Module - I

INTRODUCTION

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.

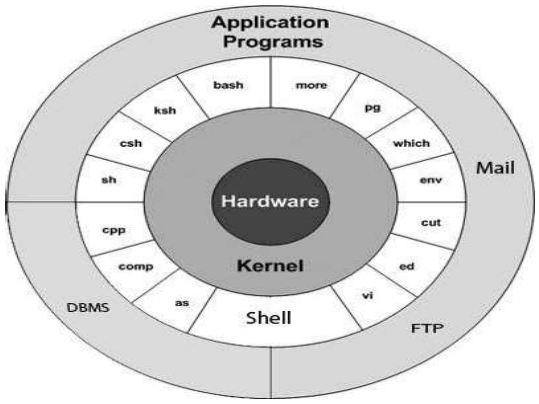
BRIEF HISTORY

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 oder 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 .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.

UNIX 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. 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.

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.

1.1 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 users 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.

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.
- **Multiuser:** The UNIX design allows multiple users to concurrently share hardware and software
- **Multitasking:** UNIX allows a user to run more than one program at a time. In fact more than one program can be running in the background while a user is working foreground.
- **Networking:** While UNIX was developed to be an interactive, multiuser, multitasking system, networking is also incorporated into the heart of the operating system. Access to another system uses a standard communications protocol known as Transmission Control Protocol/Internet Protocol (TCP/IP).
- Organized File System: UNIX has a very organized file and directory system that allows users to organize and maintain files.
- Device Independence: UNIX treats input/output devices like ordinary files. Input or output to a program can be from any device or file.The source or destination for file input and output is easily controlled through a UNIX design feature called redirection.
- Utilities: UNIX provides a rich library of utilities that can be use to increase user productivity.
- Services: UNIX also includes the support utilities for system administration and control.

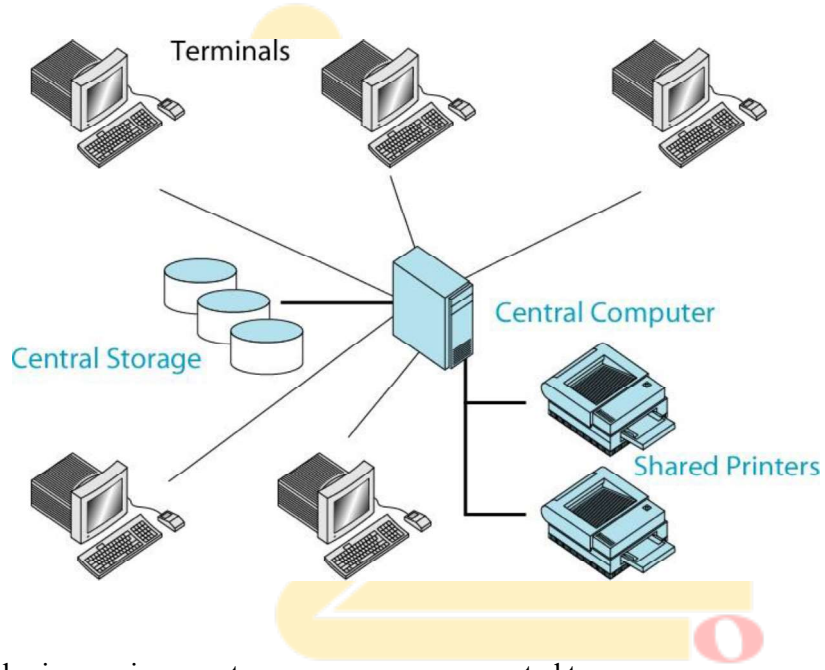
THE UNIX ENVIRONMENTS AND UNIX STRUCTURE

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 user user are installed UNIX on their personal computers this tends to personal unix system environment

Timesharing environment

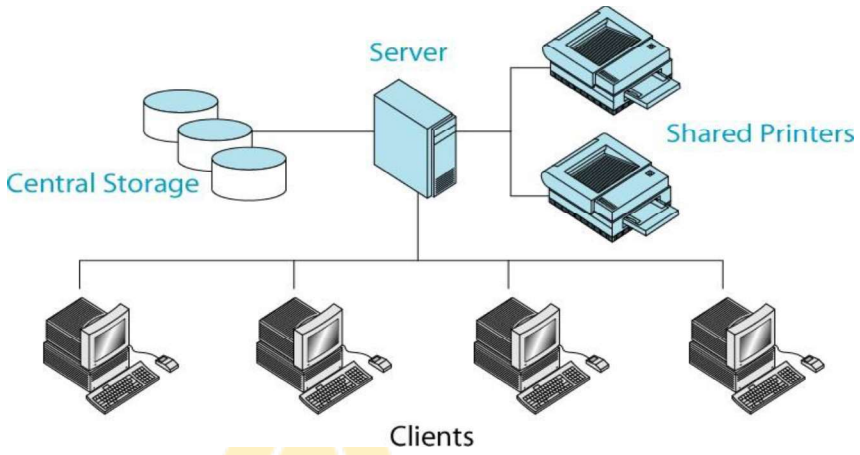


In a time-sharing environment, many users are connected to one or more computers. The output devices (such as printers) and auxiliary storage devices (such as disks) are shared by all users. In this environment all of the computing must be done by the central computer. The central computer has several 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

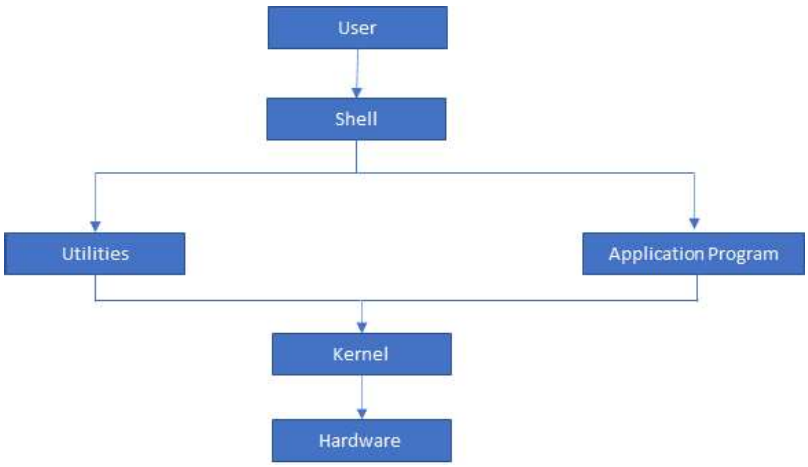
All of this work tends to keep the central computer busy so, user has to wait more time for get done their work so, it is nonproductive because of slow response.

The Client/Server Environment



- In client server environment splits the computing function between a central computer and users computers.
- User computer are personal computer or workstation central computer assigned to the workstations. In client server environment the users 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 users computer and the central computer, response time and monitor display are faster and users are more productive.

UNIX STRUCTURE



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: 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 your commands and works with the kernel to execute them.

b. Shell Programming: is a programming capability that allows you to write a shell scripts.

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 three standard shells used in UNIX today.

- Bourne shell: developed by steve bourne at AT&T labs, is the oldest.
- Bash(Bourne Again shell): An enhanced version of the Bash shell.
- C shell: developed in the Berkeley by Bill joy, Its commands look like C statements.
- Tcsh: A compatible version of C shell.
- Korn shell: developed by David Korn, also of AT&T Labs is the newest and powerful.

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.

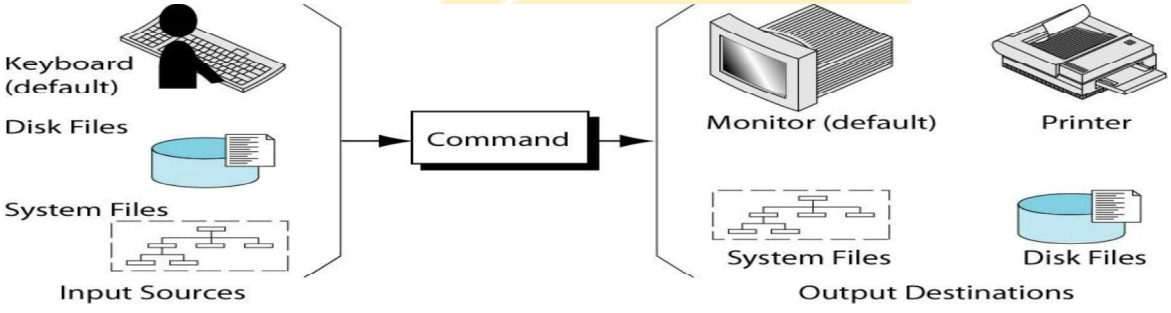
POSIX AND SINGLE UNIX SPECIFICATION

Dennis Ritchie's decision to rewrite UNIX in C didn't quite make UNIX very portable. UNIX fragmentation and absence of a single conforming standard adversely affected the development of portable applications. First ,AT &T created the System V Interface Definition(SVID). Later, X/Guide(XPG). Products conforming to this specification were branded UNIX95, UNIX98 or UNIX03 depending on the version of the specification.

Yet another group of standards, the portable operating system interface for computer environments(POSIX), were developed at the behest of the Institution of Electrical and Electronics Engineers(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 and POSIX.2. POSIX.1 specifies the C application program interface the system calls. POSIX.2 deals with the shell and utilities.

In 2001, a joint initiative of X/Open and IEEE resulted in the unification of the 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 easily ported to another POSIX- compliant UNIX machine with minimum modifications. We make reference to POSIX throughout this text, but these references should be Interpreted to mean the SUSV3 as well.

GENERAL FEATURES OF UNIX COMMANDS/ COMMAND STRUCTURE

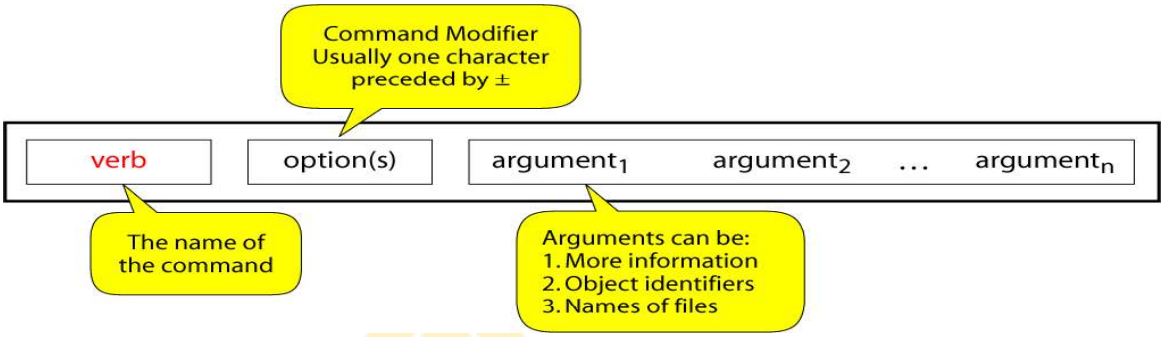


Commands are entered at shell prompt.The components of the command line are:

- 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.

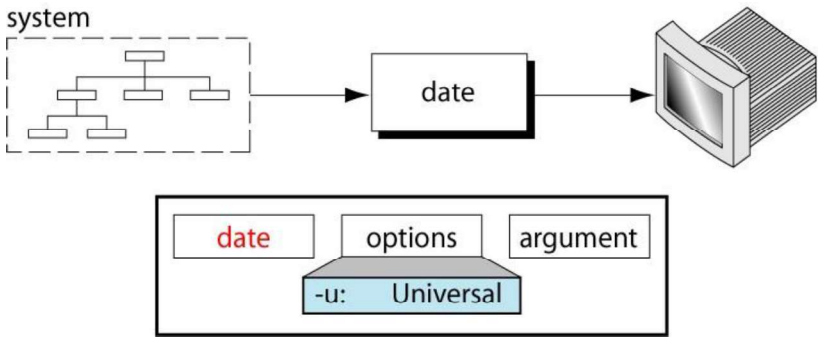
Note: Options MUST come after the command and before any command arguments. Options SHOULD NOT appear after the main argument(s). However, some options can have their own arguments

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

UNDERSTANDING OF SOME BASIC COMMANDS SUCH AS echo, printf, ls, who, date, passwd, cal.

THE DATE COMMAND:



date: displays the system date and time.If the system is local that is one in your own area-it is the current time.If the system is remote, such as across the country the reply will contain the time where the system is physically located.

The input for the date is the system itself.The date is actually maintained in the computer as a part of OS.The date command sends its response to monitor.

$date

```
Sun Aug 28 13:28:39 IST 2016
```

$date -u

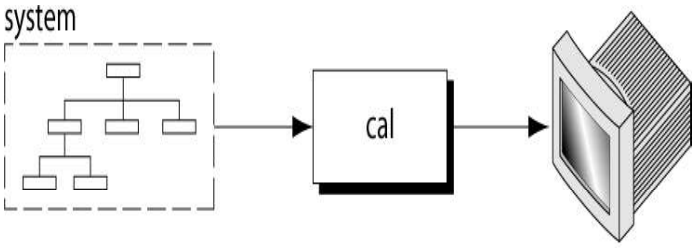
```
Sun Aug 28 08:02:13 UTC 2016
```

```
$ date "+Today's date is %D and Time is %T"
Today's date is 08/28/16 and Time is 13:33:57
```

Format code	Explanation
a	Abbreviated weekday name such as Mon
A	Full weekday name such as Monday
B	Full month name such as January
b	Abbreviated month name such as jan
d	Day of the month with two digits leading zeros (01,02…31)
e	Day of the month with spaces replacing zeros (1,2,….31)
D	Date in the format (mm/dd/yy)
H	Military time-two digit hour
I	Civilian time two digit hour
j	Julian date (day of the year) 001…366
M	Two digit minute such as 00,01,…59
m	Numeric two digit month 00,01,…12
p	Display am or pm
r	Time in format hour: minute: second with am/pm
R	Time in format hour: minute
S	Seconds as decimal number [00-61]
T	Time in format hour: min: second

U	Week number of the year
W	Week of year[00-53], with Monday being first day of the week.
Y	Year as ccyy(4 digits)
Z	Time zone name

cal: The CALENDAR COMMAND



A single parameter specifies the 4 digit year (1 - 9999) to be displayed.

Two parameters denote the Month (1 - 12) and Year (1 - 9999). If arguments are not specified, the current month is displayed. A year starts on 01 Jan.

- To display current month's calendar

$ cal

Output:

```
    April 2016
Su Mo Tu We Th Fr
             Sa 1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

- To display feb 2015 calendar

$cal 2 2015

Output :

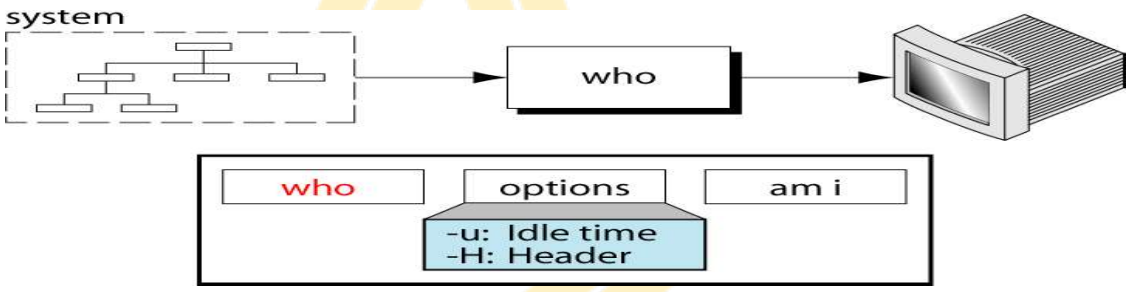
```
    February 2015
Su Mo Tu We
Th Fr Sa 1 2 3
4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
```

• To display complete year calendar.

$ cal -y

who: THE Who is online Command

The **who** command prints information about all users who are currently logged in.



who [OPTION]... [FILE] [am i]

-u –Idle time: Print the idle time for each user, and the process ID.

-H – HEADING : Print a line of column headings.

$who

Displays the username, terminal, and time and date of all currently logged-in sessions.

```
vizion   tty7         2016-08-28 13:28 (:0)
vizion   pts/0        2016-08-28 13:28 (:0.0)
```

$who am i

Displays the same information, but only for the terminal session where the command was issued, for example:

```
vizion   pts/0        2016-08-28 13:28 (:0.0)
```

$who –u

Indicates how long it has been since there was any activity on the line. This is known as

Idle time. It also returns the process id for the user.

```
vizion   tty7         2016-08-28 13:28   .          2420 (:0)
vizion   pts/0        2016-08-28 13:28   .          2623 (:0.0)
```

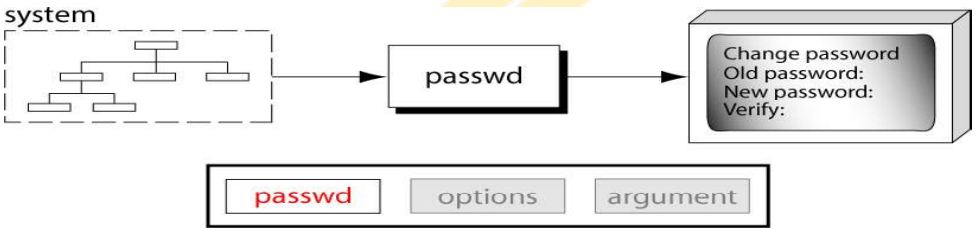
$who -uH

Displays "all" information, and headers above each column of data, for example:'

```
NAME       LINE         TIME             IDLE          PID COMMENT  EXIT
                        2016-08-28 13:26              1262 id=l5    term=0 exit=
0
LOGIN      tty1         2016-08-28 13:26              2303 id=1
LOGIN      tty2         2016-08-28 13:26              2304 id=2
LOGIN      tty3         2016-08-28 13:26              2305 id=3
LOGIN      tty4         2016-08-28 13:26              2306 id=4
LOGIN      tty5         2016-08-28 13:26              2307 id=5
LOGIN      tty6         2016-08-28 13:26              2308 id=6
                        2016-08-28 13:26              2309 id=x
vizion   + tty7         2016-08-28 13:28   .          2420 (:0)
vizion   + pts/0        2016-08-28 13:28   .          2623 (:0.0)
```

passwd command.: The change Password (PASSWD) COMMAND

The passwd command is used to change the password of a user account. A normal user can run passwd to change their own password, and a system administrator (the superuser) can use passwd to change another user's password, or define how that account's password can be used or changed.



passwd syntax

passwd [OPTION] [USER]

$passwd

Running passwd with no options will change the password of the account running the command. You will first be prompted to enter the account's current password:

- (current) UNIX password:

If it is correct, you will then be asked to enter a new password:

- Enter new UNIX password:

...and to enter the same password again, to verify it:

- Retype new UNIX password:

If the passwords match, the password will be changed.

The passwd command changes passwords for user accounts. A normal user can only change the password for their own account, but the superuser can change the password for any account. Passwd can also change or reset the account's validity period — how much time can pass before the password expires and must be changed.

Before a normal user can change their own password, they must first enter their current password for verification. (The superuser can bypass this step when changing another user's password.)

After the current password has been verified, passwd checks to see if the user is allowed to change their password at this time. If not, passwd refuses to continue, and exits.

Otherwise, the user is then prompted twice for a replacement password. Both entries must match for passwd to continue.

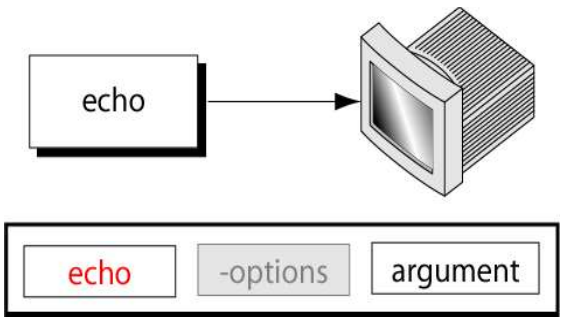
Rules for giving passwords

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

echo: Print message command

The **echo** command used to display a message.

- To display the diagnostic messages on the terminal or to issue prompts for taking user input
- To evaluate shell variable.



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

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:

\\	A 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.

Ex 1: **$echo Hello world**

Output :Hello world // Outputs the following text

Ex 2:

$x=10

$echo "The value of x is $x"

Entering these two commands will output the following text:

Output: **The value of x is 10.**

Ex 3:
$echo -e "Here**b**the**b**spaces**b**are**b**backspaced**"**

Outputs the following text:
Output :Herthspacearbackspaced //one character before backslash is deleted.

Ex 4:

$echo –e "Unix \n is \n a \n os"

Output :$Unix
is
a
Os //prints in a new line

Ex 5:
$echo –e "unix \t is \t a \t os"
Output :unix is a os //\t –prints after 8 spaces

Ex 6:
$echo –e "unix \r is a os"
Output : is a os //moving cursor to beginning of line.
//words before \r is deleted.

Ex 7:
$echo –e "unix \c is a os"
Output :unix // cursor in same line.
//words after \c deleted.

Ex 8:
$echo $SHELL
Output :/bin/bash //prints the shell name

5.2 **printf command**-alternative to echo command
$printf "No file entered"
Output :No file entered

$printf My current shell is %s\n" $SHELL
Output :My current shell is /bin/bash //Here %s act as a placeholder for the value $SHELL

%s-String
%d-Decimal integer
%o-octal integer
%f-Floating point number
%x-Hexadecimal integer

THE MAN COMMAND

The man command knowing more about Unix commands and using Unix online manual pages

man is the system's manual viewer; it can be used to display manual pages, scroll up and down, search for occurrences of specific text, and other useful functions.

Each argument given to **man** is normally the name of a program, utility or function. The manual page associated with each of these arguments is then found and displayed. A section number, if provided, will direct **man** to look only in that section of the manual. The default action is to search in all of the available sections, following a pre-defined order and to show only the first page found, even if page ex- ists in several sections.

A man page is divided into a number of compulsory optional sections. Every command doesn't have all sections, but the first three(NAME,SYNOPSIS and DESCRIPTION) are seen in all man pages.

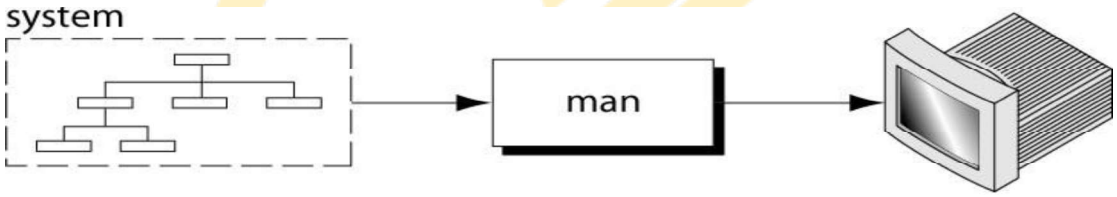
NAME presents the online introduction to the command

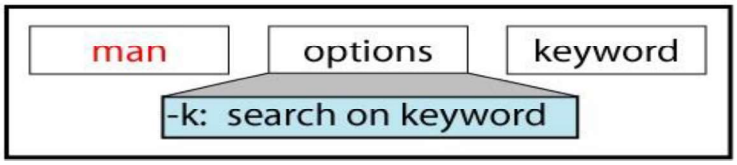
SYNOPSIS shows the syntax used by the command

DESCRIPTION provides a detailed information.

man syntax

$man [option] command name





Options

-K,**--** Search for text in all manual pages. This is a brute-force search, and is likely to take some time; if **global-** you can, you should specify a section to reduce the number of pages that need to be searched. **apropos**Search terms may be simple strings (the default), or regular expressions if the **--regex** option is used

Section Numbers

The section numbers of the manual are listed below. While reading documentation, if you see a com- mand name followed by a number in parentheses, the number refers to one of these sections. For exam- ple, man is the documentation of man found in section number 1. Some commands may have docu- mentation in more than one section, so the numbers after the command name may direct you to the cor- rect section to find a specific type of information. The section numbers, and the topics they cover, are as follows:

Section number	Description
1	Executable programs or shell commands
2	System calls (functions provided by the kernel)
3	Library calls (functions within program libraries)
4	Special files (usually found in /dev)
5	File formats and conventions eg /etc/passwd
6	Games
7	Miscellaneous (including macro packages and conventions), e.g. man, groff
8	System administration commands (usually only for root)
9	Kernel routines [Non standard]

Man Examples

$man man

View the manual page for the **man** command.

$man –s4 passwd

This displays the documentation for a configuration file from the section 4. Even this information is present in the section 1 it won't display section 1 information.

whatis command: displays short manual page descriptions.

Each manual page has a short description available within it. **whatis** searches the manual page names and displays the manual page descriptions of any name matched.

$whatis who

Display a description of what **who** is.

