enter the pattern to be searched:Enter the filename to be used:Searching for the director from emp.lst
2365|barun sengupta     |director       |personnel      |05/11/47       |7800
9876|jai sharma         |director       |production     |03/12/50       |7000
pattern is found
csr@csr-VirtualBox:~/Desktop$
```

/dev/null and /dev/tty : Two special files

/dev/null: If we like to execute a command but don't like to see its contents on the screen, we can redirect the output to a file called /dev/null. It is a special file that can accept any stream without growing in size. It's size is always zero.

Ex:

```
$ cal 1995 > /dev/null
$ cat /dev/null
$ -
```

The device file simply takes all output directed towards it . But its size always remains 0. This facility is useful in redirecting error messages away from the terminal, so that they don't appear on the screen. It is a pseudo-device, because it is not associated with any physical device.

/dev/tty: This file indicates one's terminal. In a shell script, if we wish to redirect the output of some select statements explicitly to the terminal. In such cases we can redirect these explicitly to /dev/tty inside. the script. This file can be accessed independently by several users without conflict.

Ex: Suppose kumar is working on the terminal /dev/tty01 & Sharma is working on the terminal /dev/tty02. However both kumar & sharma can refer to their own terminals with a single device file called /dev/tty. So, the following command

$ who > /dev/tty stores in /dev/tty01 on kumar's terminal & in /dev/tty02 on sharma's terminal

MODULE-3

Topics

UNIX File APIs:
General File APIs, File and Record Locking, Directory File APIs, Device File APIs, FIFO File APIs, Symbolic Link File APIs.
UNIX Processes and Process Control:
The Environment of a UNIX Process:
Introduction, main function, Process Termination, Command-Line Arguments, Environment List, Memory Layout of a C Program, Shared Libraries, Memory Allocation, Environment Variables, setjmp and longjmp Functions, getrlimit, setrlimit Functions, UNIX Kernel Support for Processes.
Process Control:
Introduction, Process Identifiers, fork, vfork, exit, wait, waitpid, wait3, wait4 Functions, Race Conditions, exec Functions

CHAPTER 6 UNIX File APIs

Introduction

In UNIX everything can be treated as files. Hence files are building blocks of UNIX operating system. When you execute a command in UNIX, the UNIX kernel fetches the corresponding executable file from a file system, loads its instruction text to memory and creates a process to execute the command.

UNIX / POSIX file Types

The different types of files available in UNIX / POSIX are:

Regular files	Example: All .exe files, C, C++, PDF Document files.
Directory files	Example: Folders in Windows.
Device files	Example: Floppy, CDROM and Printer.
FIFO files	Example: Pipes.
Link Files (only in UNIX)	Example: alias names of a file, Shortcuts in Windows.

Regular files

- It is a text or binary file.
- These files may be read or written by users with the appropriate access permission.
- For Example: All exe files, text, Document files and program files. Created and modified by using editors (vi, vim, emacs) or by compilers (cc, gcc, c++,g++).
- And these files are removed by using *rm* command.

Directory files

- A directory is like a file folder that contains other files, including other subdirectories.

- It provides a means for users to organize their files into some hierarchical structure based on file relationship or users.
- mkdir is the command used for creating the directory.
- mkdir newdir // creates an empty directory newdir under the current directory.
- An unix directory is consider to be an empty if it contains no other files except . and .. files.
- *rmdir* is a command, which is used to delete a directory(s).
- rmdir newdir // To remove the directory that directory should be empty.
- The contents of a directory files may be displayed in UNIX by the *ls* command.

Device files

- These are physical devices such as printers, tapes, floppy devices CDROMs, hard disks and terminals.
- There are two types of device files: Block and Character device files.

a. Block Device files:

- A physical device that transmits block of data at a time. For example: floppy devices CDROMs, hard disks.

b. Character Device files:

- A physical device that transmits data character by character. For example: Line printers, modems etc.
- mknod is the command used for creating the device files.
- **Syntax: $mknod Device_Filename DeviceFile_Type Major_Number Minor_Number**

  Example1: # mknod /dev/cdsk c 120 20

  This creates character device file called *cdsk* with major number 120 and minor number 20.

  Example2: # mknod /dev/bdsk b 225 15

  This creates block device file called *bdsk* with major number 225 and minor number 15.
- A major device number is an index to a kernel table that contains the addresses of all device driver functions known to the system.
- A minor device number is an integer value to be passed as an argument to a device driver function when it is called.
- The minor device number tells the device driver function what actual physical device it is taking to and the input output buffering scheme to be used for data transfer.
- In UNIX, *mknod* must be invoked through superuser privileges.

FIFO file

- It is a special pipe device file, which provides a temporary buffer for two or more processes to communicate by writing data to and reading data from the buffer.
- A process may write more than PIPE_BUF bytes of data to a FIFO file, but it may be blocked when the file buffer is filled.
- In this case the process must wait for a reader process to read data from the pipe and make room for the write operation to complete.

GANESH XEROX

- Finally, the data in the buffer is accessed in a by a first-in-first-out manner, hence the file is called a FIFO.
- These are having name hence they called named pipes.
- The size of the buffer associated with FIFO file is fixed to PIPE_BUF or _POSIX_PIPE_BUF.
- *mkfifo* is the command used for creating a FIFO file(s).

  mkfifo /usr/prog/fifo_pipe **or** by using *mknod* with the option *p*

  mknod /usr/prog/fifo_pipe p
- *rm* is the command used for removing FIFO files.

Symbolic Link files

- A symbolic link file contains a path name which references another file in either the local or a remote file system.
- These are not supported by POSIX.1
- A symbolic link may be created in UNIX via the *ln* command.
- This command creates a symbolic link /usr/csr/unix which references the file /usr/yuktha/original. The cat command which follows will print the content of the /usr/yuktha/original file:
- Example: ln –s /usr/yuktha/original /usr/csr/unix cat -n /usr/csr/unix
- It is possible to create a symbolic link to reference another symbolic link.
- rm, mv and chmod commands will operate only on the symbolic link arguments directly and not on the files that they reference.

6.1 General File APIs

The file APIs that are available to perform various operations on files in a file system are:

FILE APIs	USE
open ( )	This API is used by a process to open a file for data access.
read ( )	The API is used by a process to read data from a file
write ( )	The API is used by a process to write data to a file
lseek ( )	The API is used by a process to allow random access to a file
close ( )	The API is used by a process to terminate connection to a file
stat ( )fstat ( )	The API is used by a process to query file attributes
chmod ( )	The API is used by a process to change file access permissions.
chown ( )	The API is used by a process to change UID and/or GID of a file
utime ( )	The API is used by a process to change the last modification and access time stamps of a file
link ( )	The API is used by a process to create a hard link to a file.
unlink ( )	The API is used by a process to delete hard link of a file
umask ( )	The API is used by a process to set default file creation mask.

Open:

- It is used to open or create a file by establishing a connection between the calling process and a file.

Prototype:

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

- **path_name :** The pathname of a file to be opened or created. It can be an absolute path name or relative path name. The pathname can also be a symbolic link name.
- **access_mode:** An integer values in the form of manifested constants which specifies how the file is to be accessed by calling process. The manifested constants can be classified as access mode flags and access modifier flags.

Access mode flags:

- **O_RDONLY:** Open the file for read only. If the file is to be opened for read only then the file should already exist in the file system and no modifier flags can be used.
- **O_WRONLY:** Open the file for write only. If the file is to be opened for write only, then any of the access modifier flags can be specified.
- **O_RDWR:** Open the file for read and write. If the file is to be opened for write only, then any of the access modifier flags can be specified.

Access modifier flags are optional and can be specified by bitwise-OR ing them with one of the above access mode flags to alter the access mechanism of the file.

Access Modifier Flags:

- **O_APPEND:** Appends data to the end of the file. If this is not specified, data can be written anywhere in the file.
- **O_CREAT:** Create the file if it does not exist. If the file exists it has no effects. However if the file does not exist and O_CREATE is not specified, open will abort with a failure return status.
- **O_EXCL:** Used with O_CREAT, if the file exists, the call fails. The test for existence and the creation if the file does not exist.
- **O_TRUNC:** If the file exits, discards the file contents and sets the file size to zero.
- **O_NOCTTY:** Species not to use the named terminal device file as the calling process control terminal.
- **O_NONBLOCK:** Specifies that any subsequent read or write on the file should be non-blocking. Example, a process is normally blocked on reading an empty pipe or on writing to a pipe that is full. It may be used to specify that such read and write operations are non-blocking.

  Example:

  ```
  int fdesc = open("/usr/unix/prog1", O_RDWR|O_APPEND,0);
  ```

  If a file is to be opened for read-only, the file should already exist and no other modifier flags can be used.

O_APPEND, O_TRUNC, O_CREAT and O_EXCL are applicable for regular files, whereas O_NONBLOCK is for FIFO and device files only, and O_NOCTTY is for terminal device file only.

Permission:

- The permission argument is required only if the O_CREAT flag is set in the access_mode argument. It specifies the access permission of the file for its owner, group and all the other people.
- Its data type is *int* and its value is octal integer value, such as 0764. The left-most, middle and right-most bits specify the access permission for owner, group and others respectively.
- In each octal digit the left-most, middle and right-most bits specify read, write and execute permission respectively.
- For example 0764 specifies 7 is for owner, 6 is for group and 4 is for other.
  7 = 111 specifies read, write and execution permission for owner.
  6 = 110 specifies read, write permission for group.
  4 = 100 specifies read permission for others.
  Each bit is either 1, which means a right is granted or zero, for no such rights.
- POSIX.1 defines the permission data type as mode_t and its value is manifested constants which are aliases to octal integer values. For example, 0764 permission value should be specified as:
  S_IRWXU|S_IRGRP|S_IWGRP|S_IROTH
- *Permission* value is modified by its calling process *umask* value. An *umask* value specifies some access rights to be masked off (or taken away) automatically on any files created by process.
- The function prototype of the *umask* API is:
  mode_t umask (mode_t new_umask);
  It takes new mask value as argument, which is used by calling process and the function returns the old umask value. For example,
  mode_t old_mask = umask (S_IXGRP | S_IWOTH |S_IXOTH);
  The above function sets the new umask value to "no execute for group" and "no write-execute for others".
- The open function takes its permission argument value and bitwise-ANDs it with the one's complement of the calling process umask value. Thus,
  actual_permission = permission & ~umask_value
  Example: **actual_permission = 0557 & (~031) = 0546**
  The return value of open function is -1 if the API fails and errno contains an error status value. If the API succeeds, the return value is file descriptor that can be used to reference the file and its value should be between 0 and OPEN_MAX-1.

Creat:

- The creat system call is used to create new regular files.

  Prototype:

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

  1. The path_name argument is the path name of a file to be created.
  2. The mode argument is same as that for open API.
- Since O_CREAT flag was added to open API it was used to both create and open regular files. So, the creat API has become obsolute. It is retained for backward-compatibility with early versions of UNIX.
- The creat function can be implemented using the open function as:

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

read:

- This function fetches a fixed size block of data from a file referenced by a given file descriptor.

  Prototype:

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

  - **fdesc:** is an integer file descriptor that refers to an opened file.
  - **buf:** is the address of a buffer holding any data read.
  - **size:** specifies how many bytes of data are to be read from the file.

  **Note: read function can read text or binary files. This is why the data type of buf is a universal pointer (void *).

  For example, the following code reads, sequentially one or more record of struct sample-typed data from a file called dbase:

  ```
  struct sample { int x; double y; char* a;} varX;
  int fd = open("dbase", O_RDONLY);
  while ( read(fd, &varX, sizeof(varX))>0)
  ```

- The return value of *read* is the number of bytes of data successfully read and stored in the *buf* argument. It should be equal to the *size* value.
- If a file contains less than *size* bytes of data remaining to be read, the return value of *read* will be less than that of *size*. If end-of-file is reached, *read* will return a zero value.
- *size_t* is defined as *int* in <sys/types.h> header, users should not set *size* to exceed INT_MAX in any *read* function call.
- If a read function call is interrupted by a caught signal and the OS does not restart the system call automatically, POSIX.1 allows two possible behaviors:
  1. The read function will return -1 value, errno will be set to EINTR, and all the data will be discarded.

2. The read function will return the number of bytes of data read prior to the signal interruption. This allows a process to continue reading the file.

- The read function may block a calling process execution if it is reading a FIFO or device file and data is not yet available to satisfy the read request. Users may specify the O_NONBLOCK or O_NDELAY flags on a file descriptor to request nonblocking read operations on the corresponding file.

write:

- The write function puts a fixed size block of data to a file referenced by a file descriptor

Its prototype is:-

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

- **fdesc:** is an integer file descriptor that refers to an opened file.
- **buf:** is the address of a buffer which contains data to be written to the file.
- **size:** specifies how many bytes of data are in the buf argument.

**Note: write function can read text or binary files. This is why the data type of buf is a universal pointer (void *). For example, the following code fragment writes ten records of struct sample-types data to a file called dbase2:

```
struct sample { int x; double y; char* a;} varX[10];
int fd = open("dbase2", O_WRONLY);
write(fd, (void*)varX, sizeof varX);
```

- The return value of *write* is the number of bytes of data successfully written to a file. It should be equal to the *size* value.
- If the write will cause the file size to exceed a system imposed limit or if the file system disk is full, the return value of write will be the actual number of bytes written before the function was aborted.
- If a signal arrives during a write function call and the OS does not restart the system call automatically, the write function may either return a -1 value and set errno to EINTR or return the number of bytes of data written prior to the signal interruption.
- The write function may perform nonblocking operation if the O_NONBLOCK or O_NDELAY flags are set on the fdesc argument to the function.

close:

- The close function disconnects a file from a process.

Prototype:

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

- **fdesc:** is an integer file descriptor that refers to an opened file.
- The return value of close is zero if the call succeeds or -1 if it fails.

- The close function frees unused file descriptors so that they can be reused to reference other files.
- The close function will deallocate system resources which reduces the memory requirement of a process.
- If a process terminates without closing all the files it has opened, the kernel will close files for the process.

fcntl:

➢ The fcntl function helps to query or set access control flags and the close-on-exec flag of any file descriptor. Users can also use fcntl to assign multiple file descriptors to reference the same file.

Prototype:

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

- **fdesc:** is an integer file descriptor that refers to an opened file.
- **cmd:** specifies which operation to perform on a file referenced by the fdesc argument.
- The third argument value, which may be specified after cmd is dependent on the actual cmd value.
- The possible cmd values are defined in the <fcntl.h> header.
- These values and their uses are:

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 and 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 the return value is nonzero and the flag is on. The close-on-exec flag of a newly opened file is off by default.
F_SETFD	Sets or clears the close-on-exec flag of a file descriptor fdesc. The third argument to fcntl is integer value, which is 0 to clear, or 1 to set the flag.
F_DUPFD	Duplicates the 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, in this case 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 nonblocking and in write-append mode, it can call fcntl on the file's descriptor as:

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


GANESH XEROX RNSIT

- The close-on-exec flag of a file descriptor specifies that if the process that owns the descriptor calls the exec API to execute different program, the fdesc should be closed by the kernel before the new program runs or not.
- The example reports the close-on-exec flag of a fdesc, sets it to on afterwards:
```
cout<<fdesc<<"close-on-exec:"<<fcntl(fdesc,F_GETFD)<<endl;
(void)fcntl(fdesc, F_SETFD, 1);
```
- The fcntl function can also be used to duplicate a fdesc with another fdesc. The results are two fdesc reference the same file with same access mode and share the same file pointer to read or write the file. This is useful in the redirection of standard input or output to reference a file.

  Example: Reference 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
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)
```
  The dup function duplicates a fdesc with the lowest unused fdesc of a calling process:
  The dup2 function will duplicate a fdesc using a fd2 fdesc, regardless of whether fd2 is used to reference another file.

lseek:

- The lseek system call can be used to change the file offset to a different value. It allows a process to perform random access of data on any opened file. lseek is incompatible with FIFO files, character device files and symbolic link files.

  Prototype:
```
#include <sys/types.h>
#include <unistd.h>
off_t lseek (int fdesc , off_t pos, int whence);
```
  - **fdesc:** is an integer file descriptor that refers to an opened file.
  - **pos:** specifies a byte offset to be added to a reference location in deriving the new file offset value.
  - **whence:** specifies 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

***NOTE:**

a. It is illegal to specify a negative pos value with the whence value set to SEEK_SET as this will set negative offset.

b. If an lseek call will result in a new file offset that is beyond end-of-file, two outcomes are possible:

1. If a file is opened for read only the lseek will fail.
2. If a file is opened for write access, lseek will succeed and it will extend the file size up to the new file offset address.

- The return value of lseek is the new file offset address where the next read of write operation will occur, or -1 if lseek call fails.
- The iostream class defines *tellg* and *seekg* functions to allow users to randomly access data from any isotream class.
- These functions can be implemented using the *lseek* function as follows:

```
#include<iostream.h>
#include<sys/types.h>
#include<unistd.h>
streampos iostream::tellg()
{
return (streampos)lseek(this->fileno(),(off_t)0,SEEK_CUR);
}
iostream&iostream::seekg(streampos pos,seek_dir ref_loc)
{
if(ref_loc == ios::beg)
        (void)lseek(this->fileno(), (off_t)pos, SEEK_SET);
else if(ref_loc == ios::cur)
        (void)lseek(this->fileno(), (off_t)pos, SEEK_CUR);
else if(ref_loc == ios::end)
        (void)lseek(this->fileno(), (off_t)pos, SEEK_END);
return *this;
}
```

- The iostream::tellg simply calls lseek to return the current file pointer associated with an iostream object. The file descriptor of an iostream object **const char*** is obtained from the fileno member function.
- The iostream::seekg relies on lseek to alter the file pointer associated with an iostream object. The arguments are file offset and a reference location for the offset. This function also converts seek_dir value to an lseek whence value.
- There is one-to-one mapping of the seek_dir values to the whence values used by lseek:

GANESH XEROX RNSIT

seek_dir value	lseek whence value
ios::beg	SEEK_SET
ios::cur	SEEK_CUR
ios::end	SEEK_END

link:

- The link function creates a new link for an existing file . This function does not create a new file. It create a new path name for an existing file. Its prototype is:

```
#include <unistd.h>

int link (const char* cur_link ,const char* new_link)
```

- **cur_link:** is a path name of an existing file.
- **new_link:** is a new path name to be assigned to the same file.
- If this call succeeds, the hard link count attribute of the file will be increased by 1.
- link cannot be used to create hard links across file systems. It cannot be used on directory files unless it is called by a process that has superuser previlege.

The *ln* command is implemented using the link API. The program is given below:

```
#include<stdio.h>
#include<unistd.h>
int main(int argc,char* argv[])
{
  if(argc!=3)
  {
            printf("usage:%s",argv[0]);
            printf("<src_file><dest_file>\n");
            return 0;
  }
  if(link(argv[1],argv[2]) == -1)
  {
            perror("link");
            return 1;
  }
  return 0;
}
```

unlink:

- This function deletes a link of an existing file. It decreases the hard link count attributes of the named file, and removes the file name entry of the link from a directory file.
- If this function succeeds the file can no longer be referenced by that link.
- File will be removed by the file system if the hard link count of the file is zero and no process has fdesc referencing that file.

Prototype:

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

- **cur_link:** is a path name of an existing file.

- The return value is 0 if it succeeds or -1 if it fails.
- The failure can be due to invalid link name and calling process lacks access permission to remove the path name.
- It cannot be used to remove directory files unless the calling process has superuser privilege.

