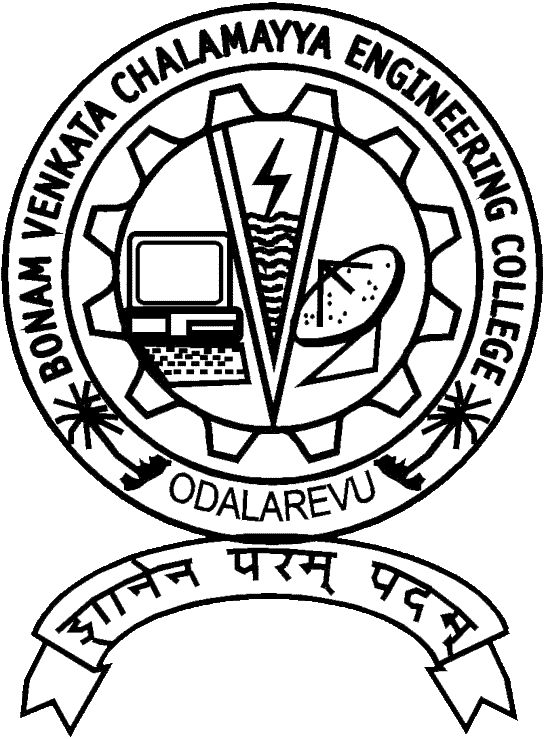
Department of Computer Science and Engineering

Sample Program on

UNIX Programming Laboratory

**FOR III B TECH CSE R-16 REGULATION**



Prepared by

Prof. Gunamani Jena, Ph D. FIE

**Sr. Member IEEE**

BONAM VENKATA CHALAMAYYA ENGINEERING COLLEGE**, AUTONOMOUS**

(Permanently affiliated to JNTUK, Kakinada, Approved by AICTE New Delhi, Accredited by NAAC with A grade and NBA)

ALLAVARAM MANDAL,ODALAREVU,DISTRICT : EAST GODAVARI. PIN: 533210. AP, INDIA

**2018-19**

**Contents**

**Sl. No Name of Experiment Page No.**

1. a) Study of Unix/Linux general purpose utility command list

man, who, cat, cd, cp, ps, ls, mv, rm, mkdir, rmdir, echo, more, date, time, kill, history, chmod, chown, finger, pwd, cal, logout, shutdown. [3]

b) Study of vi editor.

c) Study of Bash shell, Bourne shell and C shell in Unix/Linux operating system.

d) Study of Unix/Linux file system (tree structure).

e) Study of .bashrc, /etc/bashrc and Environment variables.

2. Write a C program that makes a copy of a file using standard I/O, and system calls [21]

3. Write a C program to emulate the UNIX ls –l command. [22]

4. Write a C program that illustrates how to execute two commands concurrently

with a command pipe. Ex: - ls –l | sort [22]

5. Write a C program that illustrates two processes communicating using shared memory [23]

6. Write a C program to simulate producer and consumer problem using semaphores [25]

7. Write C program to create a thread using pthreads library and let it run its function. [26]

8. Write a C program to illustrate concurrent execution of threads using pthreads library. [27]

1. a) Study of Unix/Linux general purpose utility command list:

man, who, cat, cd, cp, ps, ls, mv, rm, mkdir, rmdir, echo, more, date, time, kill, history, chmod, chown, finger, pwd, cal, logout, shutdown.

Ans.

man

This is an interface to the online reference manuals

$man ls

**Who you are**

The easiest way to find out "who you are" is to enter the **whoami** command −

$ whoami

Sometime you might be interested to know who is logged in to the computer at the same time.

$ who

## Display Content of a File

You can use the **cat** command to see the content of a file. Following is a simple example to see the content of the above created file −

$ cat filename

You can go in your home directory anytime using the following command −

$cd ~

Here **~** indicates the home directory. Suppose you have to go in any other user's home directory, use the following command −

$cd ~username

To go in your last directory, you can use the following command −

$cd -

To make a copy of a file use the **cp** command. The basic syntax of the command is −

$ cp source\_file destination\_file

ps: It reports a snapshot of the current process

To list the files and directories stored in the current directory, use the following command −

$ls

## Listing Directories

To list the files in a directory, you can use the following syntax −

$ls dirname

Following is the example to list all the files contained in **/usr/local** directory −

$ls /usr/local

The command **ls** supports the **-l** option which would help you to get more information about the listed files −

$ls -l

To change the name of a file, use the **mv** command. Following is the basic syntax −

$ mv old\_file new\_file

The following program will rename the existing file **filename** to **newfile**.

