CAMELLIA INSTITUTE OF TECHNOLOGY

LAB MANUAL

OPERATING SYSTEM

CODE CS - 693

CONTENTS

UNIX AND SHELL COMMANDS

CODE CS – 591

Assignment 1

(use options of commands as per as possible)

1. Display the users logged in.
2. How can you change the password of the user.
3. How can you know your terminal.
4. Your shell screen is cluttered, how can you clean it.
5. Display the systems date.
6. Display the calendar of October 2010.
7. How can you calculate the following in the shell terminal:
   i) 2 X 12
   ii) 2^{24}
   iii) 24/5 – show the result in two decimal places.
   iv) Convert 11001010 to decimal.
8. How can you know your present working directory.
9. How can you know your home directory.
10. How can you know your shell.
11. Create a file with echo.
12. Display the file you created with echo.
13. Create a file with cat.
14. Create a directory tree like unixlab/cit/students.
15. Go to student directory and create a file named prog.
16. How can you come back to the parent directory.
17. Copy the file you created i.e prog to the parent directory.
18. Delete the directory student.
19. Create a file in the parent directory named prog1 and copy it to the directory students you created.
20. Remove the file prog.
21. Rename the file prog1 to prog2.
22. Move the file prog2 to the parent directory.
23. Rename the file prog2 to prog3 with the command mv.
24. List all the files and directories (i.e. Including the hidden files) in the current directory.
25. List all the files and directories in such a way to distinguish between files and directories.
26. How can we see the manual of any command?
27. Change the permission of the file prog2 so that only the owner can read, write and execute the file and all others and groups can only read it.
28. How can we invoke a child shell terminal from the current shell terminal?

UNIX & SHELL COMMANDS

GENERAL PURPOSE UTILITIES

- Change your password : **passwd**

  Used to change password of a user.

- Know the users : **who** & **w**

  who – used to get the information of the user.

  w – same as who but produces a more detailed information.

- Know your terminal : **tty**

  Used to see the terminal your are logged in. Since UNIX treats terminals as files so we get the output as a file pathname.

- Clear your screen : **clear**
- Display your system date: **date**

- Calendar : **cal**

  $ cal august 2009 - displays the calendar of August of the year 2009

- The calculator: **bc** - press **ctrl+d** to exit the calculator

  a). $ bc

  12+5

  17

  b). $ bc

  12*12;2^32

144

4294967296

c). $ bc

9/5

1 - decimal part truncated

d). by default division performed by **bc** calculatorproduces results with decimal part truncated. We have to use the **scale** to set the number of decimal points.

$ bc

scale=2

15/4

3.75

e). Converting numbers from one base to another

Binary to Decimal

$ bc

ibase=2

11001010

202

Decimal to Binary

$ bc

obase=2

14

1110

- Know your present working directory : **pwd**
- Know your home directory : **echo $HOME**
- Know your shell : **echo $SHELL**
- Directing output to terminal : **echo**

$ echo Hello

Hello

$

1. Creating file with **echo:**

$ echo This is how we learn UNIX > file1

2. Displaying contents of a file with **cat**:

$ cat file1

This is how we learn UNIX

$

3. Creating file with **cat** :

$ cat > file2
> This is how we learn UNIX
[Ctrl+d] **-** ctrl+d is used to terminate input to any console
$

4. Invoking an extra(child) terminal : **xterm &** - the terminal invoked is the child of the present shell

FILE SYSTEM

5. Listing files : **ls**

- a option is used to view all files and directories including the hidden files and directories.
- x to have a columnar view of the listing.
- F provides listing from where directories and files can be identified.
- l details listing is provided.
- R recursive, list all files and subdirectories in a directory tree.

6. Creating Directories : **mkdir**

$ mkdir unixlab

$ mkdir file1 file2 -- multiple directories can be created at once.

7. Changing directories : **cd**

$ cd unixlab

$cd --when cd is used without arguments it redirects to the parent directory.

$cd .. -- moves one level up in the file system hierarchy

$ cd ../.. --moves two levels up

8. Removing directories : rmdir

$ rmdir unixlab

9. Deleting files : **rm**

$ rm file1

$ rm –i file1 --deletes file in an interactive way. Produces a confirmation message.

$ rm –r unixlab -- removes a directory tree in a recursive manner.

10. Renaming or moving a file : **mv**

The mv command has two functions:

1. It can be used to rename a file
2. It can be used to move a file

$ mv file 1 file2 -- renames a file from file1 to file2

$ mv file1 progs -- moves file1 to progs directory

11. Manual : **man**

The man command preceded by a shell command is used to view the manual to that command

Use pgup and pgdown to scroll and press q to quit.

$ man ls

Enter the manual page……...

12. Listing files : **ls**

ls options:

-a -> used to view all the files i.e including the hidden files

-x -> to have a columnar view of the listing

-F -> produces a listing where the directories and executable files are shown

-l -> details listing is provided.

-R -> recursive, lists all files and subdirectories in a directory tree

13. Changing file permission : **chmod**

Abbreviation used by chmod

Category	Operations	Permission
u- user	+ - assign permission	r - read
g – group	- - remove permission	w - write
a – all	= - assign absolute permission	x - executable
o - others		

$ chmod u+rwx,g+rw,o+r file1

Absolute assignment

$ chmod ugo=r file1

$chmod a=x file1

vi EDITOR

13. Opening the vi editor : **vi**

$ vi file1 -- file1 is the name of the file

14. Going into insert mode : **i**

on opening the vi editor we are by default in the command mode, we press i to goto the insert mode. In this mode we can enter our contents.

15. Escaping to command mode: **ESC**

After entering the contents in the vi editor press the ESC button and we move to the command mode.

16. Command mode commands:

Command	Function
:w	Saves file and remains in editing mode
:x	Saves the file and quits editing mode
:wq	Saves the file and quits editing mode
:q	Quits editing mode when changes are saved or no changes are made
:q!	Quits editing mode without saving changes
:w file1	Saves to file named file1
:w! file1	Saves to file named file1, but overwrites file
:w >> file1	Appends current file contents to file note1
:n1,n3w file1	Writes lines n1 to n3 to file named file1
:.w file1	Writes current line to file named file1
:$w file1	Writes last line to file named file1
:sh	Escapes to UNIX shell use exit to return to vi
:dd	Deletes a complete line where the cursor is placed
:dw	Deletes a complete word where the cursor is placed at the first character of the word
:yy	copies a complete line where the cursor is placed
:yw	copies a complete word where the cursor is placed
:p	Paste
:u	Undo
:x	Delete
:a	Append (moves to insert mode)

SHELL PROGRAMMING

CODE CS – 591

Assignment 2

1. Write a shell script that will take input your name and then display it.
2. Write a shell script which will add two integer number and display the result.
3. Write shell script which when run will show the system's current date and the current months calendar.
4. Write a shell script which will copy one file into another, it will take both the source and destination file names from the command line input.
5. Write a shell script which displays a file's content.
6. Write a shell script which displays a file's attributes.
7. Write a shell script which will write and append to a file.
8. Write a shell script which will copy one file into another use if – then – else – fi.
9. Write a shell script to reverse a number use while – do – done.
10. Write a shell script to demonstrate for loops.
11. Write a shell script to print the following pattern:

```
    *
   * *
  * * *
 * * * *
```

12. Write a shell script to demonstrate nested for loops.
13. Write a shell script to demonstrate case control structure.
14. Write a shell script to demonstrate continue and break statement.
15. Write a shell script which uses the "test" command to check whether the inputted number is between 1 and 10.
16. Write a shell script to find an element in an array (linear search).
17. Write a shell script to perform a bubble sort on an array of elements.
18. Write a shell script to demonstrate function.