ANSI C defines remove function which does the similar operation of unlink. If the argument to the remove functions is empty directory it will remove the directory.

Prototype:

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

The rename will fail when the new link to be created is in a different file system than the original file.

The *mv* command can be implemented using the link and unlink APIs by the program given below:

```
#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;
}
```

stat, fstat:

These functions retrieve the file attributes of a given file. The first argument of *stat* is file path name where as *fstat* is a file descriptor.

The prototype is given below:

```
#include <sys/types.h>
#include <unistd.h>
int stat (const char* path_name,struct stat* statv)
int fstat (const int fdesc,struct stat* statv)
```

The second argument to stat & fstat is the address of a struct stat-typed variable. The declaration of struct stat is given below:

```
struct stat
  {
    dev_ts    t_dev;//file system ID
    ino_t     st_ino;        //File inode number
    mode_t    st_mode;       //contains file type and access flags
    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 numbers
    off_t     st_size;         //file size in number of 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 it succeeds or -1 if it fails.
- Possible failures may be that a given file path name of file descriptor is invalid, the calling process lacks permission to access the file, or the function interrupted by a signal.
- If a path name argument specified to stat is a symbolic link file, stat will resolve the link and access the non symbolic link file. Both the functions cannot be used to obtain the attributes of symbolic link file.To obtain the attributes of symbolic link file lstat function was invented.

Prototype:

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

The UNIX ls command is implemented by the program given below:

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

static char xtbl[10] = "rwxrwxrwx";

static void display_file_type ( ostream& ofs, int st_mode )
{
   switch (st_mode &S_IFMT)
   {
      case S_IFDIR:   ofs << 'd'; return;        /* directory file */
      case S_IFCHR:   ofs << 'c'; return;        /* character device file */
      case S_IFBLK:   ofs << 'b'; return;        /* block device file */
      case S_IFREG:   ofs << ' '; return;        /* regular file */
      case S_IFLNK:   ofs << 'l'; return;        /* symbolic link file */
      case S_IFIFO:   ofs << 'p'; return;        /* FIFO file */
   }
}
/* Show access permission for owner, group, others, and any special flags */
static void display_access_perm ( ostream& ofs, int st_mode )
{
        char amode[10];
        for (int i=0, j= (1 << 8); i < 9; i++, j>>=1)
        amode[i] = (st_mode&j) ? xtbl[i] : '-';        /* set access permission */
```

```
        if (st_mode&S_ISUID) amode[2] = (amode[2]=='x') ? 'S' : 's';
        if (st_mode&S_ISGID) amode[5] = (amode[5]=='x') ? 'G' : 'g';
        if (st_mode&S_ISVTX) amode[8] = (amode[8]=='x') ? 'T' : 't';
        ofs << amode << ' ';
}
/* List attributes of one file */
static void long_list (ostream& ofs, char* path_name)
{
    struct stat          statv;
    struct group*gr_p;
    struct passwd*pw_p;
    if (lstat (path_name, &statv))
    {
        cerr<<"Invalid path name:"<< path_name<<endl;
        return;
    }
    display_file_type( ofs, statv.st_mode );
    display_access_perm( ofs, statv.st_mode );
    ofs << statv.st_nlink;                  /* display hard link count */
    gr_p = getgrgid(statv.st_gid);          /* convert GID to group name */
    pw_p = getpwuid(statv.st_uid);          /*convert UID to user name */

    ofs << ' ' <<(pw_p->pw_name ? pw_p->pw_name:statv.st_uid)
        << ' ' <<(gr_p->gr_name ? gr_p->gr_name:statv.st_gid)<< ' ';
    if ((statv.st_mode&S_IFMT) == S_IFCHR || (statv.st_mode&S_IFMT)==S_IFBLK)
        ofs << MAJOR(statv.st_rdev) << ',' <<   MINOR(statv.st_rdev);
    else ofs << statv.st_size;                  /* show file size or major/minor no. */
    ofs << ' ' << ctime (&statv.st_mtime);      /* print last modification time */
    ofs << ' ' << path_name << endl;            /* show file name */
}
/* Main loop to display file attributes one file at a time */
int main (int argc, char* argv[])
{
        if (argc==1)
            cerr << "usage: " << argv[0] << " <file path name> ...\n";
        else while (--argc >= 1) long_list( cout, *++argv);
            return 0;
}
```

access:

The access function checks the existence and/or access permission of user to a named file.

Prototype:

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

path_name: The pathname of a file.

flag: contains one or more of the following bit-flags.

Bit Flag	Use

F_OK	Checks whether a named file exists.
R_OK	Checks whether a calling process has read permission
W_OK	Checks whether a calling process has write permission
X_OK	Checks whether a calling process has execute permission

The flag argument value to access call is composed by bitwise-ORing one or more of the above bit-flags. The following statement checks whether a user has read and write permissions on a file /usr/sjb/file1.doc:

int rc = access("/usr/sjb/file1.doc", R_OK|W_OK);

- If a flag value is F_OK, the function returns 0 if the file exists and -1 otherwise.
- If a flag value is any combination of R_OK, W_OK and X_OK, the access function uses the calling process real user ID and real group ID to check against the file user ID and group ID. The function returns 0 if all the requested permission is permitted and -1 otherwise.

The following program uses access to determine, for each command line argument, whether a named file exists. If a ~~named~~ file does not exist, it will be created and initialized with a character string "Hello world".

```
#include<sys/types.h>
#include<unistd.h>
#include<fcntl.h>
int main(int argc, char*argv[])
{
    char buf[256];
    int fdesc,len;
    while(--argc>0) {
        if (access(*++argv,F_OK)) {                //a brand new file
         . fdesc = open(*argv, O_WRONLY|O_CREAT, 0744);
          write(fdesc, "Hello world\n", 12);
        }
        else  {
        fdesc = open(*argv, O_RDONLY);
        while(len = read(fdesc, buf,256))
           write(1, buf, len);
              }
              close(fdesc);
    }
}
```

chmod, fchmod:

The chmod and fcmod functions change file access permissions for owner, group and others and also set-UID, set-GID and sticky flags.

A process that calls one of these functions should have the effective user ID of either the super user or the owner of the file.

The prototype of these functions is given below:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int chmod (const char* path_name, mode_t flag);
int fchmod (int fdsec, mode_t flag);
```

The chmod function uses path name of a file as a first argument whereas fchmod uses fdesc as the first argument.

The flag argument contains the new access permission and any special flags to be set on the file.

For example: The following function turns on the set-UID flag, removes group write permission and others read and execute permission on a file named /usr/sjb/prog1.c

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

void change_mode( )
{
    struct stat statv;
    int flag = (S_IWGRP|S_IROTH|S_IXOTH);
    if (stat("/usr/sjb/prog1.c", &statv))
       perror("stat");
    else {
            flag = (statv.st_mode & ~flag) | S_ISUID;
            if (chmod("usr/sjb/prog1.c", flag))
                perror("chmod");
        }
}
```

chown, fchown, lchown:

- The chown and fchown functions change the user ID and group ID of files. They differ only in their first arguments which refer to a file by either a path name or a file descriptor.
- The lchown function changes the ownership of symbolic link file. The chown function changes the ownership of the file to which the symbolic link file refers.

  The function prototypes of these functions are given below:

```
#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);
```

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

GANESH XEROX RNSIT

- If the actual value of uid or gid argument is -1 the ID of the file is not changed.

In BSD UNIX:

- A process with super user privilege can use these functions to change any file user and group ID.
- If a process effective user ID matches a file user ID and its effective group ID or one of its supplementary group IDs match the file group ID, the process can change the file group ID only.

In UNIX System V:

- A process whose effective user ID matches either the user ID of a file or the user ID of a super user can change the file user ID and group ID.

In POSIX.1:

- Specifies that if the _POSIX_CHOWN_RESTRICTED variable is defined with a non -1 value, *chown* should behave as in BSD UNIX.
- If the _POSIX_CHOWN_RESTRICTED variable is undefined, *chown* should behave as in UNIX System V.

The following *test_chown.C* program implements the UNIX *chown* program:

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

int main (int argc, char" argv0)
 {
  if(argc<3) {
    cerr << "Usage: " << argv[0] << " <usr_name> <file> ...\n"; return 1;
  }
 }
 struct passwd *pwd = getpwuid( argv(1 ] );    /* r convert user name to UID **/
 uid t            UID = pwd ? pwd->pw_uid : -1;
 struct stat      state:
 if (UID == (uid_t)-1)
    cerr<< "Invalid user name\n";
 else for (int i=2; i < argc; i++)       /* do for each file specified *r
 if (stat(argv[i],&state)==o) {
       if (chown( argv(i], UID, staty.st_gid) )
          perror( "chown" );
    } else perror( "stat" );
 return 0;
}
```

This program takes, at least two command line arguments: the first one is a user name to be assigned to files, and the second and any subsequent arguments are file path names. The program

first converts a given user name to a user ID via the *getpwuid* function. If this succeeds, the program processes each named file as follows: it calls *stat to* get the file group ID, then calls *cltown* to change the file user ID. If either the stator *chown* API fails, *perror* will be called to print a diagnostic message.

utime:

The *utime* function modifies the access and modification time stamps of a file.
The prototype of the *utime* function is:

```
#include <sys/types.h>
#include <uni$td.h>
#include <utime.h>
int utime (const char* path_name, struct utimbuf* times );
```

The *path_name* argument is 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 utimbut
{    time_t  actime;          /* access time */
     time_t  modtime          /' modification time'/
}
```

If *times* is specified as 0, the API will set e named file access time and modification time to the current time. This requires that the calling process have write access to the named file, its effective user ID match either the file user ID or that of the super user.

The return value of *utime* is 0 if it succeeds or -1 if it fails. Possible failures of the API may be: The *path_name* argument is invalid, the process has no access permission and ownership to a named file, or the *times* argument has an invalid address.

The following *test_touch.C* program uses the *utime* function to change the access and modification time stamps of files. The time stamp to set is also defined by users:

```
/* Usage: a.out <offset in seconds> <file> ... */
#include <iostream.h>
#include <stdio.h>
#include <sys/types.h>
#include <utime.h>
#include <time.h>

int main (int argc, char* argv[] ) {
  struct utimbuf times;
  int            offset;

  if (argc < 3 II sscanf( argv[1], "%d", &offset) I= 1) {
    cerr << "usage:" << argv[0] << " <offset> <file> _An"; return 1;
  }

  /* new time is current time + offset in seconds */ times.actime =
  times.modtime = time( 0) + offset;
```

```
    for ( -i=1; i < argc; i ++)          /* touch each named file */

    if ( utime ( argv[i], &times)) perror( "utime" );
    return 0;
}
```

6.2 File and Record Locking:

- UNIX systems allow multiple processes to read and write the same file concurrently which provides data sharing among processes. It also renders difficulty for any process in determining when data in a file can be overridden by another process.
- In some of the applications like a database manager, where no other process can write or read a file while a process is accessing a database file. To overcome this drawback, UNIX and POSIX systems support a file locking mechanism.
- File locking is applicable only for regular files. It allows a process to impose a lock on a file so that other processes cannot modify the file until it is unlocked by the process.
- A process can impose a write lock or a read lock on either a portion of a file or an entire file.
- The difference between write locks and read locks is that when a write lock is set, it prevents other processes from setting any overlapping read or write locks on the locked region of a file. On the other hand, when a read lock is set, it prevents other processes from setting any overlapping write locks on the locked region of a file.
- The intention of a 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. A write lock is also known as an *exclusive lock*.
- The use of a 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 a *shared lock*.

Mandatory Lock

- Mandatory locks 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 data on the locked region.
- If a mandatory shared lock is set on a region of a file, no process can use the *write* system call to modify the locked region.
- It is used to synchronize reading and writing of shared files by multiple processes: If a process locks up a file, other processes that attempts to write to the locked regions are blocked until the former process releases its lock.
- Mandatory locks may cause problems: If a runaway process sets a mandatory exclusive lock on a file and never unlocks it, no other processes can access the locked region of the file until either the runaway process is killed or the system is rebooted.
- System V.3 and V.4 support mandatory locks.

GANESH XEROX PRINTING

Advisory Lock

- An advisory lock is not enforced by a kernel at the system call level.
- This means that even though lock (read or write) may be set on a file, other processes can still use the *read* or *write* APIs to access the file.
- To make use of advisory locks, processes that manipulate the same file must cooperate such that they follow this procedure for every read or write operation to the file:
  1. Try to set a lock at the region to be accessed. If this fails, a process can either wait for the lock request to become successful or go do something else and try to lock the file again later.
  2. After a lock is acquired successfully, read or write the locked region release the lock
- The drawback of advisory locks is that programs that create processes to share files must follow the above file locking procedure to be cooperative. This may be difficult to control when programs are obtained from different sources.
- All UNIX and POSIX systems support advisory locks.

UNIX System V and POSIX.I use the *fcntl* API for file locking.

The prototype of the *fcntl* API is:

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

The *fdesc* argument is a file descriptor for a file to be processed. The *cmd flag argu*ment defines which operation is to be performed.

cmd Flag	Use
F_SETLK	Sets a file lock. Do not block if this cannot succeed immediately
F_SETLKW	Sets a file lock and blocks the calling process until the lock is acquired
F_GETLK	Queries as to which process locked a specified region of a file

For file locking, the third argument to *fcntl is* an address of a *struct flock-typed* variable. This variable specifies a region of a file where the lock is to be set, unset, or queried. The *struct flock* is declared in the <fcntl.h> as:

```
struct flock
{
        short l_type;   // what lock to be set or to unlock file
        short l_whence;         // a reference address for the next field
        off_t l_start;  //offset from the l_whence reference address
        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 possible values of l_type are:

***l_type* value**	**Use**
F_RDLCK	Sets a a read (shared) lock on a specified region
F_WRLCK	Sets a write (exclusive) lock on a specified region
F_UNLCK	Unlocks a specified region

The possible values of l_whence and their uses are:

l_whence value	Use
SEEK_CUR	The *l_start* value is added to the current file pointer address
SEEK_SET	The *l_start* value is added to byte 0 of the file
SEEK_END	The *l_start* value ts'added to the end (current size) of the file

Lock Promotion and Lock splitting:

- 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 the process now unlocks 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.*

The procedure for setting the mandatory locks for UNIX system V3 and V4 are:

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

The following *file_lock.C* program illustrates a use of *fcntl* for file locking:

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

int main (int argc, cnar* argv[]) {
  struct flock        fvar;
  int                 fdesc;
  while (--argc > 0) {              /* do the following for each file */
   if ((fdesc=open(*++argv,O_RDWR ))==-1 ) {
        perror("open"); continue;
  }
  fvar.l_type      = F_WRLCK;
  fvar.l_whence = SEEK_SET;
  fvar.l_start     = 0;
  fvar.l_len       = 0;
     /* Attempt to set an exclusive (write) lock on the entire file */
     while (fcntl(fdesc, FSETLK,&fvar)==-1) {
        /* Set lock falls, find out who has locked the file */
        while (fcntl(fdesc,F_GETLK,&fvar)!=-1 && fvar.l_type != F_UNLCK){
        cout<<*argv<<"locked by"<<fvar.l_pid<<"from"<<fvar.l_start<<"for"<<fvar.l_len
            <<"byte for"<<(fvar.l_type == F_WRLCK ? 'w':'r')<<endl;
           if (!fvar.l_len) break;
           fvar.l_start += fvar.l_len;
           fvar.l_len   = 0;
           }/* while there are locks set by other processes */
       } /* while set lock un-successful */
```

```
        Lock the file OK. Now process data in the file */

      /* Now unlock the entire file */
      fvar.l_type      = F_UNLCK;
      fvar.l_whence    = SEEK_SET;
      fvar.l_start     = 0;
      fvar.l_len       = 0;
      if (fcntl(fdosc, F_SETLKW,&fear)==-1) perror("fcntl");
    }
    return 0;
  ) /* main */
```

6.3 Directory File APIs

- Directory files in UNIX and POSIX systems are used to help users in organizing their files into some structure based on the specific use of file.
- They are also used by the operating system to convert file path names to their inode numbers.
- Directory files are created in BSD UNIX and POSIX.1 by mkdir API:
  - #include <sys/stat.h>
  - #include <unistd.h>
  - int *mkdir (* **const char*** path_name, **mode** t mode);

1. The path_name argument is the path name of a directory to be created.
2. The mode argument specifies the access permission for the owner, group and others to be assigned to the file.
3. The return value of mkdir is 0 if it succeeds or -1 if it fails.

- UNIX System V.3 uses the *mknod* API to create directory files.
- UNIX System V.4 supports both the *mkdir* and *mknod APIs* for creating directory files.
- The difference between the two *APIs is* that a directory created by ***mknod*** **does not contain the "."** and ".." links. On the other hand, a directory created by *mkdir* has the "." and ".." links created in one atomic operation, and it is ready to be used.
- A directory file is a record-oriented file, where each record stores a file name and the mode number of a file that resides in that directory.
- The following portable functions are defined for directory file browsing. These functions are defined in both the <dirent.h> and <sys/dir.h> lieaders.

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

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

- The uses of these functions are:

  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.

  closedir: Closes a directory file referenced by *dir_fdesc*.

  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.

- UNIX systems support additional functions for random access of directory file records. These functions are not supported by POSIX.1:

  telldir: Returns the file pointer of a given dir_fdesc.

  seekdir: Changes the file pointer of a given dir_fdesc to a specified address.

- Directory files are removed by the *rmdir* API. Its prototype is given below:

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

- The following *list_dir.C* program illustrates uses of the *mkdir, opendir, readdir, closedir,* and rmdirAPIs:

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

int main (int argc, char* argv[ ])
 {
  Dirent*        dp;
  DIR* dir_fdesc;
  while (--argc > 0 ) {      /* do the following for each file */
  if ( !(dir_fdesc = opendir( *++argv ) )) {
        if (mkdir( *argv, S_IRWXU|S_IRWXG|S_IRWXO) == -1 )
         perror( "opendir" );
        continue;
      }
```

```
/*scan each directory file twice*/
    for (int i=0;i <2 ; i++) {
        for ( int cnt=0; dp=readdir( dir_fdesc );) {
            if (i) cout << dp->d_name << endl;
            if (strcmp( dp->d_name, ".") && strcmp( dp->d_name, ".. ") )
             cnt++;              /*count how many files in directory*/
        if (!cnt) { rmdir( *argv ); break;) /* empty directory */
        rewinddir( dir fdesc );      / reset pointer for second round */
    }
    closedir( dir fdesc );
  }
}
```

6.4 Device File APIs

- Device files are used to interface physical devices with application programs.
- Specifically, 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 files may be character-based or block-based.
- UNIX systems define the ***mknod*** API **to create device** files.