$ mv filename newfile

To delete an existing file, use the **rm** command. Following is the basic syntax −

$ rm filename

You can remove multiple files at a time with the command given below −

$ rm filename1 filename2 filename3

## Creating Parent Directories

$mkdir /tmp/amrood/test

## Removing Directories

Directories can be deleted using the **rmdir** command as follows −

$rmdir dirname

## Changing Directories

You can use the **cd** command to do more than just change to a home directory. You can use it to change to any directory by specifying a valid absolute or relative path. The syntax is as given below −

$cd dirname

## Renaming Directories

The **mv (move)** command can also be used to rename a directory. The syntax is as follows −

$mv olddir newdir

You can rename a directory **mydir** to **yourdir** as follows −

$mv mydir yourdir

echo: it displays a line of text

$echo Gunamani

Gunamani

more: more [option] <file>

### System Shutdown

The most consistent way to shut down a Unix system properly via the command line is to use one of the following commands −

|  |  |
| --- | --- |
|  | **Halt** Brings the system down immediately |
|  | **init 0** Powers off the system using predefined scripts to synchronize and clean up the system prior to shutting down |
|  | **init 6** Reboots the system by shutting it down completely and then restarting it |
|  | **Poweroff** Shuts down the system by powering off |
|  | **Reboot** Reboots the system |
|  | **Shutdown** Shuts down the system |

date: Wed Apr 11 13:14:48 IST 2018

time: run programs and summarize system resource usage

history: keep track of command already used, make it easy to repeat commands

File Permission

* **Owner permissions** − The owner's permissions determine what actions the owner of the file can perform on the file.
* **Group permissions** − The group's permissions determine what actions a user, who is a member of the group that a file belongs to, can perform on the file.
* **Other (world) permissions** − The permissions for others indicate what action all other users can perform on the file.

## File Access Modes

The permissions of a file are the first line of defense in the security of a Unix system. The basic building blocks of Unix permissions are the **read**, **write**, and **execute** permissions, which have been described below −

### Read

Grants the capability to read, i.e., view the contents of the file.

### Write

Grants the capability to modify, or remove the content of the file.

### Execute

User with execute permissions can run a file as a program.

$ls -l testfile

$ls -l testfile

-rwxrwxr-- 1 amrood users 1024 Nov 2 00:10 testfile

$ chmod 755 testfile

$ls -l testfile

-rwxr-xr-x 1 amrood users 1024 Nov 2 00:10 testfile

$chmod 743 testfile

$ls -l testfile

-rwxr---wx 1 amrood users 1024 Nov 2 00:10 testfile

$chmod 043 testfile

$ls -l testfile

----r---wx 1 amrood users 1024 Nov 2 00:10 testfile

## Changing Owners and Groups

While creating an account on Unix, it assigns a **owner ID** and a **group ID** to each user. All the permissions mentioned above are also assigned based on the Owner and the Groups.

Two commands are available to change the owner and the group of files −

* **chown** − The **chown** command stands for **"change owner"** and is used to change the owner of a file.
* **chgrp** − The **chgrp** command stands for **"change group"** and is used to change the group of a file.

## Changing Ownership

The **chown** command changes the ownership of a file. The basic syntax is as follows −

$ chown user filelist

## Changing Group Ownership

The **chgrp** command changes the group ownership of a file. The basic syntax is as follows −

$ chgrp group filelist

The value of group can be the **name of a group** on the system or **the group ID (GID)** of a group on the system.

Following example helps you understand the concept −

$ chgrp special testfile

Changes the group of the given file to **special** group.

## Setting the PATH

When you type any command on the command prompt, the shell has to locate the command before it can be executed.

The PATH variable specifies the locations in which the shell should look for commands. Usually the Path variable is set as follows −

$PATH = /bin:/usr/bin

Here, each of the individual entries separated by the colon character **(:)** are directories. If you request the shell to execute a command and it cannot find it in any of the directories given in the PATH variable, a message similar to the following appears −

$hello

hello: not found

$

There are variables like PS1 and PS2 which are discussed in the next section.