19. Write a shell script to find factorial of a number using recursion.

SHELL PROGRAMMING

A simple script to demonstrate echo and read

```
#!/bin/bash
echo enter your name
read name
echo your name is $name
```

Program to demonstrate use of expr to perform calculations

```
#!/bin/bash
#program to perform addition
echo enter first number
read a
echo –e “\n \t enter second number \c”
read b
c = `expr $a + $b`
echo –e “\n \t the sum is $c”
```

Program to demonstrate Command substitution

```
#!/bin/bash
echo todays Date is `date`
echo calendar of this month is `cal 9 2009`
```

Program to demonstrate if – then – fi

```
#!/bin/bash
#program to copy a file
echo enter the source file and target file
read f1 f2
if cp $f1 $f2
then
echo file copied successfully
fi
```

Program to display a file's content

i.

```
#!/bin/bash
#program to display a file's content
echo enter the file name
read f
if cat $f
then
echo file found and displayed
fi
```

ii.

```
#!/bin/bash
#program to display a file's content
echo enter the file name
read flname
terminal='tty'
exec<$flname
while read line
```

```
do
echo $line
done
exec<$terminal
```

Program to demonstrate file attributes

```
#!/bin/bash/
echo enter filename
read flname
if [ ! –r $flname ]
then
echo the file is not readable
elif [ ! –w $flname ]
then
echo the file is not writable
elif [ ! –x $flname ]
then
echo the file is not executable
else
echo the file is readable, writable and executable
fi
```

Program to write / append to a file (create a file and give it a write permission by using chmod)

```
#!/bin/bash/
echo enter file name
read flname
```

```
if [ -w $flname ]
then
echo type the matter to append, press CTRL+D to stop
cat>>$flname
else
echo no write permission
fi
```

Program to demonstrate if – then – else - fi

```
#!/bin/bash
#program to copy a file
echo enter the source file and target file
read f1 f2
if cp $f1 $f2
then
echo file copied successfully
else
echo file not found
fi
```

Program to demonstrate while – do – done

```
#!/bin/bash
#program to reverse a number
Sum=0
clear
echo –e “\n \tenter the number \c”
```

```
read n

while [ $n –ne 0]

do

q=`expr $n / 10`

r=`expr $n % 10`

n = $q

sum = `expr $sum \* 10 + r`

done

echo –e “\n the reverse is $sum \c”
```

Program to demonstrate for loops

```
#!/bin/bash
# Listing the planets.
for planet in Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune Pluto
do
echo $planet  # Each planet on a separate line.
done
echo "... Ooops Pluto is not the planet anymnore"
echo
for planet in "Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune Pluto"
# All planets on same line.
# Entire 'list' enclosed in quotes creates a single variable.
do
echo $planet
done
for i in 1 2 3 4 5 6 7  8 9
do
echo $i  # Each planet on a separate line.
done
echo
```

Program to print the pattern of stars

```
#!/bin/bash/
echo enter no of lines
read n
for((i=0;i<=n;i++))
do
for((k=i;k<=n;k++))
do
echo -e " \c"
done
```

```
      *
    *   *
  *   *   *
*   *   *   *
```

```
for((j=0;j<=i;j++))
do
echo -e " *\c"
done
echo -e "\n"
done
```

Program to demonstrate nested for loops

```
# Beginning of outer loop.
for a in 1 2 3 4 5
do
  echo "Pass $a in outer loop."
  echo "---------------------"

  # ============================================
  # Beginning of inner loop.
  for b in 1 2 3
  do
    echo "Pass $b in inner loop."
  done
  # End of inner loop.
  # ============================================

  echo # Space between output blocks in pass of outer loop.
done
# End of outer loop.
```

Program to demonstrate Case control structure

```
#!/bin/bash

#case control structure demonstration

echo –e “MENU\n

            List of files\n  2. Process of user\n

            3.Todays date\n 4. User of system\n

            5.Quit to shell\n  Enter your choice: \c”

      read choice
      case “$choice” in
      1) ls –l;;
        2) ps –f;;
        3) date;;
        4) who;;

        5) exit;;
```

```
    *) echo invalid option

esac
```

Program to demonstrate Continue & Break statement

```
LIMIT=19  # Upper limit
echo
echo "Printing Numbers 1 through 20 (but not 3 and 11)."
a=0
while [ $a -le "$LIMIT" ]
do
 a=$(($a+1))
 if [ "$a" -eq 3 ] || [ "$a" -eq 11 ]  # Excludes 3 and 11.
 then
   continue      # Skip rest of this particular loop iteration.
 fi
 echo -n "$a "   # This will not execute for 3 and 11.
done
# Exercise:
# Why does loop print up to 20?
echo; echo
echo Printing Numbers 1 through 20, but something happens after 2.
##############################################################
# Same loop, but substituting 'break' for 'continue'.
a=0
while [ "$a" -le "$LIMIT" ]
do
 a=$(($a+1))
 if [ "$a" -gt 2 ]
 then
   break  # Skip entire rest of loop.
 fi
 echo -n "$a "
done
echo; echo; echo
```

Program to demonstrate Test command

```
#!/bin/bash

#program to demonstrate test command

echo enter number between 1 to 10

read n
```

```
if test $n –lt 10

then

echo you have entered no. within the range

else

echo you have not entered no. within the range

fi
```

Program to find an element from an array (linear search)

```
#!/bin/bash/

#Linear search
echo enter no of terms
read n
count=0
for((i=0;i<n;i++))
do
echo -e "\n\tenter `expr $i + 1` element"
read v
a[$count]=$v
count=`expr $count + 1`
done
echo -e "\n\t enter elements to find"
read x
count=0
flag=0
while [ $count -lt $n -a $flag -eq 0 ]
do
if [ $x -eq ${a[$count]} ]; then
echo -e "\n\t element found at `expr $count + 1` position"
flag=1
exit
fi
count=`expr $count + 1`
done
if [ $flag -eq 0 ]
then
echo -e "\n\t elment not found"
fi
```