```
#include <sys/stat.h>
#include <unistd.h>
int mknod ( const char* path_name, mode t mode, int device_id );
```

1. The path_name argument is the path name of a directory to be created.
2. The mode argument specifies the access permission for the owner, group and others to be assigned to the file.
3. The *device_id* contains the major and minor device numbers and is constructed in most UNIX systems as follows: The lowest byte of *a device_id* is set to a minor device number and the next byte is set to the major device number. For example, to create a block device file called SCSI5 with major and minor numbers of 15 and 3, respectively, and access rights of read-write-execute for everyone, the *mknod* system call is:
   mknod("SCSI5", S_IFBLK | S_IRWXU | S_IRWXG | S_IRWXO, (15<<8) 13);
4. The major and minor device numbers are extended to fourteen and eighteen bits, respectively.
5. In UNIX, if a calling process has no controlling terminal and it opens a character device file, the kernel will set this device file as the controlling terminal of the process. However, if the O_NOCTTY flag is set in the *open* call, such action will be suppressed.
6. The O_NONBLOCK flag specifies that the *open* call and any subsequent *read* or *write* calls to a device file should be nonblocking to the process.

The following *test mknod.C* program illustrates 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>\n";
        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);
 }
```

6.5 FIFO File APIs

- FIFO files are also known as named pipes.
- They are special pipe device files used for inter process communication. Any process can attach to a FIFO file to read, write, or read-write data.
- Data written to a FIFO file are stored in a fixed-size (PIPE BUF) buffer and are retrieved in a FIFO order.

BSD UNIX and POSIX.1 define the *mkfifo* API to create FIFO files:

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

1. The path_name argument is the path name of a directory to be created.
2. The mode argument specifies the access permission for the owner, group and others to be assigned to the file, as well as the S_IFIFO flag to indicate that this is a FIFO file.
3. For example, to create a FIFO file called *FIFO5* with access permission of read-write-execute for everyone, the *mkfifo* call is:

```
mkfifo( "FIFO5", S_IFIFO | S_IRWXU | S_IRWXG | S_IRWXO );
```

- When a process opens a FIFO file for read-only, the kernel will block the process until there is another process that opens the same file for write.

- Similarly, if a process opens a FIFO for write, it will be blocked until another process opens the FIFO file for read. This provides a method for processes synchronization.
- Another special thing about FIFO files is that if a process writes to a FIFO file that has no other process attached to it for read, the kernel will send a SIGPIPE signal to the process to notify it of the illegal operation.
- If a process does not desire to be blocked by a, FIFO file, it can specify the O_NONBLOCK flag in the *open* call to the-FIFO file. With this flag, the *open* API will not block the process even though there is no process attached to the other end of the FIFO file.

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

```
#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>]\n";
     return 0;
  }
  int     fd;
  char    but [256];
  if (argue=2) {                    /' reader process * /
  (void)mkfifo( argv[1], S_IFIFO | S_IRWXU | S_IRWXG | S_IRWXO );
  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 {                          /* writer process' */ fd = open( argv[1], O_WRONLY
       );
       write( fd, argv[2], strlen( argv[2] ) );
  }
  close(fd);
}
```

6.6 Symbolic Link API:

- Symbolic links are developed to overcome several shortcomings of hard links:
  - Symbolic links can link files across file systems.
  - Symbolic links can link directory files.
  - Symbolic links always reference the latest version of the files to which they link.
- Symbolic links are being proposed to be included the POSIX.1 standard. BSD UNIX defines the following APIs for symbolic links manipulation:

```
#include<sys/types.h>
#include<sys/stat.h>
#include<unistd.h>
int symlink ( const char* org_link, const char* symlink );
int readlink ( const char* sym_link, char* buf, int size );
int lstat ( const char* sym_link, struct stat* state );.
```

symlink Function

- 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.
- For example, to create a symbolic-link called */usr/joe/lnk* for a file called */usr/go/testl*, the *symlink* call will be:

  symlink ("/usr/go/testl", "usr/roe/lnk");

- The return value of *symlink* is 0 if it succeeds or -1 if it fails, Possible causes of failure are: The path name specified is illegal, the *sym_link* file already exists, or the calling process lacks permission to create the new file.

readlink Function

- To query the path name to which a symbolic link refers, users must use the *readlink* API.
- The arguments to the *readlink* API are: *sym_link* is the path name of a symbolic link, *buf* is a character array buffer that holds the return path name referenced by the link, and *size* specifies the maximum capacity (in number of bytes) of the *buf* argument.
- The return value of *readlink is -1* if it fails or the actual number of characters of a path name that is placed in the *buf* argument.

The following function takes a symbolic link path name as argument, and it will call *readlink* repeatedly to resolve all links to the file. The *while* loop terminates when *readlink* returns -1, and the *buf* variable contains the nonlink file path name, which is then printed to the standard output:

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

int resolve_link( const char* sym_link )
{
  char* buf[256], tname[256];
  strcpy(tname,sym_link);
  while ( readlink( tname, buf, sizeof( buf)) > 0 )
  strcpy( tname, buf );
  cout <<sym_link << " --> " <<buf << endl;
}
```

lstat Function

- The *lstat* function is used to query the file attributes of symbolic links.
- The UNIX Is -1 command uses *lstat* to display information of all file type, including symbolic links.

The following *test_symin.C* program emulates the UNIX In command. The main function of the program is to create a link to a file. The names of the original file and new link are specified as the arguments to the program, and if the -s option is not specified, the program will create a hard link. Otherwise, it will create a symbolic link:

```
#include <iostream.h>
#include <sys/types.h>
#include <unistd.h>
 #include <string.h>
/* Emulate the UNIX In command */
 int main (int argc, char* argv[ ])
 {
    char* buf[256], tname[256];
    if ((argc< 3 && argc > 4) || (argc==4 && strcmp(argv[1 ];'-s")))
    {
     cout << "usage: " << argv[0] << " [-s] <orig file> <new_link>\n";
     return 1;
    }
    if (argc==4)
      return symlink( argv[2], argv[3]); /* create a symbolic link */
      else
      return link(argv[1]. argv[2]); /* create a hard link */
 }
```

UNIX PROCESSES AND PROCESS CONTROL

Chapter 7 The Environment of a UNIX Process

7.1 Introduction

- Before looking at the process control primitives, we need to examine the environment of a single process.
- We'll see how the main function is called when the program is executed.
- How command-line arguments are passed to the new program.
- What the typical memory layout looks like, how to allocate additional memory, how the process can use environment variables, and various ways for the process to terminate.
- Additionally, we'll look at the longjmp and setjmp functions and their interaction with the stack.
- Lastly we see the resource limits of a process.

7.2 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,
  → 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.

7.3 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

The start-up routine is also written so that if the main function returns, the exit function is called. If the start-up routine were coded in C (it is often coded in assembler) the call to main could look like

```
exit(main(argc, argv));
```

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>
void _exit(int status);
```

- The reason for the different headers is that exit and _Exit are specified by ISO C, whereas _exit is specified by POSIX.1.
- Historically, the exit function has always performed a clean shutdown of the standard I/O library: the fclose function is called for all open streams. This causes all buffered output data to be flushed (written to the file).
- All three exit functions expect a single integer argument, which we call the exit status. Most UNIX System shells provide a way to examine the exit status of a process. If
  (a) Any of these functions is called without an exit status,
  (b) main does a return without a return value, or
  (c) The main function is not declared to return an integer,
- The exit status of the process is undefined.
- However, if the return type of main is an integer and main "falls off the end" (an implicit return), the exit status of the process is 0.
- This behavior is new with the 1999 version of the ISO C standard. Historically, the exit status was undefined if the end of the main function was reached without an explicit return statement or call to the exit function.
- 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.

Example

The program below is the classic "hello, world" example.

```
#include <stdio.h>
main()
{
        printf("hello, world\n");
}
```

- When we compile and run this program, we see that the exit code is random. If we compile the same program on different systems, we are likely to get different exit codes, depending on the contents of the stack and register contents at the time that the main function returns:

```
$ cc hello.c
$ ./a.out
hello, world
$ echo $?              print the exit status
13
```

- Now if we enable the 1999 ISO C compiler extensions, we see that the exit code changes:

```
$ cc -std=c99 hello.c      enable gcc's 1999 ISO C extensions
hello.c:4: warning: return type defaults to 'int'
$ ./a.out
hello, world
$ echo $?              print the exit status
0
```

- Note the compiler warning when we enable the 1999 ISO C extensions. This warning is printed because the type of the main function is not explicitly declared to be an integer.

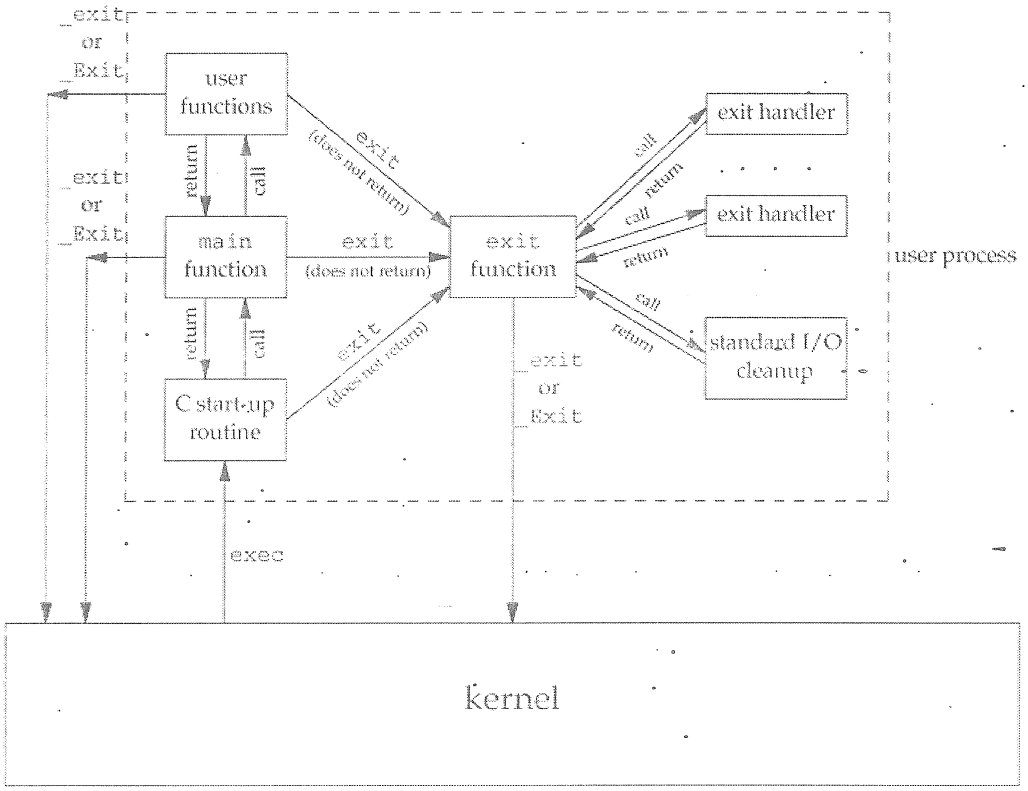
atexit Function

- 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.
- These exit handlers first appeared in the ANSI C Standard in 1989. Systems that predate ANSI C, such as SVR3 and 4.3BSD, did not provide these exit handlers.
- ISO C requires that systems support at least 32 exit handlers. The sysconf function can be used to determine the maximum number of exit handlers supported by a given platform.
- With ISO C and POSIX.1, exit first calls the exit handlers and then closes (via fclose) all open streams.
- POSIX.1 extends the ISO C standard by specifying that any exit handlers installed will be cleared if the program calls any of the exec family of functions.

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



- Note that the only way a program is executed by the kernel is when one of the exec functions is called. The only way a process voluntarily terminates is when _exit or _Exit is called, either explicitly or implicitly (by calling exit). A process can also be involuntarily terminated by a signal (not shown in Figure).

Example

1. The program in below demonstrates the use of the atexit function.

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

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

int main(void)
{
    if (atexit(my_exit2) != 0){
      printf("can't register my_exit2");
    exit(1);
    }
```

```
    if (atexit(my_exit1) != 0){
      printf("can't register my_exit1");
    exit(1);
    }

    if (atexit(my_exit1) != 0){
      printf("can't register my_exit1");
    exit(1);
    }

    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");

}
```

Executing this program in yields

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

An exit handler is called once for each time it is registered. In the program above, the first exit handler is registered twice, so it is called two times. Note that we don't call exit; instead, we return from main.

7.4 Command-Line Arguments

- When a program is executed, the process that does the exec can pass command-line arguments to the new program.
- This is part of the normal operation of the UNIX system shells.

Example

The program below echoes all its command-line arguments to standard output.

```
#include <stdio.h>
#include <stdlib.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);
}
```

If we compile this program and name the executable echoarg, we have

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

We are guaranteed by both ISO C and POSIX.1 that argv[argc] is a null pointer. This lets us alternatively code the argument-processing loop as

```
for (i = 0; argv[i] != NULL; i++)
```

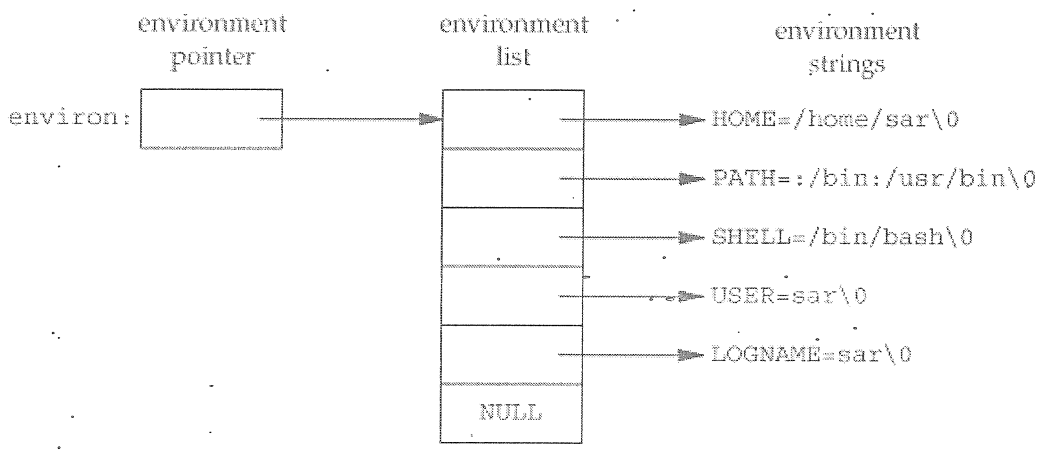
7.5 Environment List

- 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;
```


GANESH XEROX RNSIT

For example, if the environment consisted of five strings, it could look like as shown in figure below.



- By convention, the environment consists of name=value strings, as shown in figure above. Most predefined names are entirely uppercase, but this is only a convention.
- Historically, most UNIX systems have provided a third argument to the main function that is the address of the environment list:
  - int main(int argc, char *argv[], char *envp[]);
- Because ISO C specifies that the main function be written with two arguments, and because this third argument provides no benefit over the global variable environ, POSIX.1 specifies that environ should be used instead of the (possible) third argument. Access to specific environment variables is normally through the getenv and putenv functions, instead of through the environ variable. But to go through the entire environment, the environ pointer must be used.

7.6 Memory Layout of a C Program

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

- ***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.
- ***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.

- ***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

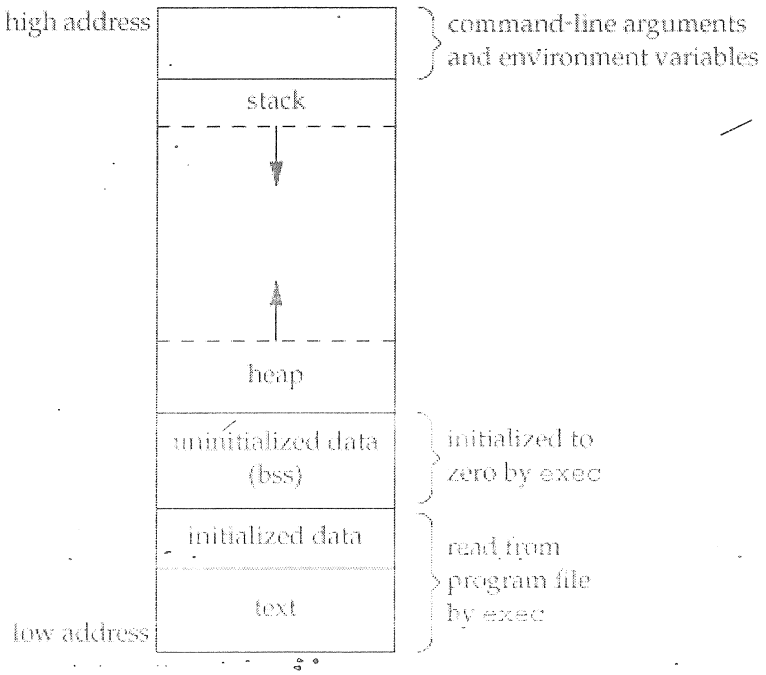
  long sum[1000];

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

- ***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.

- ***Heap,*** where dynamic memory allocation usually takes place. Historically, the heap has been located between the uninitialized data and the stack.

The figure below shows the typical arrangement of these segments.



- This is a logical picture of how a program looks; there is no requirement that a given implementation arrange its memory in this fashion. Nevertheless, this gives us a typical arrangement to describe.

- With Linux on an Intel x86 processor, the text segment starts at location 0x08048000, and the bottom of the stack starts just below 0xC0000000. (The stack grows from higher-numbered addresses to lower-numbered addresses on this particular architecture.) The unused virtual address space between the top of the heap and the top of the stack is large.
- Several more segment types exist in an a.out, containing the symbol table, debugging information, linkage tables for dynamic shared libraries, and the like. These additional sections don't get loaded as part of the program's image executed by a process.
- Note from then figure above that the contents of the uninitialized data segment are not stored in the program file on disk.
- This is because the kernel sets it to 0 before the program starts running. The only portions of the program that need to be saved in the program file are the text segment and the initialized data.
- The **size** command reports the sizes (in bytes) of the text, data, and bss segments.

  For example:

```
$ size /usr/bin/cc /bin/sh
   text    data   bss    dec     hex     filename
  79606    1536   916   82058   1408a    /usr/bin/cc
 619234   21120 18260  658614   a0cb6    /bin/sh
```

  The fourth and fifth columns are the total of the three sizes, displayed in decimal and hexadecimal, respectively.

7.7 Shared Libraries

- Most UNIX systems today 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 relink edit every program that uses the library. (This assumes that the number and type of arguments haven't changed.)
- Different systems provide different ways for a program to say that it wants to use or not use the shared libraries.
- Options for the **cc** and **ld** commands are typical. As an example of the size differences, the following executable file—the classic hello.c program—was first created without shared libraries:

```
$ cc -static hello1.c        prevent gcc from using shared libraries
$ size a.out
   text    data    bss     dec     hex    filename
 375657    3780   3220  382657   5d6c1    a.out
```

- If we compile this program to use shared libraries, the text and data sizes of the executable file are greatly decreased:

```
$ cc hello1.c            gcc defaults to use shared libraries
$ size a.out
   text    data    bss    dec     hex   filename
   872     256     4      1132    46c   a.out
```

7.8 Memory Allocation

ISO C specifies three functions for memory allocation:

a. *malloc*, which allocates a specified number of bytes of memory. The initial value of the memory is indeterminate.
b. *calloc*, which allocates space for a specified number of objects of a specified size. The space is initialized to all 0 bits.
c. *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. For example, if the most restrictive alignment requirement on a particular system requires that doubles must start at memory locations that are multiples of 8, then all pointers returned by these three functions would be so aligned.
- The allocation routines are usually implemented with the ***sbrk*** system call. This system call expands (or contracts) the heap of the process.
- 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.
- Other possible errors that can be fatal are freeing a block that was already freed and calling free with a pointer that was not obtained from one of the three alloc functions.
- If a process calls malloc, but forgets to call free, its memory usage continually increases; **this is called leakage.** By not calling free to return unused space, the size of a process's address space

slowly increases until no free space is left. During this time, performance can degrade from excess paging overhead.