```
who                  (1p)  - display who is on the system
who                  (1)   - show who is logged on
```

$whatis cal

```
cal                  (1)   - displays a calendar
cal                  (1p)  - print a calendar
```

apropos command:

searches the manual pages for a keyword or regular expression. Each manual page has a short descrip- tion included with it. apropos searches these descriptions for instances of keyword.

$apropos find

aa_find_mountpoint (2) - find where the apparmor interface filesystem is mounted

chkdupexe (1) - find duplicate executables

ffs (3) - find first bit set in a word

ls COMMAND

The ls command lists all files in the directory that match the name. If name is left blank, it will list all of the files in the directory.

Syntax

The syntax for the ls command is:

```
ls  [options]   [names]
```

Option	Description
-a	Displays all files.
-b	Displays nonprinting characters in octal.
-c	Displays files by file timestamp.
-C	Displays files in a columnar format (default)
-d	Displays only directories.
-f	Interprets each *name* as a directory, not a file.
-F	Flags filenames.
-g	Displays the long format listing, but exclude the owner name.
-i	Displays the inode for each file.
-l	Displays the long format listing.
-L	Displays the file or directory referenced by a symbolic link.
-m	Displays the names as a comma-separated list.
-n	Displays the long format listing, with GID and UID numbers.
-o	Displays the long format listing, but excludes group name.
-p	Displays directories with /
-q	Displays all nonprinting characters as **?**
-r	Displays files in reverse order.
-R	Displays subdirectories as well.
-t	Displays newest files first. (based on timestamp)
-u	Displays files by the file access time.
-x	Displays files as rows across the screen.
-1	Displays each entry on a line.

To show long listing information about the file/directory.

$ls -l

```
total 1340
-rwxrwxr-x 1 vizion vizion    6961 2015-09-17 16:17 a.out
-rw-rw-r-- 1 vizion vizion       0 2016-08-08 16:03 cal
drwxr-xr-x 2 vizion vizion    4096 2016-08-28 13:29 Desktop
drwxr-xr-x 2 vizion vizion    4096 2008-03-06 13:08 Documents
drwxr-xr-x 2 vizion vizion    4096 2008-03-06 13:08 Download
-rw-rw-r-- 1 vizion vizion    1129 2016-08-11 00:49 ecos.c
drwxrwxr-x 2 vizion vizion    4096 2016-08-24 05:53 fedora
-rw-rw-r-- 1 vizion vizion      29 2016-08-10 13:22 ff.c
```

a. Field 1:

- 1st Character – File Type: First character specifies the type of the file. In the example above the hyphen (-) in the 1st character indicates that this is a normal file. Following are the possible file type options in the 1st character of the ls -l output.
  - Field Explanation
  - – normal file
  - d directory
  - s socket file
  - l link file
- **2nd to 9th character -- File Permissions:** Next 9 character specifies the files permission. Each 3 characters refers to the read, write, execute permissions for owner, group and other.

b. Field 2 – Number of links: Second field specifies the number of links for that file. In this example, 1 indicates only one link to this file ecos.c and 2 links to directory named fedora

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

d. Field 4 – Group: Fourth field specifies the group of the file. In this example, the file ecos.c belongs to "vizion' group.

e. Field 5 – Size: Fifth field specifies the size of file. In this example, '1129' indicates the ecos.c file size.

f. Field 6 – Last modified date & time: Sixth field specifies the date and time of the last modification of the file.

g. Field 7 – File name: The last field is the name of the file.

h. Display Directory Information Using ls -ld

When you use "ls -l" you will get the details of directories content. But if you want the details of direc- tory then you can use -d option as., For example, if you use ls -l /etc will display all the files under etc directory. But, if you want to display the information about the /etc/ directory, use -ld option as shown below.

```
$ ls -l /etc
total 3344
-rw-r--r-- 1 root root 15276 Oct 5 2004 a2ps.cfg
-rw-r--r--  1 root root  2562 Oct 5 2004 a2ps-site.cfg
drwxr-xr-x  4 root root  4096 Feb  2  2007 acpi
-rw-r--r--  1 root root    48 Feb  8  2008 adjtime
drwxr-xr-x  4 root root  4096 Feb 2 2007 alchemist
$ ls -ld /etc
drwxr-xr-x 21 root root 4096 Jun 15 07:02 /etc
```

FLEXIBILITY OF COMMAND USAGE 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

tThe 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
```

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 exe- cuted.

- 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.

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'

$type 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!

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

BECOMING THE SUPER USER: SU COMMAND

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

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

Though the current directory does not change the # prompt indicates the GMIT 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 pro- gram 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 – GMIT

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

UNIX FILES

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

NAMING FILES OR WHAT'S IN A (FILE) NAME

a. UNIX permits file names to use most characters, but avoid spaces, tabs and characters that have a special meaning to the shell, such as:

& ; ( ) | ? \ ' " ` [ ] { } < > $ - ! /

b. It is recommended that only the following characters be used in filenames.

c. Alphabetic characters and numerals

d. The period(.), the hyphen(-) and underscore(_)

e. Case Sensitivity: uppercase and lowercase are not the same! These are three different files:

NOVEMBER November november

f. Length: can be up to 256 characters

g. Extensions: may be used to identify types of files

libc.a - *archive, library file*

program.c - *C language source file*
alpha2.f - *Fortran source file*
xwd2ps.o - *Object/executable code*
mygames.Z - *Compressed file*

BASIC FILE TYPES/CATEGORIES FILES

File is a collection of records. So, files are divided into three categories

a. Ordinary file

b. Directory file

c. Device file

The UNIX file system contains several different types of files:

a.Ordinary Files or regular files

It contains only data as a stream of characters.

An ordinary files itself divided into 2 types

Text file: contains only printable characters, and you can often view the contents and make sense out of them. All C and Java files are example of text file. A text file contains lines of characters where every line is terminated with the newline character, also known as **linefeed** (LF) when you press Enter while inserting text, the LF character is appended to every line. You won't see this character normally, but there is command (od) which can make it visible.

Binary file: it contains both printable and unprintable characters that cover the entire ASCII range(0 to 255).most UNIX commands are example of binary files.

b. Directory files

i. Contains no data, but keeps some details of the files and subdirectories that it contains.

ii. A directory file contains an entry for every file and sub directory that it houses. Each entry has two components

- The filename
- A unique Identification number for the file or directory(called the inode number)

iii. A directory contains the filename but not the contents of file.

iv. When you create or remove a file the kernel automatically updates its corresponding directory by adding or removing the enter i.e inode number associated with that file.

c. Device files

i. Used to represent a real physical device such as a printer, tape drive or terminal, used for Input/Ouput (I/O) operations

ii. Unix considers any device attached to the system to be a file - including your terminal:

iii. By default, a command treats your terminal as the standard input file (stdin) from which to read its input

iv. Your terminal is also treated as the standard output file (stdout) to which a command's output is sent.

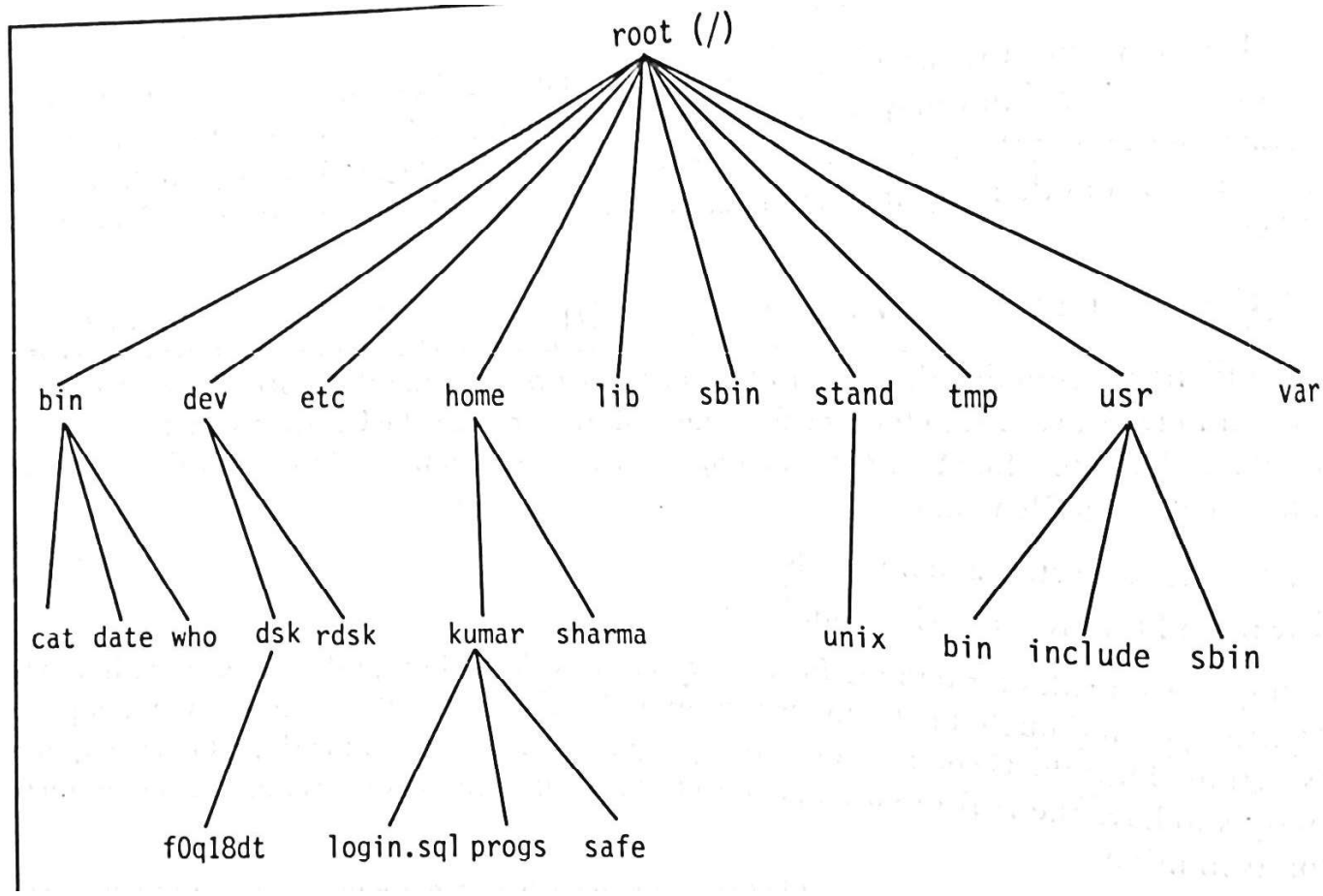
v. stdin and stdout will be discussed in more detail later

vi. Two types of I/O: character and block

vii. Usually only found under directories named /dev

PARENT CHILD RELATIONSHIP/ ORGANIZATION OF FILES

All files in UNIX are related to one another. The file system in unix is a collection of all ordinary, directory and device files and organized in a hierarchical structure as shown in below fig.



The implicit feature of every UNIX file system is that there is a top which serves as reference point for all files.This top is called **root** & is represented by a /(front slash). Root is actually a directory. The root

directory has a number of sub directories under it. These sub directories in turn have more sub directories and others files under them.

For instance bin and usr are two directories directly under root, while a second bin and kumar are sub directories under usr.

Every file apart from root must have a parent. Thus the home directory is the parent of kumar , while / is the parent of home and grandparent of kumar. If you create a file login.sql under the kumar directory ,then kumar will be the parent of this file.

The first group contains the files that are made available during system installation

- **/bin and /usr/bin:** these are the directories where all the commonly used UNIX commands are found.
- **/sbin and /usr/sbin:** If there's a command that you can't execute but the system administrator can execute, it would be probably in one of these directories.
- **/etc:** this directory contains the configuration files of the system. You can change a very important aspect of system functioning by editing a text file in this directory. Your login name and password are stored in files /etc/passwd and etc/shadow
- **/dev:** This directory contains all device files. These files don't occupy space on disk.there could be more sub directories like pts, dsk and rdsk in this directory
- **/lib and /usr/lib:** Contains shared library files and sometimes other kernel-related files.
- **/usr and /include:** contains the standard header files used by C programs. The statement #include<stdio.h> used in most C programs referes to the file stdio.h in this directory.
- **/usr/share/man:** this is where the man pages are stored. There are separate subdirectories here(like man1,man2 etc) that contains the pages for each section. For instance, the man page of **ls** can be found in /usr/share/man/man1

User also work with their own files, they write programs, send and receive mail and also create temporary files. These files are available in the second group shown below

- **/tmp:** the directory where users are allowed to create temporary files. These files are wiped away regularly by the system
- **/var:** The variable part of the file system. Contains all your print jobs and your outgoing and incoming mail.
- **/home:**On many systems users are housed here.Kumar would have his home directory in /home/kumar

THE home directory and the HOME VARIABLE

HOME DIRECTORY : When log on to the system, UNIX automatically places you 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 sharma , you will land up in a directory that could have the pathname

  /home/sharma
- The shell variable HOME known's yours home directory

  $echo $HOME

  /home/sharma

  You will be doing much of your work in your home directory and subdirectories.
- Home variable: it is also called environment variables. **Environment variables** are a set of dynamic named values that can affect the way running processes will behave on a computer.
- Here **$HOME** is a environment variable it indicates the home directory of the current user: the default argument for the cd built-in command.

PATH VARIABLE:

- The PATH environment variable is a colon-delimited list of directories that your shell searches through when you enter a command.
- Program files (executables) are kept in many different places on the Unix system. Yourpath tells the Unix shell where to look on the system when you request a particular program.
- To find out what your path is, at the Unix shell prompt echo $PATH
- Your path will look something like the following.

```
/usr2/username/bin:/usr/local/bin:/usr/bin:.
```

You will see your username in place of username. Using the above example path, if you enter the ls command, your 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).

RELATIVE AND ABSOLUTE PATHNAMES

THE dot(.) and double dots(..) notation to represent present and parent directories and their usage in relative pathnames

RELATIVE PATHNAMES

- Pathnames that don't begin with / specify locations relative to your current working directory.
- Uses either the current or parent directory as reference and specifies path relative to it.
- A relative pathname uses one of these cryptic symbols.
  - **.** (a single dot)-→ this represents the current directory.
  - **..** (two dots)→this represents the parent directory

Command	Function
cd	Returns you to your login directory
cd ~	Also returns you to your login directory
cd /	Takes you to the entire system's root directory
cd /root	Takes you to the home directory of the root or superuser,account created at installation, you must be root user to access this directory.
cd /home	Takes you to the home directory where user login directories are usually stored
cd ..	Moves you up one directory
cd ~otheruser	Takes you to the otheruser's login directory
cd /dir/subdirfoo	Regardless of which directory you are in, the absolute path takes you directly to subdirfoo, a subdirectory of dir.

Ex .1: Assume the current directory is /home/kumar/progs/data/text, using cd .. will move one level up

$pwd
/home/kumar/progs/data/text
$ cd ..
$pwd
/home/kumar/progs/data

Ex 2 : To move two levels up

$pwd
/home/kumar/progs
$ cd ../..
$pwd
/home

Ex 3: My present location is /etc/samba and now I want to change directory to /etc.
Using relative path: $ **cd ..**
Using absolute path: $**cd /etc**

Ex 4: My present location is /var/ftp/ and I want to change the location to /var/log
Using relative path: **cd ../log**
Using absolute path: **cd /var/log**

Ex 5: My present location is /etc/lvm and I want to change my location to /opt/oradba
Using relative path: **cd ../../opt/oradba**
Using absolute path: **cd /opt/oradba**

ABSOLUTE PATHNAMES:

- 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/kumar

- When you have more than one / in a pathname for such / you have to descend one level in the file system. Thus Kumar is one level below home and two levels below root.
- When you specify a file y using frontslashes to demarcate the various levels,you have a mechanism of identifying a file uniquely.No two files in a UNIX system can have same absolute pathnames.
- When you 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 you 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

DIRECTORY COMMANDS – PWD, CD, MKDIR, RMDIR COMMANDS

pwd (PRINT WORKING DIRECTORY) (checking your current directory)

As the name states, command '**pwd**' prints the current working directory or simply the directory user is, at present. It prints the current directory name with the complete path starting from root (/). This command is built in shell command and is available on most of the shell – bash, Bourne shell, ksh,zsh, etc.

Basic syntax
$pwd [option]

Options	Description
-L (logical)	Use PWD from environment, even if it contains symbolic links
-P (physical)	Avoid all symbolic links
–help	Display this help and exit
–version	Output version information and exit

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.

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 assume gmit is a directory under user directory Kumar.To change from Kumar directory to gmit directory , issue the command as follows

$pwd
/home/kumar
$cd gmit
$pwd
/home/kumar/gmit

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

$pwd
/home/kumar/gmit
$cd
cd without argument will change directory from gmit to its home directory Kumar
$pwd
/home/kumar

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

$pwd
/home/kumar
$cd /bin
$pwd
/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 gmit, issue the following command**.**

$mkdir gmit

gmit directory is created under present working directory.

Assume that pwd is /home/kumar , then gmit directory is created under kumar directory.

Ex 2: To create three directories at a time, named patch, dbs, doc, pass directory names as argu- ments.

$mkdir patch dbs doc

Ex 3:To create a directory tree:

To create a directory named gmit and create two subdirectories named cse and ise under gmit, issue the command. gmit is a parent directory.

$mkdir parent directory sub-directories

$mkdir gmit gmit/cse gmit/ise

Ex 4: Error while creating a directory tree

$mkdir gmit/cse gmit/ise

mkdir: Failed to make a directory “gmit/cse”; no such file or directory

mkdir: Failed to make a directory “gmit/ise”; no such file or directory

---

Error is due to the fact that the parent directory named gmit is not created before creating sub directo- ries cse and ise.

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

Ex 6.4.1: **$rmdir progs**

removes the directory named progs

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	**gmit/cse**	**gmit/ise**	**gmit**

- You cant delete a directory with rmdir unless it is empty.In this example gmit directory cannot be removed until the sub directories cse and ise 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.

ls : listing directory contents:

To obtain a list of all filenames in the current directory.

Numerals first

Uppercase next

Lowercase Then

$ls

Output:08_packets.html
calendar
dept.lst
emp.lst
helpdir
uskdsk06

ls options:

- Output in multiple columns(-x):

$ls -x

08_packets.html	**calendar**	**dept.lst**	**emp.lst**
helpdir	**progs**	**usdsk07**	**usdsk07**

Identifying directories and executables(-F)

$ ls –Fx

08_packets.html calendar* cptodos.sh* dept.lst
emp.lst helpdir / progs / usdsk07

The * indicates the file contains the executable code and / refers to directory

- Showing Hidden files also(-a):

$ls –axF

.profile .exrc .kshrc .xinitrc
08_packets.html calendar* cptodos.sh* dept.lst emp.lst
helpdir / progs / usdsk07

The hidden files are indicated by **.** (dot) displayed before filename.

- Listing directory contents:

$ls –x helpdir progs

helpdir:
forms.obd graphics.obd
progs:
array.pl n2words.pl

If we specify two directories named helpdir and progs , the contents of the directory i,e filenames are listed out.

- Recursive listing(-R)

The recursive option lists all sub-directories and files in a directory tree structure.

$ls -xR

08_packets.html calendar cptodos.sh dept.lst
emp.lst helpdir progs usdsk07
./helpdir
forms.hlp graphics.hlp
./progs
arrays.pl n2words.pl

FILE RELATED COMMANDS

Cat command : Displaying and creating files

cat is one of the most well known commands of UNIX system.

Cat is useful for creating a file .

Its mainly used to display the contents of a small file on the terminal.

- **Using cat to create a file:**
  Enter the command cat, followed by >(right chevron) character and the filename.
  Example: take a filename named foo
  $ cat > foo
  > Symbol following command means that the output goes to filename following it.
  [ctrl+d] /* to terminate or to signify end of the input.
  $
- **Using cat to display a file**
  Enter the cat command followed by filename
  $ cat foo
  Symbol following command means that the output goes to filename following it.

Cat options (-v and -n)

Displaying Non printing characters(-v)

cat is normally used for displaying text files only. If you have non-printing ASCII characters in your input , you can see cat with -v option to display these characters

Numbering lines(-n)

The -n option numbers lines.

Cat with more than one filename as arguments:

cat filename1 filename2 ....

cat chap01 chap02

The contents of second file are displayed immediately after the first file without any header information.

---

cp: copying a file

- cp command copies a file or a group of files.it creates an exact image of the file on the disk with the different name.
- The syntax requires atleast two filenames to be specified in the command line.
- When both are ordinary files, the first is copied to second file.

  cp source file destination file

cp chap01 unit1

if destination file i.e unit1 does not exist, first it will be created before copying.if not it will be simply overwritten without any warning.

- Copying a file to another directory
  ex: assume there is a file named chap01 and it has to be copied to progs directory
  cp chap01 progs
  output: chap01 is now copied to directory named progs with the same name chap01.
- Copying a file to another directory with different name
  ex: assume there is a file named chap01 and it has to be copied to progs directory with chap01 file renamed as unit1
  cp chap01 progs/unit1
  output: chap01 is now copied to directory named progs with the same name unit1

- Copy more than one file with a single command.
  cp chap01 chap02 chap03 progs
  chap01, chap02, chap03 files are copied to directory named progs.
- Copy all files beginning with chap
  cp chap* progs

cp options:

Interactive copying (-i): the -i option warns the user before overwriting the destination file.

Ex: **$ cp -i chap01 unit1**
cp: overwrite unit1(yes/no)? y
A y at this prompt will overwrite the file.

Copying directory structure(-R) : the -R command behaves recursively to copy an entire directory structure say progs to newprogs.

Ex say progs directory contains three files kernel, bash, korn. To copy all three files under progs to newprogs directory

$ cp -R progs newprogs

---

rm : deleting files

The rm command deletes one or more files.

Ex 1: The following command deletes three files chap01, chap02, chap03.

$ rm chap01 chap02 chap03

Ex 2: to delete files named chap01 and chap02 under progs directory

$ rm progs/chap01 progs/chap02

Ex 3: to remove all file

$ rm*

rm options:

Interactive deletion (-i): the –i optin makes the command ask the user for confirmation before removing each file.

$rm -i chap01 chap02 chap03

rm: remove chap01(yes/no)?y
rm: remove chap01(yes/no)?y
rm: remove chap01(yes/no)?y

Recursive deletion(-r or -R) deletes all subdirectories and files recursively. Rm wont normally remove directories but when used with -r or -R option it will.

$ rm -r *

Forcing removal: rm prompts for removal, if a file is write protected. The -f option overrides this minor protection and forces removal.

$ rm -rf * /*(deletes everything in the current directory and below)

---

mv: RENAMING FILES.

The mv command renames or moves files. It has two distinct functions:

h. It renames a file or directory
i. it moves a group of files to a different directory

To rename a file chap01 to man01

$ mv chap01 man01

mv replace the filename in the existing directory entry with the new name.
No additional space is consumed on disk during renaming.

To rename a directory:

$ mv pts perdir

pts directory is renamed as perdir

To move group of files to a directory

mv chap01 chap02 chap03 progs

to move three files chap01, chap02, chap03 to directory named progs

wc command: COUNTING LINES,WORDS,CHARACTERS

wc command takes one or more filenames as arguments and displays four columnar output.

First we will create a file named infile

$ cat > infile
I am the wc command
I count characters,words and lines
[ctrl+D]

$wc infile

2 10 55 infile

wc counts lines in first column ,words in second column,characters in third column and filename in fourth column..

A line is any group of characters not containing a newline

A word is group of characters not containing a space tab or newline.