Program to perform bubble sort

```
clear

#set -v
```

```
printf "\n\tEnter the number of elements..."
read num

if test $num -le 0 ; then
  printf "\n\tThe operation can't be computed"
else
  printf "\n\tEnter the elements"
  count=0
  while [ $count -lt $num ]
  do
     printf "\n\tARR[`expr $count + 1`]="
     read ARR[$count]
     count=`expr $count + 1`
  done
  count=0
  printf "\n\tThe inserted elements of the list as follows"
  while [ $count -lt $num ]
  do
     printf "\n\t${ARR[$count]}"
     count=`expr $count + 1`
  done
fi
 count=0
 while [ $count -lt $num ]
 do
    j=0
    n=`expr $num - $count - 1`
    while [ $j -lt $n ]
    do
       if test ${ARR[$j]} -gt ${ARR[$j + 1]} ; then
             temp=${ARR[$j]}
             ARR[$j]=${ARR[$j + 1]}
             ARR[$j + 1]=$temp
       fi
        j=`expr $j + 1`
```

```
  done
 count=`expr $count + 1`
done
count=0

printf "\n\tThe sorted list is as follows"
while [ $count -lt $num ]
do
   printf "\n\t${ARR[$count]}"
   count=`expr $count + 1`
done
printf "\n"
```

Program to demonstrate functions

```
#!/bin/bash
JUST_A_SECOND=1
funky ()
{ # This is about as simple as functions get.
  echo "This is a funky function."
  echo "Now exiting funky function."
} # Function declaration must precede call.
fun ()
{ # A somewhat more complex function.
  i=0
  REPEATS=2
  echo
  echo "And now the fun really begins."
  echo
  sleep $JUST_A_SECOND    # Hey, wait a second!
  while [ $i -lt $REPEATS ]
  do
    echo "----------FUNCTIONS---------->"
    echo "<------------ARE-------------"
    echo "<------------FUN------------>"
    echo
    let "i+=1"
  done
}
  # Now, call the functions.
funky
fun
```

Program to find factorial of a number using recursion (function)

```
#!/bin/bash/
factorial()
{
 If [ “$1” –gt “1” ]; then
l=`expr $1 – 1`
j=`factorial $i`
k=`expr $1 \* $j`
echo $k
   else
echo 1
fi
}
a=0
while [ a –gt 0 ]
do
echo “enter a no.”
read x
factorial $x
done
```

PROCESSES IN UNIX

CODE CS – 591

Assignment 3

1. Write a c program which will print the process ID of the currently running process.
2. Write a program which uses fork() system call to create a child process and display an appropriate message.
3. Write a program to duplicate a program's process.
4. Write a program which uses fork and execv together.
5. Wtite a program which uses exit status of parent and child and writes into a file supplied as command line arguments.
6. Write a program to create a Zombie Process
7. Write a program which cleans up the children by handling SIGCHILD.

PROCESSES IN UNIX

C program to print the Process ID

```
#include <stdio.h>
#include <unistd.h>
int main ()
{
printf (“The process ID is %d\n”, (int) getpid ());
printf (“The parent process ID is %d\n”, (int) getppid ());
return 0;
}
```

Program using fork() to create child processes and display an appropriate message

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

int main()
{
pid_t pid;
char *message;
int n; printf("fork program startingn");
pid = fork();
switch(pid)
{
case -1:
exit(1);
case 0:
message = "this is the child process";
n = 3;
break;
default:
message = "this is the parent process";
n = 6;
break;
}

for(; n > 0; n-) {
puts(message);
sleep(1);
}
exit(0);
}
```

Program using *fork* to Duplicate a Program's Process

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main ()
{
pid_t child_pid;
printf ("the main program process ID is %d\n", (int) getpid ());
child_pid = fork ();
if (child_pid != 0) {
printf ("this is the parent process, with id %d\n", (int) getpid ());
printf ("the child's process ID is %d\n", (int) child_pid);
}
else
printf ("this is the child process, with id %d\n", (int) getpid ());
return 0;
}
```

Program Using *fork* and *exec* Together

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
/* Spawn a child process running a new program. PROGRAM is the name
of the program to run; the path will be searched for this program.
ARG_LIST is a NULL-terminated list of character strings to be
passed as the program's argument list. Returns the process ID of
the spawned process. */
int spawn (char* program, char** arg_list)
{
pid_t child_pid;
/* Duplicate this process. */
child_pid = fork ();
if (child_pid != 0)
/* This is the parent process. */
return child_pid;
else {
/* Now execute PROGRAM, searching for it in the path. */
execvp (program, arg_list);
/* The execvp function returns only if an error occurs. */
fprintf (stderr, "an error occurred in execvp\n");
abort ();
}
}
int main ()
```

```
{
/* The argument list to pass to the “ls” command. */
char* arg_list[] = {
“ls”, /* argv[0], the name of the program. */
“-l”,
“/”,
NULL /* The argument list must end with a NULL. */
};
/* Spawn a child process running the “ls” command. Ignore the
returned child process ID. */
spawn (“ls”, arg_list);
printf (“done with main program\n”);
return 0;
}
```

Program which uses exit status of parent and child and writes into a file supplied as command line arguments.

```
 /* the parent opens a file and writes one line of data to it.*/
/*After forking, the child writes one more line to the same file.*/
/*This should be possible since the child inherits the parent's file descriptors.*/
/*meanwhile, the parent waits for the child to die, and when it does, it invokes*/
/*the WEXITSTATUS macro to gather the exit status from the process table.*/
/*Finally, the parent writes a third line to the file*/
/*wait used to obtain child's termination status*/
/*WEXITSTATUS macro fetches the exit status*/

#include<stdio.h>
#include<fcntl.h>
#include<sys/wait.h>
int main(int argc, char **argv)
{
int fd, exitstatus;
int exitval = 10;       /*value to be returned by the child*/
fd=open(argv[1],O_WRONLY | O_CREAT | O_TRUNC, 0644);
write(fd,"Original process writes\n",24);           /*First write*/
switch(fork())
{
case 0:
      write(fd,"Child writes\n",13);               /*Second write*/
      close(fd);                    /*closing here doesnt affect parent copy*/
      printf("CHILD: terminating with exit value %d\n",exitval);
      exit(exitval);
default:
      wait(&exitstatus);        /* waits for child to die*/
```