- 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.

Alternate Memory Allocators

Many replacements for malloc and free are available. We discuss some of the alternatives here.

a. **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.

b. **Vmalloc:** Vo, Kiem-Phong [1996] 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.

c. **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. Weinstock and Wulf [1988] describe the algorithm, which is based on splitting up memory into buffers of various sizes and maintaining unused buffers on different free lists, depending on the size of the buffers. Free implementations of malloc and free based on quick-fit are readily available from several FTP sites.

d. **alloca Function:** he 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. Nevertheless, many software packages use it, and implementations exist for a wide variety of systems.

7.9. 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.

SRI SAIVA XEROX
#147, 1st Floor, Opp. JSS College, Near UCO Bank, Dr.Vishnuvardhan Road, Kalpa Layout, Bangalore - 560 060. Ph: 9535001507, 7378870591.

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.

ISO C defines a function that we can use to fetch values from the environment, but this standard says that the contents of the environment are implementation defined.

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

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

Table below shows the functions that are supported by the various standards and implementations.

Function	ISO C	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
getenv	•	•	•	•	•	•
putenv		XSI	•	•	•	•
setenv		•	•	•	•	
unsetenv		•	•	•	•	
clearenv				•		

Note: clearenv is not part of the Single UNIX Specification. It is used to remove all entries from the environment list.

The prototypes for the middle three functions listed in table above 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 operation of these three functions is as follows.

a. 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.

b. 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.

c. The unsetenv function removes any definition of name. It is not an error if such a definition does not exist.

7.10 setjmp and longjmp Functions

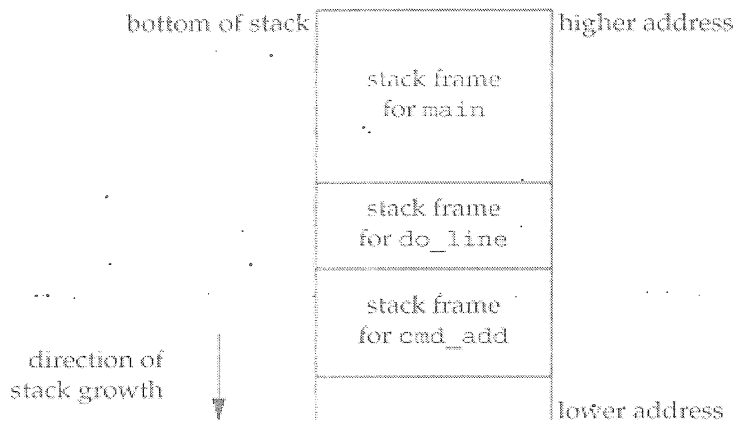
- In C, we can't goto a label that's in another function. Instead, we must use the setjmp and longjmp functions to perform this type of branching.
- These two functions are useful for handling error conditions that occur in a deeply nested function call.

Consider the skeleton of a program as shown below:

```
#include <stdio.h>

#define TOK_ADD   5

void    do_line(char *);
void    cmd_add(void);
int     get_token(void);

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

    while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
    exit(0);
}

char    *tok_ptr;    /* global pointer for get_token() */

void do_line(char *ptr)    /* process one line of input */
{
    int    cmd;

    tok_ptr = ptr;
    while ((cmd = get_token()) > 0) {
        switch (cmd) { /* one case for each command */
        case TOK_ADD:
            cmd_add();
            break;
        }
    }
}

void cmd_add(void)
{
    int    token;

    token = get_token();
    /* rest of processing for this command */
}

int get_token(void)
{
    /* fetch next token from line pointed to by tok_ptr */
```

It consists of a main loop that reads lines from standard input and calls the function do_line to process each line. This function then calls get_token to fetch the next token from the input line. The first token of a line is assumed to be a command of some form, and a switch statement selects each command. For the single command shown, the function cmd_add is called.

Figure below shows what the stack could look like after cmd_add has been called.



- Storage for the automatic variables is within the stack frame for each function. The array *line* is in the stack frame for main, the integer *cmd* is in the stack frame for do_line, and the integer *token* is in the stack frame for cmd_add.
- The coding problem that's often encountered with programs like the one shown above is how to handle nonfatal errors.
- For example, if the cmd_add function encounters an error—say, an invalid number—it might want to print an error, ignore the rest of the input line, and return to the main function to read the next input line.
- But when we're deeply nested numerous levels down from the main function, this is difficult to do in C.
- In this example, in the cmd_add function, we're only two levels down from main, but it's not uncommon to be five or more levels down from where we want to return to. It becomes messy if we have to code each function with a special return value that tells it to return one level.
- The solution to this problem is to use the setjmp and longjmp functions.

The prototypes of the two functions are as:

```
#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);
```


GANESH XEROX RNSIT

- We call setjmp from the location that we want to return to, which in this example is in the main function. In this case, setjmp returns 0 because we called it directly. In the call to setjmp, the argument env is of the special type jmp_buf. This data type is some form of array that is capable of holding all the information required to restore the status of the stack to the state when we call longjmp. Normally, the env variable is a global variable, since we'll need to reference it from another function.
- When we encounter an error—say, in the cmd_add function—we call longjmp with two arguments. The first is the same env that we used in a call to setjmp, and the second, val, is a nonzero value that becomes the return value from setjmp.
- The reason for the second argument is to allow us to have more than one longjmp for each setjmp. For example, we could longjmp from cmd_add with a val of 1 and also call longjmp from get_token with a val of 2. In the main function, the return value from setjmp is either 1 or 2, and we can test this value, if we want, and determine whether the longjmp was from cmd_add or get_token.
- The program below shows the modification in main and cmd_add functions. The other two functions, do_line and get_token, haven't changed.

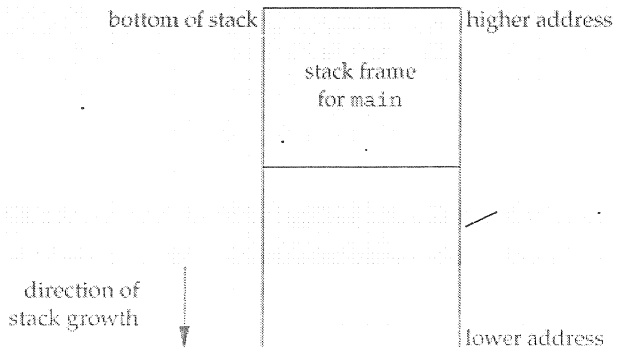
```
#include <stdio.h>
#include <setjmp.h>

#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 */
}
```

- When main is executed, we call setjmp, which records whatever information it needs to in the variable jmpbuffer and returns 0. We then call do_line, which calls cmd_add, and assume that an error of some form is detected.

- Before the call to longjmp in cmd_add, the stack looks like that in figure above. But longjmp causes the stack to be "unwound" back to the main function, throwing away the stack frames for cmd_add and do_line (figure below). Calling longjmp causes the setjmp in main to return, but this time it returns with a value of 1 (the second argument for longjmp).

Automatic, Register, and Volatile Variables

- We've seen what the stack looks like after calling longjmp. The next question is, "what are the states of the automatic variables and register variables in the main function?"
- When main is returned to by the longjmp, do these variables have values corresponding to when the setjmp was previously called (i.e., are their values rolled back), or are their values left alone so that their values are whatever they were when do_line was called (which caused cmd_add to be called, which caused longjmp to be called)?
- Unfortunately, the answer is "it depends." Most implementations do not try to roll back these automatic variables and register variables, but the standards say only that their values are indeterminate. If you have an automatic variable that you don't want rolled back, define it with the volatile attribute. Variables that are declared global or static are left alone when longjmp is executed.



Example

The program below demonstrates the different behavior that can be seen with automatic, global, register, static, and volatile variables after calling longjmp.

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

static void f1(int, int, int, int);
static void f2(void);

static jmp_buf jmpbuffer;
static int     globval;
int main(void)
{
    int          autoval;
    register int   regival;
    volatile int   volaval;
```

```
    static int      statval;

    globval = 1; autoval = 2; regival = 3;
    volaval = 4; statval = 5;

    if (setjmp(jmpbuffer) != 0) {
        printf("after longjmp:\n");
        printf("globval = %d, autoval = %d, regival = %d,"
            " volaval = %d, statval = %d\n",
            globval, autoval, regival, volaval, statval);
        exit(0);
    }
    /* Change variables after setjmp, but before longjmp.*/
    globval = 95; autoval = 96; regival = 97; volaval = 98;
    statval = 99;

    f1(autoval, regival, volaval, statval); /* never returns */
    exit(0);
}

static void f1(int i, int j, int k, int l)
{
    printf("in f1():\n");
    printf("globval = %d, autoval = %d, regival = %d,"
        " volaval = %d, statval = %d\n", globval, i, j, k, l);
    f2();
}

static void f2(void)
{
    longjmp(jmpbuffer, 1);
}
```

1. If we compile and test the above program with and without compiler optimizations, the results are different:

```
$ cc testjmp.c          compile without any optimization
$ ./a.out
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99

$ cc -O testjmp.c       compile with full optimization
$ ./a.out
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 2, regival = 3, volaval = 98, statval = 99
```

- The setjmp(3) manual page on one system states that variables stored in memory will have values as of the time of the longjmp, whereas variables in the CPU and floating-point registers are restored to their values when setjmp was called.
- This is indeed what we see when we run the program above. Without optimization, all five variables are stored in memory (the register hint is ignored for regival). When we enable

optimization, both autoval and regival go into registers, even though the former wasn't declared register, and the volatile variable stays in memory.

- The thing to realize with this example is that you must use the volatile attribute if you're writing portable code that uses nonlocal jumps. Anything else can change from one system to the next

7.11. getrlimit and setrlimit Functions

- Every process has a set of resource limits, some of which can be queried and changed by the getrlimit and setrlimit functions.
- The prototypes of these functions are:

```
#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

- The resource limits for a process are normally established by process 0 when the system is initialized and then inherited by each successive process. Each implementation has its own way of tuning the various limits.
- 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) A process can change its soft limit to a value less than or equal to its hard limit.
b) 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.
c) Only a superuser process can raise a hard limit.

- An infinite limit is specified by the constant RLIM_INFINITY.

The resource argument takes on one of the following values.

RLIMIT_AS	The maximum size in bytes of a process's total available memory. This affects the sbrk function and the mmap function.
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 mlock.
RLIMIT_NOFILE	The maximum number of open files per process. Changing this limit affects the value returned by the sysconf function for its _SC_OPEN_MAX argument.
RLIMIT_NPROC	The maximum number of child processes per real user ID. Changing this limit affects the value returned for _SC_CHILD_MAX by the sysconf 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 RLIMIT_AS.

- 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

The program below prints out the current soft limit and hard limit for all the resource limits supported on the system.

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

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

#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)
        printf("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');
}
```

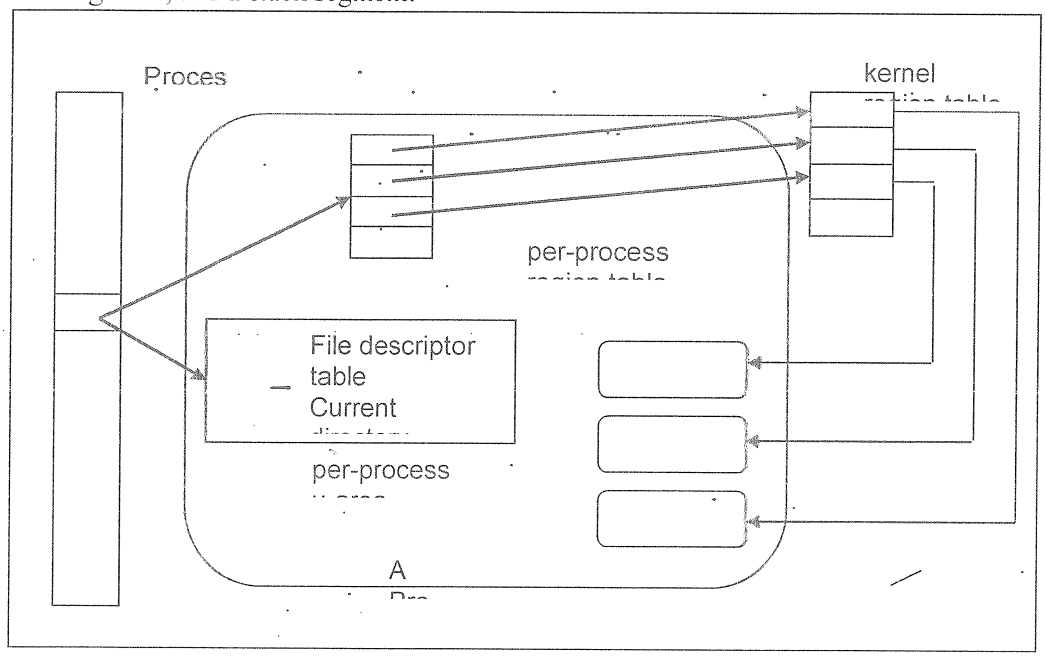
Running this program under FreeBSD gives us the following:

```
$ ./a.out
RLIMIT_CORE      (infinite) (infinite)
RLIMIT_CPU       (infinite) (infinite)
RLIMIT_DATA      536870912  536870912
RLIMIT_FSIZE     (infinite) (infinite)
RLIMIT_MEMLOCK   (infinite) (infinite)
RLIMIT_NOFILE        1735       1735
RLIMIT_NPROC          867        867
RLIMIT_RSS       (infinite) (infinite)
RLIMIT_SBSIZE    (infinite) (infinite)
```

RLIMIT_STACK 67108864 67108864
RLIMIT_VMEM (infinite) (infinite)

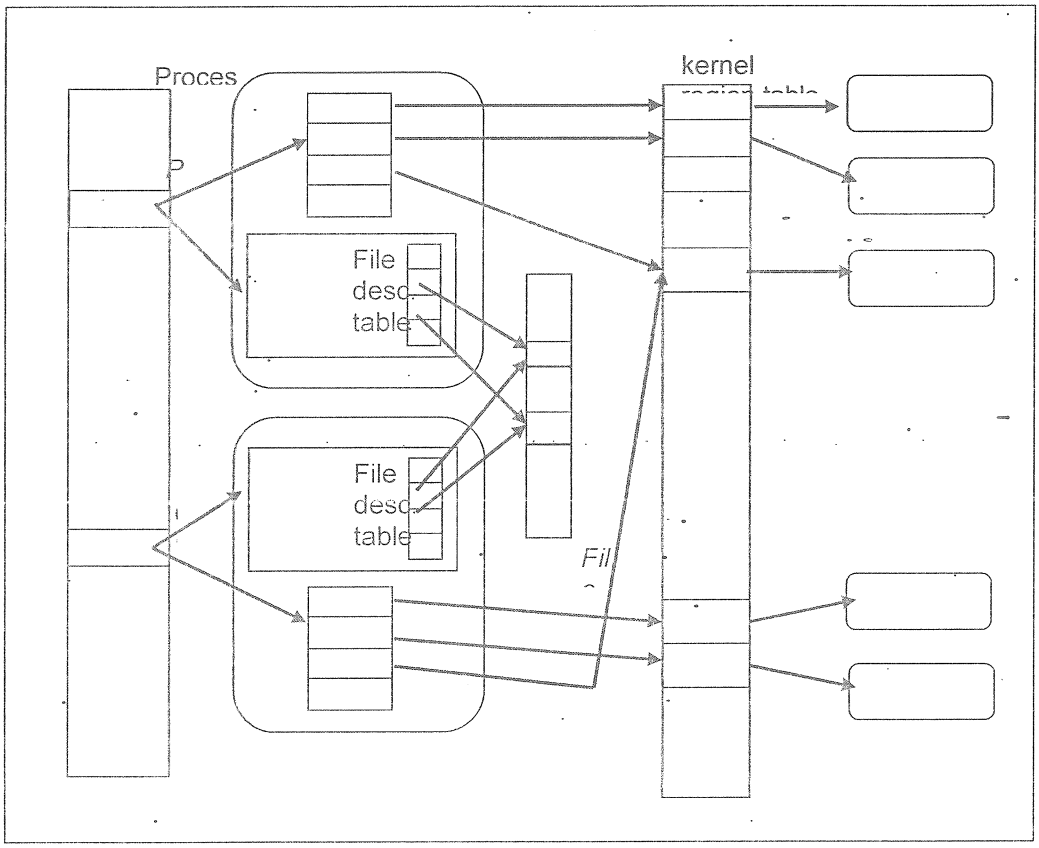
7.12 UNIX Kernel Support for processes.

- The data structure and execution of processes are dependent on operating system implementation.
- As shown in the figure below, a UNIX process consists minimally of a text segment, 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 contains the program text of a process in machine—executable instruction code format. A data segment contains static and global variables and their corresponding data. A stack segment contains a run-time stack. A stack provides storage for function arguments, automatic variables, and return address of all active functions for a process at any time.
- A UNIX kernel has a process table that keeps track of all active processes. Some of the processes belong to the kernel, they are called *system processes*. The majority of processes are associated with the users who are logged in.
- Each entry in the process table contains pointers to the text, data, stack segments and the U-area of a process.
- The U-area is an extension of a process table entry and contains other process specific data, such as the file descriptor table, current root, and working directory inode numbers, and a set of system –imposed process resource limits, etc.
- All processes in UNIX system, except the first process (process 0) which is created by the system boot code, are created via the *fork* system call.
- After the fork system call both the parent and child processes resume execution at the return of the fork function.

- As shown in the figure below, when a process is created by fork, it contains duplicate copies of the text, data, and stack segments of its parent. Also, it has an FDT that contains references to the same opened files as its parent, such that they both share the same file pointer to each opened file.



Furthermore, the process is assigned the following attributes which are either inherited by from its parent or set by the kernel.

a) ***A real user identification number (rUID):*** the user ID of a user who created the parent process.
b) ***A real group identification number (rGID):*** the group ID of a user who created the parent process.
c) ***An effective user identification number (eUID):*** this is normally the same as the real UID, except when the file that was executed to create the process has its set UID flag turned on, in that case the eUID will take on the UID of the file.
d) ***An effective group identification number (eGID):*** this is normally the same as the real GID, except when the file that was executed to create the process has its set UID flag turned on, in that case the eGID will take on the GID of the file.

GANESH XEROX RNSIT

e) ***Saved set-UID and saved set-GID***: these are the assigned eUID and eGID, respectively of the process.

f) ***Process group identification number (PGID) and session identification number (SID)***: these identify the process group and session of which the process is member.

g) ***Supplementary group identification numbers***: this is a set of additional group IDs for a user who created the process.

h) ***Current Directory***: this is the reference (inode number) to a working directory file.

i) ***Root Directory***: this is the reference (inode number) to a root directory file.

j) ***Signal handling***: the signal handling settings.

k) ***Signal mask***: a signal mask that specifies which signals are to be blocked.

l) ***Umask***: a file mode mask that is used in creation of files to specify which accession rights should be taken out.

m) ***Nice value***: the process scheduling priority value.

n) ***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.
  a) ***Process identification number (PID)***: an integer identification number that is unique per process in an entire operating system.
  b) ***Parent process identification number (PPID)***: the parent process ID.
  c) ***Pending signals***: the set of signals that are pending delivery to the parent process. This is reset to none in the child process.
  d) ***Alarm clock time***: the process alarm clock time is reset to zero in the child process.
  e) ***File locks***: the set of file locks owned by the parent process is not inherited by the child process.
- After *fork* a parent process may choose to suspend its execution until its child process terminates by calling the *wait* or *waitpid* system call, or it may continue execution independently of its child process. In the later case, the parent process may use the *signal* or *sigaction* function to detect or ignore the child process termination.
- A process terminates its execution by calling the *_exit* system call. The argument to the *_exit* call is the exit status code of the process. By convention, an exit status code of zero means the process has completed its execution successfully, and any non zero exit code indicates failure has occurred.
- A process can execute a different program by calling the *exec* system call. If the call succeeds, the kernel will replace the process's existing text, data and stack segments with a new set that represents the new program to be executed.
- However, the process is still the same process (the PID and PPID are the same), and its FDT opened directory stream remain the same (except that those FDs which have their *close-on-exec* flag set via *fcntl* system call will be closed upon *exec*'ing).
- When the *exec*'ed program completes its execution, it terminates the process. The exit status code of the program may be polled by the process's parent via the *wait* or *waitpid* function.

- *fork and exec* are commonly used together to spawn a subprocess to execute a different program. For example UNIX shell executes each user command by calling *fork and exec* to execute the requested command in a child process.

The advantage of this method are:

- The 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.

- Two or more related processes (parent to child, or child to child with the same parent) may communicate with others by setting up unnamed pipes among them.
- For unrelated processes, they can communicate using named pipe or interprocess communication methods.

GANESH XEROX RICHE

Chapter 8 PROCESS CONTROL

8.1. Introduction

Process control includes the creation of new processes, program execution, and process termination. We will also look at the various IDs that are the property of the process—real, effective, and saved; user and group IDs—and how they're affected by the process control primitives. Interpreter files and the system function and the process accounting provided by most UNIX systems.

8.2. Process Identifiers

Every process has a unique process ID, a non-negative integer. Although unique, processes IDs are reused. As processes terminate, their IDs become candidates for reuse. Most UNIX systems implement algorithms to delay reuse, however, so that newly created processes are assigned IDs different from those used by processes that terminated recently. This prevents a new process from being mistaken for the previous process to have used the same ID.

1. Process ID 0 is usually the scheduler process and is often known as the swapper. No program on disk corresponds to this process, which is part of the kernel and is known as a system process.

2. Process ID 1 is usually the init process and is invoked by the kernel at the end of the bootstrap procedure. The program file for this process was /etc/init in older versions of the UNIX System and is /sbin/init in newer versions. This process is responsible for bringing up a UNIX system after the kernel has been bootstrapped.

init usually reads the system-dependent initialization files—the /etc/rc* files or /etc/inittab and the files in /etc/init.d—and brings the system to a certain state, such as multi-user. The init process never dies. It is a normal user process, not a system process within the kernel, although it does run with *superuser* privileges.

Each UNIX System implementation has its own set of kernel processes that provide operating system services. For example, on some virtual memory implementations of the UNIX System, process ID 2 is the *page daemon*. This process is responsible for supporting the paging of the virtual memory system.

In addition to the process ID, there are other identifiers for every process. The following functions return these 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

Note that none of these functions has an error return.

8.3. fork Function

An existing process can create a new one by calling the fork function. The prototype for the fork function is:

```
#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. 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.

Current implementations don't perform a complete copy of the parent's data, stack, and heap, since a fork is often followed by an exec. Instead, a technique called *copy-on-write (COW)* is used. These regions are shared by the parent and the child and have their protection changed by the kernel to read-only. If either process tries to modify these regions, the kernel then makes a copy of that piece of memory only, typically a "page" in a virtual memory system.

Example

The program below demonstrates the fork function, showing how changes to variables in a child process do not affect the value of the variables in the parent process.

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
int     glob = 6;     /* external variable in initialized data */
char    buf[] = "a write to stdout\n";
int main(void)
{
   int      var;     /* automatic variable on the stack */
   pid_t    pid;
   var = 88;
   if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
      perror("write");
   printf("before fork\n");    /* we don't flush stdout */
   if ((pid = fork()) < 0) {
      perror("fork");
```

```
    } else if (pid == 0) {      /* child */
        glob++;                 /* modify variables */
        var++;
    } else {
        sleep(2);               /* parent */
    }
    printf("pid= %d, glob= %d, var= %d\n", getpid(), glob, var);
    exit(0);
}
```

```
$ ./a.out
a write to stdout
before fork
pid = 430, glob = 7, var = 89     child's variables were changed
pid = 429, glob = 6, var = 88     parent's copy was not changed

$ ./a.out > temp.out
$ cat temp.out
a write to stdout
before fork
pid = 432, glob = 7, var = 89
before fork
pid = 431, glob = 6, var = 88
```

We never know whether the child starts executing before the parent or vice versa. This depends on the scheduling algorithm used by the kernel. If it's required that the child and parent synchronize, some form of inter-process communication is required. In the program shown above, we simply have the parent put itself to sleep for 2 seconds, to let the child execute. The write function is not buffered. Because write is called before the fork, its data is written once to standard output. The standard I/O library, however, is buffered and that the standard output is line buffered if it's connected to a terminal device; otherwise, it's fully buffered.

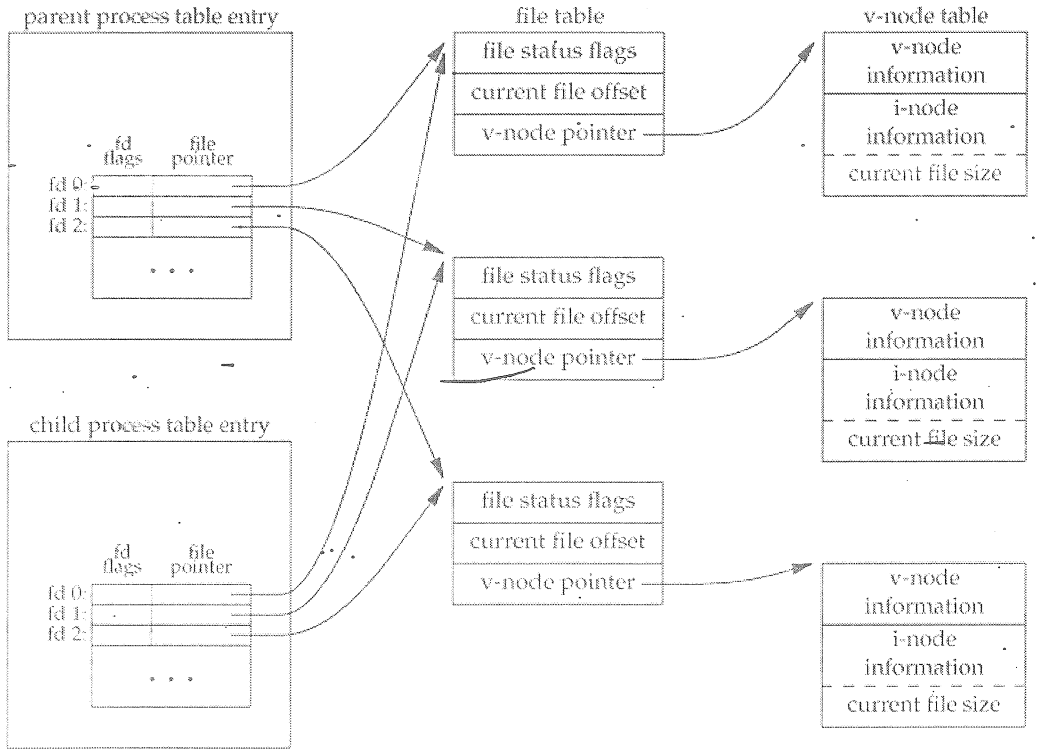
When we run the program interactively, we get only a single copy of the printf line, because the standard output buffer is flushed by the newline. But when we redirect standard output to a file, we get two copies of the printf line. In this second case, the printf before the fork is called once, but the line remains in the buffer when fork is called. This buffer is then copied into the child when the parent's data space is copied to the child. Both the parent and the child now have a standard I/O buffer with this line in it. The second printf, right before the exit, just appends its data to the existing buffer. When each process terminates, its copy of the buffer is finally flushed.

File Sharing

When we redirect the standard output of the parent from the above program, the child's standard output is also redirected. One characteristic of fork is that all file descriptors that are open in the parent are duplicated in the child. The parent and the child share a file table entry for every open descriptor.

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 below. It is important that the parent and the child share the same file offset. Consider a process that forks a child, and 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.



1. If both parent and child write to the same descriptor, without any form of synchronization, such as having the parent wait for the child, their output will be intermixed
2. There are two normal cases for handling the descriptors after a fork.
   a. **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.
   b. **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.

Besides the open files, there are numerous other properties of the parent that are inherited by the child:

a. Real user ID, real group ID, effective user ID, effective group ID
b. Supplementary group IDs
c. Process group ID
d. Session ID
e. Controlling terminal
f. The set-user-ID and set-group-ID flags

g. Current working directory
h. Root directory
i. File mode creation mask
j. Signal mask and dispositions
k. The close-on-exec flag for any open file descriptors
l. Environment
m. Attached shared memory segments
n. Memory mappings
o. 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
e. File locks set by the parent are not inherited by the child
f. Pending alarms are cleared for the child
g. The set of pending signals for the child is set to the empty set

The two main reasons for fork to fail are:

(a) If too many processes are already in the system, which usually means that something else is wrong.

(b) If the total number of processes for this real user ID exceeds the system's limit. Recall that CHILD_MAX specifies the maximum number of simultaneous processes per real user ID.

There are two uses for fork:

1. When a process wants to duplicate itself so that the parent and child can each execute different sections of code at the same time. 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.
2. 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.

8.4. vfork Function

The function vfork has the same calling sequence and same return values as fork. But the semantics of the two functions differ. The vfork function originated with 2.9BSD. It 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. (**This can**

lead to deadlock if the child depends on further actions of the parent before calling either of these two functions.)

Example

The program below is a modified version of the program from fork function. We've replaced the call to fork with vfork and removed the write to standard output. Also, we don't need to have the parent call sleep, as we're guaranteed that it is put to sleep by the kernel until the child calls either exec or exit.

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.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) {
      perror("vfork");
   } 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);
}
```

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

Here, the incrementing of the variables done by the child changes the values in the parent. Because the child runs in the address space of the parent. This behavior, however, differs from fork.

8.5. exit Functions

As we have seen a process can terminate normally in five ways:

a. Executing a return from the main function. This is equivalent to calling exit.

b. Calling the exit function. This function is defined by ISO C and includes the calling of all exit handlers that have been registered by calling atexit and closing all standard I/O streams. Because ISO C does not deal with file descriptors, multiple processes (parents and children), and job control, the definition of this function is incomplete for a UNIX system.

c. Calling the _exit or _Exit function. ISO C defines _Exit to provide a way for a process to terminate without running exit handlers or signal handlers. Whether or not standard I/O streams are flushed depends on the implementation. On UNIX systems, _Exit and _exit are synonymous and do not flush standard I/O streams. The _exit function is called by exit and handles the UNIX system-specific details; _exit is specified by POSIX.1.

d. Executing a return from the start routine of the last thread in the process. The return value of the thread is not used as the return value of the process, however. When the last thread returns from its start routine, the process exits with a termination status of 0.

e. Calling the pthread_exit function from the last thread in the process. As with the previous case, the exit status of the process in this situation is always 0, regardless of the argument passed to pthread_exit.

The three forms of abnormal termination are as follows:

a. Calling abort. This is a special case of the next item, as it generates the SIGABRT signal.

b. When the process receives certain signals. The signal can be generated by the process itself—for example, by calling the abort function—by some other process, or by the kernel. 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.

c. 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.

Regardless of how a process terminates, the same code in the kernel is eventually executed. This kernel code closes all the open descriptors for the process, releases the memory that it was using, and the like.

For any of the preceding cases, we want the terminating process to be able to notify its parent how it terminated. For the three exit functions (exit, _exit, and _Exit), this is done by passing an exit status as the argument to the function.
In the case of an abnormal termination, however, the kernel, not the process, generates a termination status to indicate the reason for the abnormal termination. In any case, the parent of the process can obtain the termination status from either the wait or the waitpid function.

Note that we differentiate between the exit status, which is the argument to one of the three exit functions or the return value from main, and the termination status. The exit status is converted into a termination status by the kernel when _exit is finally called.
The Table 5a below describes the various ways the parent can examine the termination status of a child. If the child terminated normally, the parent can obtain the exit status of the child.

Table 8a: Macros to examine the termination status returned by wait and waitpid

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 (XSI extension to POSIX.1; waitpid only).

But what happens if the parent terminates before the child?
The init process becomes the parent process of any process whose parent terminates.
Whenever a process terminates, the kernel goes through all active processes to see whether the terminating process is the parent of any process that still exists. If so, the parent process ID of the surviving process is changed to be 1 (the process ID of init). This way, we are guaranteed that every process has a parent.

Another condition we have to worry about is when a child terminates before its parent. If the child completely disappeared, the parent wouldn't be able to fetch its termination status when and if the parent were finally ready to check if the child had terminated. The kernel keeps a small amount of information for every terminating process, so that the information is available when the parent of the terminating process calls wait or waitpid. Minimally, this information consists of the process ID, the termination status of the process, and the amount of CPU time taken by the process.

The kernel can discard all the memory used by the process and close its open files. In UNIX System terminology, a process that has terminated, but whose parent has not yet waited for it, is called a *zombie*.

The ps command prints the state of a zombie process as Z. If we write a long-running program that forks many child processes, they become zombies unless we wait for them and fetch their termination status.

The final condition to consider is this: **what happens when a process that has been inherited by init terminates? Does it become a zombie?**
The answer is "no," because init is written so that whenever one of its children terminates, init calls one of the wait functions to fetch the termination status. By doing this, init prevents the system from being clogged by zombies.

8.6. 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

GANESH XEROX RNSIT

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. The default action for this signal is to be ignored.

A process that calls wait or waitpid can

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

If the process is calling wait because it received the SIGCHLD signal, we expect wait to return immediately. But if we call it at any random point in time, it can block.

The prototypes of wait and waitpid are as:

```
#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.

- The wait function can block the caller until a child process terminates, whereas waitpid has an option that prevents it from blocking.
- 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. We can always tell which child terminated, because the process ID is returned by the function.

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.

The integer status that these two functions return has been defined by the implementation, with certain bits indicating the exit status (for a normal return), other bits indicating the signal number (for an abnormal return), one bit to indicate whether a core file was generated, and so on.

POSIX.1 specifies that the termination status is to be looked at using various macros that are defined in <sys/wait.h>. Four mutually exclusive macros tell us how the process terminated, and they all begin with WIF as shown in **Table 8a.**

Example

1. The function pr_exit defined below uses the macros from table 8a to print a description of the termination status. Note that this function handles the WCOREDUMP macro, if it is defined.

```
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.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));
}
```

2. The program shown below calls the pr_exit function, demonstrating the various values for the termination status.

```
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

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

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

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

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

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

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

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

    exit(0);
}
```

```
$ ./a.out
normal termination, exit status = 7
abnormal termination, signal number = 6 (core file generated)
abnormal termination, signal number = 8 (core file generated)
```

3. **What if we want to wait for a specific process to terminate (assuming we know which process ID we want to wait for)?**
   - In older versions of the UNIX System, we would have to call wait and compare the returned process ID with the one we're interested in. If the terminated process wasn't the one, we wanted, we would have to save the process ID and termination status and call wait again.
   - We would need to continue doing this until the desired process terminated. The next time we wanted to wait for a specific process, we would go through the list of already terminated processes to see whether we had already waited for it, and if not, call wait again.

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.

The waitpid function returns the process ID of the child that terminated and stores the child's termination status in the memory location pointed to by statloc.

With wait, the only real error is if the calling process has no children.

With waitpid, however, it's also possible to get an error if the specified process or process group does not exist or is not a child of the calling process.

The options argument lets us further control the operation of waitpid. This argument is either 0 or is constructed from the bitwise OR of the constants in table 8b.

Table 8b: The options constants for waitpid

Constant	Description
WCONTINUED	If the implementation supports job control, the status of any child specified by pid that has been continued after being stopped, but whose status has not yet been reported, is returned (XSI extension to POSIX.1).
WNOHANG	The waitpid function will not block if a child specified by pid is not immediately available. In this case, the return value is 0.
WUNTRACED	If the implementation supports job control, the status of any child specified by pid that has stopped, and whose status has not been reported since it has stopped, is returned. The WIFSTOPPED 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.

a. 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 popen function.

b. 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.

c. The waitpid function provides support for job control with the WUNtrACED and WCONTINUED options.

Example

If we want to write a process so that it forks a child but we don't want to wait for the child to complete and we don't want the child to become a zombie until we terminate, the trick is to call fork twice.

```
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
  pid_t  pid;

  if ((pid = fork()) < 0) {
    perror("fork error");
  } else if (pid == 0) {    /* first child */
    if ((pid = fork()) < 0)
      perror("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 */
        perror("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);
}
```

We call sleep in the second child to ensure that the first child terminates before printing the parent process ID. After a fork, either the parent or the child can continue executing; we never know which will résume execution first. If we didn't put the second child to sleep, and if it resumed execution after the fork before its parent, the parent process ID that it printed would be that of its parent, not process ID 1.

Executing the program above gives us

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

8.7. waitid Function

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

The prototype of the function is as

```
#include <sys/wait.h>
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

Returns: 0 if OK, 1 on error

Like waitpid, waitid allows a process to specify which children to wait for. Instead of encoding this information in a single argument combined with the process ID or process group ID, two separate arguments are used. The id parameter is interpreted based on the value of idtype. The types supported are summarized in table 8c.

Table 8c. The idtype constants for waitid

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 shown in table 8d. These flags indicate which state changes the caller is interested in.

Table 8d. The options constants for waitid

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 wait, waitid, or waitpid.
WSTOPPED	Wait for a process that has stopped and whose status has not yet been reported.

The infop argument is a pointer to a siginfo structure. This structure contains detailed information about the signal generated that caused the state change in the child process.

8.8. wait3 and wait4 Functions

Most UNIX system implementations provide two additional functions: wait3 and wait4. The only feature provided by these two functions that is not 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 prototype 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, 0, or 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, and the like.

8.9. 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. The fork function is a lively breeding ground for race conditions, if any of the logic after the fork either explicitly or implicitly depends on whether the parent or child runs first after the fork.
In general, we cannot predict which process runs first. Even if we knew which process would run first, what happens after that process starts running depends on the system load and the kernel's scheduling algorithm.

We see a potential race condition in the program below when the second child printed its parent process ID.

```
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
  pid_t  pid;

  if ((pid = fork()) < 0) {
     perror("fork error");
  } else if (pid == 0) {    /* first child */
     if ((pid = fork()) < 0)
        perror("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 */
     perror("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);
}
```

If the second child runs before the first child, then its parent process will be the first child. But if the first child runs first and has enough time to exit, then the parent process of the second child is init. Even calling sleep, as we did, guarantees nothing. If the system was heavily loaded, the second child could resume after sleep returns, before the first child has a chance to run.

Problems of this form can be difficult to debug because they tend to work "most of the time." A process that wants to wait for a child to terminate must call one of the wait functions.

If a process wants to wait for its parent to terminate, as in the program above, a loop of the following form could be used:

```
while (getppid() != 1)
        sleep(1);
```

The problem with this type of loop, *called polling*, is that it wastes CPU time, as the caller is awakened every second to test the condition. To avoid race conditions and to avoid polling, some form of signaling is required between multiple processes. Signals can be used, and various forms of interprocess communication (IPC) can also be used.