A character is the smallest unit of information and includes a space, tab and newline

wc options:

$ wc -l infile

2

$wc -w infile

10

$wc -c infile

55

- When two filenames are passed as wc argument

```
[vizion@localhost ~]$ cat > chap01
unix is a multitasking os
its is a multiuser os
[vizion@localhost ~]$ cat > chap02
who cal date
ls rm mv
[vizion@localhost ~]$ wc chap01 chap02
 2 10 48 chap01
 2  6 22 chap02
 4 16 70 total
```

First line : number of lines, words and characters of chap01

Second line: number of lines, words and characters of chap02

Third line: Total number of lines, words and characters of both.

---

od Command: DISPLAYING DATA IN OCTAL.

$ cat odfile

White space includes a

The ^G character rings a bell

$ od -b odfile

The -b option displays the octal values for each character.

000000 127 150 151 164 145 040 163 160 141 143 145 040 151 156 143 154

000000 165 144 145 163 040 141 040 011 012 124 150 145 040 007 040 143

Each line displays 16 bytes of data in octal , preceded by the offset in the file of the first byte in the line.

$od -bc odfile

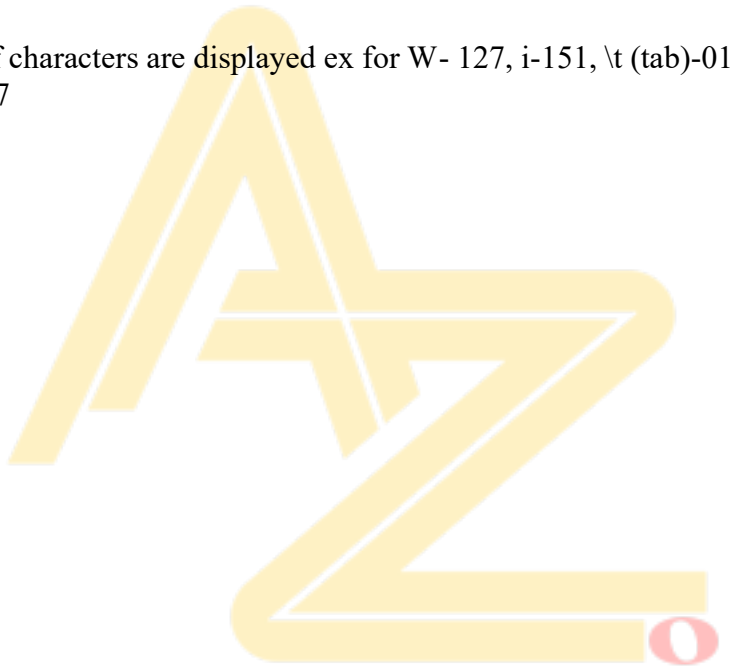
The -b and -c option combined

Each line is now replaced with two.

The octal values are shown in first line and printable characters and escape sequences are shown in second line

```
000000 127 150 151 164 145 040 163 160 141 143 145 040 151
         W   h   i   t   e       s   p   a   c   e       i
       156 143 154
         n   c   l
000000 165 144 145 163 040 141 040 011 012 124 150 145 040
         u   d   e   s       a       \t  \n  T   h   e

       007 040 143
       007     c
```

The octal equivalent of characters are displayed ex for W- 127, i-151, \t (tab)-011, \n(newline)-012 ^G(Bell character)- 007



MODULE 2

FILE ATTRIBUTES AND PERMISSIONS

The ls command with options

ls –l: LISTING FILE ATTRIBUTES

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

File type and Permissions

The file type and its 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. In unix, file system has 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.

Ownership: The third column shows the owner of files. The owner has full authority to tamper with files content and permissions. Similarly, you can create, modify or remove files in a directory if you are the owner of the directory.

Group ownership: The fourth column represents the group owner of the file. When opening a user account, the system admin also assigns the user to some group. The concept of a group of users also owning a file has acquired importance today as group members often need to work on the same file.

File size: The fifth column shows the size of the file in bytes. The important thing to remember here is that it only a character count of the file and not a measure of the disk space that it occupies.

Last modification time: The sixth, seventh and eighth 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 content have changed in any way.If the file is less than a year old since its last modification time, the year won't be displayed.

Filename: The last column displays the filename arranged in ASCII collating sequence.

For example, $ ls –l

total 72

-rw-r--r--	1	kumar	metal	19514	may	10 13:45	chap01
-rw-r--r--	1	kumar	metal	4174	may	10 15:01	chap02
-rw-rw-rw-	1	kumar	metal	84	feb	12 12:30	dept.lst
-rw-r--r--	1	kumar	metal	9156	mar	12 1999	genie.sh
drwxr-xr-x	2	kumar	metal	512	may	09 10:31	helpdir
drwxr-xr-x	2	kumar	metal	512	may	09 09:57	progs

FILE PERMISSIONS

- UNIX has a simple and well defined system of assigning permissions to files.
- Lets issue the ls –l command once again to view the permissions of a few lines .

```
$ls   -l    chap02    dept.lst    dateval.sh
-rwxr-xr--    1    kumar    metal    25000    May 10  19:21    chap02
-rwxr-xr-x    1    kumar    metal    890      Jan 10  23:17    dept.lst
-rw-rw-rw-    1    kumar    metal    84       Feb 18  12:20    dateval.sh
```

Consider the first column.

- rwx r-x r—

Each group here represents the category and contain three slots representing the read, write and execute permissions of the file.

r indicates the read permission; **w** indicates write permission; **x** indicates execute permission

- (hyphen) indicates the absence of the corresponding permission.

In the above example, the file permissions of chap02 file is

-	**rwx**	**r-x**	**r - -**
File	**owner/user**	**group**	**others**

First group(rwx) has all the three permissions.

- The file is readable, writable and executable by the **owner** of the file, kumar.
- The third column shows the owner of the file.
- The first permissions group applies to kumar.
- You have to login with the name kumar for the privileges to apply to you.

Second group(r-x):

- has a hyphen in the middle slot, which indicates the absence of write permissions by the **group** owner of the file.
- The group owner is metal and all users belonging to group metal has only read and execute permissions.

Third group(r--):

- has the write and execute bits absent.
- This set is applicable to **others** i,e those who are neither the owner nor group.
- This category is referred to as the world.

CHANGING FILE PERMISSIONS:RELATIVE AND ABSOLUTE PERMISSIONS

chmod command.

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

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)

The below shows the abbreviations used by **chmod** command

Category	**operation**	**permission**
u - user	+ assign	r - read
g – group	- remove	w - write
o - others	= absolute	x - execute
a - all (ugo)		

Ex 1:
$ls –l xstart
-rw-r--r-- 1 kumar metal 1906 sep 23:38 xstart
Here user is having the only read and execute permission .

Using relative file permission need to add the execute permission to user

chmod	category	operation(+,-)	permission	filename.
$chmod	u	+	x	xstart

$chmod u+x xstart

$ ls –l xstart

-rwxr--r-- 1 kumar metal 1906 sep 23:38 xstart

After executing the **chmod** command, the command assigns (+) execute (x) permission to the user (u), other permissions remain unchanged.

Ex 2: To remove execute permission from all and assign read permission to group and others

$chmod a-x, go+r xstart /*to remove execute permission from all(a)ie user, group, others
/*to assign read permission to group and others (go+r)

Ex 3: To assign write and execute permissions for others.

$chmod o+wx xstart

ABSOLUTE PERMISSIONS

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
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.

Ex 1: To assign read,write permissions to all .Using **relative** permission, we have,

$chmod a+rw xstart

Using **absolute** permission, we have,

$chmod 666 xstart /* 6 for r-w

/* first digit 6 for user, second 6 for group and third 6 for others

Ex 2:
To assign read and write for user and remove write, execute permissions from group and others

- Here to assign rw- corresponds to digit 6
- Remove write , execute permissions is nothing but assigning only read option to group and others
- Only read permission is r—corresponds to 4

$chmod 644 xstart

Ex 3:
To assign all permissions to the owner, read and write to group and only execute for others.

$chmod 761 xstart

Ex 4
To assign all permissions to all categories.

$chmod 777 xstart

5.2 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

After Executing above any one command the output will be removes the all permission from all categories as shown below

----

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

On the other hand,

chmod a+rwx xstart or

chmod 777 xstart

After Executing either of the one command it adds the all permission to all categories as shown below

-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.

RECURSIVELY CHANGING FILE PERMISSIONS

use chmod Recursively.(-R)

- It possible to make **chmod** descend directory hierarchy and apply the expression to every file and sub directory it finds. This is done by using -R option

$chmod -R a+x shell_scripts

This makes all the files and subdirectories found in the shell_scripts directory, executable by all users.

DIRECTORY PERMISSIONS

- Directories also have their own permissions and the significance of these permissions differ from those of ordinary files.
- The default permissions of a directory are,

  rwxr-xr-x (755)
- A directory must never be writable by group and others

Ex1:

$mkdir c_progs ; ls –ld c_progs

drwxr-xr-x 2 kumar metal 512 may 9 09:57 c_progs

Here the c_progs directory is created (mkdir c_progs) and then the attributes of directory is listed out(ls –ld c_progs)

If a directory has write 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.

CHANGING FILE OWNERSHIP

There are two commands meant to change the ownership of a file or directory.

chown changing file owner and

chgrp changing group owner

chown : Changing file owner

- The syntax is

  chown options owner [:group] file(s)

- For changing ownership requires super user permission. So let's first change our status to that of super user with the **su** command:
  $su
  Password: ******
  #_
- After the password is successfully entered, **su** returns a # prompt, the prompt used by root. **su** lets us acquire super user status if we know the root password.
- Consider the file note owned by kumar.To now renounce the ownership of the file note to Sharma, use **chown** in the following way:
  # ls -l note
  -rwxr----x 1 kumar metal 347 may 10 20:30 note

  #chown sharma note; ls -l note

  -rwxr----x 1 sharma metal 347 may 10 20:30 note

Once ownership of the file has been given away to sharma, the user file permissions that previously applied to Kumar now apply to sharma. Thus, Kumar can no longer edit *note* since there is no write privilege for group and others. He cannot get back the ownership either. But he can copy the file to his own directory, in which case he becomes the owner of the copy.

chgrp :Changing Group Owner

- This command changes the file's group owner. No superuser permission is required.

$ ls –l dept.lst
-rw-r--r-- 1 kumar **metal** 139 jun 8 16:43 dept.lst
Here the group owner of dept.lst is metal

$chgrp dba dept.lst; ls –l dept.lst
-rw-r--r-- 1 kumar **dba** 139 jun 8 16:43 dept.lst

The group owner of the file dept.lst is changed from metal to dba by issuing the command
$chgrp dba dept.lst

The file attributes of dept.lst is listed by issuing the command
$ls –l dept.lst

- ***Using chown to change both file owner and group :***
  The syntax requires two arguments to be separated by :
  chown sharma:dba dept.lst
  Here the ownership of dept.lst is changed to sharma and group to dba

THE SHELLS INTERPRETIVE CYCLE

The SHELL'S INTERPRETIVE CYCLE

When you log onto a UNIX machine , you see a prompt.This prompt remains until you key in something.Even though it may appear that the system is idling, a UNIX command is in fact running at the terminal.But this command is special its with you all the time and never terminate unless you log out. This command is **shell**.

If you provide the input in the form of ps command (that shows processes owned by you), you will see shell running

```
$ps
PID   TTY   TIME CMD
328   pts/2 0:00  bash
```

- The bash shell is running at the terminal /pts/2.
- When you key in the command it goes to shell as input.
- The shell scans the command line for metacharacters.These are the characters that mean nothing to the command but has special meaning to the shell.
- For example if the shell encounters metacharacters like *,| etc in the command line.
- If the metacharacter is *, then shell replaces it with all the filenames in the current directory.
- When all preprocessing is complete , the shell passes on the command to the kernel for ultimate execution.
- While the command is running, the shell has to wait for notice of its termination from the kernel.
- After the command is complete with its execution, then shell once again issues the prompt to take up your next command.

ACTIVITIES PERFORMED BY THE SHELL IN ITS INTERPRETIVE CYCLE.

1.The shell issues the prompt and waits for you to enter a command.

2.After a command is entered, the shell scans the command line for metacharacters and expands abbreviations (like * in rm *) to recereate the simplified command line.

3.It then passes on the command line to the kernel for execution

4.The shell waits for the command to complete and normallty cant do any work while the command is running.

5. After command execution is complete, the prompt reappers and shell returns to its waiting role to start the next cycle. Yo can now enter the next command.

WILD CARDS

The metacharacters taht are used to construct the generalized pattern for matching filenames belong to the category called **wild cards.**

The * and ?

- **The metacharacter *** is one of the characters of the shell wild card set. It matches any number of characters including none.
- For example : to match filenames chap chap01 chap02 chap03 chap04

  $ ls chap*

  Output : // the * matches all strings along with none

  chap
  chap01
  chap02
  chap03
  chap04

- **The metacharacter ?** matches a single character
  For example: to match filenames chapx chapy chapz

  $ ls chap?
  Output: //the ? replaces single character i,e ? is replaced by x, y,z
  chapx
  chapy
  chapz

Matching the dot(.)

- The **.** dot metachacter can be used to match all the hidden files in your directory.
- Example: To list all hidden files in your directory having atleast three characters after the dot.

  $ ls .???*
  Output:
  .bash_profile
  .exrc
  .netscape
  .profile

The character class [ ]

- This class comprises a set of charcters enclosed by the rectangular brackets [and],but it matches only a single character in the class.
- The **pattern [abcd]** is a character class and it matches a single character a, b, c or d
- For example to match chap01 chap02 chap03 chap04

  $ ls chap0[1234]
  output:
  chap01

chap02
chap03
chap04

Negating the character class(!)

- It is used to reverse the matching criteria.
- For example:
  To match all the filenames with single character extensions but not .c or .o files
  $ ls *.[!co]
  *** to match any filename**
  . extension
  ! except
  [!co]---> .c extension or .o extension
- To match filenames that does not begin with a digit
  $ ls [!0-9]*
- To match filename with 3 character that does not begin with an Upper Case letter
  $ ls [!A-Z]??

REMOVING THE SPECIAL MEANINGS OF WILD CARDS

ESCAPING and QUOTING

Escaping: providing a \(backslash character) before the wild card to remove or escape its special meaning.

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

ESCAPING

- Placing a \ immediately before a metacharacter turns off its special meaning.
- For instance * , matches * itself. Its special meaning of matching zero or more occurrencesof character is turned off.

  Ex 1:

  $rm chap*

  removes all the filenames starting with chap. Chap, chap01,chap02 and chap03 are removed.

  $rm chap* // * metacharacter meaning is turned off

  removes the filename with chap* /*name of the file itself is chap*.

Ex 2:

- If there are files with names chap01, chap02,chap03.
- To list the filenames starting with chap0

$ ls chap0[1-3]
Output:
chap01
chap02
chap03

Ex 3:
To match the file named as chap0[1-3]
$ ls chap0\[1-3\]
Output:
chap0[1-3]

Escaping the space: To remove the file My document.doc, which has space embedded,
$rm My\ document.doc

Escaping the newline character.
$echo -e "The newline charcter is \n Enter the command"
Output:
The newline character /n – newline character is interpreted and cursor moves to next line
Enter the command

$ echo "the newline character is \\n enter the command"
Output:

the newline character is \\n enter the command /* here newline character interpretation is turned off and the \n is printed as it is*/

Escaping the \ itself
**$echo **
Output
\

QUOTING:

- This is the another way of turning off the meaning of metacharacter.
- When a command argument is enclosed within quotes, the meaning of all enclosed special characters are turned off

$rm 'chap*' // * metacharacter meaning is turned off

removes the filename with chap* /*name of the file itself is chap*.

$rm "My\ document.doc" /* To remove the file My document.doc, which has space embedded.

$echo '\'

output

\

REDIRECTION: THE THREE STANDARD FILES

Redirection is the process by which we specify that a file is to be used in place of one of the standard files.

- With input files, we call it input redirection;
- With output file, we call it output redirection
- With the error file, we call it error redirection.

Standard input: The file representing the input, which is connected to the keyboard

Standard output: The file representing the output, which is connected to the display.

Standard error: the file representing the error messages that emanate from the command or shell. This is also connected to display.

Each of the three standard files are represented by a number called a **file descriptor.**

The first three slots are generally allocated to three standard streams in this manner

0: standard input

1: standard output

2: standard error

STANDARD INPUT

This file is indeed special

- The keyboard, the default source
- **a file using redirection with the < symbol**
- another program using the pipeline

- The input redirection operator is less than character (<).
- When you use wc without an argument , it prompts you to provide the input from standard input keyboard

  $ wc

  Unix is a multiuser multitasking OS

  [ctrl-d]
- When wc is used with argument. Filename is passed as an argument i,e wc takes the input from the filename we have specified

  For example: Create a file with name sample.txt

  $ vi sample.txt

  Unix is a multiuser multitasking OS

  :wq /*saving and quitting from the file

  $ wc < sample.txt /*wc command takes input from the file sample.txt

  output

  1 6 36 /* count of characters, words, lines of file sample.txt

STANDARD OUTPUT

- All commands displaying the output on the terminal actually write to the standard output file as a stream of characters and not directly to the terminal as such.
- There are three possible destinations of this stream
- The terminal, the default destination
- **A file using the redirection symbol > and >>**
- As input to another program using a pipeline
- There are two basic redirection operators for standard output.

a. greater than character(>):

If you want the file to contain only the output from this execution of the command, you can use greater than token.

Ex: consider the file **sample.txt**

$ cat sample.txt /* to display the content of sample.txt

Unix is a multiuser multitasking OS

$ wc sample.txt > newfile

> symbol redirects the output of wc command to a file named newfile

The output is now stored in newfile

$ cat newfile /* To view the content of newfile

1 6 37

b Two greater than characters>> (append)

If you want the output of the command to be appended without overwriting the existing content.

$who >> newfile

The output of who command is appended to newfile without overwriting the existing content.

$cat newfile

1 6 37 /* output of wc command executed previously

root console aug 1 07:51 (:0) /* output of who command

kumar pts/10 aug 1 02:51 (:0)

sharma pts/6 aug 1 03:51 (:0)

STANDARD ERROR

When you enter an incorrect command or try to open a non existent file, certain diagnostic messages show up on the screen. This is the standard error stream whose default destination is the terminal.

Standard output and error on monitor

Ex 1: Consider two files file1 and file2 , where file1 exist and file2 do not exist

$ ls –l file1 file2

-rwxr—r-- 1 gilberg staff 1234 oct file1

Cannot access file2: no such file or directory /*error because file2 do not exist

---

Ex 2: To redirect standard output to same file

$ls -l file1 file2 1>filelist 2>filelist

The 1st argument is file1 and 2nd argument is file2.

The output and errors are sent to file named filelist

$cat filelist

-rwxr—r-- 1 gilberg staff 1234 oct file1

Cannot access file2: no such file or directory

Ex 3: To redirect standard output to different files

$ls -l file1 file2 1>stdout 2>stderr

The 1st argument is file1 and 2nd argument is file2.

The output of 1st file is sent to filename stdout and errors are sent to file named stderr.

$cat stdout

-rwxr—r-- 1 gilberg staff 1234 oct file1

$cat stderr

Cannot access file2: no such file or directory

CONNECTING COMMANDS: PIPE.

- We often need to use a series of commands to complete a task. For example, if we need to see list of users logged into the system, we use who command.However if we need a hard copy of the list, we need two commands.
- First use **who** command to get the list and store the result in file using redirection
- We then use **lpr** command to print the file
- We can avoid the creation of intermediate file by using a pipe

Pipe is an operator that temporarily saves the output of one command in a buffer that is being used at the same time as the input of the next command.

- The first command must be able to send its output to standard output. The second command must be able to read its input from standard input.
- **The token for a pipe is vertical bar(|)**

  $ who | lpr

  We use the **who command** because it reads from the system and sends the list of users to standard output.The pipe command uses a buffer to send the piped data to next command.The recieving command must recieve its data from standard input.

  $who

  root console aug 1 07:51 (:0)

  kumar pts/10 aug 1 02:51 (:0)

  sharma pts/6 aug 1 03:51 (:0)

  rajath pts/8 aug 1 06:51 (:0)

vikas pts/14 aug 1 09:51 (:0)

$who | wc -l
Output: 5 /* count of number of lines of who command

Here the output of **who** command has been passed directly as the input to **wc** command and **who** is said to be piped to **wc.**

The grep,egrep

grep(globally search regular expression and print)

Unix has a special family of commands for handling search requirements and the principal member of the family is the **grep command**.

grep scans its input for a pattern and displays lines containing the pattern, the line numbers or filename where the pattern occurs.

Syntax:

grep options pattern filename(s)

grep searches for pattern in one or more filename or the standard input if no filename is specified.

The first argument(barring the options) is the pattern and the remaining arguments are filenames.

Consider three files emp.lst, emp1.lst, emp2.lst

$cat emp.lst

```
101|sharma|general manager|sales|10/09/61|6700
102|kumar|director|Sales|09/09/63|7700
103|aggarwal|manager|sales|03/05/70|5000
104|rajesh|manager|marketing|12/04/72|5800
105|adarsh|executive|sales|07/09/57|5300
```

$cat emp1.lst

```
201|anil|director|sales|05/01/59|5000
202|sunil|director|marketing|12/06/51|6000
203|gupta|director|production|09/08/55|7000
```

$cat emp2.lst

```
301|anil Agarwal|manager|sales
302|sudhir agarwal|director|production
303|rajath Agarwal|manager|sales
304|v.k.agrawal|director|marketing
305|v.s.agrawal|deputy manager|sales
306|agrawal|executive|sales
401|sumith|general manager|marketing
402|sudhir agarwal|director|production
```

Ex 1: Now to search the pattern **sales** in **emp.lst** file

```
$ grep "sales" emp.lst
101|sharma|general manager|sales|10/09/61|6700
103|aggarwal|manager|sales|03/05/70|5000
105|adarsh|executive|sales|07/09/57|5300
```

Ex 2: To search the pattern director from 2 files i,e emp.lst and emp2.lst

```
grep "director" emp.lst emp2.lst
emp.lst:102|kumar|director|Sales|09/09/63|7700
emp2.lst:302|sudhir agarwal|director|production
emp2.lst:304|v.k.agrawal|director|marketing
emp2.lst:402|sudhir agarwal|director|production
```

grep along with options

Table 13.1 Options Used by **grep**

Option	*Significance*
-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. Can use multiple times. Also used for matching expression beginning with a hyphen.
-x	Matches pattern with entire line (doesn't match embedded patterns)
-f *file*	Takes patterns from *file*, one per line
-E	Treats pattern as an extended regular expression (ERE)
-F	Matches multiple fixed strings (in **fgrep**-style)

Ignoring case: when you look for a name but are not sure of the case, use the –i option to ignore case for pattern matching.

```
$ grep -i "sales" emp.lst
101|sharma|general manager|sales|10/09/61|6700
102|kumar|director|Sales|09/09/63|7700
103|aggarwal|manager|sales|03/05/70|5000
105|adarsh|executive|sales|07/09/57|5300
```

Deleting lines(-v): The –v option selects all lines except those containing the pattern

```
$ grep -v "director" emp2.lst
301|anil Agarwal|manager|sales
303|rajath Agarwal|manager|sales
305|v.s.agrawal|deputy manager|sales
306|agrawal|executive|sales
401|sumith|general manager|marketing
```

The lines containing the pattern **director** are deleted in the output.

Dislaying line numbers(-n).

The –n option displays the line numbers containing the pattern along with the line

```
$ grep -n "sales" emp.lst
1:101|sharma|general manager|sales|10/09/61|6700
3:103|aggarwal|manager|sales|03/05/70|5000
5:105|adarsh|executive|sales|07/09/57|5300
```

Counting lines containing patter(-c)

The –c option counts the number of lines containing the pattern.

```
$ grep -c "manager" emp2.lst
4
```

Dislaying filenames(-l)

The –l option displays only the names of the files containing the pattern.