```
        printf("PARENT: Child terminated with exit value %d\n",WEXITSTATUS(exitstatus));
        /*Extracting exit status*/
        write(fd,"Parent writes\n",14);  /*Third write*/
        exit(20);     /*Value returned to shell; try echo $?*/
}
}
```

Program to create a Zombie Process

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main ()
{
pid_t child_pid;
/* Create a child process. */
child_pid = fork ();
if (child_pid > 0) {
/* This is the parent process. Sleep for a minute. */
sleep (60);
}
else {
/* This is the child process. Exit immediately. */
exit (0);
}
return 0;
}
```

Cleaning Up Children by Handling *SIGCHLD*

```
#include <signal.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
sig_atomic_t child_exit_status;
void clean_up_child_process (int signal_number)
{
/* Clean up the child process. */
int status;
wait (&status);
/* Store its exit status in a global variable. */
child_exit_status = status;
}
int main ()
{
/* Handle SIGCHLD by calling clean_up_child_process. */
struct sigaction sigchld_action;
```

```
memset (&sigchld_action, 0, sizeof (sigchld_action));
sigchld_action.sa_handler = &clean_up_child_process;
sigaction (SIGCHLD, &sigchld_action, NULL);
/* Now do things, including forking a child process. */
/* ... */
return 0;
}
```

SIGNAL PROGRAMMING IN UNIX

CODE CS – 591

Assignment 4

1. How can we see details of all (including system processes) processes (with their PID and other details) running currently on my system.
2. List the entire signal available in UNIX along with their numbers.
3. How can we send a signal to a specific process.
4. Write a program that will catch the following signal:

   i) SIGINT

   ii) SIGKILL

   iii) SIGHUP

   iv) SIGTERM

   and will display an appropriate message.

SIGNAL

The following command shows you all the processes (including system processes) running in the system along with their PIDs.

ps -e

This program is a simple demonstration of signals. Signals are like sofware interrupts, that are used to inform a process of the occurence of an event. Programs can be designed to catch a signal, by providing a function to handle it.

For example, when shutting down a Linux box, the SIGTERM signal is sent to all processes. Processes that catch this signal, can properly terminate (e.g. de-allocate resources, close all open files).

sighandler_t signal(int signum, sighandler_t handler);

The signal() function associates a specific signal identified by signum with the new function specified by handler.

The signal() function can be used to install a handler for specific signals like SIGINT. SIGINT is the signal sent to the program when the user presses Ctrl-C. If the program should not exit when this happens, and should instead do something else, a new handler function must be associated with the SIGINT signal.

There are a number of signals available. In the program below, three signals are caught by the provided signal handler functions.

Try sending signals to this program through the command-line using the kill utility. First run the program in the background:

./signals-ex &

Then try sending a signal, for example:

kill -SIGHUP pid

Try killing the process:

kill pid

What signal does kill send a process by default?

Replace pid with the one the program displays upon startup.

Associating a signal with SIG_IGN will cause the program to ignore the signal. To reset the default
signal handler, use SIG_DFL, as the second parameter of signal().

Macros defined in <signal.h> header file for common signals.

These include:

SIGHUP 1 /* hangup */
SIGQUIT 3 /* quit */
SIGABRT 6 /* used by abort */
SIGALRM 14 /* alarm clock */
SIGCONT 19 /* continue a stopped process */
SIGINT 2 /* interrupt */
SIGILL 4 /* illegal instruction */
SIGKILL 9 /* hard kill */

Example Program: Program to catch the signals

```
/* Includes */
#include <stdio.h>      /* Input/Output */
#include <stdlib.h>     /* General Utilities */
#include <signal.h>     /* Signal handling */

/* This will be our new SIGINT handler.
   SIGINT is generated when user presses Ctrl-C.
   Normally, program will exit with Ctrl-C.
   With our new handler, it won't exit. */
void mysigint()
{
    printf("I caught the SIGINT signal!\n");
    return;
}

/* Our own SIGKILL handler */
void mysigkill()
{
    printf("I caught the SIGKILL signal!\n");
    return;
}

/* Our own SIGHUP handler */
void mysighup()
{
    printf("I caught the SIGHUP signal!\n");
    return;
}
```

```
/* Our own SIGTERM handler */
void mysigterm()
{
   printf("I caught the SIGTERM signal!\n");
   return;
}

int main()
{
   /* Use the signal() call to associate our own functions with
      the SIGINT, SIGHUP, and SIGTERM signals */
   if (signal(SIGINT, mysigint) == SIG_ERR)
      printf("Cannot handle SIGINT!\n");
   if (signal(SIGHUP, mysighup) == SIG_ERR)
      printf("Cannot handle SIGHUP!\n");
   if (signal(SIGTERM, mysigterm) == SIG_ERR)
      printf("Cannot handle SIGTERM!\n");

   /* can SIGKILL be handled by our own function? */
   if (signal(SIGKILL, mysigkill) == SIG_ERR)
      printf("Cannot handle SIGKILL!\n");

   while(1);  /* infinite loop */

   /* exit */
   exit(0);
} /* main() */
```

When the above program is run, press ctrl+c which is the signal – SIGINT and the program outputs as follows:

OUTPUT:

I caught the SIGINT signal!

Similarly when a kill signal is sent to the program from another terminal the program outputs as follows:

OUTPUT:

I caught the SIGTERM signal

SEMAPHORES IN UNIX

CODE CS – 591

Assignment 5

1. Discuss and explain the following system call functions:
    - semget();
    - semctl();
    - semop();

2. Write a program to demonstrate semget() system call.

3. Write a program to demonstrate semctl() system call.

4. Write a program to demonstrate semop() system call.

SEMAPHORES

Semaphores are the classic method for restricting access to shared resources (e.g. storage) in a multi-processing environment. They were invented by Dijkstra and first used in T.H.E operating system.

A semaphore is a protected variable (or abstract data type) which can only be accessed through the following two atomic (i.e. no two processes can modify a semaphore value concurrently and two statements of P are executed without any intermediate interruption) operations:

```
P(s)
 Semaphore s;
{
  while (s <= 0) do skip;
  s = s-1;
}

V(s)
Semaphore s;
{
  s = s+1;
}

Init(s, v)
Semaphore s;
Int v;
{
  s = v;
}
```

The main system calls that will be needed for the following experiments are:

semget() -- to create a semaphore set

semctl() -- to initialize, view or remove a semaphore set

semop() -- to perform operations on a semaphore set

Creating a Semaphore Set Using semget()

SYSTEM CALL: semget();

PROTOTYPE: int semget (key_t key, int nsems, int semflg);

With `semget()`, the return is a semaphore set id.

- The first argument to semget() is the key value . This key value is then compared to existing key values that exist within the kernel for other semaphore sets. At that point, the open or access operation is dependent upon the contents of the semflg argument.

- **`nsems`** is the number of semaphores in the set

- **`semflg`** might be something like: IPC_CREAT | IPC_EXCL | 0666. This means that you want to create the semaphore, and fail if that semaphore is already created. The digits represent the permissions on the semaphore.