For a parent and child relationship, we often have the following scenario.

After the fork, both the parent and the child have something to do. For example, the parent could update a record in a log file with the child's process ID, and the child might have to create a file for the parent.
In this example, we require that each process tell the other when it has finished its initial set of operations, and that each wait for the other to complete, before heading off on its own. The following code illustrates this scenario:

```
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

TELL_WAIT();    /* set things up for TELL_xxx & WAIT_xxx */

if ((pid = fork()) < 0) {
    perror("fork");
} else if (pid == 0) {      /* child */

    /* child does whatever is necessary ... */

    TELL_PARENT(getppid());    /* tell parent we're done */
    WAIT_PARENT();             /* and wait for parent */

    /* and the child continues on its way ... */

    exit(0);
}

/* parent does whatever is necessary ... */
TELL_CHILD(pid);           /* tell child we're done */
WAIT_CHILD();              /* and wait for child */

/* and the parent continues on its way ... */

exit(0);
```

The five routines TELL_WAIT, TELL_PARENT, TELL_CHILD, WAIT_PARENT, and WAIT_CHILD can be either macros or functions. The ways to implement these TELL and WAIT routines may be any forms of IPC or signals. Let's look at an example that uses these five routines.

Example

The program below outputs two strings: one from the child and one from the parent.

```
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static void charatatime(char *);

int main(void)
{
    pid_t   pid;
```


GANESH XEROX RNSIT ISE

```
    if ((pid = fork()) < 0) {
        perror("fork");
    } else if (pid == 0) {
        charatatime("output from child\n");
    } else {
        charatatime("output from parent\n");
    }
    exit(0);
}

static void charatatime(char *str)
{
    char    *ptr;
    int     c;

    setbuf(stdout, NULL);       /* set unbuffered */
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}
```

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.

```
$ ./a.out
ooutput from child
utput from parent
$ ./a.out
ooutput from child
utput from parent
$ ./a.out
output from child
output from parent
```

We need to change the program above to use the TELL and WAIT functions.

The program below does this. The lines preceded by a plus sign are new lines.

```
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static void charatatime(char *);

int main(void)
{
    pid_t   pid;

+   TELL_WAIT();

    if ((pid = fork()) < 0) {
        perror("fork error");
    } else if (pid == 0) {
+       WAIT_PARENT();      /* parent goes first */
        charatatime("output from child\n");
    } else {
```

```
        charatatime("output from parent\n");
+       TELL_CHILD(pid);
    }
    exit(0);
}
static void charatatime(char *str)
{
    char    *ptr;
    int     c;

    setbuf(stdout, NULL);          /* set unbuffered */
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}
```

In the program shown above, the parent goes first. The child goes first if we change the lines following the fork to be

```
    } else if (pid == 0) {
        charatatime("output from child\n");
        TELL_PARENT(getppid());
    } else {
        WAIT_CHILD();       /* child goes first */
        charatatime("output from parent\n");
    }
```

8.10. exec Functions

One use of the fork function is to create a new process (the child) that causes another program to be executed by calling one of the 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 six different exec functions

These six functions round out the UNIX System process control primitives. With fork, we can create new processes; and with the exec functions, we can initiate new programs.

The prototypes of the exec functions are as:

```
#include <unistd.h>
int execl(const char *pathname, const char *arg0,... /* (char *)0 */ );
int execv(const char *pathname, char *const argv []);
int execle(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

The first difference in these functions is that the first four take a pathname argument, whereas the last two take a filename argument.

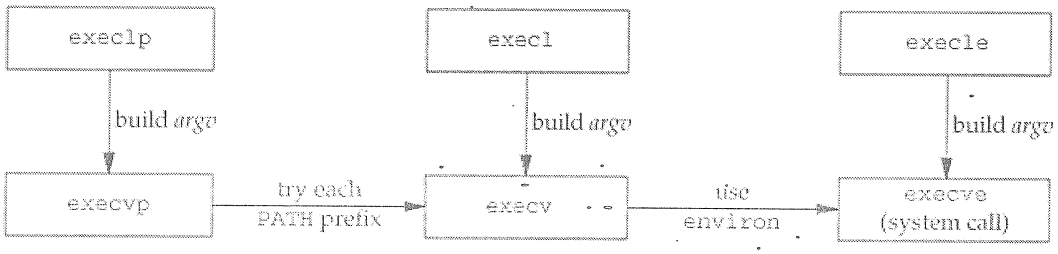
When a filename argument is specified

- 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.

The PATH variable contains a list of directories, called path prefixes that are separated by colons. For example, the **name=value** environment string

PATH=/bin:/usr/bin:/usr/local/bin/:.



specifies four directories to search. The last path prefix specifies the current directory.

The Second difference concerns the passing of the argument list (l stands for list and v stands for vector). The functions execl, execlp, and execle require each of the command-line arguments to the new program to be specified as separate arguments. We mark the end of the arguments with a null pointer. 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.

Before using ISO C prototypes, the normal way to show the command-line arguments for the three functions execl, execle, and execlp was

char *arg0, char *arg1, ..., char *argn, (char *)0

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.

In many UNIX system implementations, only one of these six functions, execve, is a system call within the kernel. The other five are just library functions that eventually invoke this system call. We can illustrate the relationship among these six functions as shown in figure below.

In this arrangement, the library functions execlp and execvp process the PATH environment variable, looking for the first path prefix that contains an executable file named filename.

Example

The program in below demonstrates the exec functions.

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

char  *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };
```

```
int main(void)
{
   pid_t   pid;

   if ((pid = fork()) < 0) {
      perror("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)
         perror("execle error");
   }

   if (waitpid(pid, NULL, 0) < 0)
      perror("wait error");

   if ((pid = fork()) < 0) {
      perror("fork error");
   } else if (pid == 0) {  /* specify filename, inherit environment */
      if (execlp("echoall", "echoall", "only 1 arg", (char *)0) < 0)
         perror("execlp error");
   }
   exit(0);
}
```

We first call execle, which requires a pathname and a specific environment. The next call is to execlp, which uses a filename and passes the caller's environment to the new program. The program echoall that is executed twice in the above program and is listed below. It is a trivial program that echoes all its command-line arguments and its entire environment list.

```
#include <stdio.h>

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

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

   for (ptr = environ; *ptr != 0; ptr++)   /* and all env strings */
      printf("%s\n", *ptr);

   exit(0);
}
```

When we execute the program, we get

```
$ ./a.out
  argv[0]: echoall
```


GANESH XEROX RNSIT

```
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
```

MODULE-4

Chapter 8 Process Control

Topics

Process Control Continuation:
Changing User IDs and Group IDs, Interpreter files, System function, Process Accounting, User Identification, Process Times, I/O Redirection.
IPC Methods:
Pipes, popen, pclose functions, Coprocesses, FIFOs, System V IPC, Message Queues, Semaphores.
Shared Memory:
Client-Server Properties, Stream Pipes, Passing File Descriptors, An Open Server-Version 1, Client-Server Functions.

8.11 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.

- 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..
- 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 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.
- 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 execstores the effective user ID from the file's user ID.

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 table summarizes 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.)

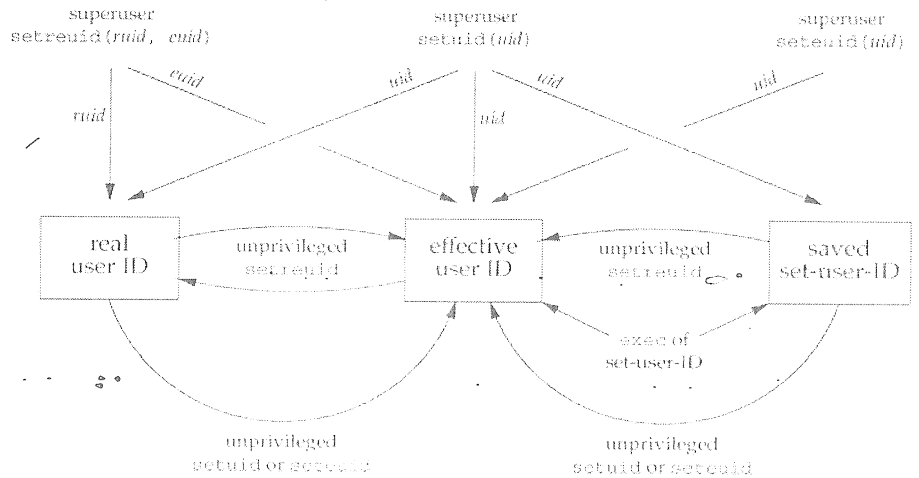


Figure: Summary of all the functions that set the various user Ids

8.12 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);
}
```

8.13 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 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);
}
```

8.14 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 */
};
```

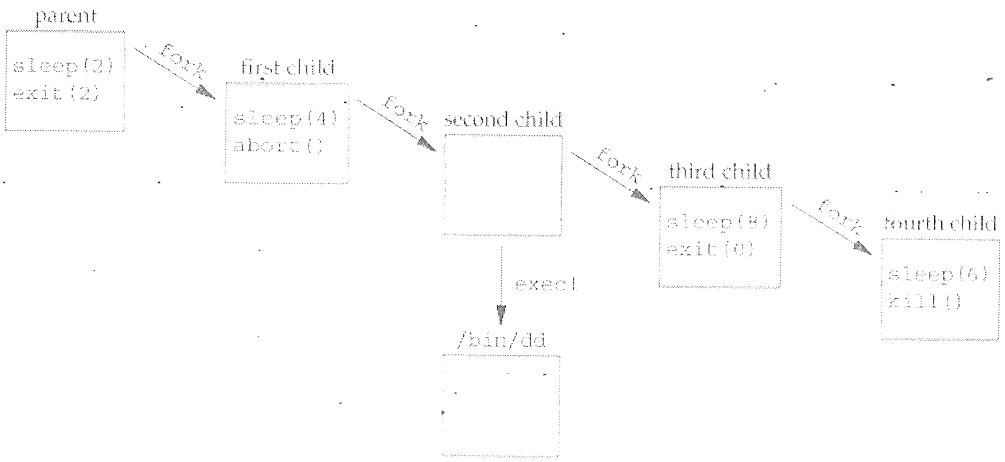
Values for ac_flag from accounting record

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

8.15 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 getloginfunction 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.

- *getlogin* can fail if the process is not attached to a terminal that a user logged in to.
- These types of processes are called *daemons*.
- Given login name, we can use the information to look up the user in the password file using *getpwnam*.
- Example – to determine the login shell.
- LOGNAME environment variable is initialized with user's login name by *login* and inherited by the login shell.
- But it can be modified by user, so *getlogin* should be used instead.

8.16 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 timesfunction 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 tmsstructure pointed to by buf:

```
struct tms {
clock_t tms_utime;    /* 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. *Example*: call *times* and save the return value. At a later time, call *times* again and subtract the earlier return value from the new value. Difference between the two times = wall clock time. Two structure fields for child processes (*tms_cutime*, *tms_cstime*) contain values only for child processes that were waited for using *wait*, *waitid*, or *waitpid*. All *clock_t* values returned by *times* function are converted to seconds using the number of clock ticks per second – the _SC_CLK_TCK value returned by *sysconf*.

8.17 I/O REDIRECTION

Process can use the C library function *reopen* to change its standard input and standard output ports to refer to text files instead of the console.

- Example to change process standard output to file *foo*:

```
FILE *fptr = freopen("foo", "w", stdout);
printf("Greeting message to foo\n");
```

- Example to change process standard input to file *foo*:

```
char buf[256];
FILE *fptr = freopen("foo", "r", stdin);
while(gets(buf))
        puts(buf);
```

- *freopen* function relies on *open* and *dup2* system calls to do redirection of standard input or standard output.
- To redirect standard input of a process from file *src_stream:*

```
#include<unistd.h>
int fd = open("src_stream", O_RDONLY);
if(fd != -1)
        dup2(fd, STDIN_FILENO), close(fd);
```

- *src_stream* file is now referenced by the STDIN_FILENO descriptor of the process.
- To redirect standard output of a process to file *dest_stream:*

```
#include<unistd.h>
int fd = open("dest_stream", O_WRONLY|O_CREAT|O_TRUNC, 0644);
if(fd != -1)
        dup2(fd, STDOUT_FILENO), close(fd);
```

- *dest_stream* file is now referenced by the STDOUT_FILENO descriptor of the process.

Implementation of *freopen* function

```
FILE *freopen(const char* filename, const char *mode, FILE *old_fstream){
    if(strcmp(mode,"r") && strcmp(mode,"w"))
        return NULL; //invalid mode
    int fd = open(file_name, *mode=="r" ? O_RDONLY:
                O_WRONLY|O_CREAT|O_TRUNC, 0644);
    if(fd == -1)
        return NULL;
    if(!old_stream)
        return fdopen(fd, mode);
    fflush(old_fstream);
```

GANESH XEROX RNSIT

Chapter 9 Overview of IPC Methods

9.1 INTRODUCTION

IPC enables one application to control another application, and for several applications to share the same data without interfering with one another. IPC is required in all multiprocessing systems, but it is not generally supported by single-process operating systems.

The various forms of IPC that are supported on a UNIX system are as follows:

1) Half duplex Pipes.
2) FIFO's
3) Full duplex Pipes.
4) Named full duplex Pipes.
5) Message queues.
6) Shared memory.
7) Semaphores.
8) Sockets.
9) STREAMS.

The first seven forms of IPC are usually restricted to IPC between processes on the same host. The final two i.e. Sockets and STREAMS are the only two that are generally supported for IPC between processes on different hosts.

PIPES

Pipes are the oldest form of UNIX System IPC. Pipes have two limitations.

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 pipefunction.

```
#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 below Figure. 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.

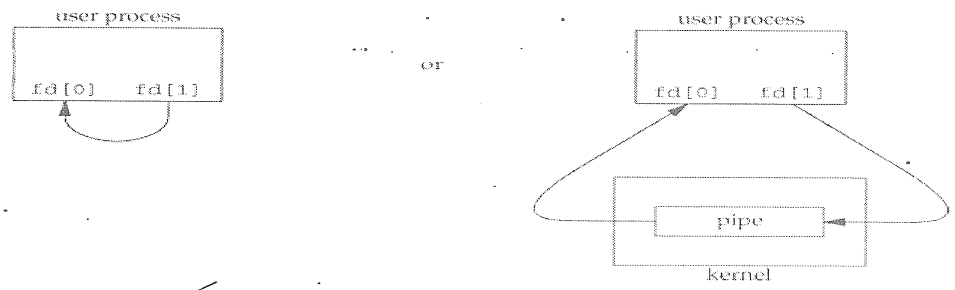


Figure: Two ways to view a half-duplex pipe

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. Below Figure shows this scenario.

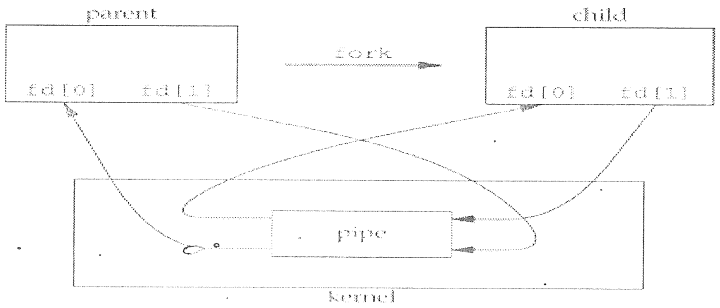


Figure: Half-duplex pipe after a fork

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 shows the resulting arrangement of descriptors.

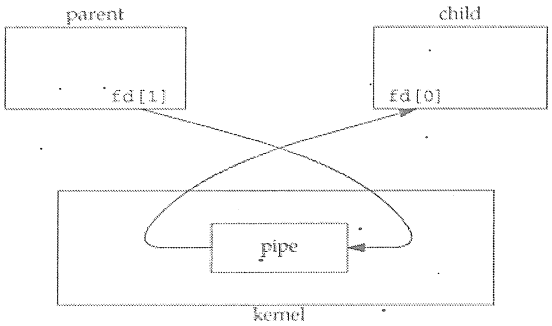


Figure: Pipe from parent to child

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 SIGPIPEis generated. If we either ignore the signal or catch it and return from the signal handler, writereturns 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);
}
```

9.2 popen AND pclose FUNCTIONS

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, NULLon error

```
int pclose(FILE *fp);
```

Returns: termination status of cmdstring, or 1 on error

The function popendoes a forkand execto 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.

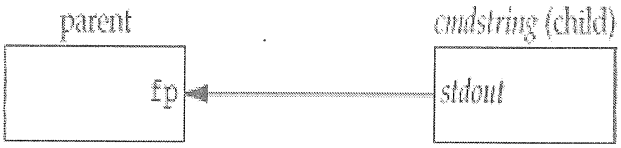


Figure: Result of fp = popen(cmdstring, "r")

If type is "w", the file pointer is connected to the standard input of cmdstring, as shown:

Figure: Result of fp = popen(cmdstring, "w")

9.3 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.