Here the pattern **manager** is searched in all files ending with .lst (*.lst)

```
$ grep -l "manager" *.lst
emp2.lst
emp.lst
```

Matching multiple patterns(-e)

By using –e option, you can match multiple patterns

```
$ grep -e "agarwal" -e "Agarwal" -e "agrawal" emp2.lst
301|anil Agarwal|manager|sales
302|sudhir agarwal|director|production
303|rajath Agarwal|manager|sales
304|v.k.agrawal|director|marketing
305|v.s.agrawal|deputy manager|sales
306|agrawal|executive|sales
402|sudhir agarwal|director|production
```

BASIC AND EXTENDED REGULAR EXPRESSIONS
TYPICAL EXAMPLES INVOLVING DIFFERENT REGULAR EXPRESSIONS

BASIC REGULAR EXPRESSIONS

Table 13.2 The Basic Regular Expression (BRE) Character Subset

Symbols or Expression	*Matches*
*	Zero or more occurrences of the previous character
g*	Nothing or g, gg, ggg, etc.
.	A single character
.*	Nothing or any number of characters
[*pqr*]	A single character *p*, *q* or *r*
[*c1-c2*]	A single character within the ASCII range represented by *c1* and *c2*
[1-3]	A digit between 1 and 3
[^*pqr*]	A single character which is not a *p*, *q* or *r*
[^a-zA-Z]	A nonalphabetic character
^*pat*	Pattern *pat* at beginning of line
pat$	Pattern *pat* at end of line
bash$	bash at end of line
^bash$	bash as the only word in line
^$	Lines containing nothing

The character class []

A regular expression lets you specify a group of characters enclosed within a pair of rectangle brackets [ ], in which the match is performed for a single character in the group.

Thus the expression

[aA] Matches either a or A

```
$ grep "[aA]garwal" emp2.lst
301|anil Agarwal|manager|sales
302|sudhir agarwal|director|production
303|rajath Agarwal|manager|sales
402|sudhir agarwal|director|production
```

Negating a class(^)

Regular expressions use the **caret(^)** to negate the character class, while the shell uses **bang(!)**

Ex:

[^a-zA-Z] matches a non-alphabetic character

The *(asterisk)

The * refers to the immediately preceding character.Here it indicates that the previous character can occur many times or not at all.

The patttern g*

Matches none, g, gg, gg, ggg,.....

```
$ cat file.lst
ourrences of a character
ocurrences of a character
occurrences of a character
occcurrences of a character

$ grep "oc*urrences" file.lst
ourrences of a character
ocurrences of a character
occurrences of a character
occcurrences of a character
```

The dot (.)

A **.** matches a single character where as the shell uses **?** to indicate that.

```
$ grep "10." emp.lst
101|sharma|general manager|sales|10/09/61|6700
102|kumar|director|Sales|09/09/63|7700
103|aggarwal|manager|sales|03/05/70|5000
104|rajesh|manager|marketing|12/04/72|5800
105|adarsh|executive|sales|07/09/57|5300
```

Here the . matches single character. It list all files beginning with 10 followed by single character.

It displays lines with id 101,102,103,104,105.

7.1.1 Specifying pattern locations(^ and $)

The two regular expressions characters that match pattern at the beginning or end of line .

^(caret) Matching at the beginning of the line

$(dollar) Matching at the end of the line

```
$ grep "^3" emp2.lst
301|anil Agarwal|manager|sales
302|sudhir agarwal|director|production
303|rajath Agarwal|manager|sales
304|v.k.agrawal|director|marketing
305|v.s.agrawal|deputy manager|sales
306|agrawal|executive|sales
```

^3 matches all the lines beginning with digit 3.

```
$ grep "5...$" emp.lst
103|aggarwal|manager|sales|03/05/70|5000
104|rajesh|manager|marketing|12/04/72|5800
105|adarsh|executive|sales|07/09/57|5300
```

5...$ matches all the lines ending with four digit number beginning with 5.

EXTENDED REGULAR EXPRESSIONS(ERE)

ERE make it possible to match dissimilar patterns with a single expression.
The ERE has to be used with **–E option.**

Table 13.3 The Extended Regular Expression (ERE) Set Used by **grep**, **egrep** and **awk**

Expression	*Significance*
ch+	Matches one or more occurrences of character *ch*
ch?	Matches zero or one occurrence of character *ch*
exp1|exp1	Matches *exp1* or *exp2*
GIF|JPEG	Matches GIF or JPEG
(x1|x2)x3	Matches *x1x3* or *x2x3*
(lock|ver)wood	Matches lockwood or verwood

The + and ?

+ Matches one or more occurrences of the previous character

? Matches zero or one occurrences of the previous character.

```
$ grep -E "oc+urrences" file.lst
ocurrences of a character
occurrences of a character
occcurrences of a character
```

The + symbol matches one or more occurrences of character c i,e **c, cc, ccc**
The occurrences of a character is not matched by +c , since there is no c in the occurrences.

```
$ grep -E "oc?urrences" file.lst
ourrences of a character
ocurrences of a character
```

The ? symbol matches zero or one occurrences of character c i,e **0 c , 1 c.**
The other two lines occurrences (2 c) and occurrences (3 c) are not matched .

Matching multiple patterns(**|, (and)**)

- The | is the delimiter for the multiple patterns.
- Consider the file ere.lst with two lines of employee details.

```
$ cat ere.lst
235|barun sengupta|director|sales
279|s.n. dasgupta|manager|marketing
```

- To locate both sengupta and dasgupta from file ere.lst

```
$ grep -E "sengupta|dasgupta" ere.lst
235|barun sengupta|director|sales
279|s.n. dasgupta|manager|marketing
```

- The characters (and) lets you group patterns and use of | inside the parenthesis, you can frame more compact pattern

```
$ grep -E "(sen|das)gupta" ere.lst
235|barun sengupta|director|sales
279|s.n. dasgupta|manager|marketing
```

SHELL PROGRAMMING

ORDINARY AND ENVIRONMENT VARIABLES

A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at command prompt.

VARIABLE NAMES

- A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.
- A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.
- The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_).
- By convention, Unix Shell variables would have their names in UPPERCASE.
- The following examples are valid variable names −

  VAR_1

  VAR_2

  TOKEN _A

DEFINING VARIABLES:

- Variables are defined as follows –
- variable_name=variable_value
  For example:
  NAME="Sumitabha Das"

ACCESSING VARIABLES:

- To access the value stored in a variable, prefix its name with the dollar sign ($) –
- For example, following script would access the value of defined variable NAME and would print it on STDOUT –

```
#!/bin/sh
NAME="Sumitabha Das"
echo $NAME
```

ENVIRONMENT VARIABLES –

An environment variable is a variable that is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually a shell script defines only those environment variables that are needed by the programs that it runs.

SHELL: points to the shell defined as default.

DISPLAY : Contains the idenifier for the display that X11 programs should use by default.

HOME: Indicates the home directory of the current user; the default argument for the cd built in command

IFS: Indicates the Internal Field Separator that is used by the parser for word splitting after expansion.

PATH : Indicates search path for commands.It is a colon separated list of directories in which the shell looks for commands.

PWD: Indicates the current working directory as set by the cd command.

RANDOM: Generates a random integer between 0 and 32767 each time it is referenced.

SHLVL: Increments by one each time an instance of bash is created.

UID: Expands to the numeric user ID of the current user initialized at shell prompt.

- Following is the sample example showing few environment variables −

```
$ echo $HOME
/root
]$ echo $DISPLAY

$ echo $TERM
xterm
$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/home/amrood/bin:/usr/local/bin
$
```

PS1(Prompt String one) and PS2 Environment Variables

The characters that the shell displays as your command prompt are stored in the variable PS1. You can change this variable to be anything you want. As soon as you change it, it'll be used by the shell from that point on.

For example, if you issued the command −

```
$PS1='=>'
=>
=>
```

Your prompt would become =>.

When you issue a command that is incomplete, the shell will display a secondary prompt and wait for

you to complete the command and hit Enter again.

The default secondary prompt is > (the greater than sign), but can be changed by re-defining the **PS2** shell variable −

Following is the example which uses the default secondary prompt −

```
$ echo "this is a
> test"
this is a
test
$
```

```
$PS= '-->'
$ echo "this is a
--> test"
```

The .profile File

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

  A list of directories in which to locate commands

  A list of variables effecting look and feel of your terminal.
- You can check your **.profile** available in your home directory. Open it using **vi**editor and check all the variables set for your environment.

SHELL SCRIPTS

When a group of commands have to be executed regularly they should be stored in a file and the file itself executed as a **shell script or shell program.**

Structure of shell script:

```
#!/bin/sh
# script.sh: Sample shell script
echo " Todays date: `date`"
echo   "This month calendar"
cal   `date` "+%m 20%y"
echo "My Shell: $SHELL"
```

output:

```
$sh script.sh
Todays date : Mon Nov 7 10:03:42 IST 2016
```

This month's calendar:
November 2016

Su	Mo	Tu	We	Th	Fr	Sa
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30			

My shell: /bin/sh

- Use your vi editor to create the shell script script.sh.
- The script runs three echo commands and shows the use of variable evaluation and command substitution. It also prints the calendar of the current month.
- Note that the # is comment character, that can be placed anywhere in a line; the shell ignores all characters placed on its right.
- However this doesnot apply to the first line, which also begins with a #. This is interpreter line that was mentioned previously.
- It always begins with #! and is followed by pathname of the shell to be used for running the script. However this line specifies the bourne shell.
- To run the script, make it executable first and then invoke the script name

  $chmod a+x script.sh
  $sh script.sh
- Shell scripts are executed in a separate child shell process and this sub shell need not be of the same type as your login shell.By default child and parent shell belongs to the same type, but you can provide a interpreter line in the first line of the script to specify a different shell for your script.

read and readonly commands.

read:MAKING SRIPTS INTERACTIVE

- The read statement is the shell internal tool for taking the input from the user ie making scripts interactive.
- It is used with one or more variables. Input is supplied through the standard input is read into these variables.
- When you use statement like:

  read name

  the script pauses at that point to take input from the keyboard. whatever you enter is stored in the variable name. since this is a form of assignment , no $ is used before the name.

- A single read statement can be used with one or more variables to let you enter multiple arguments.

  read pname flname

- The script asks for a pattern to be entered. Input the string director, which is assigned to the variable pname. Next the script asks for the filename enter the string emp.lst which is assigned to the variable flname.
- grep runs with these two variables as arguments

```
#!/bin/sh
#emp1.sh
#
echo "Enter the pattern to be searched : \c"
read pname
echo " Enter the file to be used : \c"
read flname
echo " Searching for $pname from file $flname"
grep "$pname"  $flname
echo "Selected rows shown above"
```

Output:

$sh emp1.sh

Enter the pattern to be searched:**director**

Enter the file to be used: **emp.lst**

Searching for **director** from file emp.lst

101 sharma|**director**|production|12/03/70|7000

102|barun|**director**|marketing|11/06/67|7800

selected rows shown above

COMMAND LINE ARGUMENTS

- When arguments are specified with a shell script they are assigned to certain special variables- positional parameters.
- $*→ store the complete set of positional parameters as a single string.
- $#--> It is set to the number of arguments specified.
- $0-→ holds the command name itself.
- When arguments are specified in this way the first word (the command itself) is assigned to $0, the second word(the first argument) to $1, the third word(the second argument) to $2.

```
#!/bin/sh
#emp2.sh
#
echo "Program:$0
The number of arguments specified is $#
The arguments are $*"
```

```
grep "$1" $2
echo "\n job over"
```

Output:
$ sh emp2.sh director emp.lst
Program: **emp2.sh**
The number of arguments specified is:**2**
The arguments are **director emp.lst**
101| sharma|**director**|production|12/03/70|7000
102|barun|**director**|marketing|11/06/67|7800
job over

SPECIAL PARAMETERS USED BY THE SHELL.

Shell Parameter	Significance
$1, $2...	Positional parameters representing command line arguments
$#	Number of arguments specified in command line
$0	Name of executed command
$*	Complete set of positional parameters as a single string
"$@"	Each quoted string treated as separate argument
$?	Exit status of last command
$$	PID of the current shell
$!	PID of the last background job

Exit and exit status of Command.

- The shells exit command
  exit 0 Used when everything went fine.
  exit 1 Used when something went wrong

Its through the exit command or function that every command returns an exit status to the caller. Further a command is said to return true exit status if it executes successfully and false if its fails.

- THE PARAMETER $? :It stores the exit status of the last command. It has the value 0 if the command succeeds and a non zero value if it fails. This parameter is set by exit's argument If no exit status is specified then $? is set to zero(true).

- Consider two files file1 which exist in current directory and file2 which does not exist

$ ls –l file1; echo $? **/*file1 attributes are listed**
Output :0 /*exit status $?=0, since cmd executed successfully

$ ls –l file2; echo $? **/*error since file2 doesnot exist**
Output: 1 /*exit status $?=1, since cmd execution failed.

THE LOGICAL OPERATORS && and || - CONDITIONAL EXECUTION

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

cmd1 && cmd2

cmd1 || cmd2

- Consider a file emp.lst

$cat emp.lst

1066| sharma | **director** |sales |03/09/66 | 7000

1098| Kumar |**director|** production|0/08/67 | 8200

1082|sumith| **manager**|marketing|09/09/73| 7090

- The && delimits two commands ; the command cmd2 is executed only when cmd1 succeeds.

$ grep "director" emp.lst && echo "Pattern found in file"

Output:

1066| sharma | **director** |sales |03/09/66 | 7000

1098| Kumar |**director**| production|0/08/67 | 8200

Pattern found in file

- The || operator plays inverse role. The second command is executed only when the first fails.

$grep " deputy manager" emp.lst || echo "Pattern not found"

Output:

Pattern not found /* cmd1 -**deputy manager** is not found in emp.lst.
Hence cmd1 fails. Therefore cmd2 "**pattern not found**" executes.

$grep "manager" emp.lst || echo "Pattern not found"

Output

1082|sumith| **manager**|marketing|09/09/73| 709 /*Here cmd1 is executed successfully i,e manager is found ,therefore cmd2 will not be executed.

CONDITIONAL STATEMENTS:

The if CONDITIONAL

If command is successful then execute commands else execute commands fi	If command is successful then execute commands fi	If command is successful then execute commands elif command is successful then .. else .. fi

The **if** statement makes two way decision making depending on the fulfillment of a certain condition.

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

```
#!/bin/sh
a=10
b=20
if [ $a==$b ]
then
echo "a is equal to b"
elif [ $a –gt  $b  ]
then
echo " a is greater than b"
elif [ $a -lt $b  ]
then
echo " a is lesser than b"
else
echo " None of the conditions met"
fi
```

output:
a is lesser than b

The case CONDITIONAL

- The case statement is the second conditional offered by the shell
- The statement matches an expression for more than one alternative and uses a compact construct to permit multiway branching.

The general syntax is

case *expression* in

pattern1) commands1 ;;

pattern2) commands2 ;;

pattern3) commands3 ;;

……..

esac

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

```
#!/bin/sh
#menu.sh
# echo " MENU \n
1.List of files\n 2.Processes of user\n 3.Todays date\n
4.Users of system\n 5.Quit\n
Enter your option: \"

read choice
case "$choice" in
1 ) ls –l ;;
2 ) ps –f ;;
3 ) date  ;;
4 ) who  ;;
5 ) exit ;;
* ) echo "invalid option"
esac
```

To run the program:

$ sh menu.sh

Output:

MENU

1. List of files

2. Processes of user

```
3. Todays date
4.Users of system
5. Quit
Enter your option : 3
Sun  Nov  6    18:03:06    IST    2016
```

Matching multiple patterns:

- case can also specify same action for more than one pattern.
- For example the expression y|Y can be used to match y in both upper and lower case letters.

```
echo "Do you wish to continue? : \c"
read answer
case "$answer" in
y|Y )         ;;
n|N ) exit ;;
esac
```

Wild cards: case uses them

- case has a string matching feature that uses wild cards.
- It uses the filename matching meta characters *, ? and the character class but only to match strings but not the files in the current directory.

```
case "$answer" in
[yY][eE] * )         ;;
[nN][oO]   )  exit  ;;
* ) echo "Invalid response"
esac
```

USING test command and its shortcut

USING test and [] to evaluate the expressions.

Test uses the certain operators to evaluate the condition on its right and returns either true or false exit status which is then used by if for making decision .

Test works in 3 ways:

Compares two numbers (NUMERIC COMPARISION)

compares two strings or a single one for a null value.(STRING COMPARISION)

checks a file attributes. (FILE TEST)

NUMERIC COMPARISION:

The numeric comparison operators used by test are

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 to

Numeric comparison in the shell is confined to integer values only , decimal values are simply truncated.

$ x=5; y=7; z=7.2
$ test $x –eq $y ; echo $?
Output : 1

$ test $x –lt $y ; echo $?
Output: 0

$ test $z –gt $y ; echo $?
Output: 1

STRING COMPARISION

test can be used to compare strings with yet another set of operators.

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

Example:

```
#!/bin/sh
a=”abc”
b=”efg”
if [ $a = $b ]
then
echo “a is equal to b”
else
echo “a is not equal to b”
fi
```

output:
a is not equal to b

FILE TESTS

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

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

$ls –l emp.lst
-rw-rw-rw- 1 kumar group 870 Sep 8 15:52 emp.lst

$ [-f emp.lst] ; echo $? // **-f** (emp.lst file exist and its regular file)
0 // Yes
$ [-x emp.lst] ; echo $? // **-x** (emp.lst file is executable or not)
1 //No

while Looping

- The while statement repeatedly performs a set of instructions until the control command return a true exit status .
- The general syntax is

  while *condition* is true

  do

  commands

  done

- The commands enclosed by do and done are executed repeatedly as long as condition remains true.

Using while to wait for a file

- There are situations when a program needs to read a file that is created by another program, but it has to wait until the file is created.
- The script, monitfile.sh periodically monitors the disk for the existence of the file. And then executes the program once the file has been located.
- It makes use of the external sleep command that makes the script pauses for the duration in seconds as specified in its arguments .
- The loop executes repeatedly as long as the file invoice.lst cannot be read.
- If the file becomes readable the loop is terminated and the program alloc.pl is executed.
- We use the sleep command to check every 60 seconds for the existence of the file.

Setting up an infinite loop

Suppose you as system administrator want to see the free space available on your disk every five minutes

```
while true
do
df –t
sleep 300
done &
```

df reports free space on disk . sleep command is used to hek for every 300 seconds(5 minutes).

& after done runs loop in background

---

for : LOOPING WITH A LIST

The shells for loop differs in structure from the ones used in other programming languages.

There is no three part structure.

```
for variables in list
do
commands
done
```

The loop body also uses the keyword do and done. But the additional parameters here are variable and list. Each whitespace separated word in list is assigned to variable and commands are executed until list is executed .

Ex:

```
$for file in chap20 chap21 chap22
 do
 cp $file  {$file}.bak
 echo $file copied to $file.bak
 done
```

Output:
chap20 copied to chap20.bak
chap21 copied to chap21.bak
chap22 copied to chap22.bak

set and shift: MANIPULATING THE POSITIONAL PARAMETERS

- set assigns its argument to positional parameters $1,$2 and so on.
  $set 989 878 779
  $_
  This assigns the value 989 to the positional parameter $1, 878 to the positional parameter $2 and 779 to $3

Ex:
$echo "\$1 is $1, \$2 is $2, \$3 is $3"
Output: $1 is 989, $2 is 878, $3 is 779
$echo "The $# arguments are $*"
Output: The 3 arguments are 989 878 779

Shift : Shifting Arguments left
Shift transfers the contents of a positional parameter to its immediate lower numbered one.
$ set `date'
$echo "$*
Output: Wed Nov 9 09:04:30 IST 2016
$shift
$ echo $1 $2 $3 $4 $5
Output: Nov 9 09:04:30 IST 2016

$shift 2
$echo $1 $2 $3
Output: 09:04:30 IST 2016

The HERE DOCUMENT (<<)

- The shell uses << symbols to read data from the same file containing the script.
- This is referred to as here document , signifying that the data is here rather than in a separate file .

- If the message is short you can have both the command and message in the same script.

```
mail sharma << MARK
Your program for printing the invoices has been executed
on `date`. The updated file is $flname
MARK
```

- The here document symbol(<<) followed by three lines of data and a delimiter (the string MARK)
- The shell treats every line following the command and delimited by MARK as input to the command.
- Sharma at the other end will see the three lines of message text with the date inserted by command substitution and the evaluated filename.

---

trap: INTERRUPTING A PROGRAM

- By default shell scripts terminate whenever the interrupt key is pressed. It may leave a lot of temporary files on disk .
- The trap statement lets you do things you want in case the script receives a signal. The statement is normally placed at the beginning of a shell script and uses two lists

trap 'command _list' signal_list

- When a script is sent any of the signals in signal_list, trap executes the commands in command_list
- The signal_list can contain the integer values or names of one or more signals.

trap **'rm $$* ; eho "Program Interrupted" ; exit '** HUP INT TERM

trap 'cmd_list' sig_list

- trap is a signal handler.
- Here it first removes all the files expanded from $$*, echoes a message and finally terminates the script when the signals SIGHUP(1), SIGINT(2), SIGTERM(15) are sent to the shell process running the script.
- When the interrupt key is pressed it sends the signal number 2.

MODULE 3
UNIX FILE APIs

General file API's

Files in a UNIX and POSIX system may be any one of the following types:

- Regular file
- Directory File
- FIFO file
- Block device file
- character device file
- Symbolic link file.

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

❖ **open**

✓ This is used to establish a connection between a process and a file i.e. it is used to open an existing file for data transfer function or else it may be also be used to create a new file.

✓ The returned value of the open system call is the file descriptor (row number of the file table), which contains the inode information.

✓ The prototype of open function is

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

✓ If successful, open returns a nonnegative integer representing the open file descriptor.

✓ If unsuccessful, open returns –1.

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

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

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

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

O_RDONLY	- open for reading file only
O_WRONLY	- open for writing file only
O_RDWR	- opens for reading and writing file.

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

✓ To illustrate the use of the above flags, the following example statement opens a file called /usr/divya/usp for read and write in append mode:
int fd=open("/usr/divya/usp",O_RDWR | O_APPEND,0);

✓ If the file is opened in read only, then no other modifier flags can be used.

✓ If a file is opened in write only or read write, then we are allowed to use any modifier flags along with them.

✓ The third argument is used only when a new file is being created. The symbolic names for file permission are given in the table in the previous page.

❖ **creat**

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

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

- Returns: file descriptor opened for write-only if OK, -1 on error.
- The first argument pathname specifies name of the file to be created.
- The second argument mode_t, specifies permission of a file to be accessed by owner group and others.
- The creat function can be implemented using open function as:
  - **#define creat(path_name, mode)**
  - **open (pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);**

❖ **read**

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

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

- If successful, read returns the number of bytes actually read.
- If unsuccessful, read returns –1.
- The first argument is an integer, fdesc that refers to an opened file.
- The second argument, buf is the address of a buffer holding any data read.
- The third argument specifies how many bytes of data are to be read from the file.
- The size_t data type is defined in the <sys/types.h> header and should be the same as unsigned int.
- There are several cases in which the number of bytes actually read is less than the amount requested:
  - When reading from a regular file, if the end of file is reached before the requested number of bytes has been read. For example, if 30 bytes remain until the end of file and we try to read 100 bytes, read returns 30. The next time we call read, it will return 0 (end of file).
  - When reading from a terminal device. Normally, up to one line is read at a time.
  - When reading from a network. Buffering within the network may cause less than the requested amount to be returned.
  - When reading from a pipe or FIFO. If the pipe contains fewer bytes than requested, read will return only what is available.

❖ **write**

- The write system call is used to write data into a file.
- The write function puts data to a file in the form of fixed block size referred by a given file descriptor.
- The prototype of write is

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

- If successful, write returns the number of bytes actually written.
- If unsuccessful, write returns –1.
- The first argument, fdesc is an integer that refers to an opened file.
- The second argument, buf is the address of a buffer that contains data to be written.
- The third argument, size specifies how many bytes of data are in the buf argument.
- The return value is usually equal to the number of bytes of data successfully written to a file. (*size* value)

❖ **close**

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

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

- If successful, close returns 0.
- If unsuccessful, close returns –1.
- The argument fdesc refers to an opened file.
- Close function frees the unused file descriptors so that they can be reused to reference other files. This is important because a process may open up to OPEN_MAX files at any time and the close function allows a process to reuse file descriptors to access more than OPEN_MAX files in the course of its execution.
- The close function de-allocates system resources like file table entry and memory buffer allocated to hold the read/write.