  - **IPC_PRIVATE:** if this is specified, the value of the key is ignored, and a private semaphore set is created, that can be used by a process and its siblings through semaphore id inheritance. Since there's no key, other unrelated processes have no way to access the semaphore set.

  - **IPC_CREAT:** if this is specified, and a semaphore with the given key does not exist, it is created, otherwise the call returns with -1, which indicates failure to create semaphore

  - **IPC_EXCL:** When used with IPC_CREAT, fail if semaphore set already exists.

If IPC_CREAT is used alone, semget() either returns the semaphore set identifier for a newly created set, or returns the identifier for a set which exists with the same key value. If IPC_EXCL is used along with IPC_CREAT, then either a new set is created, or if the set exists, the call fails with -1. IPC_EXCL is useless by itself, but when combined with IPC_CREAT, it can be used as a facility to guarantee that no existing semaphore set is opened for access.

Sample Program: Creating sets of semaphores

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <unistd.h>
main(void)
{
        int sem1, sem2, sem3;
        key_t ipc_key;
        ipc_key = ftok(".", 'S');
        if (( sem1 = semget (ipc_key, 3, IPC_CREAT | 0666))==-1)
        {
                perror("semget: IPC_CREAT | 0666");
        }
        printf("sem1 identifier %d\n", sem1);
        if ((sem2=semget(ipc_key, 3, IPC_CREAT| IPC_EXCL| 0666)) ==-1)
        {
                perror("semget: IPC_CREAT | IPC_EXCL | 0666");
        }
        printf("sem2 identifier %d\n", sem2);
        if ((sem3=semget(IPC_PRIVATE, 3, 0600)) == -1)
        {
                perror("semget: IPC_PRIVATE");
        }
        printf("sem3 identifier %d\n", sem3);
}
```

OUTPUT:
sem1 identifier 0
semget: IPC_CREAT | IPC_EXCL | 0666: File exists
sem2 identifier -1
sem3 identifier 32769

Note the following:

- Each time `semget (ipc_key, 3, IPC_CREAT | 0666)` is called (using the same `ipc_key`), the same semaphore id will be returned. The permission on this semaphore is read and write for user, group and others.

- If `semget(ipc_key, 3, IPC_CREAT| IPC_EXCL| 0666)` is called after the above `semget()` call, an error will occur. This is because the `IPC_EXCL` flag prevents you from generating a semaphore id that is already in use.

- A call like `semget(IPC_PRIVATE, 3, 0600)` will generate a unique semaphore id. This is because `IPC_PRIVATE`, when used as key, guarantees that a unique semaphore id is created.

Initializing, Viewing, and Removing Semaphore Sets Using semctl()

SYSTEM CALL: semctl();

PROTOTYPE: int semctl (int semid, int semnum, int cmd, union semun arg);

This function has three or four arguments.

- The first two arguments identify the semaphore set (semid), and, if relevant, the specific semaphore in the set (semnum, an index).
- cmd specifies what you want to do. These commands are listed in the table that follows.
- The fourth optional argument is of type union semun

The arg argument represents an instance of type semun. This particular union is declared in linux/sem.h as follows:

```
/* arg for semctl system calls. */
union semun {
        int val;               /* value for SETVAL */
        struct semid_ds *buf;   /* buffer for IPC_STAT & IPC_SET */
        ushort *array;         /* array for GETALL & SETALL */
        struct seminfo *__buf;  /* buffer for IPC_INFO */
        void *__pad;
};
```

For a list of commands used with `semctl`, you can refer to the following table:

Command	Description
IPC_STAT	Return the current values of the *semid_ds* structure for the indicated semaphore identifier. The returned information is stored in a user-generated structure referenced by the fourth argument to **semctl**. To specify IPC_STAT the process must have **read** permission for the semaphore set associated with the semaphore identifier.
IPC_SET	Modify a restricted number of members in the *semid_ds* structure. The members *sem_perm.uid*, *sem_perm.gid* and *sem_perm.mode* (in the permission structure within semid_ds) can be changed if the effective ID of the accessing process is that of the super user, or is the same as the ID value stored in sem_perm.cuid or sem_perm.uid. To make these changes, a structure of the type semid_ds must be allocated. The appropriate members' values are then assigned and a reference to the modified structure is passed as the fourth argument to the **semctl** system call.
IPC_RMID	Remove the semaphore set associated with the semaphore identifier
GETALL	Return the current value of the semaphore set. The values are returned via the array reference passed as the fourth argument to **semctl** The user is responsible for allocating the array of the proper size and type prior to passing its address to **semctl** **Read** permissions for the semaphore is required to specify GETALL
SETALL	Initialize all semaphores in a set to the values stored in the array referenced by the fourth argument to **semctl**. Again, the user must allocate the initializing array and assign values prior to passing the address of the array to **semctl**. The process must have **alter** access for the semaphore set to use SETALL.
GETVAL	Return the current value of the individual semaphore referenced by the value of the semnum argument.
SETVAL	Set the value of the individual semaphore referenced by the semnum argument to the value specified by the fourth argument to **semctl**
GETPID	Return the process ID from the sem_perm structure within the semid_ds structure
GETNCNT	Return the number of processes waiting for the semaphore referenced by the semnum argument to increase in value
GETZCNT	Return the number of processes waiting for the semaphore referenced by the semnum argument to become zero

Sample Program: using semctl

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>
#define NS 3
```

```
union semun
{
	int val;
	struct semid_ds *buf;
	ushort  *array;
	struct seminfo *__buf;
};

main(void)
{
	int sem_id, sem_value, i;
	key_t ipc_key;
	struct semid_ds sem_buf;
	static ushort sem_array[NS] = {3,1,4};
	union semun arg;
	ipc_key = ftok(".", 'S');
	/*
	 * Create the semaphore
	 */
	if (( sem_id = semget (ipc_key, NS, IPC_CREAT | 0666))==-1)
	{
		perror("semget: IPC_CREAT | 0666");
		exit (1);
	}
	printf("Semaphore identifier %d\n", sem_id);

	/*
	 * Set arg (the union) to the addr of the storage location for
	 * returned semid_ds values.
	 */
	arg.buf= &sem_buf;
	if (semctl(sem_id, 0, IPC_STAT, arg) == -1) //get info
	{
		perror("semctl:IPC_STAT");
		exit (2);
	}

	printf("Created %s", ctime(&sem_buf.sem_ctime));

	/*
	 * Set arg (the union) to the addr of the initializing vector
	 */
	arg.array=sem_array;
	if (semctl(sem_id, 0, SETALL, arg) == -1)
	{
		perror("semctl: SETALL");
		exit (3);
	}

	for (i = 0; i < NS; ++i) //display contents
```

```
        {
                if ((sem_value = semctl(sem_id, i, GETVAL, 0)) == -1)
                {
                        perror("semctl: GETVAL");
                        exit (4);
                }
                printf("Semaphore %d has value of %d\n", i, sem_value);
        }
        if (semctl(sem_id, 0, IPC_RMID, 0) == -1) //remove semaphore
        {
                perror("semctl: IPC_RMID");
                exit (5);
        }
}
```