$ echo "this is a

> test"

this is a

test

$

$ PS2="secondary prompt->"

$ echo "this is a

secondary prompt->test"

this is a

test

$

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

$

## Listing Running Processes

It is easy to see your own processes by running the **ps** (process status) command as follows −

$ps

One of the most commonly used flags for ps is the **-f** ( f for full) option, which provides more information as shown in the following example −

$ps -f

## Stopping Processes

Ending a process can be done in several different ways. Often, from a console-based command, sending a CTRL + C keystroke (the default interrupt character) will exit the command. This works when the process is running in the foreground mode.

If a process is running in the background, you should get its Job ID using the **ps** command. After that, you can use the **kill** command to kill the process as follows −

$ps -f

UID PID PPID C STIME TTY TIME CMD

amrood 6738 3662 0 10:23:03 pts/6 0:00 first\_one

amrood 6739 3662 0 10:22:54 pts/6 0:00 second\_one

amrood 3662 3657 0 08:10:53 pts/6 0:00 -ksh

amrood 6892 3662 4 10:51:50 pts/6 0:00 ps -f

$kill 6738

Terminated

Here, the **kill** command terminates the **first\_one** process. If a process ignores a regular kill command, you can use **kill -9** followed by the process ID as follows −

$kill -9 6738

Terminated

## The finger Utility

The **finger** command displays information about users on a given host. The host can be either local or remote.

Finger may be disabled on other systems for security reasons.

Following is the simple syntax to use the finger command −

Check all the logged-in users on the local machine −

$ finger

Login Name Tty Idle Login Time Office

amrood pts/0 Jun 25 08:03 (62.61.164.115)

pwd:print name of current/working directory

cal:displays a calender

shutdown

shutdown -c

b) Study of vi editor.

An improved version of the vi editor which is called the **VIM** has also been made available now. Here, VIM stands for **Vi IM**proved.

Following is an example to create a new file **testfile** if it already does not exist in the current working directory −

$vi testfile

The above command will generate the following output −

|

~

~

~

~

~

~

~

~

~

~

~

~

"testfile" [New File]

A tilde represents an unused line. If a line does not begin with a tilde and appears to be blank, there is a space, tab, newline, or some other non-viewable character present.

You now have one open file to start working on. Before proceeding further, let us understand a few important concepts.

vi always starts in the **command mode**. To enter text, you must be in the insert mode for which simply type **i**. To come out of the insert mode, press the **Esc** key, which will take you back to the command mode.

**Hint** − If you are not sure which mode you are in, press the Esc key twice; this will take you to the command mode. You open a file using the vi editor. Start by typing some characters and then come to the command mode to understand the difference.

## Getting Out of vi

The command to quit out of vi is **:q**. Once in the command mode, type colon, and 'q', followed by return. If your file has been modified in any way, the editor will warn you of this, and not let you quit. To ignore this message, the command to quit out of vi without saving is **:q!**. This lets you exit vi without saving any of the changes.

The command to save the contents of the editor is **:w**. You can combine the above command with the quit command, or use **:wq** and return.

The easiest way to **save your changes and exit vi** is with the ZZ command. When you are in the command mode, type **ZZ**. The **ZZ** command works the same way as the **:wq** command.

If you want to specify/state any particular name for the file, you can do so by specifying it after the **:w**. For example, if you wanted to save the file you were working on as another filename called **filename2**, you would type **:w filename2** and return.

c) Study of Bash shell, Bourne shell and C shell in Unix/Linux operating system.

**Shell** − The shell is the utility that processes your requests. When you type in a command at your terminal, the shell interprets the command and calls the program that you want. The shell uses standard syntax for all commands. C Shell, Bourne Shell and Korn Shell are the most famous shells which are available with most of the Unix variants.

## Shell Prompt

The prompt, **$**, which is called the **command prompt**, is issued by the shell. While the prompt is displayed, you can type a command.

Shell reads your input after you press **Enter**. It determines the command you want executed by looking at the first word of your input. A word is an unbroken set of characters. Spaces and tabs separate words.

Following is a simple example of the **date** command, which displays the current date and time −

$date

Thu Jun 25 08:30:19 MST 2009

You can customize your command prompt using the environment variable PS1 explained in the Environment tutorial.

## Shell Types

In Unix, there are two major types of shells −

* **Bourne shell** − If you are using a Bourne-type shell, the **$** character is the default prompt.
* **C shell** − If you are using a C-type shell, the % character is the default prompt.