❖ **fcntl**

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

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

- The first argument is the file descriptor.
- The second argument cmd specifies what operation has to be performed.
- The third argument is dependent on the actual cmd value.
- The possible cmd values are defined in <fcntl.h> header.

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

- The fcntl function is useful in changing the access control flag of a file descriptor.
- For example: after a file is opened for blocking read-write access and the process needs to change the access to non-blocking and in write-append mode, it can call:

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

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

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

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

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

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

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

❖ **lseek**

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

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

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

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

- They are defined in the <unistd.h> header.
- If an lseek call will result in a new file offset that is beyond the current end-of-file, two outcomes possible are:
  - If a file is opened for read-only, lseek will fail.
  - If a file is opened for write access, lseek will succeed.
  - The data between the end-of-file and the new file offset address will be initialised with NULL characters.

❖ **link**

- The link function creates a new link for the existing file.
- The prototype of the link function is

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

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

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

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

❖ **unlink**

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

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

- If successful, the unlink function returns 0.
- If unsuccessful, unlink returns –1.
- The argument cur_link is a path name that references an existing file.
- ANSI C defines the rename function which does the similar unlink operation.
- The prototype of the rename function is:

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

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

```
#include <iostream.h>
#include <unistd.h>
#include<string.h>
int main ( int argc, char *argv[ ])
{
        if (argc != 3 || strcmp(argv[1],argcv[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**

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

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

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

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

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

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

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

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

Note: refer UNIX lab program 3(b) for example

❖ **access**

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

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

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

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

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

- example to check whether a file exists:

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

❖ **chmod, fchmod**

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

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

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

- The prototypes of these functions are

- The pathname argument of chmod is the path name of a file whereas the fdesc argument of fchmod is the file descriptor of a file.
- The chmod function operates on the specified file, whereas the fchmod function operates on a

file that has already been opened.

- To change the permission bits of a file, the effective user ID of the process must be equal to the owner ID of the file, or the process must have super-user permissions. The mode is specified as the bitwise OR of the constants shown below.

❖ **chown, fchown, lchown**

- The chown functions changes the user ID and group ID of files.
- The prototypes of these functions are

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

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

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

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

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

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

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

```
else
        perror
        ("stat");
```

- The above program takes at least two command line arguments:
  - The first one is the user name to be assigned to files
  - The second and any subsequent arguments are file path names.
- The program first converts a given user name to a user ID via *getpwuid* function. If that succeeds, the program processes each named file as follows: it calls *stat* to get the file group ID, then it calls *chown* to change the file user ID. If either the stat or chown fails, error is displayed.

❖ utime Function

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

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

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

- On success it returns 0, on failure it returns –1.
- The path_name argument specifies the path name of a file.
- The times argument specifies the new access time and modification time for the file.
- The struct utimbuf is defined in the <utime.h> header as:

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

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

File and Record Locking

- Multiple processes performs read and write operation on the same file concurrently.
- This provides a means for data sharing among processes, but it also renders difficulty for any process in determining when the other process can override data in a file.

- So, in order to overcome this drawback UNIX and POSIX standard support file locking mechanism.
- File locking is applicable for regular files.
- Only a process can impose a write lock or read lock on either a portion of a file or on the entire file.
- The differences between the read lock and the write lock is that when write lock is set, it prevents the other process from setting any over-lapping read or write lock on the locked file.
- Similarly when a read lock is set, it prevents other processes from setting any overlapping write locks on the locked region.
- The intension of the write lock is to prevent other processes from both reading and writing the locked region while the process that sets the lock is modifying the region, so write lock is termed as "**Exclusive lock**".
- The use of read lock is to prevent other processes from writing to the locked region while the process that sets the lock is reading data from the region.
- Other processes are allowed to lock and read data from the locked regions. Hence a read lock is also called as "**shared lock** ".
- File lock may be **mandatory** if they are enforced by an operating system kernel.
- If a mandatory exclusive lock is set on a file, no process can use the read or write system calls to access the data on the locked region.
- These mechanisms can be used to synchronize reading and writing of shared files by multiple processes.
- If a process locks up a file, other processes that attempt to write to the locked regions are blocked until the former process releases its lock.
- Problem with mandatory lock is – if a runaway process sets a mandatory exclusive lock on a file and never unlocks it, then, no other process can access the locked region of the file until the runway process is killed or the system has to be rebooted.
- If locks are not mandatory, then it has to be **advisory** lock.
- A kernel at the system call level does not enforce advisory locks.
- This means that even though a lock may be set on a file, no other processes can still use the read and write functions to access the file.
- To make use of advisory locks, process that manipulate the same file must co-operate such that they follow the given below procedure for every read or write operation to the file.
    1. Try to set a lock at the region to be accesses. If this fails, a process can either wait for the lock request to become successful.
    2. After a lock is acquired successfully, read or write the locked region.
    3. Release the lock.
- If a process sets a read lock on a file, for example from address 0 to 256, then sets a write lock on the file from address 0 to 512, the process will own only one write lock on the file from 0 to 512, the previous read lock from 0 to 256 is now covered by the write lock and the process does not own two locks on the region from 0 to 256. This process is called "**Lock Promotion**".
- Furthermore, if a process now unblocks the file from 128 to 480, it will own two write locks on the file: one from 0 to 127 and the other from 481 to 512. This process is called "**Lock Splitting**".
- UNIX systems provide fcntl function to support file locking. By using fcntl it is possible to impose read or write locks on either a region or an entire file.
- The prototype of fcntl is

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

- The first argument specifies the file descriptor.
- The second argument cmd_flag specifies what operation has to be performed.
- If fcntl is used for file locking then it can values as
- For file locking purpose, the third argument to fctnl is an address of a *struct flock* type variable.
- This variable specifies a region of a file where lock is to be set, unset or queried.

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

- The l_type field specifies the lock type to be set or unset.
- The possible values, which are defined in the <fcntl.h> header, and their uses are
- The l_whence, l_start & l_len define a region of a file to be locked or unlocked.
- The possible values of l_whence and their uses are
- A lock set by the fcntl API is an advisory lock but we can also use fcntl for mandatory locking purpose with the following attributes set before using fcntl
    1. Turn on the set-GID flag of the file.
    2. Turn off the group execute right permission of the file.
- In the given example program we have performed a read lock on a file “divya” from the 10th byte to 25th byte.

Example Program

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

Directory File API's

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

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

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

- The first argument is the path name of a directory file to be created.
- The second argument mode, specifies the access permission for the owner, groups and others to be assigned to the file. This function creates a new empty directory.
- The entries for "." and ".." are automatically created. The specified file access permission, mode, are modified by the file mode creation mask of the process.
- To allow a process to scan directories in a file system independent manner, a directory record is defined as
  struct dirent in the <dirent.h> header for UNIX.
- Some of the functions that are defined for directory file operations in the above header are

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

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

The uses of these functions are

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

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

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

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

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

The following list_dir.C program illustrates the uses of the mkdir, opendir, readdir, closedir and rmdir APIs:

```
#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 dirent Dirent;
#else
        #include<dirent.h>
        typedef struct dirent Dirent;
#endif
int main(int agc, char* argv[])
{
Dirent* dp;
DIR*
dir_fdesc;
while(--argc>0)
{
if(!(dir_fdesc=opendir(*++argv)))
{
if(mkdir(*argv,S_IRWXU | S_IRWXG |
S_IRWXO)==-1) perror("opendir");
continue;
}
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++;
    }
    if(!cnt)
    {
    rmdir(*argv)
    ; break;
    }
    rewinddir(dir_fdesc);
    }
    closedir(dir_fdesc);
    }
    }
```

Device file APIs

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

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

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

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

  mknod("SCSI5",S_IFBLK | S_IRWXU | S_IRWXG | S_IRWXO,(15<<8) | 3);
- The above function creates a block device file "divya", to which all the three i.e. read, write and execute permission is granted for user, group and others with major number as 8 and minor number 3.
- On success mknod API returns 0 , else it returns -1

The following test_mknod.C program illustrates the use of the mknod, open, read, write and close APIs on a block device file.

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

int main(int argc, char* argv[])
{
        if(argc!=4)
        {
                cout<<"usage:"<<argv[0]<<"<file><major_no><minor_
                no>"; return 0;
        }
        int major=atoi(argv[2],minor=atoi(argv[3]);
        (void) mknod(argv[1], S_IFCHR | S_IRWXU | S_IRWXG | S_IRWXO, (major<<8) | minor);

        int rc=1,fd=open(argv[1],O_RDW | 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<<end
        l; close(fd);
}
```

FIFO file API's

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

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

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

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

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

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

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

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

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

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

int main(int argc,char* argv[])
{
        if(argc!=2 && argc!=3)
        {
                cout<<"usage:"<<argv[0]<<"<file>
                [<arg>]"; return 0;
        }
        int fd;
```

```
        char buf[256];
        (void) mkfifo(argv[1], S_IFIFO | S_IRWXU | S_IRWXG |
        S_IRWXO ); if(argc==2)
        {
                fd=open(argv[1],O_RDONLY | O_NONBLOCK);
                while(read(fd,buf,sizeof(buf))==-1 &&
                errno==EAGAIN)
                        sleep(1);
                while(read(fd,buf,sizeof(buf))>0)
        }               cout<<buf<<endl;
        else
        {

                fd=open(argv[1],O_WRONLY);
        }       write(fd,argv[2],strlen(argv[2]));

        close(fd);
}
```

Symbolic Link File API's

- A symbolic link is an indirect pointer to a file, unlike the hard links which pointed directly to the inode of the file.
- Symbolic links are developed to get around the limitations of hard links:
  - 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
  - There are no file system limitations on a symbolic link and what it points to and anyone can create a symbolic link to a directory.
  - Symbolic links are typically used to move a file or an entire directory hierarchy to some other location on a system.
  - A symbolic link is created with the symlink.
  - The prototype is

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

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

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

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

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

UNIX PROCESSES

INTRODUCTION

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

mainFUNCTION

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

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

where argc is the number of command-line arguments, and argv is an array of pointers to the arguments.

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

PROCESS TERMINATION

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

- Return from main
- Calling exit
- Calling _exitor _Exit
- Return of the last thread from its start routine
- Calling pthread_exit from the last thread

Abnormal termination occurs in three ways:

- Calling abort
- Receipt of a signal
- Response of the last thread to a cancellation request

Exit Functions

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

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

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

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

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

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

undefined. Example:

atexitFunction

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 atexitfunction.

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

Returns: 0 if OK, nonzero on error

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

Example of exit handlers

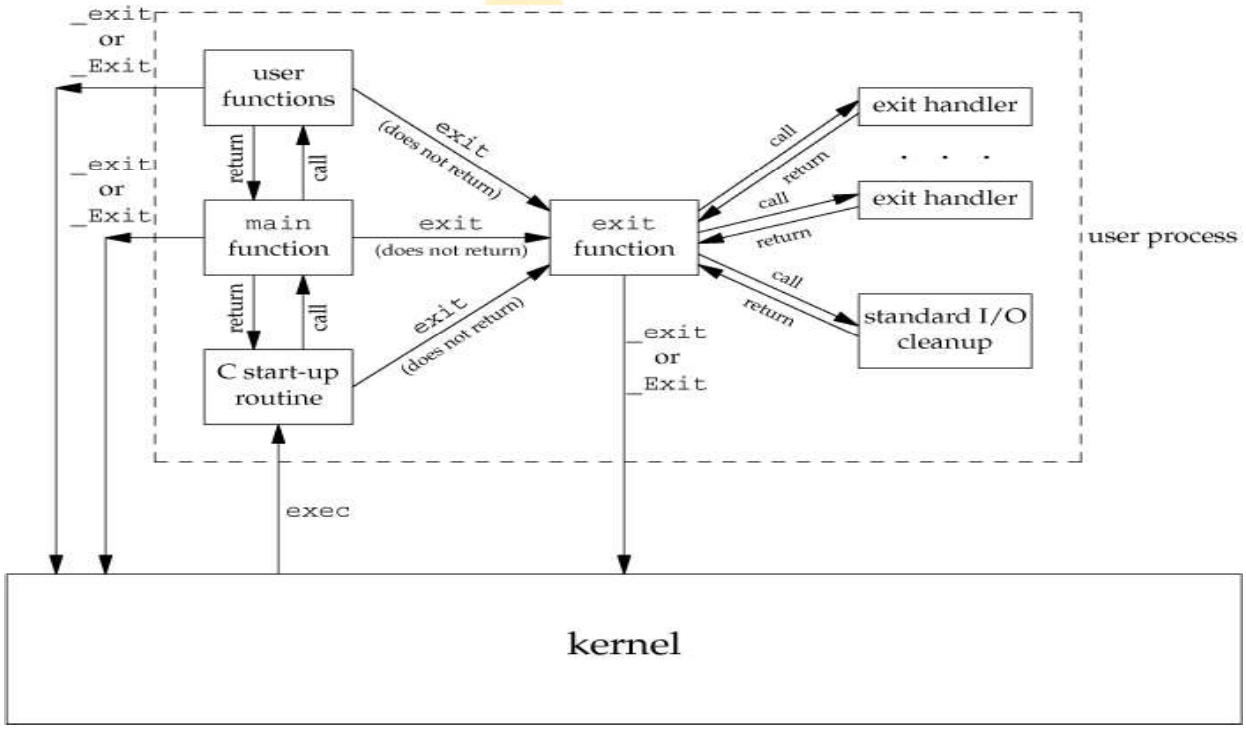
```
#include "apue.h"

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

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

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

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

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

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

static void
my_exit2(void)
```

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

Output:

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

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



COMMAND-LINE ARGUMENTS

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

Example: Echo all command-line arguments to standard output

```
#include "apue.h"

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

    for (i = 0; i < argc;           i++)        /* echo all command-line args */
```

```
            printf("argv[%d]: %s\n", i, argv[i]);
        exit(0);
}
```

Output:

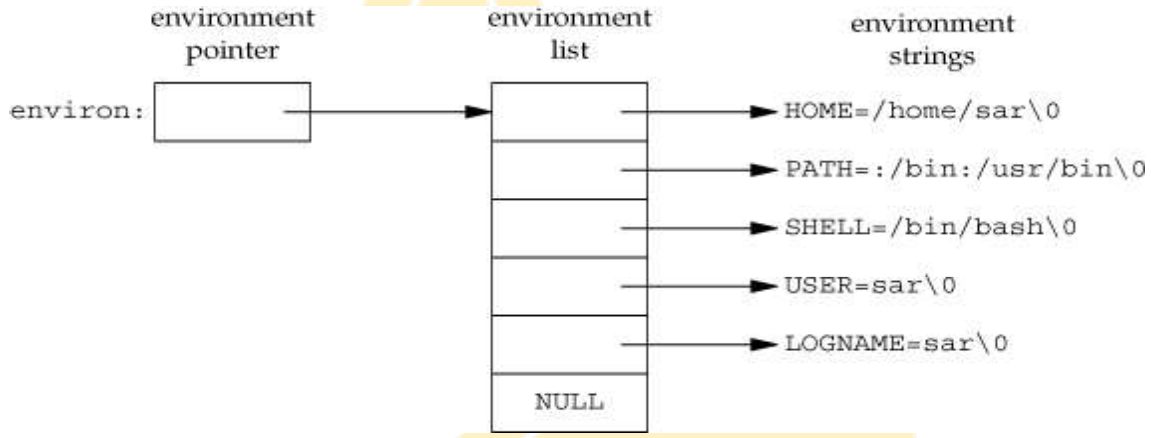
```
$ ./echoarg arg1 TEST foo
      argv[0]: ./echoarg
      argv[1]: arg1 argv[2]:
      TEST argv[3]: foo
```

ENVIRONMENT LIST

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

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

Figure: Environment consisting of five C character string



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

MEMORY LAYOUT OF A C PROGRAM

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

- **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

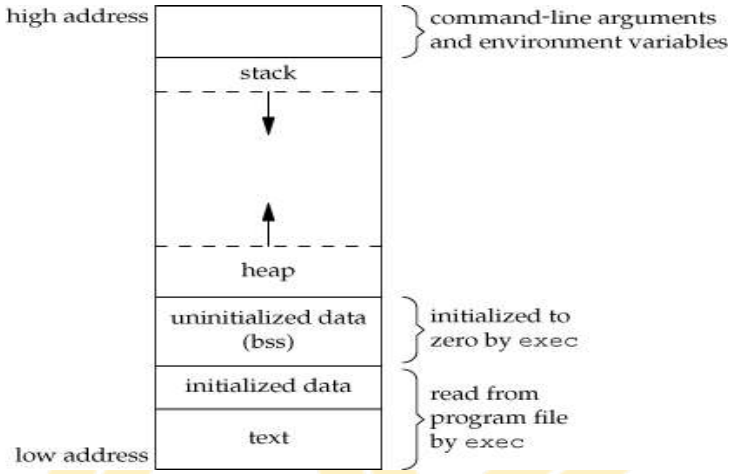
  ```
  int     maxcount = 99;
  ```

  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.



SHARED LIBRARIES

Nowadays most UNIX systems support shared libraries. Shared libraries remove the common library routines from the executable file, instead maintaining a single copy of the library routine somewhere in memory that all processes reference. This reduces the size of each executable file but may add some runtime overhead, either when the program is first executed or the first time each shared library function is called. Another advantage of shared libraries is that, library functions can be replaced with new versions without having to re-link, edit every program that uses the library. With cc compiler we can use the option –g to indicate that we are using shared library.

MEMORY ALLOCATION

ISO C specifies three functions for memory allocation:

- malloc, which allocates a specified number of bytes of memory. The initial value of the memory is indeterminate.
- calloc, which allocates space for a specified number of objects of a specified size. The space is initialized to all 0 bits.
- 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

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

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.
All three return: non-null pointer if OK, NULLon error

void free(void *ptr);

The pointer returned by the three allocation functions is guaranteed to be suitably aligned so that it can be used for any data object. Because the three alloc functions return a generic void * pointer, if we #include <stdlib.h> (to obtain the function prototypes), we do not explicitly have to cast the pointer returned by these functions when we assign it to a pointer of a different type.

The function free causes the space pointed to by ptr to be deallocated. This freed space is usually put into a pool of available memory and can be allocated in a later call to one of the three allocfunctions.

The reallocfunction lets us increase or decrease the size of a previously allocated area. For example, if we allocate room for 512 elements in an array that we fill in at runtime but find that we need room for more than 512 elements, we can call realloc. If there is room beyond the end of the existing region for the requested space, then realloc doesn't have to move anything; it simply allocates the additional area at the end and returns the same pointer that we passed it. But if there isn't room at the end of the existing region, reallocallocates another area that is large enough, copies the existing 512-element array to the new area, frees the old area, and returns the pointer to the new area.

The allocation routines are usually implemented with the sbrk(2) system call. Although sbrk can expand or contract the memory of a process, most versions of malloc and freenever 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 mallocpool.

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

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

Alternate Memory Allocators

Many replacements for mallocand freeare available.

- **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 libmalloclibrary 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.

- **vmalloc**

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

emulations of the ISO C memory allocation functions.

- **quick-fit**

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

- **allocaFunction**

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

ENVIRONMENT VARIABLES

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

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

can use to set and fetch values from the variables are setenv, putenv, and getenv functions. The prototype of these functions are

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

Note that this function returns a pointer to the value of a **name=value** string. We should always use getenv to fetch a specific value from the environment, instead of accessing environ directly. In addition to fetching the value of an environment variable, sometimes we may want to set an environment variable. We may want to

```
#include <stdlib.h>
int putenv(char *str);
int setenv(const char *name, const char *value, int rewrite);
int unsetenv(const char *name);
```

change the value of an existing variable or add a new variable to the environment. The prototypes of these functions are

All return: 0 if OK, nonzero on error.

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

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

environment.

Environment variables defined in the Single UNIX Specification

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

NOTE:

- If we're modifying an existing name:
  - If the size of the new value is less than or equal to the size of the existing value, we can just copy the new string over the old string.
  - If the size of the new value is larger than the old one, however, we must malloc to obtain room for the new string, copy the new string to this area, and then replace the old pointer in the environment list for name with the pointer to this allocated area.
- If we're adding a new name, it's more complicated. First, we have to call malloc to allocate room for the name=value string and copy the string to this area.
  - Then, if it's the first time we've added a new name, we have to call mallocto obtain room for a new list of pointers. We copy the old environment list to this new area and store a pointer to the name=value string at the end of this list of pointers. We also store a null pointer at the end of this list, of course. Finally, we set environto point to this new list of pointers.
  - If this isn't the first time we've added new strings to the environment list, then we know that we've already allocated room for the list on the heap, so we just call realloc to allocate room for one more pointer. The pointer to the new name=value string is stored at the end of the list (on top of the previous null pointer), followed by a null pointer.

setjmpAND longjmpFUNCTIONS

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

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

that occur in a deeply nested function call.

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

void longjmp(jmp_buf env, int val);

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

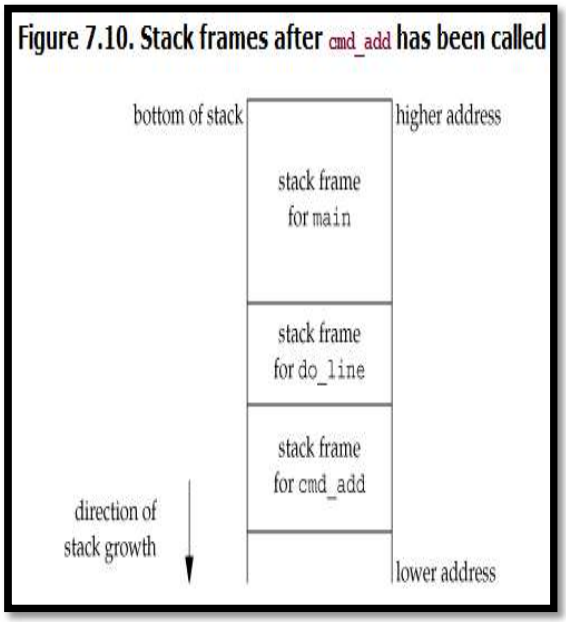
Example of setjmpand longjmp

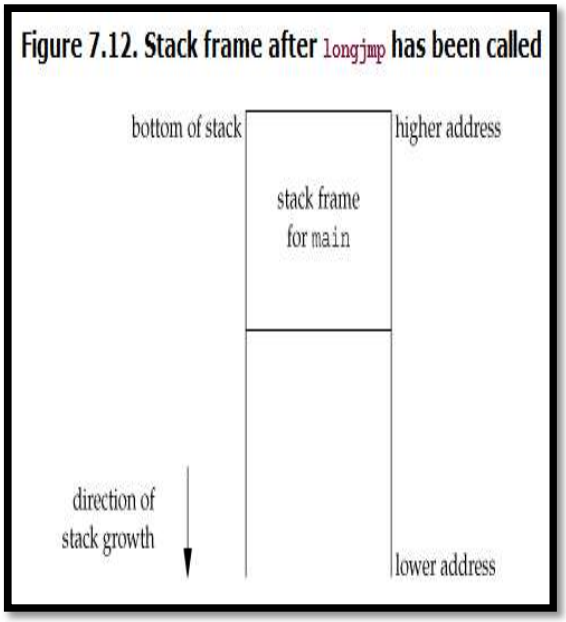
```
#include "apue.h"
#include <setjmp.h>

#define TOK_ADD    5
jmp_buf jmpbuffer;