OUTPUT:

```
Semaphore identifier 0
Created Thu Apr 22 13:40:03 2010
Semaphore 0 has value of 3
Semaphore 1 has value of 1
Semaphore 2 has value of 4
```

Performing Operations on a Semaphore Set Using semop()

SYSTEM CALL: semop();

PROTOTYPE: int semop (int semid, struct sembuf *sops, unsigned nsops);

- `semid` is the semaphore set identifier
- `sops` is a structure specifying the operation(s) to be performed
- `nsops` is the number of operations in `sops`

The `sops` argument points to an array of type `sembuf`. This structure is declared in `linux/sem.h` as follows:

```
/* semop system call takes an array of these */
struct sembuf {
        ushort  sem_num;        /* semaphore index in array */
        short   sem_op;         /* semaphore operation */
        short   sem_flg;        /* operation flags */
};
```

- **`sem_num`** : The number of the semaphore you wish to deal with
- **`sem_op`** : The operation to perform (positive, negative, or zero)
- **`sem_flg`** : Operational flags

Sample Program: demonstration of semop()

```
/*  semop.c -- demonstrates semaphore usage as a file locking mechanism */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#define MAX_RETRIES 10
union semun {
	int val;
	struct semid_ds *buf;
	ushort *array;
};

/*
** initsem() -- more-than-inspired by W. Richard Stevens' UNIX Network
** Programming 2nd edition, volume 2, lockvsem.c, page 295.
*/
int initsem(key_t key, int nsems)  /* key from ftok() */
{
	int i;
	union semun arg;
	struct semid_ds buf;
	struct sembuf sb;
	int semid;

	semid = semget(key, nsems, IPC_CREAT | IPC_EXCL | 0666);

	if (semid >= 0) { /* we got it first */
		sb.sem_op = 1; sb.sem_flg = 0;
		arg.val = 1;
		printf("press return\n"); getchar();
		for(sb.sem_num = 0; sb.sem_num < nsems; sb.sem_num++) {
			/* do a semop() to "free" the semaphores. */
			/* this sets the sem_otime field, as needed below. */
			if (semop(semid, &sb, 1) == -1) {
				int e = errno;
				semctl(semid, 0, IPC_RMID); /* clean up */
				errno = e;
				return -1; /* error, check errno */
			}
		}

	} else if (errno == EEXIST) { /* someone else got it first */
```

```
                int ready = 0;
                semid = semget(key, nsems, 0); /* get the id */
                if (semid < 0) return semid; /* error, check errno */
                /* wait for other process to initialize the semaphore: */
                arg.buf = &buf;
                for(i = 0; i < MAX_RETRIES && !ready; i++) {
                        semctl(semid, nsems-1, IPC_STAT, arg);
                        if (arg.buf->sem_otime != 0) {
                                ready = 1;
                        } else {
                                sleep(1);
                        }
                }
                if (!ready) {
                        errno = ETIME;
                        return -1;
                }
        } else {
                return semid; /* error, check errno */
        }
        return semid;
}
int main(void)
{
        key_t key;
        int semid;
        struct sembuf sb;
        sb.sem_num = 0;
        sb.sem_op = -1;  /* set to allocate resource */
        sb.sem_flg = SEM_UNDO;
        if ((key = ftok("semdemo.c", 'J')) == -1) {
                perror("ftok");
                exit(1);
        }

        /* grab the semaphore set created by seminit.c: */
        if ((semid = initsem(key, 1)) == -1) {
                perror("initsem");
                exit(1);
        }

        printf("Press return to lock: ");
        getchar();
        printf("Trying to lock...\n");
```

```
        if (semop(semid, &sb, 1) == -1) {
                perror("semop");
                exit(1);
        }

        printf("Locked.\n");
        printf("Press return to unlock: ");
        getchar();

        sb.sem_op = 1; /* free resource */
        if (semop(semid, &sb, 1) == -1) {
                perror("semop");
                exit(1);
        }

        printf("Unlocked\n");

        return 0;
}
```

OUTPUT:

```
[root@localhost]# ./a.out
Press return to lock:
Trying to lock...
Locked.
Press return to unlock:
Unlocked
[root@localhost]#
```

POSIX THREADS IN UNIX

CODE CS – 591

Assignment 6

1. Write a Program which creates two threads and then terminates the threads.

2. Write a program which joins two threads.

3. Discuss the function pthread_create, pthread_exit and pthread_join, you have used in your programs above.

POSIX THREADS

Thread Creation and Termination:

Function call: pthread_create

```
int pthread_create(pthread_t * thread,
                   const pthread_attr_t * attr,
                   void * (*start_routine)(void *),
                   void *arg);
```

- **Arguments:**
  - thread - returns the thread id. (unsigned long int defined in bits/pthreadtypes.h)
  - attr - Set to NULL if default thread attributes are used. (else define members of the struct pthread_attr_t defined in bits/pthreadtypes.h) Attributes include:
    - detached state (joinable? Default: PTHREAD_CREATE_JOINABLE. Other option: PTHREAD_CREATE_DETACHED)
    - scheduling policy (real-time? PTHREAD_INHERIT_SCHED,PTHREAD_EXPLICIT_SCHED,SCHED_OTHER)
    - scheduling parameter
    - inheritsched attribute (Default: PTHREAD_EXPLICIT_SCHED Inherit from parent thread: PTHREAD_INHERIT_SCHED)
    - scope (Kernel threads: PTHREAD_SCOPE_SYSTEM User threads: PTHREAD_SCOPE_PROCESS Pick one or the other not both.)
    - guard size
    - stack address (See unistd.h and bits/posix_opt.h _POSIX_THREAD_ATTR_STACKADDR)
    - stack size (default minimum PTHREAD_STACK_SIZE set in pthread.h),
  - void * (*start_routine) - pointer to the function to be threaded. Function has a single argument: pointer to void.
  - *arg - pointer to argument of function. To pass multiple arguments, send a pointer to a structure.