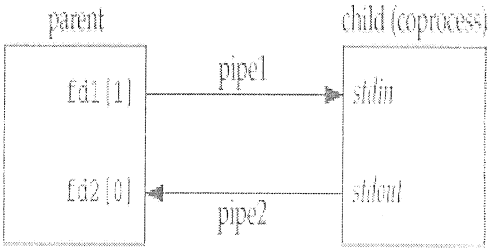


Figure: Driving a coprocess by writing its standard input and reading its standard output

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_FILEN
                    O, line, n) != n)
                    err_sys("write
                    error");
            } else {
                if (write(STDOUT_FILENO, "invalid
                    args\n", 13) != 13)
                    err_sys("write error");
            }
        }
        exit(0);
    }
```

9.4 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_NONBLOCKis specified, an openfor read-only returns immediately. But an openfor write-only returns 1 with errnoset to ENXIOif no process has the FIFO open for reading.

There are two uses for FIFOs.

- FIFOs are used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files.
- 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.

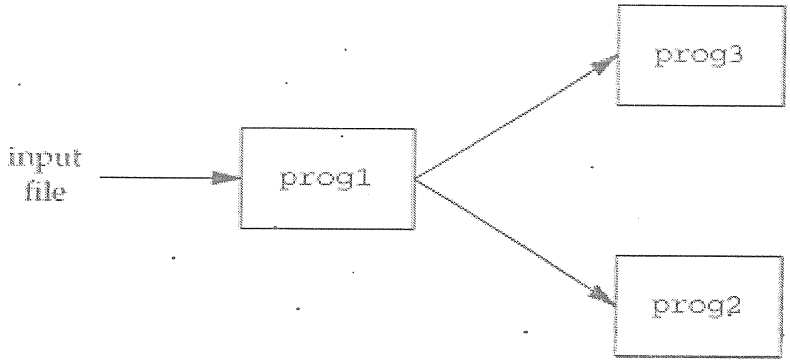


FIGURE: Procedure that processes a filtered input stream twice

With a FIFO and the UNIX program tee(1), we can accomplish this procedure without using a temporary file. (The teeprogram 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 teeto send its input to both the FIFO and prog2. Figure shows the process arrangement.

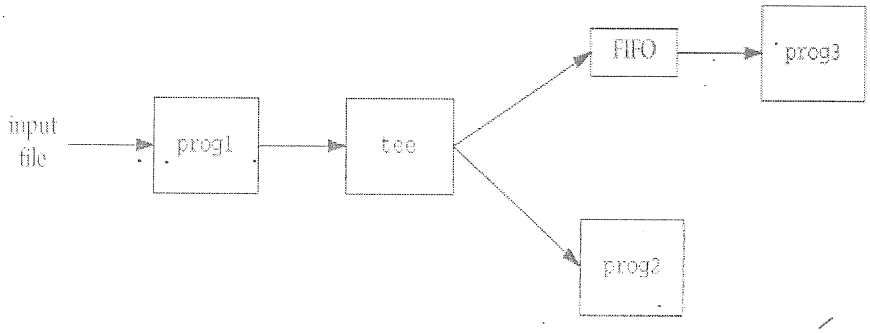


FIGURE: Using a FIFO and teeto send a stream to two different processes

Example Client-Server Communication Using a FIFO

- 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. 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.
- 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.
- 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 crash. This causes the client-specific FIFOs to be left in the file system.

- 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.

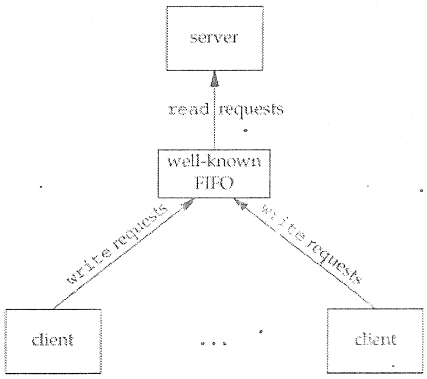


Figure: Clients sending requests to a server using a FIFO

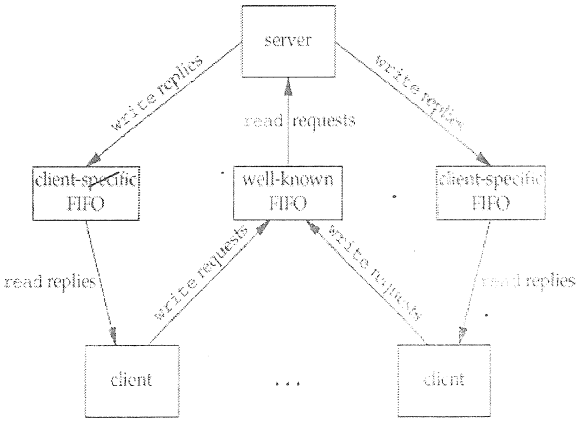


Figure: Client-server communication using FIFOs

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

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

- 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.
- 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.
- 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.

9.5 SYSTEM V IPC

Linux supports three types of interprocess communication mechanisms which first appeared in Unix System V (1983). These are message queues, semaphores and shared memory. These System V IPC mechanisms all share common authentication methods. Processes may access these resources only by passing a unique reference identifier to the kernel via system calls. Access to these System V IPC objects is checked using access permissions, much like accesses to files are checked. The access rights to the System V IPC object is set by the creator of the object via system calls. The object's reference identifier is used by each mechanism as an index into a table of resources. It is not a straight forward index but requires some manipulation to generate the index. All Linux data structures representing System V IPC objects in the system include an ipc_perm structure which contains the owner and creator processes user and group identifiers. the access mode for this object (owner, group and other) and the IPC object's key. The key is used as a way of locating the System V IPC object's reference identifier. Two sets of keys are supported: public and private. If the key is public then any process in the system, subject to rights checking, can find the reference identifier for the System V IPC object. System V IPC objects can never be referenced with a key, only by their reference identifier.

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 msgsndwhen 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 msggetto 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 msqid_dsstructure are initialized.

- ✓ The ipc_perm structure is initialized. The mode member of this structure is set to the corresponding permission bits of flag.
- ✓ msg_qnum, msg_lspid, msg_lrpid, msg_stime, and msg_rtimeare all set to 0.
- ✓ msg_ctimeis set to the current time.
- ✓ msg_qbytesis 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 msgctlfunction performs various operations on a queue.

```
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf );
```

Returns: 0 if OK, 1 on error.

The cmd argument specifies the command to be performed on the queue specified by msqid.

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);
```

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_dsstructure 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 semopatomically 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 semop array 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.

- 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.
- 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.
- 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.

Chapter 10 Shared Memory

Shared memory allows two or more processes to share a given region of memory.

- This is the fastest form of IPC, because the data does not need to be copied between the client and the server.
- The only trick in using shared memory is synchronizing access to a given region among multiple processes. If the server is placing data into a shared memory region, the client shouldn't try to access the data until the server is done. Often, semaphores are used to synchronize shared memory access.

The kernel maintains a structure with at least the following members for each shared memory segment:

```
struct shmid_ds {
struct ipc_perm shm_perm;        /* see Section 15.6.2 */
size_t shm_segsz;                /* size of segment in bytes */
pid_t shm_lpid;                  /* pid of last shmop() */
pid_t shm_cpid;                  /* pid of creator */
shmatt_t shm_nattch;             /* number of current attaches */
time_t shm_atime;                /* last-attach time */
time_t shm_dtime;                /* last-detach time */
time_t shm_ctime;                /* last-change time */
...
};
```

The type shmatt_t is defined to be an unsigned integer at least as large as an unsigned short.

The first function called is usually shmget, to obtain a shared memory identifier.

```
#include <sys/shm.h>
int shmget(key_t key, size_t size, int flag);
```

Returns: shared memory ID if OK, -1 on error

we described the rules for converting the *key* into an identifier and whether a new segment is created or an existing segment is referenced. When a new segment is created, the following members of the shmid_ds structure are initialized.

- The ipc_perm structure is initialized and described previously. The mode member of this structure is set to the corresponding permission bits of *flag*.
  - shm_lpid, shm_nattch, shm_atime, and shm_dtime are all set to 0.
  - shm_ctimeis set to the current time.
  - shm_segszis set to the *size* requested.

The *size* parameter is the size of the shared memory segment in bytes. Implementations will usually round up this size to a multiple of the system's page size, but if an application specifies *size* as a value other than an integral multiple of the system's page size, the remainder of the last page will be unavailable for use. If a new segment is being created (typically by the server), we must specify its *size*.

If we are referencing an existing segment (a client), we can specify *size* as 0. When a new segment is created, the contents of the segment are initialized with zeros. The shmctl function is the catchall for various shared memory operations.

```
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf)
```

Returns: 0 if OK, -1 on error

The *cmd* argument specifies one of the following five commands to be performed, on the segment specified by *shmid*.

IPC_STAT — Fetch the shmid_ds structure for this segment, storing it in the structure pointed to by *buf*.

IPC_SET — Set the following three fields from the structure pointed to by *buf* in the shmid_ds structure associated with this shared memory segment: shm_perm.uid, shm_perm.gid, and shm_perm.mode. This command can be executed only by a process whose effective user ID equals shm_perm.cuid or shm_perm.uid or by a process with superuser privileges.

IPC_RMID — Remove the shared memory segment set from the system. Since an attachment count is maintained for shared memory segments (the shm_nattch field in the shmid_ds structure), the segment is not removed until the last process using the segment terminates or detaches it.Regardless of whether the segment is still in use, the segment's identifier is immediately removed so that shmat can no longer attach the segment. This command can be executed only by a process whose effective user ID equals shm_perm.cuid or shm_perm.uid or by a process with superuser privileges.

Two additional commands are provided by Linux and Solaris, but are not part of the Single UNIX Specification.

- SHM_LOCK Lock the shared memory segment in memory. This command can be executed only by the superuser.
- SHM_UNLOCK Unlock the shared memory segment. This command can be executed only by the superuser.

Once a shared memory segment has been created, a process attaches it to its address space by calling shmat.

```
#include <sys/shm.h>
void *shmat(int shmid, const void *addr, int flag);
```

Returns: pointer to shared memory segment if OK, -1 on error

The address in the calling process at which the segment is attached depends on the *addr* argument and whether the SHM_RND bit is specified in *flag*.

- If *addr* is 0, the segment is attached at the first available address selected by the kernel. This is the recommended technique.
- If *addr* is nonzero and SHM_RND is not specified, the segment is attached at the address given by *addr*.
- If *addr* is nonzero and SHM_RND is specified, the segment is attached at the address given by (*addr* -(*addr* modulus SHMLBA)). The SHM_RND command stands for "round:" SHMLBA stands for "low boundary address multiple" and is always a power of 2. What the arithmetic does is round the address down to the next multiple of SHMLBA.

- If the SHM_RDONLY bit is specified in *flag*, the segment is attached as read-only. Otherwise, the segment is attached as read–write.
- The value returned by shmat is the address at which the segment is attached, or -1 if an error occurred.

GANESH XEROX RNSIT

- If shmat succeeds, the kernel will increment the shm_nattch counter in the shmid_ds structure associated with the shared memory segment.

When we're done with a shared memory segment, we call shmdt to detach it. Note that this does not remove the identifier and its associated data structure from the system. The identifier remains in existence until some process (often a server) specifically removes it by calling shmctl with a command of IPC_RMID.

```
#include <sys/shm.h>
int shmdt(const void *addr);
```

Returns: 0 if OK, -1 on error

The *addr* argument is the value that was returned by a previous call to shmat. If successful, shmdt will decrement the shm_nattch counter in the associated shmid_ds structure.

10.1 CLIENT SERVER PROPERTIES

Some of 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. 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. We 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, we 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 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 system wide 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.

10.2 STREAMS-BASED 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.

Below Figure shows the two ways to view a STREAMS pipe. The only difference between this picture and Figure of half duplex is that the arrows have heads on both ends; since the STREAMS pipe is full duplex, data can flow in both directions

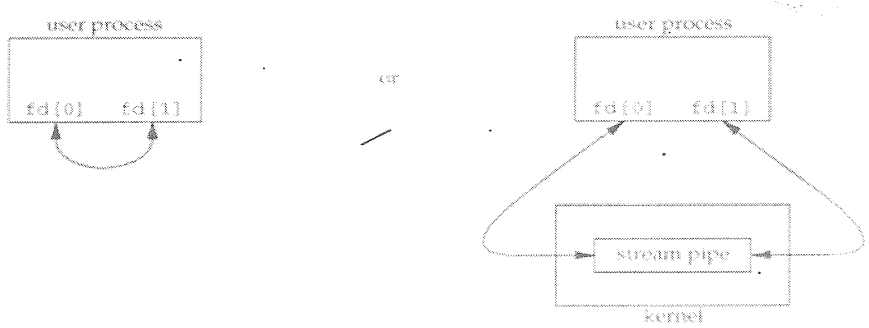


Figure: Two ways to view the stream pipes

If we look inside a STREAMS pipe (Below Figure), 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.

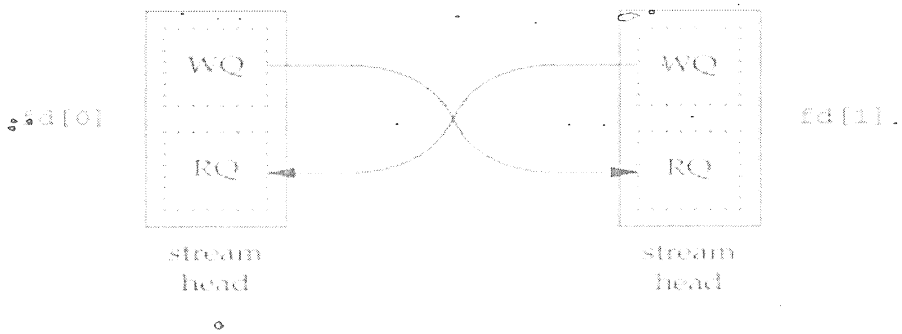


Figure: Inside a STREAMS pipe

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 (Below Figure). 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.

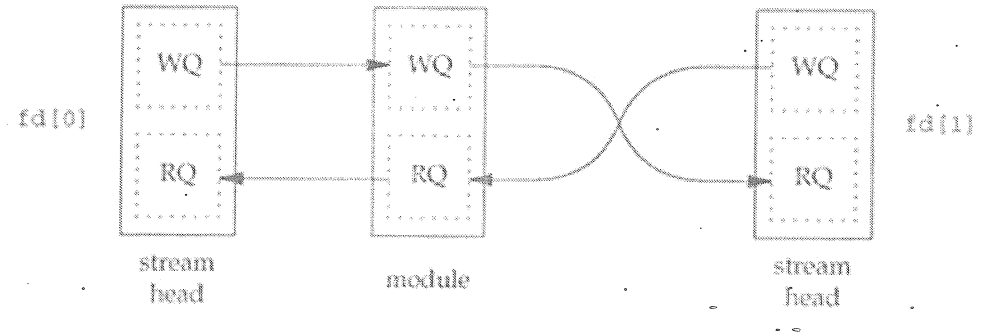


Figure: Inside a STREAMS pipe with a module

Assuming that we don't do anything fancy, such as pushing modules, a STREAMS pipe behaves just like a non-STREAMS pipe, except that it supports most of the STREAMS ioctl commands described in streamio.

Naming Streams Pipes

Normally, pipes can be used only between related processes: child processes inheriting pipes from their parent processes. In Section 15.5, we saw 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 bypass 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.

Below 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.

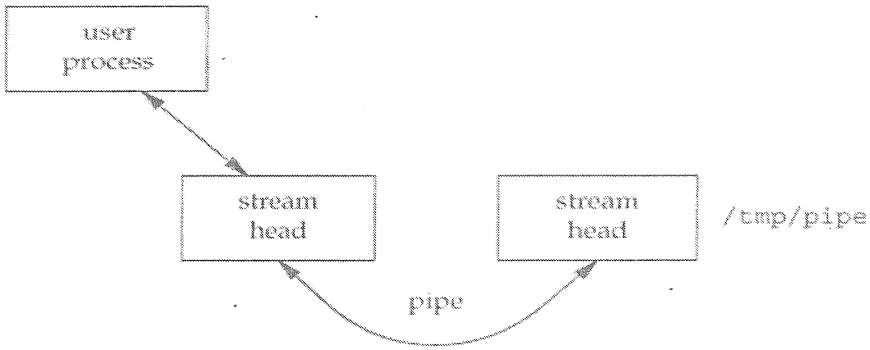


Figure: A pipe mounted on a name in the file system

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 below figure.

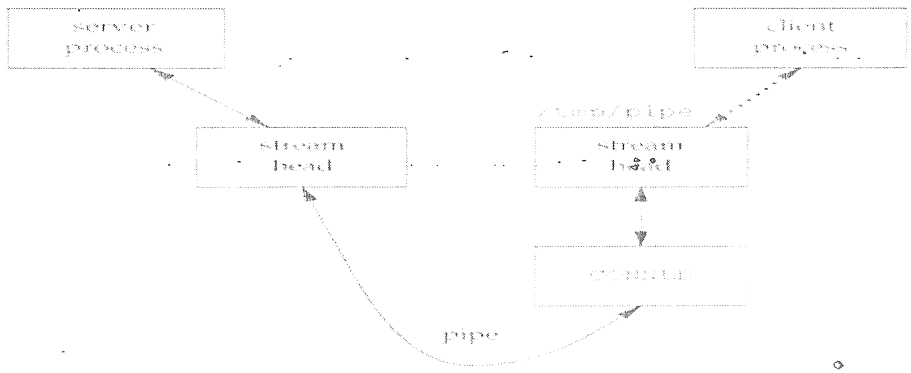


Figure: Setting up connld for unique connections

In above Figure, 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 below Figure.

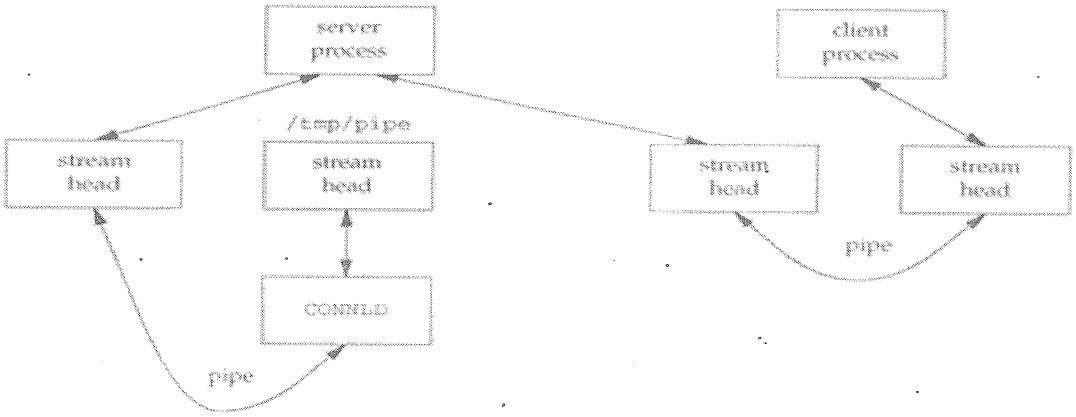


Figure: Using connld to make unique connections

The client process never receives an open file descriptor for the end of the pipe that it opened. Instead, the operating system creates a new pipe and returns one end to the client process as the result of opening /tmp/pipe. The system sends the other end of the new pipe to the server process by passing its file descriptor over the existing (attached) pipe, resulting in a unique connection between the client process and the server process. We'll see the mechanics of passing file descriptor using STREAMS pipes.

The fattach function is built on top of the mount system call. This facility is known as *mounted streams*. Mounted streams and the connld module were developed by Presotto and Ritchie [1990] for the Research UNIX system. These mechanisms were then picked up by SVR4.

We will now develop three functions that can be used to create unique connections between unrelated processes. These functions mimic the connection-oriented socket functions.

We use STREAMS pipes for the underlying communication mechanism here, but we'll see alternate implementations of these functions that use UNIX domain sockets.

```
#include "apue.h"
int serv_listen(const char *name);
```

Returns: file descriptor to listen on if OK, negative value on error

```
int serv_accept(int listenfd, uid_t *uidptr);
```

Returns: new file descriptor if OK, negative value on error

```
int cli_conn(const char *name);
```

Returns: file descriptor if OK, negative value on error

The serv_listen 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.

10.3 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.

We must be more specific about what we mean by "passing an open file descriptor" from one process to another. Although they share the same v-node, each process has its own file table entry. 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. Below figure shows the desired arrangement.

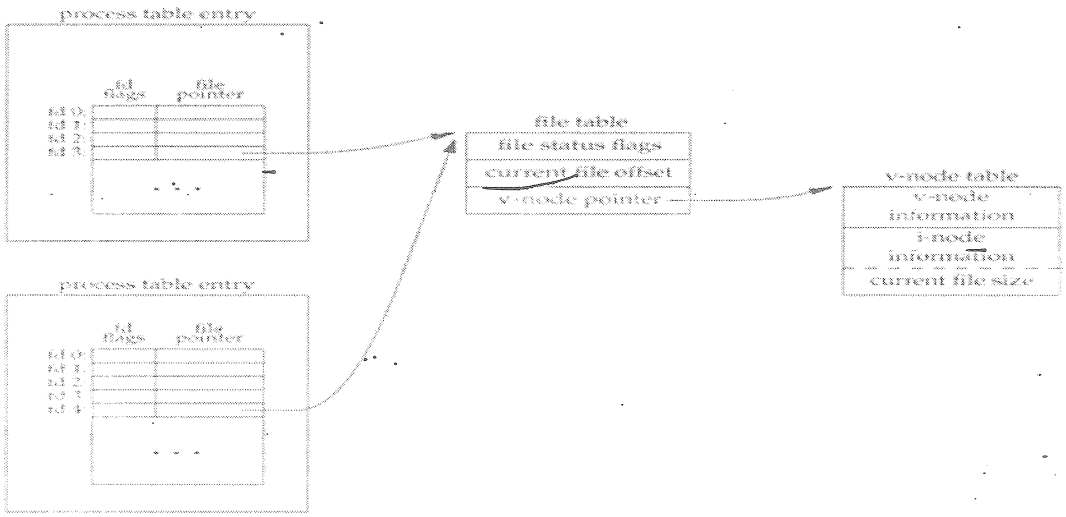


Figure: Passing an open file from the top process to the bottom process

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.) Having two processes share an open file table is exactly what happens after a fork.