The Bourne Shell has the following subcategories −

* Bourne shell (sh)
* Korn shell (ksh)
* Bourne Again shell (bash)
* POSIX shell (sh)

The different C-type shells follow −

* C shell (csh)
* TENEX/TOPS C shell (tcsh)

## Shell Scripts

The basic concept of a shell script is a list of commands, which are listed in the order of execution. A good shell script will have comments, preceded by **#** sign, describing the steps.

## Example Script

Assume we create a **test.sh** script. Note all the scripts would have the **.sh** extension. Before you add anything else to your script, you need to alert the system that a shell script is being started. This is done using the **shebang** construct. For example −

#!/bin/sh

This tells the system that the commands that follow are to be executed by the Bourne shell. *It's called a shebang because the* ***#*** *symbol is called a hash, and the ! symbol is called a bang*.

To create a script containing these commands, you put the shebang line first and then add the commands −

#!/bin/bash

pwd

ls

## Shell Comments

You can put your comments in your script as follows −

#!/bin/bash

# Author : Zara Ali

# Copyright (c) Tutorialspoint.com

# Script follows here:

pwd

ls

Save the above content and make the script executable −

$chmod +x test.sh

The shell script is now ready to be executed −

$./test.sh

Upon execution, you will receive the following result −

/home/amrood

index.htm unix-basic\_utilities.htm unix-directories.htm

test.sh unix-communication.htm unix-environment.htm

**Note** − To execute a program available in the current directory, use **./program\_name**

## Extended Shell Scripts

Shell scripts have several required constructs that tell the shell environment what to do and when to do it. Of course, most scripts are more complex than the above one.

The shell is, after all, a real programming language, complete with variables, control structures, and so forth. No matter how complicated a script gets, it is still just a list of commands executed sequentially.

The following script uses the **read** command which takes the input from the keyboard and assigns it as the value of the variable PERSON and finally prints it on STDOUT.

#!/bin/sh

# Author : Zara Ali

# Copyright (c) Tutorialspoint.com

# Script follows here:

echo "What is your name?"

read PERSON

echo "Hello, $PERSON"

Here is a sample run of the script −

$./test.sh

What is your name?

Zara Ali

Hello, Zara Ali

$

# Unix - Using Shell Variables

In this chapter, we will learn how to use Shell variables in Unix. 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.

## Variable Names

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 will have their names in UPPERCASE.

The following examples are valid variable names −

\_ALI

TOKEN\_A

VAR\_1

VAR\_2

Following are the examples of invalid variable names −

2\_VAR

-VARIABLE

VAR1-VAR2

VAR\_A!

The reason you cannot use other characters such as **!**, **\***, or **-** is that these characters have a special meaning for the shell.

## Defining Variables

Variables are defined as follows −

variable\_name=variable\_value

For example −

NAME="Zara Ali"

The above example defines the variable NAME and assigns the value "Zara Ali" to it. Variables of this type are called **scalar variables**. A scalar variable can hold only one value at a time.

Shell enables you to store any value you want in a variable. For example −

VAR1="Zara Ali"

VAR2=100

## Accessing Values

To access the value stored in a variable, prefix its name with the dollar sign (**$**) −

For example, the following script will access the value of defined variable NAME and print it on STDOUT −

#!/bin/sh

NAME="Zara Ali"

echo $NAME

The above script will produce the following value −

Zara Ali

## Read-only Variables

Shell provides a way to mark variables as read-only by using the read-only command. After a variable is marked read-only, its value cannot be changed.

For example, the following script generates an error while trying to change the value of NAME −

#!/bin/sh

NAME="Zara Ali"

readonly NAME

NAME="Qadiri"

The above script will generate the following result −

/bin/sh: NAME: This variable is read only.

## Unsetting Variables

Unsetting or deleting a variable directs the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you cannot access the stored value in the variable.

Following is the syntax to unset a defined variable using the **unset** command −

unset variable\_name

The above command unsets the value of a defined variable. Here is a simple example that demonstrates how the command works −

#!/bin/sh

NAME="Zara Ali"

unset NAME

echo $NAME

The above example does not print anything. You cannot use the unset command to **unset** variables that are marked **readonly**.

## Variable Types

When a shell is running, three main types of variables are present −

* **Local 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 the command prompt.
* **Environment Variables** − An environment variable 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 Variables** − A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.