Example Program 1: pthread1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *print_message_function( void *ptr );
main()
{
    pthread_t thread1, thread2;
    char *message1 = "Thread 1";
    char *message2 = "Thread 2";
    int  iret1, iret2;
   /* Create independent threads each of which will execute function */
    iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
```

```
    iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);
    /* Wait till threads are complete before main continues. Unless we  */
    /* wait we run the risk of executing an exit which will terminate   */
    /* the process and all threads before the threads have completed.   */
    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);
    printf("Thread 1 returns: %d\n",iret1);
    printf("Thread 2 returns: %d\n",iret2);
    exit(0);
}
void *print_message_function( void *ptr )
{
    char *message;
    message = (char *) ptr;
    printf("%s \n", message);
}
```

Compile:

- gcc -lpthread pthread1.c

Run: ./a.out

OUTPUT:

```
        Thread 1
  Thread 2
  Thread 1 returns: 0
  Thread 2 returns: 0
```

Details:

- In this example the same function is used in each thread. The arguments are different. The functions need not be the same.
- Threads terminate by explicitly calling pthread_exit, by letting the function return, or by a call to the function exit which will terminate the process including any threads.

Function call: pthread_exit

void pthread_exit(void *retval);

Arguments:

- retval - Return value of thread.

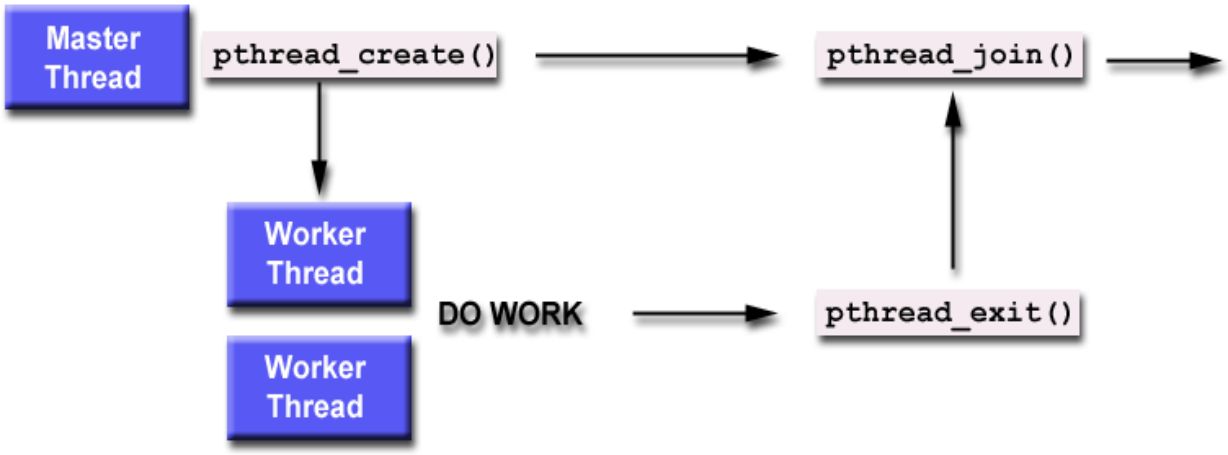
This routine kills the thread. The pthread_exit function never returns. If the thread is not detached, the thread id and return value may be examined from another thread by using pthread_join.
Note: the return pointer *retval, must not be of local scope otherwise it would cease to exist once the thread terminates.

- [C++ pitfalls]: The above sample program **will** compile with the GNU C **and** C++ compiler g++. The following function pointer representation below will work for C but not C++. Note the subtle differences and avoid the pitfall below:

```
void print_message_function( void *ptr );
...
...
iret1 = pthread_create( &thread1, NULL, (void*)&print_message_function, (void*)
message1);
...
...
```

Joining and Detaching Threads

- "Joining" is one way to accomplish synchronization between threads. For example:



- The `pthread_join()` subroutine blocks the calling thread until the specified `threadid` thread terminates.
- The programmer is able to obtain the target thread's termination return `status` if it was specified in the target thread's call to `pthread_exit()`.
- A joining thread can match one `pthread_join()` call. It is a logical error to attempt multiple joins on the same thread.
- Two other synchronization methods, mutexes and condition variables, will be discussed later.

Joinable or Not?

- When a thread is created, one of its attributes defines whether it is joinable or detached. Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.
- The final draft of the POSIX standard specifies that threads should be created as joinable. However, not all implementations may follow this.
- To explicitly create a thread as joinable or detached, the `attr` argument in the `pthread_create()` routine is used. The typical 4 step process is:
    1. Declare a pthread attribute variable of the `pthread_attr_t` data type
    2. Initialize the attribute variable with `pthread_attr_init()`

3. Set the attribute detached status with `pthread_attr_setdetachstate()`

Example Program 2: Pthread Joining

Pthread Joining

This example demonstrates how to "wait" for thread completions by using the Pthread join routine. Since some implementations of Pthreads may not create threads in a joinable state, the threads in this example are explicitly created in a joinable state so that they can be joined later.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 4

void *BusyWork(void *t)
{
   int i;
   long tid;
   double result=0.0;
   tid = (long)t;
   printf("Thread %ld starting...\n",tid);
   for (i=0; i<1000000; i++)
   {
      result = result + sin(i) * tan(i);
   }
   printf("Thread %ld done. Result = %e\n",tid, result);
   pthread_exit((void*) t);
}

int main (int argc, char *argv[])
{
   pthread_t thread[NUM_THREADS];
   pthread_attr_t attr;
   int rc;
   long t;
   void *status;

   /* Initialize and set thread detached attribute */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   for(t=0; t<NUM_THREADS; t++) {
      printf("Main: creating thread %ld\n", t);
      rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
      if (rc) {
         printf("ERROR; return code from pthread_create()
               is %d\n", rc);
         exit(-1);
```

```
        }
      }

   /* Free attribute and wait for the other threads */
   pthread_attr_destroy(&attr);
   for(t=0; t<NUM_THREADS; t++) {
      rc = pthread_join(thread[t], &status);
      if (rc) {
         printf("ERROR; return code from pthread_join()
               is %d\n", rc);
         exit(-1);
         }
      printf("Main: completed join with thread %ld having a status
           of %ld\n",t,(long)status);
      }

printf("Main: program completed. Exiting.\n");
pthread_exit(NULL);
}
```

OUTPUT:

```
Main: creating thread 0
Main: creating thread 1
Thread 0 starting...
Main: creating thread 2
Main: creating thread 3
Thread 1 starting...
Thread 2 starting...
Thread 3 starting...
Thread 0 done. Result = -3.153838e+06
Main: completed join with thread 0 having a status of 0
Thread 1 done. Result = -3.153838e+06
Main: completed join with thread 1 having a status of 1
Thread 2 done. Result = -3.153838e+06
Main: completed join with thread 2 having a status of 2
Thread 3 done. Result = -3.153838e+06
Main: completed join with thread 3 having a status of 3
Main: program completed. Exiting.
```

PIPES & NAMED PIPES IN UNIX

CODE CS – 591

Assignment 7

1. Describe the following system call:

- pipe()
- read()
- write()

2. Write a program which uses a pipe to communicate data between a parent process and its child process. The parent reads input from the user, and sends it to the child via a pipe. The child prints the received data to the screen.