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).

We define the following three functions that we use in this chapter to send and receive file descriptors. Later in this section, we'll show the code for these three functions for both STREAMS and sockets.

```
#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 recv_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 beimplemented 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 readup 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.

10.4 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. But instead of sending the contents of the file back to the calling process, the server sends back an open file descriptor. This lets the server work with any type of file (such as a device or a socket) and not simply regular files. It also means that a minimum of information is exchanged using IPC: the filename and open mode from the client to the server, and the returned descriptor from the server to the client. The contents of the file are not exchanged using IPC.

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). Shared libraries can simplify this updating.
- 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 s-pipe (either a STREAMS-based pipe or a UNIX domain socket pair) and then calls fork and exec to invoke the server. The client sends requests across the s-pipe, and the server sends back responses across the s-pipe.

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 s-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.

The server sends back an open descriptor or an error by calling either send_fd or send_err.

2. This is an example of a process sending an open descriptor to its parent. We'll modify this example to use a single daemon server, where the server sends a descriptor to a completely unrelated process.

We first have the header, open.h, which includes the standard headers and defines the function prototypes

10.5 CLIENT-SERVER CONNECTION FUNCTIONS

Functions used in client-server communication:

- socket()
- connect()
- bind()
- listen()
- accept()
- send() / write()
- recv() / read()
- close()

socket() - necessary to perform network communication. Mainly specifies protocol type and family used for communication.

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

int socket(int family, int type, int protocol);
```

Returns: socket descriptor if successful, -1 on error.

Parameters: family (protocol family), type (kind of socket), protocol (specific protocol type or 0 for system default)

- connect() - used by TCP client to establish connection with a TCP server.

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

int connect(int sockfd, struct sockaddr *serv_addr,
            int addrlen);
```

Returns: 0 if successful, -1 on error.
Parameters: sockfd (socket descriptor), serv_addr (pointer to socket structure that contains destination IP and port), addrlen (size of socket structure)

- bind() - assigns local protocol address to a socket.

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

int bind(int sockfd, struct sockaddr *my_addr,
         int addrlen);
```

Returns: 0 if successful, -1 on error.
Parameters: sockfd (socket descriptor), my_addr (pointer to socket structure that contains local IP and port), addrlen (size of socket structure)

- accept() - called by a TCP server to return the next completed connection from the front of the completed connection queue.

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

int accept(int sockfd, struct sockaddr *cli_addr,
           socklen_t *addrlen);
```

Returns: non-negative if successful, -1 on error.
Parameters: sockfd (socket descriptor), cli_addr (pointer to socket structure that contains client IP and port), addrlen (size of socket structure)

- send() / write() - used to send data over stream sockets or connected datagram sockets.

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

int send(int sockfd, const void *msg,
         int len, int flags);
```

Returns: number of bytes sent if successful, -1 on error.
Parameters: sockfd (socket descriptor), msg (pointer to data that must be sent), len (length of data to be sent), flags (set to 0)

- recv() / read() - used to receive data over stream sockets or connected datagram sockets.

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

int recv(int sockfd, void *buf,
                int len, unsigned int flags);
```

Returns: number of bytes read into the buffer if successful, -1 on error.
Parameters: sockfd (socket descriptor), buf (pointer to buffer that reads incoming information), len (maximum length of buffer), flags (set to 0)

- close() - used to close the communication between the client and the server

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

int close(int sockfd);
```

Returns: 0 if successful, -1 on error.
Parameters: sockfd (socket descriptor

GANESH XEROX RNSIT DEPT CS

MODULE-5

Chapter 11 Signals and Daemon Processes

Topics

Signals:
The UNIX kernel support for signals, signal, signal mask, sigaction, the SIGCHLD signal and the waitpid function, the sigsetjmp and siglongjmp functions, kill, alarm, interval timers, POSIX.1b Timers.
Daemon Processes:
Introduction, Daemon characteristics, coding rules, error logging, client-server model.

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.

Name	Description	Default action
SIGABRT	abnormal termination (abort)	terminate+core
SIGALRM	timer expired (alarm)	terminate
SIGBUS	hardware fault	terminate+core
SIGCANCEL	threads library internal use	ignore
SIGCHLD	change in status of child	ignore
SIGCONT	continue stopped process	continue/ignore
SIGEMT	hardware fault	terminate+core
SIGFPE	arithmetic exception	terminate+core
SIGFREEZE	checkpoint freeze	ignore
SIGHUP	hangup	terminate
SIGILL	illegal instruction	terminate+core
SIGINFO	status request from keyboard	ignore
SIGINT	terminal interrupt character	terminate
SIGIO	asynchronous I/O	terminate/ignore
SIGIOT	hardware fault	terminate+core
SIGKILL	termination	terminate
SIGLWP	threads library internal use	ignore
SIGPIPE	write to pipe with no readers	terminate
SIGPOLL	pollable event (poll)	terminate
SIGPROF	profiling time alarm (setitimer)	terminate
SIGPWR	power fail/restart	terminate/ignore
SIGQUIT	terminal quit character	terminate+core
SIGSEGV	invalid memory reference	terminate+core
SIGSTKFLT	coprocessor stack fault	terminate

SRI SEIVA XEROX
#147, NPS Tower, Opp. JSS College, Below UCO Bank, Dr. Vishnuvardhan Road, Kaveri Layout, Bangalore - 560 060. Ph: 9620063637, 7676670594.

SIGSTOP	stop	stop process
SIGSYS	invalid system call	terminate+core
SIGTERM	termination	terminate
SIGTHAW	checkpoint thaw	ignore
SIGTRAP	hardware fault	terminate+core
SIGTSTP	terminal stop character	stop process
SIGTTIN	background read from control tty	stop process

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:

- Accept the **default action** of the signal, which for most signals will terminate the process.
- **Ignore** the signal. The signal will be discarded and it has no affect whatsoever on the recipient process.
- 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.

11.1 THE UNIX KERNEL SUPPORT OF SIGNALS

- 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.

11.2 SIGNAL

The function prototype of the signal API is:

```
#include <signal.h>
void (*signal(int sig_no, void (*handler)(int)))(int);
```

The formal argument of the API are: sig_no is a signal identifier like SIGINT or SIGTERM. The handler argument is the function pointer of a user-defined signal handler function.

The following example attempts to catch the SIGTERM signal, ignores the SIGINT signal, 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>
/*signal handler function*/
void catch_sig(int sig_num)
{
        signal (sig_num,catch_sig);
        cout<<"catch_sig:"<<sig_num<<endl;
```


GANESH XEROX RNSIT ISE

```
}

/*main function*/
int main()
{
        signal(SIGTERM,catch_sig);
        signal(SIGINT,SIG_IGN);
        signal(SIGSEGV,SIG_DFL);
        pause( );                    /*wait for a signal interruption*/
}
```

The SIG_IGN specifies a signal is to be ignored, which means that if the signal is generated to the process, it will be discarded without any interruption of the process.

The SIG_DFL specifies to accept the default action of a signal.

11.3 SIGNAL MASK

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);
```

Returns: 0 if OK, 1 on error

The new_mask argument defines a set of signals to be set or reset in a calling process signal mask, and the cmd argument specifies how the new_mask value is to be used by the API. The possible values of cmd and the corresponding use of the new_mask value 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 from 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.
- If the actual argument to old_mask is a NULL pointer, no previous signal mask will be returned.
- The sigset_t contains a collection of bit flags.

  The BSD UNIX and POSIX.1 define a set of API known as sigsetops functions:

```
#include<signal.h>
int sigemptyset (sigset_t* sigmask);
int sigaddset (sigset_t* sigmask, const int sig_num);
int sigdelset (sigset_t* sigmask, const int sig_num);
int sigfillset (sigset_t* sigmask);
int sigismember (const sigset_t* sigmask, const int sig_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 the sigmask argument.
  - The sigdelset API clears the flag corresponding to the signal_num signal in the sigmask argument.

- The sigfillset API sets all the signal flags in the sigmask argument.
  [all the above functions return 0 if OK, -1 on error]
- The sigismember API returns 1 if flag is set, 0 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.

```
#include<stdio.h>
#include<signal.h> int main()
{
        sigset_t      sigmask;
        sigemptyset(&sigmask);            /*initialise set*/


        if(sigprocmask(0,0,&sigmask)==-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");
}
```

A process can query which signals are pending for it via the sigpending API:

```
#include<signal.h>
int sigpending(sigset_t* sigmask);
```

Returns 0 if OK, -1 if fails.

The sigpending API can be useful to find out whether one or more signals are pending for a process and to set up special signal handling methods for these signals before the process calls the sigprocmask API to unblock them.

The following example reports to the console whether the SIGTERM signal is pending for the process:

```
#include<iostream.h>
#include<stdio.h>
#include<signal.h>
int main()
{
        sigset_t      sigmask;
        sigemptyset(&sigmask);
        if(sigpending(&sigmask)==-1)
            perror("sigpending");
        else
            cout << "SIGTERM signal is:" << (sigismember(&sigmask,SIGTERM) ? "Set" :
        "No Set")  << endl;
}
```

In addition to the above, UNIX also supports following APIs 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);
```

11.4 SIGACTION

The sigaction API blocks the signal it is catching allowing a process to specify additional signals to be blocked when the API is handling a signal.

The sigaction API prototype is:

```
#include<signal.h>

int sigaction(int signal_num, struct sigaction* action, struct sigaction*
old_action);
```

Returns: 0 if OK, 1 on error

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 following program illustrates the uses 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();
    cout<<argv[0]<<"exists\n";
    return 0;
}
```

11.5 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 on how the parent sets up the handling of the SIGCHLD signal, different events may occur:

- Parent accepts the **default action** of the SIGCHLD signal:
  - SIGCHLD does not terminate the parent process.
  - Parent process will be awakened.
  - API will return the child's exit status and process ID to the parent.
  - Kernel will clear up the Process Table slot allocated for the child process.
  - Parent process can call the waitpid API repeatedly to wait for each child it

created.

- Parent **ignores** the SIGCHLD signal:
  - SIGCHLD signal will be discarded.
  - Parent will not be disturbed even if it is executing the waitpid system call.
  - If the parent calls the waitpid API, the API will suspend the parent until all its child processes have terminated.
  - Child process table slots will be cleared up by the kernel.
  - API will return a -1 value to the parent process.
- Process **catches** the SIGCHLD signal:
  - The signal handler function will be called in the parent process whenever a child process terminates.
  - If the SIGCHLD arrives while the parent process is executing the waitpid system call, the waitpid API may be restarted to collect the child exit status and clear its process table slots.
  - Depending on parent setup, the API may be aborted and child process table slot not freed.

11.6 THE sigsetjmp AND siglongjmp APIs

The function prototypes of the APIs are:

```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savemask);
int siglongjmp(sigjmp_buf env, int 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 only difference between these functions and the setjmp and longjmp functions is that sigsetjmp has an additional argument. If savemask is nonzero, then sigsetjmp also saves the current signal mask of the process in env. When siglongjmp is called, if the env argument was saved by a call to sigsetjmp with a nonzero savemask, then siglongjmp restores the saved signal mask. 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 the process signal mask is restored properly when "jumping out" from a signal handling function.

The following program illustrates the uses 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()
{
```

```
    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";
        return 0;
    }
    else
        cerr<<"return from first time sigsetjmp is called";
    pause();
}
```

~~11.7 KI~~LL

A process can send a signal to a related process via the kill API. This is a simple means of inter-process communication or control. The function prototype of the API is:

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

Returns: 0 on success, -1 on failure.

The signal_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 > 0	The signal is sent to the process whose process ID is pid.
pid == 0	The signal is sent to all processes whose process group ID equals the process group ID of the sender and for which the sender has permission to send the signal.
pid < 0	The signal is sent to all processes whose process group ID equals the absolute value of pid and for which the sender has permission to send the signal.
pid == 1	The signal is sent to all processes on the system for which the sender has permission to send the signal.

The following program illustrates the implementation of the UNIX kill command using the kill API:

```
#include<iostream.h>
#include<stdio.h>
#include<unistd.h>
#include<string.h>
#include<signal.h>
int main(int argc,char** argv)
{
        int pid, sig = SIGTERM;
        if(argc==3)
        {
                if(sscanf(argv[1],"%d",&sig)!=1)
                {
                        cerr<<"invalid number:" << argv[1] << endl; return -1;
                }
```

```
            argv++,argc--;
        }
        while(--argc>0)
        if(sscanf(*++argv, "%d", &pid)==1)
        {
            if(kill(pid,sig)==-1)
                perror("kill");
        }
        else
            cerr<<"invalid pid:" << argv[0] <<endl;
            return 0;
    }
```

The UNIX kill command invocation syntax is:

Kill [-<signal_num>] <pid>.....

Where signal_num can be an integer number or the symbolic name of a signal. <pid> is process ID.

11.8 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 function prototype of the API is:

```
#include<signal.h>
Unsigned int alarm(unsigned int time_interval);
```

Returns: 0 or number of seconds until previously set alarm 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(SIGALARM,&action,0)==-1)
        {
                perror("sigaction");
                return -1;
        }
        (void) alarm (timer);
        (void) pause( );
        return 0;
 }
```

11.9 INTERVAL TIMERS

The interval timer can be used to schedule a process to do some tasks at a fixed time interval, to time the execution of some operations, or to 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(SIGALARM,&action,0)==-1)
        {
              perror("sigaction"); return 1;
        }
        if(alarm(INTERVAL)==-1)
                perror("alarm"); else while(1)
        {
              /*do normal operation*/
        }
        return 0;
}
```

In addition to alarm API, UNIX also invented the setitimer API, which can be used to define up to three different types of timers in a process:

- Real time clock timer
- Timer based on the user time spent by a process
- Timer based on the total user and system times spent by a process

The getitimer API is also defined 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* arguments to the above APIs specify which timer to process. Its possible values and the corresponding timer types are:

ITIMER_REAL	decrements in real time and generates a SIGALRM signal when it expires
ITIMER_VIRTUAL	decrements in virtual time (time used by the process) and generates a SIGVTALRM signal when it expires.
ITIMER_PROF	decrements in virtual time and system time for the process and generates a SIGPROF signal when it expires.

The struct itimerval datatype is defined as:

```
struct itimerval
{
        struct timeval it_value;          /*current value*/
        struct timeval it_interval;       /* time interval*/
};
```

Example program:

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

```
#define INTERVAL  5
void callme(int sig_no)
{
        /*do scheduled tasks*/
}
int main()
{
        struct itimerval val;
        struct sigaction action;
        sigemptyset(&action.sa_mask);
        action.sa_handler=(void(*)( )) callme;
        action.sa_flags=SA_RESTART;
        if(sigaction(SIGALARM,&action,0)==-1)
        {
                perror("sigaction"); return 1;
        }
        val.it_interval.tv_sec      =INTERVAL;
        val.it_interval.tv_usec     =0;
        val.it_value.tv_sec         =INTERVAL;
        val.it_value.tv_usec        =0;

        if(setitimer(ITIMER_REAL, &val , 0)==-1)
                perror("alarm");

        else while(1)
        {
                /*do normal operation*/
        }
        return 0;
}
```

The setitimer and getitimer APIs return a zero value if they succeed or a -1 value if they fail.

11.10 POSIX.1b TIMERS

POSIX.1b defines a set of APIs for interval timer manipulations. The POSIX.1b timers are more flexible and powerful than are the UNIX timers in the following ways:

- Users may define multiple independent timers per system clock.
- The timer resolution is in nanoseconds.
- Users may specify the signal to be raised when a timer expires.
- The time interval may be specified as either an absolute or a relative time

The POSIX.1b APIs for timer manipulations 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);
```

Chapter 12 DAEMON PROCESSES

12.1 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.

12.2 DAEMON CHARACTERISTICS

The characteristics of daemons are:

- Daemons run in background.
- Daemons have super-user privilege.
- Daemons don't have controlling terminal.
- Daemons are session and group leaders.

12.3 CODING RULES

- **Call umask to set the file mode creation mask to 0.** The file mode creation mask that's inherited could be set to deny certain permissions. If the daemon process is going to create files, it may want to set specific permissions.
- **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're guaranteed that the child is not a process group leader.
- **Call setsid to create a new session.** The process (a) becomes a session leader of a new session, (b) becomes the process group leader of a new process group, and (c) has no controlling terminal.
- **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.
- **Unneeded file descriptors should be closed.** This prevents the daemon from holding open any descriptors that it may have inherited from its parent.
- **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.

Example Program:

```
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>
int daemon_initialise( )
{
pid_t pid;
```

```
if (( pid = for() ) < 0)
return -1;
else
if ( pid != 0)
exit(0); /* parent exits */
/* child continues */
setsid( );
chdir("/");
umask(0);
return 0;
}
```

12.4 ERROR LOGGING

One problem a daemon has is how to handle error messages. It can't simply write to standard error, since it shouldn't have a controlling terminal. We don't want all the daemons writing to the console device, since on many workstations, the console device runs a windowing system. A central daemon error-logging facility is required.

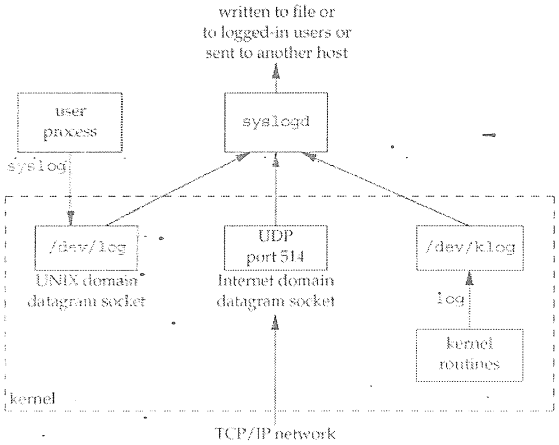


Fig 12.1 The BSD syslog facility

There are three ways to generate log messages:

- Kernel routines can call the logfunction. These messages can be read by any user process that opens and
  reads the /dev/klogdevice.
- Most user processes (daemons) call the syslog(3) function to generate log messages. This causes the message to be sent to the UNIX domain datagram socket /dev/log.
- 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);
```

11.5 CLIENT-SERVER MODEL

In general, a server is a process that waits for a client to contact it, requesting some type of service. In Figure 12.1, the service being provided by the syslogdserver is the logging of an error message.

In Figure 12.1, the communication between the client and the server is one-way. The client sends its service request to the server; the server sends nothing back to the client. In the upcoming chapters, we'll see numerous examples of two-way communication between a client and a server. The client sends a request to the server, and the server sends a reply back to the client.

SRI SHIVA XEROX
#147, RPS Tower, Opp. JSS College,
Below UCO Bank, Dr. Vishnuvardhan Road,
Relax Layout, Bangalore - 560 060.
Ph: 9620099567, 7676710591.