# Unix - Special Variables

In this chapter, we will discuss in detail about special variable in Unix. In one of our previous chapters, we understood how to be careful when we use certain nonalphanumeric characters in variable names. This is because those characters are used in the names of special Unix variables. These variables are reserved for specific functions.

For example, the **$** character represents the process ID number, or PID, of the current shell −

$echo $$

The above command writes the PID of the current shell −

29949

The following table shows a number of special variables that you can use in your shell scripts −

|  |
| --- |
| **Variable & Description** |
| **$0** The filename of the current script. |
| **$n** These variables correspond to the arguments with which a script was invoked. Here **n** is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on). |
| **$#** The number of arguments supplied to a script. |
| **$\*** All the arguments are double quoted. If a script receives two arguments, $\* is equivalent to $1 $2. |
| **$@** All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2. |
| **$?** The exit status of the last command executed. |
| **$$** The process number of the current shell. For shell scripts, this is the process ID under which they are executing. |
| **$!** The process number of the last background command. |

## Set Commands

You can change the look and feel of your vi screen using the following **:set** commands. Once you are in the command mode, type **:set** followed by any of the following commands.

|  |  |
| --- | --- |
|  | **Command & Description** |
|  | **:set ic** Ignores the case when searching |
|  | **:set ai** Sets autoindent |
|  | **:set noai** Unsets autoindent |
|  | **:set nu** Displays lines with line numbers on the left side |
|  | **:set sw** Sets the width of a software tabstop. For example, you would set a shift width of 4 with this command — **:set sw = 4** |
|  | **:set ws** If *wrapscan* is set, and the word is not found at the bottom of the file, it will try searching for it at the beginning |
|  | **:set wm** If this option has a value greater than zero, the editor will automatically "word wrap". For example, to set the wrap margin to two characters, you would type this: **:set wm = 2** |
|  | **:set ro** Changes file type to "read only" |
|  | **:set term** Prints terminal type |
|  | **:set bf** Discards control characters from input  d) Study of Unix/Linux file system (tree structure). 1. / – Root  * Every single file and directory starts from the root directory. * Only root user has write privilege under this directory. * Please note that /root is root user’s home directory, which is not same as /.  2. /bin – User Binaries  * Contains binary executables. * Common linux commands you need to use in single-user modes are located under this directory. * Commands used by all the users of the system are located here. * For example: ps, ls, ping, grep, cp.  3. /sbin – System Binaries  * Just like /bin, /sbin also contains binary executables. * But, the linux commands located under this directory are used typically by system aministrator, for system maintenance purpose. * For example: iptables, reboot, fdisk, ifconfig, swapon  4. /etc – Configuration Files  * Contains configuration files required by all programs. * This also contains startup and shutdown shell scripts used to start/stop individual programs. * For example: /etc/resolv.conf, /etc/logrotate.conf  5. /dev – Device Files  * Contains device files. * These include terminal devices, usb, or any device attached to the system. * For example: /dev/tty1, /dev/usbmon0  6. /proc – Process Information  * Contains information about system process. * This is a pseudo filesystem contains information about running process. For example: /proc/{pid} directory contains information about the process with that particular pid. * This is a virtual filesystem with text information about system resources. For example: /proc/uptime  7. /var – Variable Files  * var stands for variable files. * Content of the files that are expected to grow can be found under this directory. * This includes — system log files (/var/log); packages and database files (/var/lib); emails (/var/mail); print queues (/var/spool); lock files (/var/lock); temp files needed across reboots (/var/tmp);  8. /tmp – Temporary Files  * Directory that contains temporary files created by system and users. * Files under this directory are deleted when system is rebooted.  9. /usr – User Programs  * Contains binaries, libraries, documentation, and source-code for second level programs. * /usr/bin contains binary files for user programs. If you can’t find a user binary under /bin, look under /usr/bin. For example: at, awk, cc, less, scp * /usr/sbin contains binary files for system administrators. If you can’t find a system binary under /sbin, look under /usr/sbin. For example: atd, cron, sshd, useradd, userdel * /usr/lib contains libraries for /usr/bin and /usr/sbin * /usr/local contains users programs that you install from source. For example, when you install apache from source, it goes under /usr/local/apache2  10. /home – Home Directories  * Home directories for all users to store their personal files. * For example: /home/john, /home/nikita  11. /boot – Boot Loader Files  * Contains boot loader related files. * Kernel initrd, vmlinux, grub files are located under /boot * For example: initrd.img-2.6.32-24-generic, vmlinuz-2.6.32-24-generic  12. /lib – System Libraries  * Contains library files that supports the binaries located under /bin and /sbin * Library filenames are either ld\* or lib\*.so.