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

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

```
            do_line(line);
        exit(0);
}

  ...

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

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

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

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





getrlimitAND setrlimitFUNCTIONS

Every process has a set of resource limits, some of which can be queried and changed by the geTRlimit and setrlimitfunctions.

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

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

Both return: 0 if OK, nonzero on error

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

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

Three rules govern the changing of the resource limits.

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

An infinite limit is specified by the constant RLIM_INFINITY.

RLIMIT_AS	The maximum size in bytes of a process's total available memory.
RLIMIT_CORE	The maximum size in bytes of a core file. A limit of 0 prevents the creation of a core file.

RLIMIT_CPU	The maximum amount of CPU time in seconds. When the soft limit is exceeded, the SIGXCPU signal is sent to the process.
RLIMIT_DATA	The maximum size in bytes of the data segment: the sum of the initialized data, uninitialized data, and heap.
RLIMIT_FSIZE	The maximum size in bytes of a file that may be created. When the soft limit is exceeded, the process is sent the SIGXFSZsignal.
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(2).
RLIMIT_NOFILE	The maximum number of open files per process. Changing this limit affects the value returned by the sysconffunction for its _SC_OPEN_MAXargument
RLIMIT_NPROC	The maximum number of child processes per real user ID. Changing this limit affects the value returned for _SC_CHILD_MAXby the sysconffunction
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: Print the current resource limits

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

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

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

int main(void)
{

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

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

      if (getrlimit(resource, &limit) < 0)
            err_sys("getrlimit error for %s", name);
      printf("%-14s ", name);
      if (limit.rlim_cur ==
```

```
            RLIM_INFINITY)
            printf("(infinite) ");
        else
            printf(FMT, limit.rlim_cur);
        if (limit.rlim_max ==
            RLIM_INFINITY)
            printf("(infinite)");
        else
            printf(FMT, limit.rlim_max);
        putchar((int)'\n');
}
```

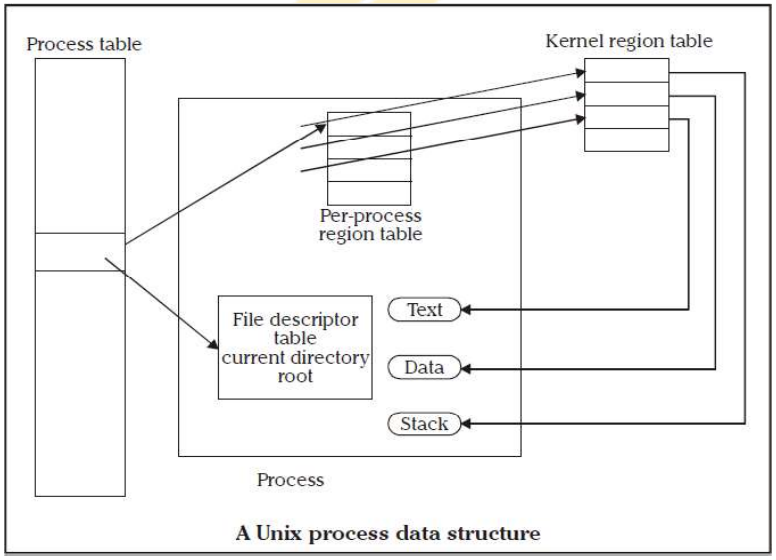
UNIX KERNEL SUPPORT FOR PROCESS

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

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

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

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



A Unix process data structure

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

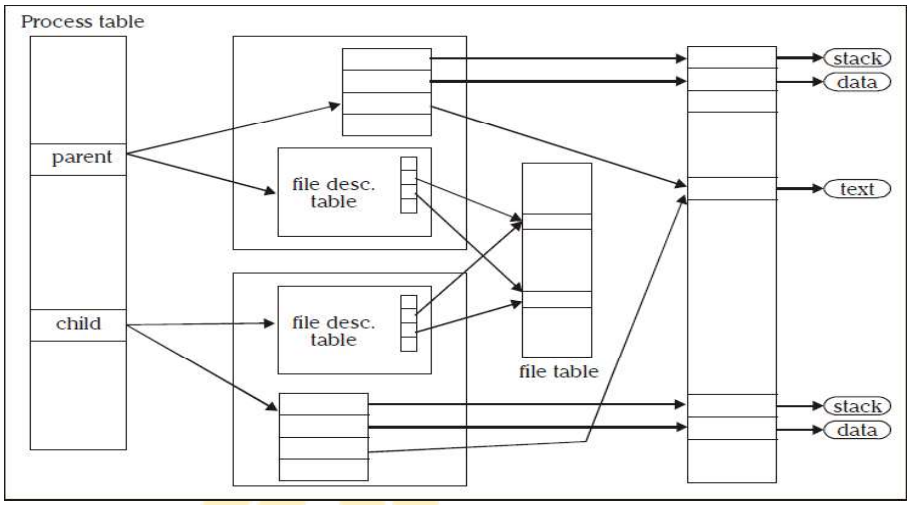


Figure: Parent & child relationship after fork

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

- **A real user identification number (rUID)**: the user ID of a user who created the parent process.
- **A real group identification number (rGID)**: the group ID of a user who created that parent process.
- **An effective user identification number (eUID)**: this allows the process to access and create files with the same privileges as the program file owner.
- **An effective group identification number (eGID)**: this allows the process to access and create files with the same privileges as the group to which the program file belongs.
- **Saved set-UID and saved set-GID**: these are the assigned eUID and eGID of the process respectively.
- **Process group identification number (PGID) and session identification number (SID):** these identify the process group and session of which the process is member.
- **Supplementary group identification numbers:** this is a set of additional group IDs for a user who created the process.

- **Current directory:** this is the reference (inode number) to a working directory file.
- **Root directory**: this is the reference to a root directory.
- **Signal handling**: the signal handling settings.
- **Signal mask**: a signal mask that specifies which signals are to be blocked.
- **Unmask**: a file mode mask that is used in creation of files to specify which accession rights should be taken out.
- **Nice value**: the process scheduling priority value.
- **Controlling terminal**: the controlling terminal of the process.

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

- **Process identification number (PID)**: an integer identification number that is unique per process in an entire operating system.
- **Parent process identification number (PPID)**: the parent process PID.
- **Pending signals**: the set of signals that are pending delivery to the parent process.
- **Alarm clock time**: the process alarm clock time is reset to zero in the child process.
- **File locks**: the set of file locks owned by the parent process is not inherited by the chid process.

fork and ***exec*** are commonly used together to spawn a sub-process to execute a different program. The advantages of this method are:

- A process can create multiple processes to execute multiple programs concurrently.
- Because each child process executes in its own virtual address space, the parent process is not affected by the execution status of its child process.

PROCESS CONTROL

INTRODUCTION

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

PROCESS IDENTIFIERS

#include <unistd.h>

pid_t getpid(void);

Returns: process ID of calling process

pid_t getppid(void);

Returns: parent process ID of calling process

uid_t getuid(void);

Returns: real user ID of calling process

uid_t geteuid(void);

Returns: effective user ID of calling process

gid_t getgid(void);

Returns: real group ID of calling process

gid_t getegid(void);

Returns: effective group ID of calling process

fork FUNCTION

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

```
#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 forkis called the child process.
- This function is called once but returns twice.
- The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child.
- The reason the child's process ID is returned to the parent is that a process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children.
- The reason fork returns 0 to the child is that a process can have only a single parent, and the child can always call getppid to obtain the process ID of its parent. (Process ID 0 is reserved for use by the kernel, so it's not possible for 0 to be the process ID of a child.)

- Both the child and the parent continue executing with the instruction that follows the call to fork.
- The child is a copy of the parent.
- For example, the child gets a copy of the parent's data space, heap, and stack.
- Note that this is a copy for the child; the parent and the child do not share these portions of memory.
- The parent and the child share the text segment .

Example programs:

Program 1

/* Program to demonstrate fork function Program name – fork1.c */

```
#include<sys/types.h>
#include<unistd.h>
int main( )
{
        fork( );
        printf("\n hello USP");
}
```

Output :

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

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

Program 2

/* Program name –
fork2.c */

```
#include<sys/types.h>
#include<unistd.h>
int main( )
{
        printf("\n 6 sem ");
        fork( );
        printf("\n hello USP");
}
```

Output :

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

6 sem
hello USP
hello USP
Note: The statement 6 sem is executed only once by the parent because it is called before fork and statement hello USP is executed twice by child and parent. [*Also refer lab program 3.sh*]

File Sharing

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

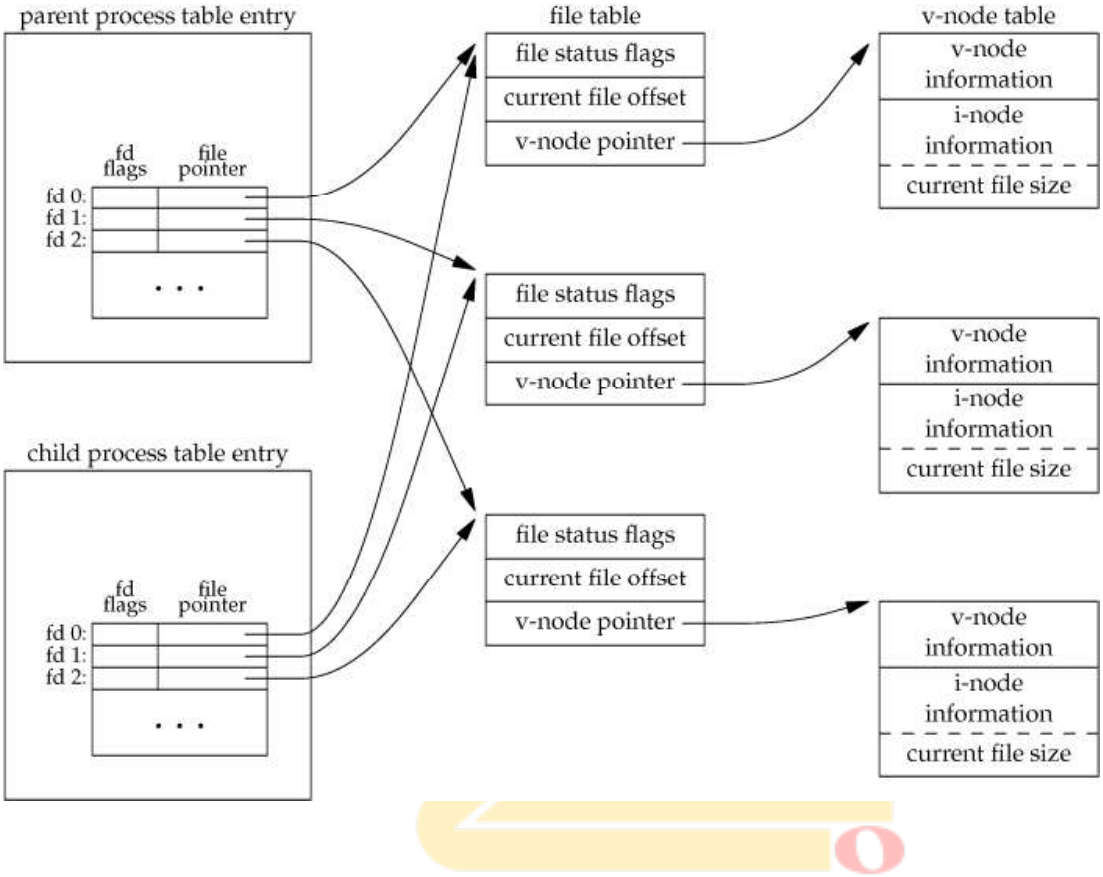


Figure 8.2 Sharing of open files between parent and child after fork

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

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

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

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

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

The differences between the parent and child

- are The return value from fork
- The process IDs are different
- 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
- The child's tms_utime, tms_stime, tms_cutime, and tms_cstimevalues are set to 0
- File locks set by the parent are not inherited by the child
- Pending alarms are cleared for the child
- The set of pending signals for the child is set to the empty set

The two main reasons for forkto fail are

(a) if too many processes are already in the system, which usually means that something else is wrong, or

(b) if the total number of processes for this real user ID exceeds the system's limit.

There are two uses for fork:

- ❖ When a process wants to duplicate itself so that the parent and child can each execute different sections of code at the same time. This is common for network servers, the parent waits for a service request

from a client. When the request arrives, the parent calls fork and lets the child handle the request. The

- ❖ parent goes back to waiting for the next service request to arrive.
- ❖ When a process wants to execute a different program. This is common for shells. In this case, the child does an execright after it returns from the fork.

vfork FUNCTION

- ✓ The function vforkhas the same calling sequence and same return values as fork.
- ✓ The vfork function is intended to create a new process when the purpose of the new process is to exec a new program.
- ✓ The vfork function creates the new process, just like fork, without copying the address space of the parent into the child, as the child won't reference that address space; the child simply calls exec (or exit) right after the vfork.
- ✓ Instead, while the child is running and until it calls either exec or exit, the child runs in the address space of the parent. This optimization provides an efficiency gain on some paged virtual-memory implementations of the UNIX System.
- ✓ Another difference between the two functions is that vfork guarantees that the child runs first, until the child calls execor exit. When the child calls either of these functions, the parent resumes.

Example of vforkfunction

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

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

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

Output:

$ ./a.out

before vfork
pid = 29039, glob = 7, var = 89

exit FUNCTIONS

A process can terminate normally in five ways:

- Executing a return from the main function.
- Calling the exit function.
- Calling the _exit or _Exit function.

  In most UNIX system implementations, exit(3) is a function in the standard C library, whereas _exit(2) is a system call.
- Executing a return from the start routine of the last thread in the process. When the last thread returns from its start routine, the process exits with a termination status of 0.
- Calling the pthread_exit function from the last thread in the

process. The three forms of abnormal termination are as follows:

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

wait AND waitpid FUNCTIONS

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

A process that calls waitor waitpidcan:

- ✓ 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.

```
#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 waitfunction can block the caller until a child process terminates, whereas waitpidhas an option

that prevents it from blocking.

- The waitpidfunction doesn't wait for the child that terminates first; it has a number of options that control which process it waits for.

If a child has already terminated and is a zombie, wait returns immediately with that child's status. Otherwise,

it blocks the caller until a child terminates. If the caller blocks and has multiple children, wait returns when one terminates.

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

Print a description of the exitstatus

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

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

```
        else if (WIFSTOPPED(status))
            printf("child stopped, signal number = %d\n",
                    WSTOPSIG(status));
}
```

Program to Demonstrate various exitstatuses

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

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

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

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



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

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

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

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

    exit(0);
}
```

The interpretation of the pid argument for waitpiddepends 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.

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

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.

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 waitpidfunction provides three features that aren't provided by the waitfunction.

- ✓ 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 popenfunction.
- ✓ 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.
- ✓ The waitpidfunction provides support for job control with the WUNTRACEDand WCONTINUED options.

Program to Avoid zombie processes by calling forktwice

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

Int main(void)
{

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

    /*
     * We're the parent (the original process); we continue executing,
```

```
         * knowing that we're not the parent of the second child.
         */
        exit(0);
}
```

Output:

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

waitidFUNCTION

The waitidfunction is similar to waitpid, but provides extra flexibility.

```
#include <sys/wait.h>
Int waited(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

Returns: 0 if OK, -1 on error

The *idtype* constants for waited are as follows:

Constant	Description
P_PID	Wait for a particular process: id contains the process ID of the child to wait for.
P_PGID	Wait for any child process in a particular process group: id contains the process group ID of the children to wait for.
P_ALL	Wait for any child process: id is ignored.

The options argument is a bitwise OR of the flags as shown below: these flags indicate which state changes the caller is interested in.

Constant	Description
WCONTINUED	Wait for a process that has previously stopped and has been continued, and whose status has not yet been reported.
WEXITED	Wait for processes that have exited.
WNOHANG	Return immediately instead of blocking if there is no child exit status available.
WNOWAIT	Don't destroy the child exit status. The child's exit status can be retrieved by a subsequent call to wait, waitid,or waitpid.
WSTOPPED	Wait for a process that has stopped and whose status has not yet been reported.

wait3AND wait4FUNCTIONS

The only feature provided by these two functions that isn't provided by the wait, waitid, and waitpid functions

is an additional argument that allows the kernel to return a summary of the resources used by the terminated process and all its child processes.

The prototypes of these functions are:

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>
pid_t wait3(int *statloc, int options, struct rusage *rusage);
pid_t wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);
```

Both return: process ID if OK,-1 on error

The resource information includes such statistics as the amount of user CPU time, the amount of system CPU time, number of page faults, number of signals received etc. the resource information is available only for terminated child process not for the process that were stopped due to job control.

RACE CONDITIONS

A race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run.

Example: The program below outputs two strings: one from the child and one from the parent. The program contains a race condition because the output depends on the order in which the processes are run by the kernel and for how long each process runs.

```
#include "apue.h"
static void charatatime(char *);
int main(void)
{
    pid_t    pid;
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } 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);
}
```

Output:

```
$ ./a.out
ooutput from child
utput from parent
$ ./a.out
ooutput from child
utput from parent
$ ./a.out
output from child
output from parent
```

program modification to avoid race condition

```
#include "apue.h"
    static void charatatime(char *);
    int main(void)
    {
        pid_t    pid;

+       TELL_WAIT();
+
        if ((pid = fork()) < 0) {
            err_sys("fork error");
        } else if (pid == 0) {
+           WAIT_PARENT();  /* parent goes first */
            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);
    }
```

When we run this program, the output is as we expect; there is no intermixing of output from the two processes.

execFUNCTIONS

When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function. The process ID does not change across an exec, because a new process is not created; exec merely replaces the current process - its text, data, heap, and stack segments - with a brand new program from disk.

There are 6 exec functions:

```
#include <unistd.h>
int execl(const char *pathname, const char *arg0,... /* (char *)0 */ );
int execv(const char *pathname, char *const argv []);
int 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 next difference concerns the passing of the argument list (l stands for list and v stands for vector). The functions execl, execlp, and execlerequire each of the command-line arguments to the new program to be specified as separate arguments. For the other three functions (execv, execvp, and execve), we have to build an array of pointers to the arguments, and the address of this array is the argument to these three functions.
- 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.

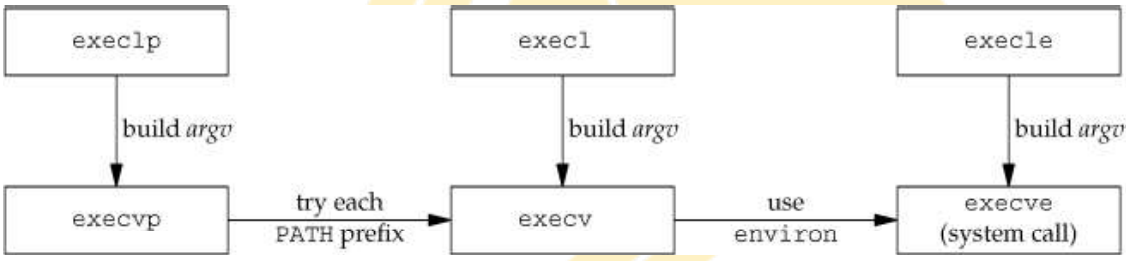
Function	pathname	filename	Arg list	argv[]	environ	envp[]
execl	•		•		•	
execlp		•	•		•	
execle	•		•			•
execv	•			•	•	
execvp		•		•	•	
execve	•			•		•
(letter in name)		p	l	v		e

The above table shows the differences among the 6 exec functions.

We've mentioned that the process ID does not change after an exec, but the new program inherits additional properties from the calling process:

- Process ID and parent process ID
- Real user ID and real group ID
- Supplementary group IDs
- Process group ID
- Session ID
- Controlling terminal
- Time left until alarm clock
- Current working directory
- Root directory
- File mode creation mask
- File locks
- Process signal mask
- Pending signals

- Resource limits
- Values for tms_utime, tms_stime, tms_cutime, and tms_cstime.



Relationship of the six exec functions

Example of exec functions

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

char    *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int main(void)
{
    pid_t    pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {          /* specify pathname, specify environment */
        if (execle("/home/sar/bin/echoall", "echoall", "myarg1",
```

```
                "MY ARG2", (char *)0, env_init) <
            0) err_sys("execle error");
    }

    if (waitpid(pid, NULL, 0) < 0)
        err_sys("wait error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {          /* specify filename, inherit environment */ if
        (execlp("echoall", "echoall", "only 1 arg", (char *)0) <0)
            err_sys("execlp error");
    }

    exit(0);
}
```

Output:

```
$ ./a.out

  argv[0]: echoall
  argv[1]: myarg1
  argv[2]: MY
  ARG2
  USER=unknown
  PATH=/tmp
  $ argv[0]: echoall
  argv[1]: only 1 arg
  USER=sar
  LOGNAME=sar
  SHELL=/bin/bash
                                  47 more lines that aren't shown
  HOME=/home/sar
```

Note that the shell prompt appeared before the printing of argv[0] from the second exec. This is because the parent did not wait for this child process to finish.

Echo all command-line arguments and all environment strings

```
#include "apue.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);
}
```

MODULE 4

CHANGING USER IDs AND GROUP IDs

When our programs need additional privileges or need to gain access to resources that they currently aren't allowed to access, they need to change their user or group ID to an ID that has the appropriate privilege or access. Similarly, when our programs need to lower their privileges or prevent access to certain resources, they do so by changing either their user ID or group ID to an ID without the privilege or ability access to the resource.

```
#include <unistd.h>

int setuid(uid_t uid);
int setgid(gid_t gid);
```

Both return: 0 if OK, 1 on error

There are rules for who can change the IDs. Let's consider only the user ID for now. (Everything we describe for the user ID also applies to the group ID.)

- If the process has superuser privileges, the setuid function sets the real user ID, effective user ID, and saved set-user-ID to uid.
- If the process does not have superuser privileges, but uid equals either the real user ID or the saved set-user- ID, setuid sets only the effective user ID to uid. The real user ID and the saved set-user-ID
- are not changed. If neither of these two conditions is true, errnois 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 figure summarises the various ways these three user IDs can be changed

setreuidand setregidFunctions

Swapping of the real user ID and the effective user ID with the setreuid function.

```
#include <unistd.h>
int setreuid(uid_t ruid, uid_t euid);
int setregid(gid_t rgid, gid_t egid);
```

Both return : 0 if OK, -1 on error

We can supply a value of 1 for any of the arguments to indicate that the corresponding ID should remain unchanged. The rule is simple: an unprivileged user can always swap between the real user ID and the effective user ID. This allows a set-user-ID program to swap to the user's normal permissions and swap back again later for set-user- ID operations.

seteuid and setegid functions :

POSIX.1 includes the two functions seteuid and setegid. These functions are similar to setuid and setgid, but only the effective user ID or effective group ID is changed.

```
#include  <unistd.h>
int seteuid(uid_t uid);
int setegid(gid_t gid);
```

Both return : 0 if OK, 1 on error

An unprivileged user can set its effective user ID to either its real user ID or its saved set-user-ID. For a privileged user, only the effective user ID is set to uid. (This differs from the setuid function, which changes all three user IDs.)

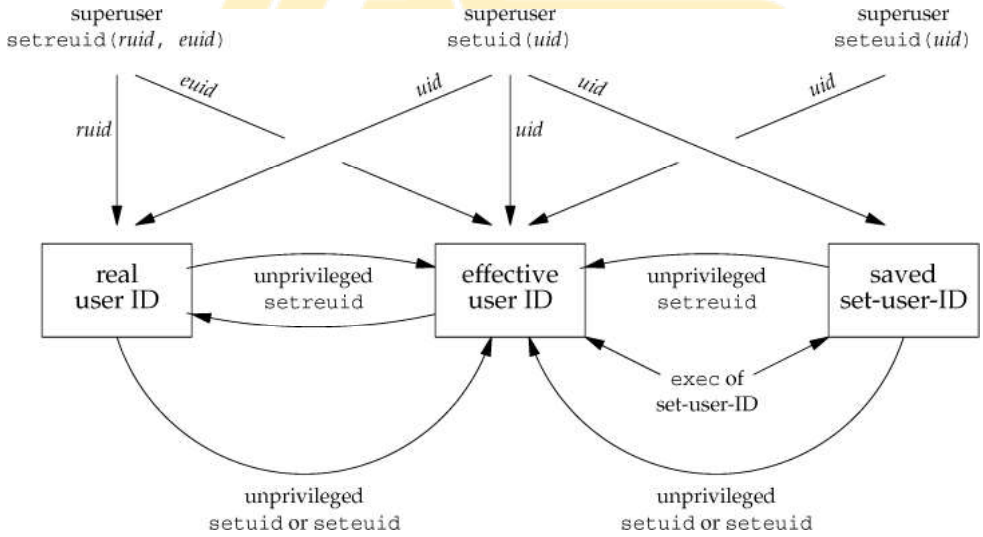


Figure: Summary of all the functions that set the various user Ids

INTERPRETER FILES

These files are text files that begin with a line of the form

#! pathname [optional-argument]

The space between the exclamation point and the pathname is optional. The most common of these interpreter files begin with the line

#!/bin/sh

The pathname is normally an absolute pathname, since no special operations are performed on it (i.e., PATH is not used). The recognition of these files is done within the kernel as part of processing the exec system call. The actual file that gets executed by the kernel is not the interpreter file, but the file specified by the pathname on the first line of the interpreter file. Be sure to differentiate between the interpreter filea text file that begins with #!and the interpreter, which is specified by the pathname on the first line of the interpreter file.
Be aware that systems place a size limit on the first line of an interpreter file. This limit includes the #!, the pathname, the optional argument, the terminating newline, and any spaces.