3. Create ca named pipe from the command prompt.

4. Write a program to demonstrate named pipe.

PIPES

A pipe is a one way mechanism that allows two related processes to send a byte stream from one of them to the other one.

The system assures us one thing: the order in which data is written to the pipe, is the same order as that in which data is read from the pipe. The system also assures that data won't get lost in the middle, unless one of the process (the sender or the receiver) exits prematurely.

Creating pipes – pipe() system call

Synopsis

#include <unistd.h>

int pipe(int *filedes***[2]);**

The pipe() system call: this system call is used to create a read- write pipe that may later be used to communicate with a process we'll fork off. The call takes as an argument an array of 2 integers that will be used to save the two file descriptors used to access the pipe. The first to read from the pipe, and the second to write to the pipe.

Example how to use this function:

```
/* first define an array to store the two file descriptors*/
int pipes[2];
/*now create the pipe*/
int rc=pipe(pipes);
if(rc==-1)    /* if the pipe call fails*/
{
  perror("pipe");
  exit(1);
}
```

If the pipe() succeeded, a pipe will be created, pipes[0] will contain the number of its read file descriptor, and pipe[1] will contain the number of its write file descriptor.

read() - read from a file descriptor

Synopsis

#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);

Description

read() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.

If *count* is zero, **read**() returns zero and has no other results. If *count* is greater than SSIZE_MAX, the result is unspecified.

write() - write to a file descriptor

Synopsis

#include <unistd.h>

ssize_t write(int *fd***, const void ***buf***, size_t** *count***);**

Description

write() writes up to *count* bytes to the file referenced by the file descriptor *fd* from the buffer starting at *buf*. POSIX requires that a read() which can be proved to occur after a write() has returned returns the new data. Note that not all file systems are POSIX conforming.

Example Program: Communication between two process using pipes.

```
/*
 * one-way-pipe.c  - example of using a pipe to communicate data between a
 *                   process and its child process. The parent reads input
 *                   from the user, and sends it to the child via a pipe.
 *                   The child prints the received data to the screen.
 */

#include <stdio.h>    /* standard I/O routines.                */
#include <unistd.h>   /* defines pipe(), amongst other things. */

/* this routine handles the work of the child process. */
void do_child(int data_pipe[]) {
    int c;  /* data received from the parent. */
    int rc; /* return status of read().       */

    /* first, close the un-needed write-part of the pipe. */
    close(data_pipe[1]);

    /* now enter a loop of reading data from the pipe, and printing it */
    while ((rc = read(data_pipe[0], &c, 1)) > 0) {
            putchar(c);
    }

    /* probably pipe was broken, or got EOF via the pipe. */
    exit(0);
}

/* this routine handles the work of the parent process. */
void do_parent(int data_pipe[])
{
    int c;  /* data received from the user. */
```

```
    int rc; /* return status of getchar().  */

    /* first, close the un-needed read-part of the pipe. */
    close(data_pipe[0]);

    /* now enter a loop of read user input, and writing it to the pipe. */
    while ((c = getchar()) > 0) {
            /* write the character to the pipe. */
        rc = write(data_pipe[1], &c, 1);
            if (rc == -1) { /* write failed - notify the user and exit */
                perror("Parent: write");
                close(data_pipe[1]);
                exit(1);
        }
    }

    /* probably got EOF from the user. */
    close(data_pipe[1]); /* close the pipe, to let the child know we're done. */
    exit(0);
}

/* and the main function. */
int main(int argc, char* argv[])
{
    int data_pipe[2]; /* an array to store the file descriptors of the pipe. */
    int pid;       /* pid of child process, or 0, as returned via fork.    */
    int rc;        /* stores return values of various routines.           */

    /* first, create a pipe. */
    rc = pipe(data_pipe);
    if (rc == -1) {
            perror("pipe");
            exit(1);
    }

    /* now fork off a child process, and set their handling routines. */
    pid = fork();

    switch (pid) {
            case -1: /* fork failed. */
                perror("fork");
                exit(1);
            case 0:              /* inside child process.  */
                do_child(data_pipe);
                /* NOT REACHED */
            default:  /* inside parent process. */
                do_parent(data_pipe);
                /* NOT REACHED */
    }
```

```
    return 0;          /* NOT REACHED */
}
```

NAMED PIPES

A named pipe is a pipe whose access point is a file kept on the file system. By opening this file for reading, a process gets access to the reading end of the pipe. By opening the file for writing, the process gets access to the writing end of the pipe. If a process opens the file for reading, it is blocked until another process opens the file for writing.

Creating a named pipe

A named pipe may be created either via the “mknod command or via the mknod() system call. To create a named pipe with the file named “pipe-file”, we can use the following command:

- mknod pipe-file p

opening a named pipe is done just like opening any other file in the system, using fopen() standard C function. If the call succeeds, we get a file pointer, which we may use either for reading or for writing, depending on the parameters passed to fopen().

- Either Read or Write – a named pipe cannot be opened for both reading and writing. The process opening it must choose one mode, and stick to it until it closes the pipe.

- Read / Write are Blocking – when a process reads from a named pipe that has no data in it, the reading process is blocked. It does not receive an end of file (EOF) value, like when reading from a file. When a process tries to write to a named pipe that has no reader, the writing process gets blocked, until a second process re-opens the named pipe.

Example Program: Writing to a named pipe.

```
#include <stdio.h>
```

```
#include <unistd.h>
#include<stdlib.h>
#define PLAN_FILE "/home/name-pipes"    /* full path to my "named-pipe" file */

main()
{
    FILE *plan;
    int count = 0;

    /* opening the named pipe for writing */
        plan = fopen(PLAN_FILE, "w");
        if (!plan)
        {
            perror("fopen");
            exit(1);
        }

          /* printing our message into it, and closing it. */
      fprintf(plan,"This is a name pipe example");
      fclose(plan);
      exit(1);
}
```

Example Program: Reading from a named pipe.

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<string.h>
main()
{
   FILE *plan;
   char a[100];

   /* opening the named-pipe for reading */
   plan = fopen("/home/name-pipes", "r");
        if (!plan)
        {
            perror("fopen");
            exit(1);
        }

/* reading contents from the named-pipe into the character array */
    fgets(a,sizeof(a),plan);

/*printing the contents to the terminal */
       fputs(a,stdout);

      fclose(plan);
      exit(1);
}
```

The above programs should be run concurrently on two shell terminal, the program to read from the named pipe should be run on one shell terminal first, then the program to write to the named pipe should be run on another terminal.

The OUTPUT is as follows:

Terminal 1: $./piperead

Terminal 2: $./pipewrite
$

Terminal 1:$ This is a named pipe example
$