\* * For example: ld-2.11.1.so, libncurses.so.5.7  13. /opt – Optional add-on Applications  * opt stands for optional. * Contains add-on applications from individual vendors. * add-on applications should be installed under either /opt/ or /opt/ sub-directory.  14. /mnt – Mount Directory  * Temporary mount directory where sysadmins can mount filesystems.  15. /media – Removable Media Devices  * Temporary mount directory for removable devices. * For examples, /media/cdrom for CD-ROM; /media/floppy for floppy drives; /media/cdrecorder for CD writer  16. /srv – Service Data  * srv stands for service. * Contains server specific services related data. * For example, /srv/cvs contains CVS related data.   e) Study of .bashrc, /etc/bashrc and Environment variables.  **What is /etc/profile used for?**  If you have been using Linux for a while you are probably familiar with the .profile or .bash\_profile files in your home directory. These files are used to set environmental items for a users shell. Items such as umask, and variables such as PS1 or PATH.  The /etc/profile file is not very different however it is used to set system wide environmental variables on users shells. The variables are sometimes the same ones that are in the .bash\_profile, however this file is used to set an initial PATH or PS1 for all shell users of the system.  **/etc/profile.d**  In addition to the setting environmental items the /etc/profile will execute the scripts within /etc/profile.d/\*.sh. If you plan on setting your own system wide environmental variables it is recommended to place your configuration in a shell script within /etc/profile.d.  **What is /etc/bashrc used for?**  Like .bash\_profile you will also commonly see a .bashrc file in your home directory. This file is meant for setting command aliases and functions used by bash shell users.  Just like the /etc/profile is the system wide version of .bash\_profile. The /etc/bashrc for Red Hat and /etc/bash.bashrc in Ubuntu is the system wide version of .bashrc.  Interestingly enough in the Red Hat implementation the /etc/bashrc also executes the shell scripts within /etc/profile.d but only if the users shell is an Interactive Shell (aka Login Shell)  **When are these files used?**  The difference between when these two files are executed are dependent on the type of login being performed. In Linux you can have two types of login shells, Interactive Shells and Non-Interactive Shells. An Interactive shell is used where a user can interact with the shell, i.e. your typical bash prompt. Whereas a non-Interactive shell is used when a user cannot interact with the shell, i.e. a bash scripts execution.  The difference is simple, the /etc/profile is executed only for interactive shells and the /etc/bashrc is executed for both interactive and non-interactive shells. In fact in Ubuntu the /etc/profile calls the /etc/bashrc directly.  **Interactive Shell vs Non-Interactive Shell**  To show an example of an interactive shell vs a non-interactive shell I will add a variable into both /etc/profile and /etc/bash.bashrc on my Ubuntu system.  **/etc/profile**  # grep TEST /etc/profile  export TESTPROFILE=1  **/etc/bash.bashrc**  # grep TEST /etc/bash.bashrc  export TESTBASHRC=1  **Interactive Shell**  The below example is showing an interactive shell, in this case both the /etc/profile and /etc/bash.bashrc was executed.  # su -  # env | grep TEST  TESTBASHRC=1  TESTPROFILE=1  **Non-Interactive Shell**  In this example we are running a command through SSH that is non-interactive; because this is a non-interactive shell only the /etc/bash.bashrc file is executed.  # ssh localhost "env | grep TEST"  root@localhost's password:  TESTBASHRC=1  **Conclusion**  In my case the applications child processes are not recognizing the umask value set in /etc/profile but do recognize the value in /etc/bashrc. This tells me that the subprocess is starting as a non-interactive shell. While the suggested route of modifying environmental variables is to add a shell script into /etc/profile.d in my case it is better to set the umask value in the /etc/bashrc. |

#### 2. Write  a C Program that makes a copy of a file using standard I/O  and system calls.

#include <stdio.h>

#include <unistd.h>

#include <fcntl.h>

#include <stdlib.h>

void typefile (char \*filename)

{

int fd, nread;

char buf[1024];

fd = open (filename, O\_RDONLY);

if (fd == -1) {

perror (filename);

return;

}

while ((nread = read (fd, buf, sizeof (buf))) > 0)

write (1, buf, nread);

close (fd);