A program that execs an interpreter file

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

Int main(void)
{
    pid_t    pid;
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {                    /* child */
        if (execl("/home/sar/bin/testinterp",
                  "testinterp", "myarg1", "MY ARG2", (char *)0) <
            0) err_sys("execl error");
    }
    if (waitpid(pid, NULL, 0) < 0) /* parent */
        err_sys("waitpid error");
    exit(0);
}
```

Output:

```
$ cat /home/sar/bin/testinterp
#!/home/sar/bin/echoarg foo
$ ./a.out
argv[0]: /home/sar/bin/echoarg
argv[1]: foo
argv[2]: /home/sar/bin/testinterp
argv[3]: myarg1
argv[4]: MY ARG2
```

systemFUNCTION

```
#include <stdlib.h>

int system(const char *cmdstring);
```

If cmdstring is a null pointer, system returns nonzero only if a command processor is available. This feature
determines whether the system function is supported on a given operating system. Under the UNIX System,
systemis always available.

Because systemis implemented by calling fork, exec, and waitpid, there are three types of return values.

- ✓ If either the forkfails or waitpidreturns an error other than EINTR, systemreturns 1 with errnoset to indicate the error.
- ✓ If the execfails, 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 systemfunction, 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 systemfunction

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

```
Int main(void)
   {
        int          status;

        if ((status = system("date")) < 0)
              err_sys("system() error");
        pr_exit(status);

       if ((status = system("nosuchcommand")) < 0)
             err_sys("system() error");
       pr_exit(status);

       if ((status = system("who; exit 44")) < 0)
             err_sys("system() error");
       pr_exit(status);

       exit(0);
   }
```

Program: Execute the command-line argument using system

```
#include "apue.h"

Int main(int argc, char *argv[])
{
      int        status;
      if (argc < 2)
            err_quit("command-line argument required");
      if ((status = system(argv[1])) < 0)
            err_sys("system() error");
      pr_exit(status);
      exit(0);
}
```

Program: Print real and effective user IDs

```
#include "apue.h"

int
main(void
)
{
      printf("real uid = %d, effective uid = %d\n", getuid(), geteuid());
      exit(0);


}
```

PROCESS ACCOUNTING

- Most UNIX systems provide an option to do process accounting. When enabled, the kernel writes an accounting record each time a process terminates.
- These accounting records are typically a small amount of binary data with the name of the command, the amount of CPU time used, the user ID and group ID, the starting time, and so on.
- A superuser executes accton with a pathname argument to enable accounting.
- The accounting records are written to the specified file, which is usually /var/account/acct. Accounting is turned off by executing accton without any arguments.
- The data required for the accounting record, such as CPU times and number of characters transferred, is kept by the kernel in the process table and initialized whenever a new process is created, as in the child after a fork.
- Each accounting record is written when the process terminates.
- This means that the order of the records in the accounting file corresponds to the termination order of the processes, not the order in which they were started.
- The accounting records correspond to processes, not programs.
- A new record is initialized by the kernel for the child after a fork, not when a new program is executed. The structure of the accounting records is defined in the header <sys/acct.h>and looks something like

```
typedef   u_short comp_t;    /* 3-bit base 8 exponent; 13-bit fraction */

struct    acct
{
  char    ac_flag;       /*  flag    */
  char    ac_stat;       /*  termination status (signal & core flag only) */
                         /*  (Solaris only) */
  uid_t   ac_uid;        /*  real user ID */
  gid_t   ac_gid;        /*  real group ID */
  dev_t   ac_tty;        /*  controlling terminal */
  time_t  ac_btime;      /*  starting calendar time */
 comp_t   ac_utime;      /*  user CPU time (clock ticks) */
 comp_t   ac_stime;      /*  system CPU time (clock ticks) */
 comp_t   ac_etime;      /*  elapsed time (clock ticks) */
 comp_t   ac_mem;        /*  average memory usage */
 comp_t   ac_io;         /*  bytes transferred (by read and write) */
                         /*  "blocks" on BSD systems */
 comp_t   ac_rw;         /*  blocks read or written */
                         /*  (not present on BSD systems) */
  char    ac_comm[8]     /*  command name: [8] for Solaris, */
          ;
                         /*  [10] for Mac OS X, [16] for FreeBSD, and */
                         /*  [17] for Linux */
};
```

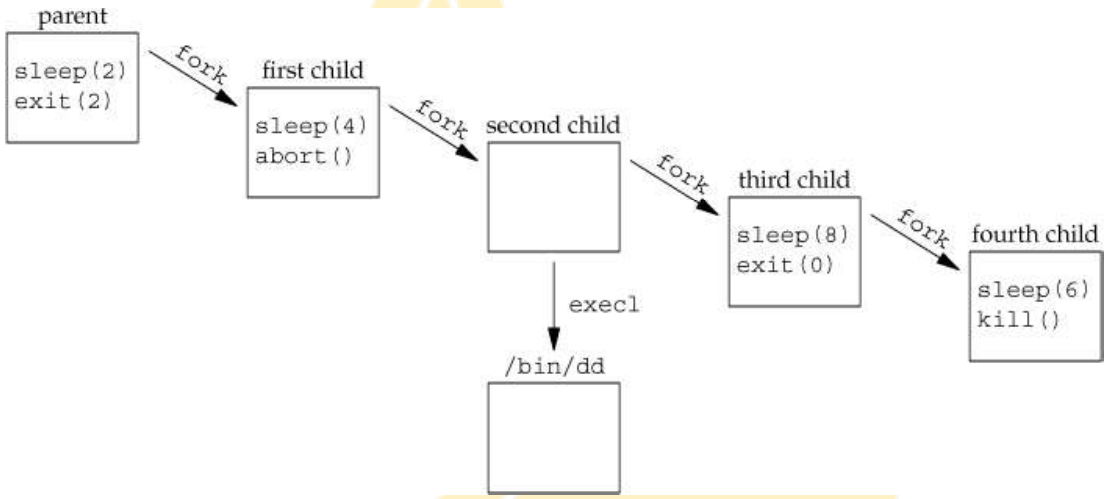
Values for ac_flagfrom accounting record

ac_flag	Description
AFORK	process is the result of fork, but never called exec
ASU	process used superuser privileges
ACOMPAT	process used compatibility mode
ACORE	process dumped core
AXSIG	process was killed by a signal
AEXPND	expanded accounting entry

Program to generate accounting data

```
#include "apue.h"

Int main(void)
{
    pid_t    pid;

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {            /* parent */
        sleep(2);
        exit(2);                    /* terminate with exit status 2 */
    }

                                    /* first child */
    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {
        sleep(4);
        abort();                    /* terminate with core dump */
    }

                                    /* second child */
  if ((pid = fork()) < 0)
      err_sys("fork error");
  else if (pid != 0) {
      execl("/bin/dd", "dd", "if=/etc/termcap", "of=/dev/null", NULL);
      exit(7);                      /* shouldn't get here */
  }
```

```
                                    /* third child */

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {
        sleep(8);
        exit(0);                    /* normal exit */
    }

                                    /* fourth child */
    sleep(6);
    kill(getpid(), SIGKILL);        /* terminate w/signal, no core dump */
    exit(6);                        /* shouldn't get here */
}
```



Process structure for accounting example

USER IDENTIFICATION

Any process can find out its real and effective user ID and group ID. Sometimes, however, we want to find out the login name of the user who's running the program. We could call getpwuid(getuid()), but what if a single user has multiple login names, each with the same user ID? (A person might have multiple entries in the password file with the same user ID to have a different login shell for each entry.) The system normally keeps track of the name we log in and the 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.

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.

PROCESS RELATIONSHIPS

INTRODUCTION

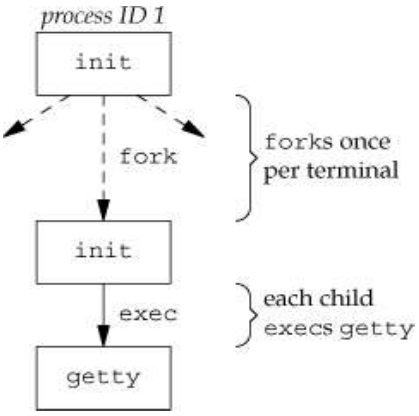
In this chapter, we'll look at process groups in more detail and the concept of sessions that was introduced by POSIX.1. We'll also look at the relationship between the login shell that is invoked for us when we log in and all the processes that we start from our login shell.

TERMINAL LOGINS

The terminals were either local (directly connected) or remote (connected through a modem). In either case, these logins came through a terminal device driver in the kernel.

The system administrator creates a file, usually /etc/ttys, that has one line per terminal device. Each line specifies the name of the device and other parameters that are passed to the getty program. One parameter is the baud rate of the terminal, for example. When the system is bootstrapped, the kernel creates process ID 1, the init process, and it is init that brings the system up multiuser. The init process reads the file /etc/ttys and, for every terminal device that allows a login, does a fork followed by an exec of the program getty. This gives us the processes shown in Figure 9.1.

Figure 9.1. Processes invoked by initto allow terminal logins



All the processes shown in Figure 9.1 have a real user ID of 0 and an effective user ID of 0 (i.e., they all have superuser privileges). The initprocess also execs the gettyprogram with an empty environment.

It is getty that calls open for the terminal device. The terminal is opened for reading and writing. If the device is a modem, the open may delay inside the device driver until the modem is dialed and the call is answered. Once the device is open, file descriptors 0, 1, and 2 are set to the device. Then getty outputs something like login: and waits for us to enter our user name.

When we enter our user name, getty's job is complete, and it then invokes the loginprogram, similar to

execle("/bin/login", "login", "-p", username, (char *)0, envp);

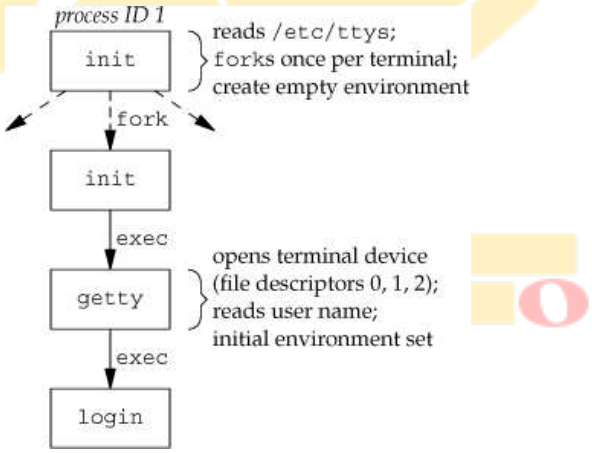


Figure 9.2. State of processes after loginhas been invoked

All the processes shown in Figure 9.2 have superuser privileges, since the original init process has superuser privileges.

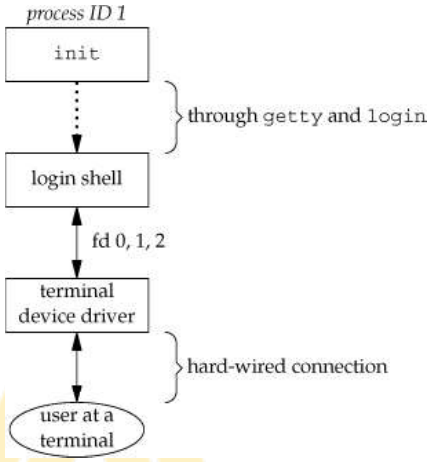
If we log in correctly, loginwill

- Change to our home directory (chdir)
- Change the ownership of our terminal device (chown) so we own it
- Change the access permissions for our terminal device so we have permission to read from and write to it
- Set our group IDs by calling setgidand initgroups
- Initialize the environment with all the information that login has: our home directory (HOME), shell (SHELL), user name (USERand LOGNAME), and a default path (PATH)

- Change to our user ID (setuid) and invoke our login shell, as in **execl("/bin/sh", "-sh", (char *)0);**

The minus sign as the first character of argv[0] is a flag to all the shells that they are being invoked as a login shell. The shells can look at this character and modify their start-up accordingly.

Figure 9.3. Arrangement of processes after everything is set for a terminal login



NETWORK LOGINS

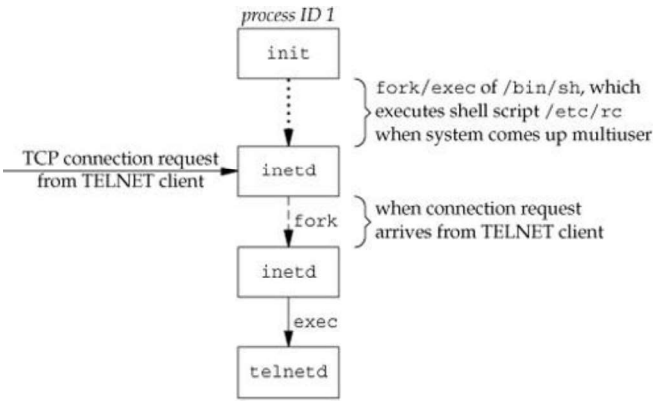
The main (physical) difference between logging in to a system through a serial terminal and logging in to a system through a network is that the connection between the terminal and the computer isn't point-to-point. With the terminal logins that we described in the previous section, init knows which terminal devices are enabled for logins and spawns a gettyprocess for each device. In the case of network logins, however, all the logins come through the kernel's network interface drivers (e.g., the Ethernet driver).

Let's assume that a TCP connection request arrives for the TELNET server. TELNET is a remote login application that uses the TCP protocol. A user on another host (that is connected to the server's host through a network of some form) or on the same host initiates the login by starting the TELNET client:

telnet hostname

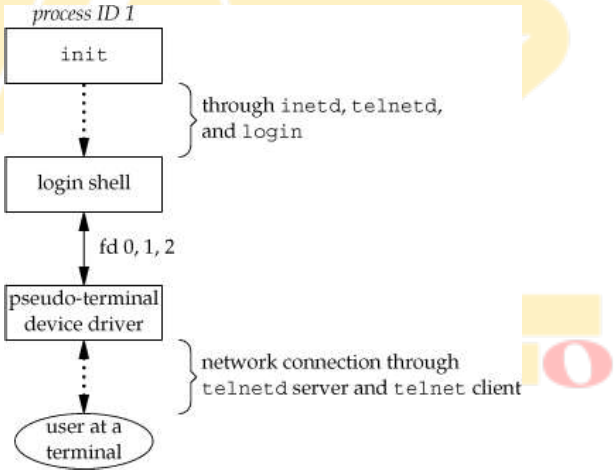
The client opens a TCP connection to hostname, and the program that's started on hostname is called the TELNET server. The client and the server then exchange data across the TCP connection using the TELNET application protocol. What has happened is that the user who started the client program is now logged in to the server's host. (This assumes, of course, that the user has a valid account on the server's host.) Figure 9.4 shows the sequence of processes involved in executing the TELNET server, called telnetd.

Figure 9.4. Sequence of processes involved in executing TELNET server



The telnetd process then opens a pseudo-terminal device and splits into two processes using fork. The parent handles the communication across the network connection, and the child does an exec of the login program. The parent and the child are connected through the pseudo terminal. Before doing the exec, the child sets up file descriptors 0, 1, and 2 to the pseudo terminal. If we log in correctly, loginperforms the same steps we described in Section 9.2: it changes to our home directory and sets our group IDs, user ID, and our initial environment. Then login replaces itself with our login shell by calling exec. Figure 9.5 shows the arrangement of the processes at this point.

Figure 9.5. Arrangement of processes after everything is set for a network login



PROCESS GROUPS

A process group is a collection of one or more processes, usually associated with the same job, that can receive signals from the same terminal. Each process group has a unique process group ID. Process group IDs are similar to process IDs: they are positive integers and can be stored in a pid_t data type. The function getpgrp returns the process group ID of the calling process.

```
#include <unistd.h>

pid_t getpgrp(void);
```

Returns: process group ID of calling process

The Single UNIX Specification defines the getpgidfunction as an XSI extension that mimics this behavior.

```
#include <unistd.h>

pid_t getpgid(pid_t pid);
```

Returns: process group ID if OK, 1 on error

Each process group can have a process group leader. The leader is identified by its process group ID being equal to its process ID.

It is possible for a process group leader to create a process group, create processes in the group, and then terminate. The process group still exists, as long as at least one process is in the group, regardless of whether the group leader terminates. This is called the process group lifetime-the period of time that begins when the group is created and ends when the last remaining process leaves the group. The last remaining process in the process group can either terminate or enter some other process group.

A process joins an existing process group or creates a new process group by calling setpgid.

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```

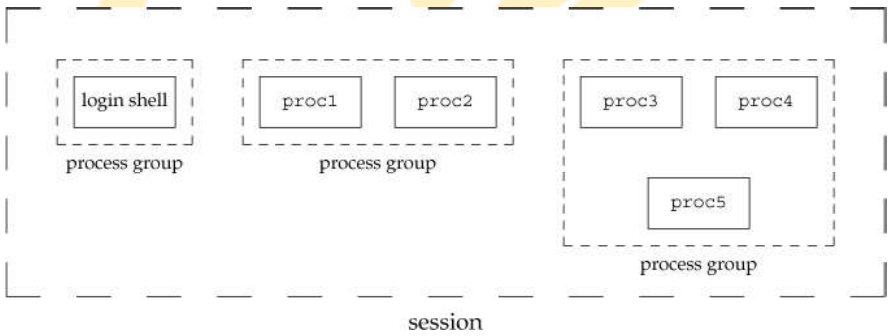
Returns: 0 if OK, 1 on error

This function sets the process group ID to pgid in the process whose process ID equals pid. If the two arguments are equal, the process specified by pid becomes a process group leader. If pid is 0, the process ID of the caller is used.

SESSIONS

A session is a collection of one or more process groups. For example, we could have the arrangement shown in Figure 9.6. Here we have three process groups in a single session.

Figure 9.6. Arrangement of processes into process groups and sessions



A process establishes a new session by calling the setsidfunction.

```
#include <unistd.h>

pid_t setsid(void);
```

Returns: process group ID if OK, 1 on error

If the calling process is not a process group leader, this function creates a new session. Three things happen.

- The process becomes the session leader of this new session. (A session leader is the process that creates a session.) The process is the only process in this new session.
- The process becomes the process group leader of a new process group. The new process group ID is the process ID of the calling process.

- The process has no controlling terminal. If the process had a controlling terminal before calling setsid, that association is broken.

This function returns an error if the caller is already a process group leader. The getsidfunction returns the process group ID of a process's session leader. The getsid function is included as an XSI extension in the Single UNIX Specification.

```
#include <unistd.h>

pid_t getsid(pid_t pid);
```

Returns: session leader's process group ID if OK, 1 on error

If pid is 0, getsidreturns the process group ID of the calling process's session leader.

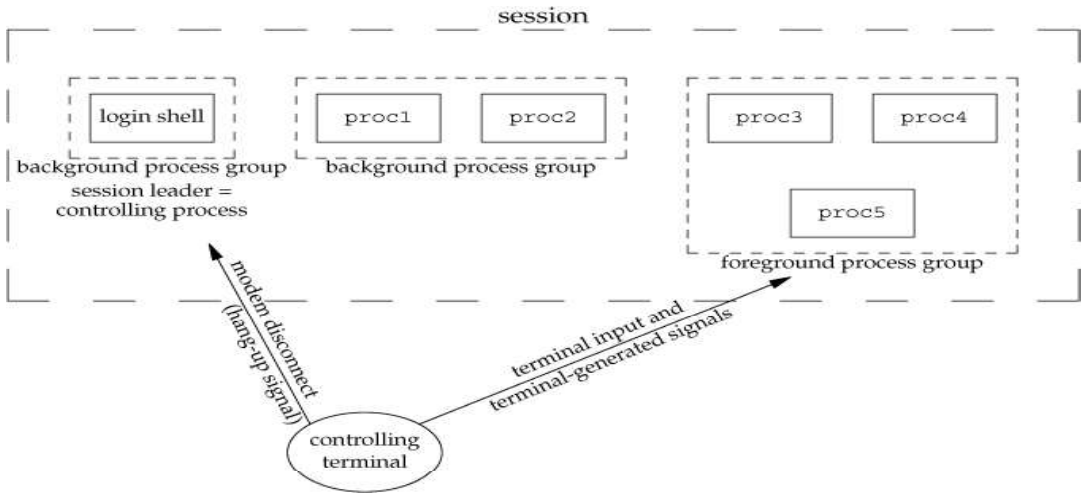
CONTROLLING TERMINAL

Sessions and process groups have a few other characteristics.

- A session can have a single controlling terminal. This is usually the terminal device (in the case of a terminal login) or pseudo-terminal device (in the case of a network login) on which we log in.
- The session leader that establishes the connection to the controlling terminal is called the controlling process.
- The process groups within a session can be divided into a single foreground process group and one or more background process groups.
- If a session has a controlling terminal, it has a single foreground process group, and all other process groups in the session are background process groups.
- Whenever we type the terminal's interrupt key (often DELETE or Control-C), this causes the interrupt signal be sent to all processes in the foreground process group.
- Whenever we type the terminal's quit key (often Control-backslash), this causes the quit signal to be sent to all processes in the foreground process group.
- If a modem (or network) disconnect is detected by the terminal interface, the hang-up signal is sent to the controlling process (the session leader).

These characteristics are shown in Figure 9.7.

Figure 9.7. Process groups and sessions showing controlling terminal



tcgetpgrp, tcsetpgrp, AND tcgetsidFUNCTIONS

We need a way to tell the kernel which process group is the foreground process group, so that the terminal device driver knows where to send the terminal input and the terminal-generated signals.
To retrieve the foreground process group-id and to set the foreground process group-id we can use tcgetprgp and tcsetpgrp function.

The prototype of these functions are :

```
#include <unistd.h>
pid_t tcgetpgrp(int filedes);
```

Returns : process group ID of foreground process group if OK, -1 on error

```
int tcsetpgrp(int filedes, pid_t pgrpid);
```

Returns: 0 if OK, -1 on error

The function tcgetpgrp returns the process group ID of the foreground process group associated with the terminal open on *filedes*. If the process has a controlling terminal, the process can call tcsetpgrp to set the foreground process group ID to pgrpid. The value of pgrpid must be the process group ID of a process group in the same session, and filedes must refer to the controlling terminal of the session.
The single UNIX specification defines an XSI extension called tcgetsid to allow an application to obtain the process group-ID for the session leader given a file descriptor for the controlling terminal.

```
#include <termios.h>
pid_t tcgetsid(int filedes);
```

Returns: session leader's process group ID if Ok, -1 on error

JOB CONTROL

This feature allows us to start multiple jobs (groups of processes) from a single terminal and to control which jobs can access the terminal and which jobs are to run in the background. Job control requires three forms of support:

- A shell that supports job control
- The terminal driver in the kernel must support job control
- The kernel must support certain job-control signals

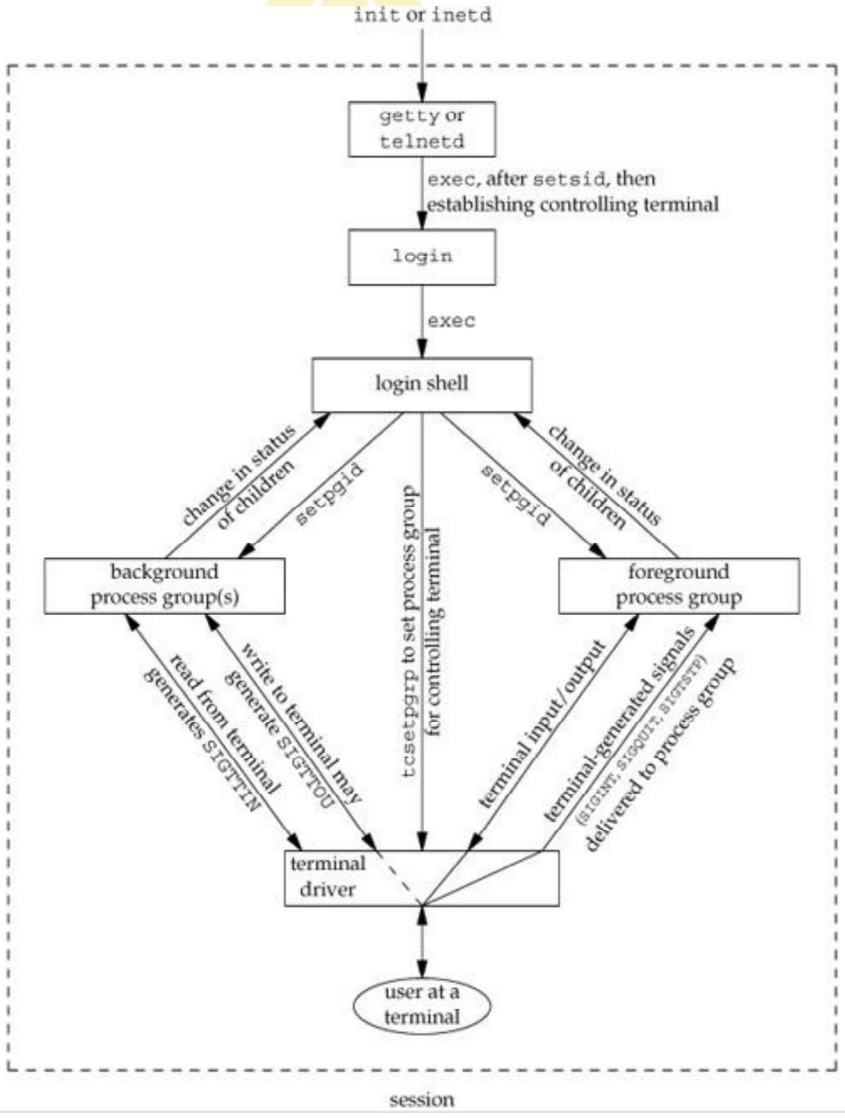
The interaction with the terminal driver arises because a special terminal character affects the foreground job: the suspend key (typically Control-Z). Entering this character causes the terminal driver to send the SIGTSTP signal to all processes in the foreground process group. The jobs in any background process groups aren't affected. The terminal driver looks for three special characters, which generate signals to the foreground process group.

- The interrupt character (typically DELETE or Control-C) generates SIGINT.
- The quit character (typically Control-backslash) generates SIGQUIT.
- The suspend character (typically Control-Z) generates SIGTSTP.

This signal normally stops the background job; by using the shell, we are notified of this and can bring the job into the foreground so that it can read from the terminal. The following demonstrates this:

```
$ cat > temp.foo &          start in background, but it'll read from standard input
  [1]       1681
  $                         we press RETURN
  [1] + Stopped (SIGTTIN)   cat > temp.foo &
  $ fg %1                   bring job number 1 into the foreground
  cat > temp.foo            the shell tells us which job is now in the foreground

  hello, world              enter one line

  ^D                        type the end-of-file character
  $ cat temp.foo            check that the one line was put into the file
  hello, world
```

Figure 9.8. Summary of job control features with foreground & background jobs & terminal driver



What happens if a background job outputs to the controlling terminal? This is an option that we can allow or disallow. Normally, we use the stty(1) command to change this option. The following shows how this works:

```
$ cat temp.foo &                    execute in background
[1]     1719
$ hello, world              the output from the background job appears after the prompt
                             we press RETURN

[1] + Done      cat temp.foo &

$ stty tostop           disable ability of background jobs to output to controlling terminal
$ cat temp.foo &            try it again in the background
[1]     1721
$                           we press RETURN and find the job is stopped

[1] + Stopped(SIGTTOU)      cat temp.foo &

$ fg %1                     resume stopped job in the foreground
cat temp.foo                the shell tells us which job is now in the foreground
hello, world                and here is its output
```

When we disallow background jobs from writing to the controlling terminal, cat will block when it tries to write to its standard output, because the terminal driver identifies the write as coming from a background process and sends the job the SIGTTOUsignal.

Figure 9.8 summarizes some of the features of job control that we've been describing. The solid lines through the terminal driver box mean that the terminal I/O and the terminal-generated signals are always connected from the foreground process group to the actual terminal. The dashed line corresponding to the SIGTTOU signal means that whether the output from a process in the background process group appears on the terminal is an option.

SHELL EXECUTION OF PROGRAMS

Example 1: **ps -o pid,ppid,pgid,sid,comm**

Output 1:

```
  PID  PPID  PGID  SID  COMMAND
  949   947   949  949  sh
 1774   949   949  949  ps
```

Example 2: **ps -o pid,ppid,pgid,sid,comm &**

Output 2:

```
  PID  PPID  PGID  SID COMMAND
  949   947   949  949 sh
 1812   949   949  949 ps
```

If we execute the command in the background, the only value that changes is the process ID of the command.

Example 3: **ps -o pid,ppid,pgid,sid,comm | cat1**

Output 3:

```
  PID  PPID  PGID  SID COMMAND
  949   947   949  949 sh
 1823   949   949  949 cat1
 1824  1823   949  949 ps
```

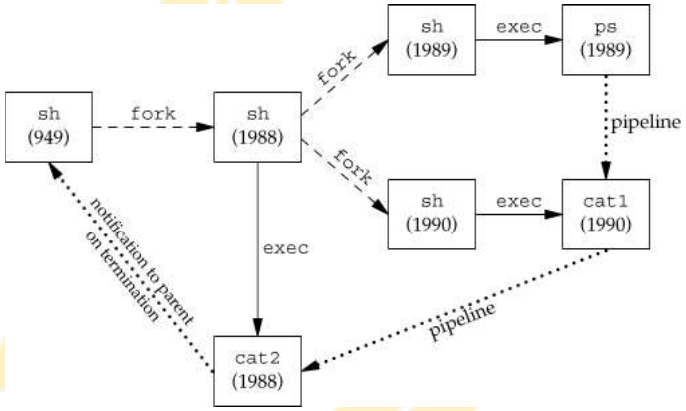
The program cat1 is just a copy of the standard cat program, with a different name. Note that the last process in the pipeline is the child of the shell and that the first process in the pipeline is a child of the last process.

Example 4: **ps -o pid,ppid,pgid,sid,comm | cat1 | cat2**

Output 4:

```
  PID  PPID  PGID  SID COMMAND
  949   947   949  949 sh
 1988   949   949  949 cat2
 1989  1988   949  949 ps
 1990  1988   949  949 cat1
```

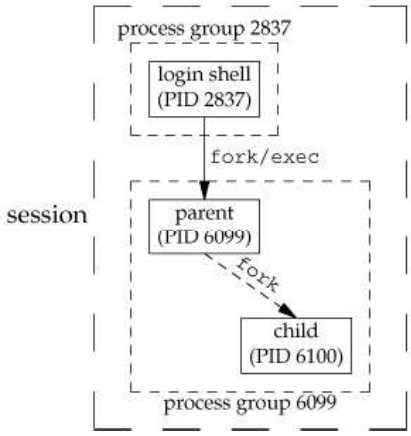
Figure 9.9. Processes in the pipeline ps | cat1 | cat2when invoked by Bourne shell



ORPHANED PROCESS GROUPS

A process whose parent terminates is called an orphan and is inherited by the initprocess.

Example of a process group about to be orphaned



Creating an orphaned process group

#include "apue.h"

#include <errno.h>

```
static void
sig_hup(int signo)
{
      printf("SIGHUP received, pid = %d\n", getpid());
}

static void
pr_ids(char
*name)
{
      printf("%s: pid = %d, ppid = %d, pgrp = %d, tpgrp = %d\n",
            name, getpid(), getppid(), getpgrp(), tcgetpgrp(STDIN_FILENO));
      fflush(stdout);
}

int
main(void
)
{
       char       c;
       pid_t      pid;

       pr_ids("parent");
       if ((pid = fork()) < 0) {
            err_sys("fork error");
       } else if (pid > 0) {           /* parent */
            sleep(5);             /*sleep to let child stop itself */
            exit(0);              /* then parent exits */
       } else {                   /* child */
            pr_ids("child");
            signal(SIGHUP, sig_hup);      /* establish signal handler */
            kill(getpid(), SIGTSTP);      /* stop ourself */
            pr_ids("child");       /* prints only if we're continued */ if
            (read(STDIN_FILENO, &c, 1) != 1)
                  printf("read error from controlling TTY, errno = %d\n",
                        errno);
            exit(0);
       }
}
```

Output:

```
$ ./a.out
parent: pid = 6099, ppid = 2837, pgrp = 6099, tpgrp = 6099
child: pid = 6100, ppid = 6099, pgrp = 6099, tpgrp = 6099
```

```
$ SIGHUP received, pid = 6100
child: pid = 6100, ppid = 1, pgrp = 6099, tpgrp = 2837 read
error from controlling TTY, errno = 5
```

The child inherits the process group of its parent (6099). After the fork,

- The parent sleeps for 5 seconds. This is our (imperfect) way of letting the child execute before the parent terminates.
- The child establishes a signal handler for the hang-up signal (SIGHUP). This is so we can see whether SIGHUP is sent to the child. The child sends itself the stop signal (SIGTSTP) with the kill function. This stops the child, similar to our stopping a foreground job with our terminal's suspend character (Control-Z).
- When the parent terminates, the child is orphaned, so the child's parent process ID becomes 1, the init process ID.
- At this point, the child is now a member of an orphaned process group If the process group is not orphaned, there is a chance that one of those parents in a different process group but in the same session will restart a stopped process in the process group that is not orphaned. Here, the parent of every process in the group belongs to another session.
- Since the process group is orphaned when the parent terminates, POSIX.1 requires that every process in the newly orphaned process group that is stopped (as our child is) be sent the hang-up signal (SIGHUP) followed by the continue signal (SIGCONT).
- This causes the child to be continued, after processing the hang-up signal. The default action for the hang-up signal is to terminate the process, so we have to provide a signal handler to catch the signal. We therefore expect the printfin the sig_hupfunction to appear before the printfin the pr_idsfunction.

OVERVIEW OF IPC METHODS

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.

- Historically, they have been half duplex (i.e., data flows in only one direction).
- 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.

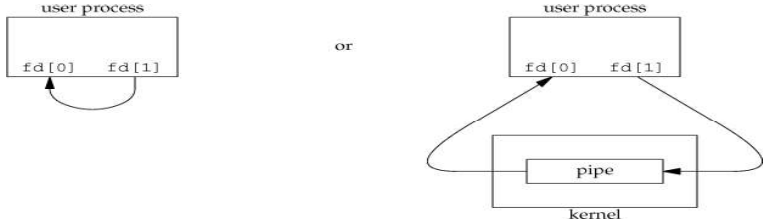
```
#include <unistd.h>

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].

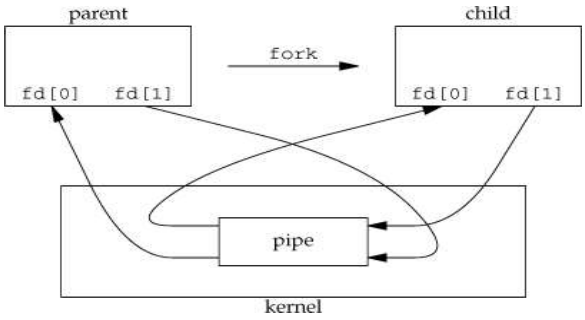
Two ways to picture a half-duplex pipe are shown in Figure 15.2. The left half of the figure shows the two ends of the pipe connected in a single process. The right half of the figure emphasizes that the data in the pipe flows through the kernel.

Figure 15.2. Two ways to view a half-duplex pipe



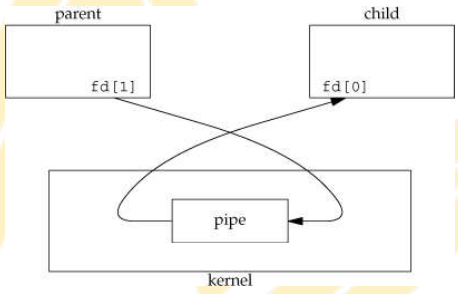
A pipe in a single process is next to useless. Normally, the process that calls pipe then calls fork, creating an IPC channel from the parent to the child or vice versa. Figure 15.3 shows this scenario.

Figure 15.3 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 15.4 shows the resulting arrangement of descriptors.

Figure 15.4. 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 errnoset 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);
}
```

popenAND pcloseFUNCTIONS

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 15.9. Result of fp = popen(cmdstring, "r")

If type is "w", the file pointer is connected to the standard input of cmdstring, as shown:

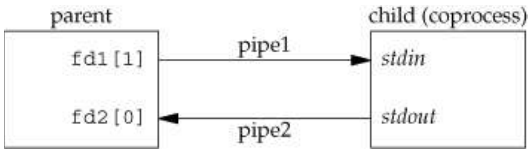
Figure 15.10. Result of fp = popen(cmdstring, "w")

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 15.16 shows this arrangement.

Figure 15.16. Driving a coprocess by writing its standard input and reading its standard output



Program: Simple filter to add two numbers

```
#include "apue.h"

Int main(void)
{
    int     n,  int1,   int2;
    char
            line[MAXLINE]
    ;

    while ((n = read(STDIN_FILENO, line, MAXLINE)) >
        0) { line[n] = 0;       /* null terminate */
        if (sscanf(line, "%d%d", &int1, &int2) == 2) {
            sprintf(line, "%d\n", int1 + int2);
            n = strlen(line);
            if (write(STDOUT_FILENO, line, n) !=
                n) err_sys("write error");
        } else {
            if (write(STDOUT_FILENO, "invalid args\n", 13) !=
                13) err_sys("write error");
        }
    }
    exit(0);
}
```

FIFOs

FIFOs are sometimes called named pipes. Pipes can be used only between related processes when a common ancestor has created the pipe.

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

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

Returns: 0 if OK, 1 on error

Once we have used mkfifo to create a FIFO, we open it using open. When we open a FIFO, the nonblocking flag (O_NONBLOCK) affects what happens.

- In the normal case (O_NONBLOCK not specified), an open for read-only blocks until some other process opens the FIFO for writing. Similarly, an open for write-only blocks until some other process opens the FIFO for reading.
- If O_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 15.20 shows this arrangement.

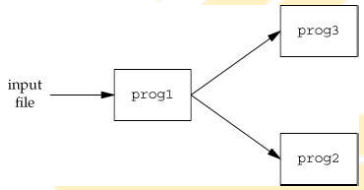


FIGURE 15.20 : Procedure that processes a filtered input stream twice

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 15.21shows the process arrangement.

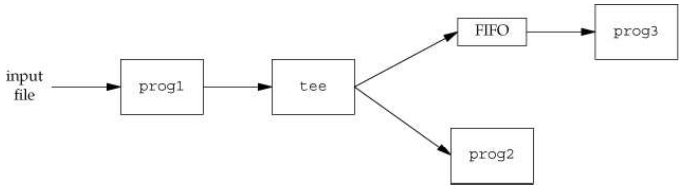


FIGURE 15.21: Using a FIFO and teeto send a stream to two different processes

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'sprocess 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 crashes. 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.

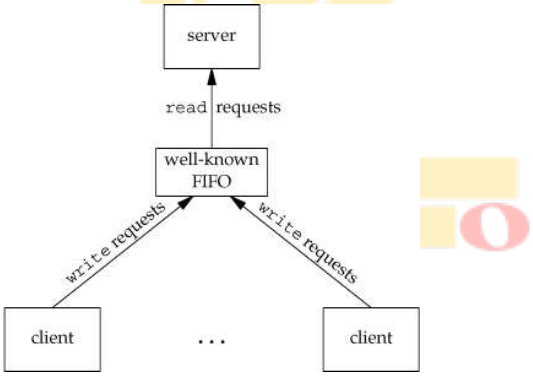


Figure 15.22. Clients sending requests to a server using a FIFO

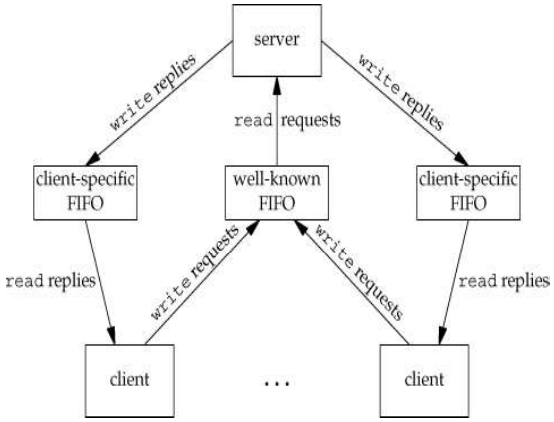


Figure 15.23. Client-server communication using FIFOs

XSI IPC

- **Identifiers and Keys**

Each IPC structure (message queue, semaphore, or shared memory segment) in the kernel is referred to by a non- negative integer identifier. The identifier is an internal name for an IPC object. Cooperating processes need an external naming scheme to be able to rendezvous using the same IPC object. For this purpose, an IPC object is associated with a key that acts as an external name.

Whenever an IPC structure is being created, a key must be specified. The data type of this key is the primitive system data type key_t, which is often defined as a long integer in the header <sys/types.h>. This key is converted into an identifier by the kernel.

There are various ways for a client and a server to rendezvous at the same IPC structure.

- The server can create a new IPC structure by specifying a key of IPC_PRIVATE and store the returned identifier somewhere (such as a file) for the client to obtain. The key IPC_PRIVATE guarantees that the server creates a new IPC structure. The disadvantage to this technique is that file system operations are required for the server to write the integer identifier to a file, and then for the clients to retrieve this identifier later.
  The IPC_PRIVATE key is also used in a parent-child relationship. The parent creates a new IPC structure specifying IPC_PRIVATE, and the resulting identifier is then available to the child after the fork. The child can pass the identifier to a new program as an argument to one of the execfunctions.
- The client and the server can agree on a key by defining the key in a common header, for example. The server then creates a new IPC structure specifying this key. The problem with this approach is that it's possible for the key to already be associated with an IPC structure, in which case the get function (msgget, semget, or shmget) returns an error. The server must handle this error, deleting the existing IPC structure, and try to create it again.
- The client and the server can agree on a pathname and project ID (the project ID is a character value between 0 and 255) and call the function ftok to convert these two values into a key. This key is then used in step 2. The only service provided by ftokis a way of generating a key from a pathname and project ID.

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

key_t ftok(const char *path, int id);
```

Returns: key if OK, (key_t)-1 on error

The path argument must refer to an existing file. Only the lower 8 bits of id are used when generating the key.

The key created by ftok is usually formed by taking parts of the st_dev and st_ino fields in the stat structure corresponding to the given pathname and combining them with the project ID. If two pathnames refer to two different files, then ftok usually returns two different keys for the two pathnames. However, because both i-node numbers and keys are often stored in long integers, there can be information loss creating a key. This means that two different pathnames to different files can generate the same key if the same project ID is used.

- **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 chownor chmodfor 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 15.24. 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 chmodcommand. 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.

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_dsstructure 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 semoparray argument is a pointer to an array of semaphore operations, represented by sembufstructures:

```
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.

MODULE 5
SIGNALS AND DAEMON PROCESSES

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
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
SIGTTOU	background write to control tty	stop process
SIGURG	urgent condition (sockets)	ignore
SIGUSR1	user-defined signal	terminate
SIGUSR2	user-defined signal	terminate
SIGVTALRM	virtual time alarm (setitimer)	terminate
SIGWAITING	threads library internal use	ignore
SIGWINCH	terminal window size change	ignore
SIGXCPU	CPU limit exceeded (setrlimit)	terminate+core/ignore
SIGXFSZ	file size limit exceeded (setrlimit)	terminate+core/ignore
SIGXRES	resource control exceeded	Ignore

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.

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.

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;
}


/*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.

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);
```

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,SIGSEG
        V); 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;
}
```

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.

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();
}
```

KILL

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.

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;
 }
```

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
                                =INTERV
        AL; val.it_interval.tv_usec  =0;
```

```
        val.it_value.tv_sec
                                =INTERV
        AL;
        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.

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);
```

DAEMON PROCESSES

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.

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.

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.

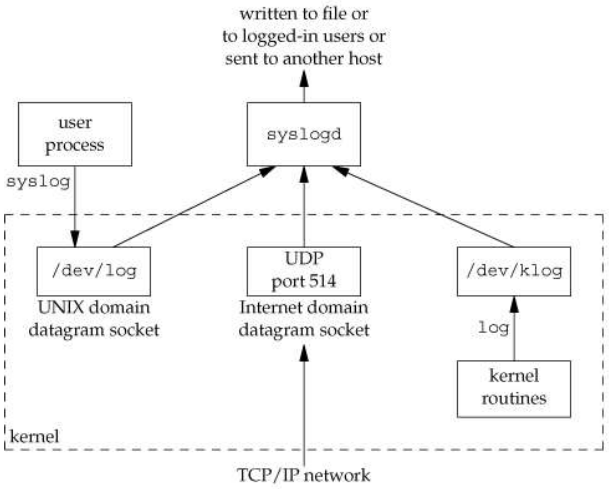
Example Program:

```
#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;
}
```

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.

Figure 13.2. 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);
```

SINGLE-INSTANCE DAEMONS

Some daemons are implemented so that only a single copy of the daemon should be running at a time for proper operation. The file and record-locking mechanism provides the basis for one way to ensure that only one copy of a daemon is running. If each daemon creates a file and places a write lock on the entire file, only one such write lock will be allowed to be created. Successive attempts to create write locks will fail, serving as an indication to successive copies of the daemon that another instance is already running.

File and record locking provides a convenient mutual-exclusion mechanism. If the daemon obtains a write-lock on an entire file, the lock will be removed automatically if the daemon exits. This simplifies recovery, removing the need for us to clean up from the previous instance of the daemon.

PROGRAM:Ensure that only one copy of a daemon is running

```
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <syslog.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>
#include <sys/stat.h>

#define LOCKFILE "/var/run/daemon.pid"
#define LOCKMODE (S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)
extern int lockfile(int); int
already_running(void)
{
    int       fd;
    char      buf[16];

    fd = open(LOCKFILE, O_RDWR|O_CREAT, LOCKMODE);
    if (fd < 0) {
```

```
            syslog(LOG_ERR, "can't open %s: %s", LOCKFILE,
            strerror(errno)); exit(1);
        }
        if (lockfile(fd) < 0) {
            if (errno == EACCES || errno ==
                EAGAIN) { close(fd);
                return(1);
            }
            syslog(LOG_ERR, "can't lock %s: %s", LOCKFILE,
            strerror(errno)); exit(1);
        }
        ftruncate(fd, 0);
        sprintf(buf, "%ld", (long)getpid());
        write(fd, buf, strlen(buf)+1);
        return(0);
}
```

DAEMON CONVENTIONS

- If the daemon uses a lock file, the file is usually stored in /var/run. Note, however, that the daemon might need superuser permissions to create a file here. The name of the file is usually name.pid, where name is the name of the daemon or the service. For example, the name of the cron daemon's lock file is
  /var/run/crond.pid.
- If the daemon supports configuration options, they are usually stored in /etc. The configuration file is named name.conf, where name is the name of the daemon or the name of the service. For example, the configuration for the syslogddaemon is /etc/syslog.conf.
- Daemons can be started from the command line, but they are usually started from one of the system initialization scripts (/etc/rc*or /etc/init.d/*). If the daemon should be restarted automatically when it exits, we can arrange for initto restart it if we include a respawnentry for it in /etc/inittab.
- If a daemon has a configuration file, the daemon reads it when it starts, but usually won't look at it again. If an administrator changes the configuration, the daemon would need to be stopped and restarted to account for the configuration changes. To avoid this, some daemons will catch SIGHUP and reread their configuration files when they receive the signal. Since they aren't associated with terminals and are either session leaders without controlling terminals or members of orphaned process groups, daemons have no reason to expect to receive SIGHUP. Thus, they can safely reuse it

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 13.2 [REFER PAGE 10], the service being provided by the syslogdserver is the logging of an error message.

In Figure 13.2, 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.