}

int

main (int argc, char \*\*argv)

{

int argno;

for (argno = 1; argno < argc; argno )

typefile (argv[argno]);

exit (0);

}

#### Output:

student@ubuntu:~$gcc –o prg10.out prg10.c

student@ubuntu:~$cat > ff

hello

hai

student@ubuntu:~$./prg10.out ff

hello

hai

3. Write a C program to emulate the Unix ls-l command.

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <stdlib.h>

int main()

{

int pid; //process id

pid = fork(); //create another process

if ( pid < 0 )

{ //fail

printf(“\nFork failed\n”);

exit (-1);

}

else if ( pid == 0 )

{ //child

execlp ( “/bin/ls”, “ls”, “-l”, NULL ); //execute ls

}

else

{ //parent

wait (NULL); //wait for child

printf(“\nchild complete\n”);

exit (0);

}

}

#### Output:

guest-glcbIs@ubuntu:~$gcc –o lsc.out lsc.c

guest-glcbIs@ubuntu:~$./lsc.out

total 100

-rwxrwx—x 1 guest-glcbls guest-glcbls 140 2012-07-06 14:55 f1

drwxrwxr-x 4 guest-glcbls guest-glcbls 140 2012-07-06 14:40 dir1

child complete

4. Write a C program that illustrates how to execute two commands concurrently

with a command pipe. Ex: - ls –l | sort

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <stdlib.h>

int main()

{

int pfds[2];

char buf[30];

if(pipe(pfds)==-1)

{

perror("pipe failed");

exit(1);

}

if(!fork())

{

close(1);

dup(pfds[1];

system (“ls –l”);

}

else

{

printf("parent reading from pipe \n");

while(read(pfds[0],buf,80))

printf("%s \n" ,buf);

}

}

#### Output:

[student@gcet ~]$ vi pipes2.c

[student@gcet ~]$ cc pipes2.c

[student@gcet ~]$ ./a.out

Parent reading from pipe

Total 24

-rwxrwxr-x l student student 5563Aug 3 10:39 a.out

-rw-rw-r—l

Student student 340 jul 27 10:45 pipe2.c

-rw-rw-r—l student student

Pipes2.c

-rw-rw-r—l student student 401 34127 10:27 pipe2.c

student

5. Write a C program that illustrates two processes communicating using shared memory

#include<stdio.h>

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/shm.h>

Struct country

{

Char name[30];

Char capital\_city [30];

Char currency[30];

Int population;

};

Int main(int argc,char\*argv[])

{

Int shm\_id;

Char\*shm\_addr;

Int\*countries\_num;

Struct country\*countries;

Struct shmid\_ds shm\_desc;

Shm\_id=shmget(100,2048,IPC\_CREAT|IPC\_EXCL\0600);

If(shm\_id==-1){

Perror(“main:shmget:”);

Exit(1);

}

Shm\_addr=shmat(shm\_id,NULL,0);

If(!shm\_addr){

Perror(“main:shmat:”);

Exit(1);

}

Countries\_num=(int\*)shm\_addr;

\*countries\_num=0;

Countries=(struct country\*)((void\*)shm\_addr sizeof(int));

Strcpy(countries[0],name,”U.S.A”);

Strcpy(countries[0],capital\_city,”WASHINGTON”);

Strcpy(countries[0],currency,”U.S.DOLLAR”);

Countries[0].population=250000000;

( countries\_num) ;

Strcpy(countries[1].name,”israel”);

Strcpy(countries[1].capital\_city,”jerushalem”);

Strcpy(countries[1].currency,”NEW ISRAEL SHEKED”);

Countries[1].population=6000000;

(\*countries\_num) ;

Strcpy(countries[2].name,”France”);

Strcpy(countries[2].capital\_city,”paris”);

Strcpy(countries[2].currency,”Frank”);

Countries[2].population=60000000;

(\*countries\_num) ;

For(i=0;i<(\*countries\_num);i )

{

Printf(“country%d:\n”,i 1);

Printf(“name:%d:\n”,i 1);

Printf(“currency:%s:\n”,countries[i].currency);

Printf(“population:%d:\n”,countries[i].population);

}

If(shmdt(shm\_addr)==-1){

Perror(“main:shmdt:”);

}

If(shmctl(shm\_id,IPC\_RMID,&SHM\_DESC)==-1)

{

Perror(“main:shmctl:”);

}

return 0;

}

#### Output:

Student@ubuntu:~$gcc shm.c

Student@ubuntu:~$ ./a.out

Shared memory ID=65537 child pointer 3086680064

Child value =1

Shared memory ID=65537 child pointer 3086680064

Parent value=1

Parent value=42

Child value=42

6. Write a C program to simulate producer and consumer problem using semaphores

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <time.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

#define NUM\_LOOPS 20

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

{

int sem\_set\_id;

union semun sem\_val;

int child\_pid;

int i;

struct sembuf sem\_op;

int rc;

struct timespec delay;

sem\_set\_id = semget(IPC\_PRIVATE, 1, 0600);

if (sem\_set\_id == -1) {

perror("main: semget");

exit(1);

}

printf("semaphore set created,

semaphore set id '%d'.\n", sem\_set\_id);

sem\_val.val = 0;

rc = semctl(sem\_set\_id, 0, SETVAL, sem\_val);

child\_pid = fork();

switch (child\_pid) {

case 1:

perror("fork");

exit(1);

case 0:

for (i=0; i<NUM\_LOOPS; i ) {

sem\_op.sem\_num = 0;

sem\_op.sem\_op = -1;

sem\_op.sem\_flg = 0;

semop(sem\_set\_id, &sem\_op, 1);

printf("consumer: '%d'\n", i);

fflush(stdout);

sleep(3);

}

break;

default:

for (i=0; i<NUM\_LOOPS; i )

{

printf("producer: '%d'\n", i);

fflush(stdout);

sem\_op.sem\_num = 0;

sem\_op.sem\_op = 1;

sem\_op.sem\_flg = 0;

semop(sem\_set\_id, &sem\_op, 1);

sleep(2);

if (rand() > 3\*(RAND\_MAX/4))

{

delay.tv\_sec = 0;

delay.tv\_nsec = 10;

nanosleep(&delay, NULL);

}

}

break;

}

return 0;

}

#### Output:

Student@ubuntu:~$ gcc sem.c

Student@ubuntu:~$ ./a.out

Semaphore set created

Consumer 0

Consumer 1

Producer 0  Producer 1

7. Write C program to create a thread using pthreads library and let it run its function.

#include<stdio.h>  
#include<stdlib.h>  
#include<pthread.h>  
  
void \*mythread(void \*vargp)  
{  
  sleep(1);  
  printf("welcome to IM CSEIAN ·\n");  
  return NULL;  
}  
int main()  
{  
  pthread\_t tid;  
  printf("before thread\n");  
  pthread\_create(&tid,NULL,mythread,NULL);  
  pthread\_join(tid,NULL);  
  exit(0);  
}  
  
Out Put ::   
  
To Compile   
cc filename.c –l pthread  
To Run  
./a.out  
  
Welcome to IM CSEIAN ·

8. Write a C program to illustrate concurrent execution of threads using pthreads library.

#include<stdio.h>  
#include<stdlib.h>  
#include<pthread.h>  
  
void \*mythread1(void \*vargp)  
{  
   int i;  
   printf("thread1\n");  
      for(i=1;i<=10;i++)  
     printf("i=%d\n",i);       
   printf("exit from thread1\n");  
  return NULL;  
}  
  
void \*mythread2(void \*vargp)  
{  
    int j;  
   printf("thread2 \n");  
   for(j=1;j<=10;j++)  
    printf("j=%d\n",j);  
     
 printf("Exit from thread2\n");  
  return NULL;  
}  
int main()  
{  
  pthread\_t tid;  
  printf("before thread\n");  
  pthread\_create(&tid,NULL,mythread1,NULL);  
  pthread\_create(&tid,NULL,mythread2,NULL);  
  pthread\_join(tid,NULL);   
  pthread\_join(tid,NULL);  
  exit(0);  
}  
  
**OUT PUT ::**  
 **$ cc w8.c – l pthread**  
 **$./a.out**  
**thread1**  
**i=1**  
**i=2;**  
**i=3**  
**thread2**  
**j=1**  
**j=2**  
**j=3**  
**j=4**  
**j=5**  
**j=6**  
**j=7**  
**j=8**  
**i=4**  
**i=5**  
**i=6**  
**i=7**  
**i=8**  
**i=9**  
**i=10**  
**exit from thread1**  
**j=9**  
**j=10**  
**exit from thread2**