

WALCHAND COLLEGE OF ENGINEERING, SANGLI



Department of Information Technology

UNIX OPERATING SYSTEM LAB (3IT 373)

Year of Studentship: 2020-21

Term: Semester-2

Class: T.Y. IT

Name

PRN. No

Mousmi Milind Suryawanshi

2018BTEIT00028

Sarita Gajanan Wadkar

2018BTEIT00047

Certificate

This is certified that

Mousmi Suryawanshi(2018BTEIT00028)

Sarita Wadkar(2018BTEIT00047)

of

*T. Y. I. T. class has completed satisfactorily 13 experiments in Unix Operating
System Lab(4IT373) during the*

Year 2020-2021

Submission Date: 31/05/2021

COURSE TEACHER

Dr. A.J.Umbarkar

HOD

Dr. A.J. Umbarkar

UNIX Operating System Lab. 4IT 373

Course Objectives:

1. To introduce design philosophy of the Unix programming, which is based on the relationship between programs.
2. To make effective use of the programming tools available in Unix environment to build efficient programs.
3. To understand inner details of working of UNIX.
4. To simulate various algorithm of OS.

Percentage of objective achieved by students

Objective No:	Not achieved	40% achieved	70% achieved	100%achieved
1				
2				
3				
4				

Please tick appropriate box.

Course Learning Outcomes:

- a. Learn about Processing Environment
- b. Use of system call to write effective programs
- c. Learn about IPC through signal.
- d. Learn about File System Internals.
- e. Learn shell programming and use it for write effective programs.
- f. Learn and understand the OS interaction with socket programming.
- g. Learn about python as scripting option.
- h. Learn about OpenMP for better use multicore system.

Percentage of Outcomes achieved by students

Outcomes	Not achieved	40% achieved	70% achieved	100%achieved
a				
b				
c				
d				
e				
f				
g				
h				

Please tick appropriate box.

Name Of Student

Roll No

1 Mousmi Suryawanshi

2018BTEIT00028

2 Sarita Wadkar

2018BTEIT00047

UNIX Operating System

Assignment list

Sr. no.	Assignments
1.	<p><u>Processing Environment:</u> fork, vfork, wait, waitpid(),exec (all variations exec), and exit,</p> <p>Objectives:</p> <ol style="list-style-type: none">1.To learn about Processing Environment.2. To know the difference between fork/vfork and various execs variations.3. Use of system call to write effective programs. <ol style="list-style-type: none">a. Write the application or program to open applications of Linux by creating new processes using fork system call. Comment on how various application's/command's process get created in linux. (B)b. Write the application or program to create Childs assign the task to them by variation exec system calls. (B)c. Write the program to use fork/ vfork system call. Justify the difference by using suitable application of fork/vfork system calls. (I)d. Write the program to use wait/ waitpid system call and explain what it do when call in parent and called in child (). Justify the difference by using suitable application. (I) http://www.yolinux.com/TUTORIALS/ForkExecProcesses.htmle. Write the program to use fork/ vfork system call and assign process to work as a shell. OR Read commands from standard input and execute them. Comment on the feature of this programe. (I) Ref: ftp://10.1013.3/pub/UOS.../ OR Ref:www.cs.cf.ac.uk/Dave/C/CE.html <i>OR System call fork/vfork search</i>

-
2. **IPC: Interrupts and Signals:** signal(any five type of signal), alarm, kill, raise, killpg, signal , sigaction, pause

Objectives:

1. To learn about IPC through signal.
 2. To know the process management of Unix/Linux OS
 3. Use of system call to write effective application programs.
-
- a. Write application or program to use alarm and signal system calls such that, it will read input from user within mentioned time (say 10 seconds) ,otherwise terminate by printing message. **(B)**
 - b. Write a application or program that communicates between child and parent processes using kill() and signal().**(I)**
 - c. Write a application or program that communicates between two process opened in two terminal using kill() and signal().**(I)**
 - d. Write a application or program to trap a ctrl-c but not quit on this signal. **(E)**
 - e. Write a program to send signal by five different signal sending system calls and identify the difference in working with example. **(E)**
 - f. Write application of signal handling in linux OS and program any one. **(E)**
- Ref: <ftp://10.1013.3/pub/UOS.../> OR
Ref: www.cs.cf.ac.uk/Dave/C/CE.html
OR System call search
Signal.ppt

-
3. **A. File system Internals:** stat, fstat, ustat, link/unlink, dup

Objectives:

1. To learn about File system Internals.
-
- a. Write the program to show file statistics using the stat system call. Take the filename / directory name from user including path. **(B)**
 - b. Write the program to show file statistics using the fstat system call. Take the file name / directory name from user including path. Print only inode no, UID, GID, FAP and File type only. **(B)**
 - c. Write a program to use link/unlink system call for creating logical link and identifying the difference using stat. **(I)**
 - d. Implement a program to print the various types of file in Linux. (Char, block etc.) **(E)**
- Ref: *System call search*

B. File locking system call : fnctl.h: flock/lockf **(Optional)**

Objectives:

1. To learn about File locking-mandatory and advisory locking.
-
- a. Write a program to lock the file using lockf system call. Check for mandatory locks, the file must be a regular file with the set-group-ID bit on and the group executes

permission off. If either condition fails, all record locks are advisory. (E)

- b. Write a program to lock the file using flock system call. (E)
- c. write a program to lock file using fcntl system call (E)

Ref:

http://techpubs.sgi.com/library/dynaweb_docs/0530/SGI_Developer/boo...

<http://docs.oracle.com/cd/E19963-01/html/821-1602/fileio-9.html>

Book : [Programming Interfaces Guide Beta, Oracle, chapter no 6.](#)

4. **Thread concept:** clone, threads of java

Objectives:

1. To learn about threading in Linux/Unix and Java and difference between them..
 2. Use of system call/library to write effective programs.
-
- a. Write a multithreaded program in java/c for chatting (multiuser and multi-terminal) using threads. (B)
 - b. Write a program which creates 3 threads, first thread printing even number, second thread printing odd number and third thread printing prime number in terminals. (B)
 1. Write program to synchronize threads using construct – monitor/serialize/semaphore of Java (In java only) (I)
 - c. Write a program in Linux to use clone system call and show how it is different from fork system call. (I)
 - d. Write a program using p-thread library of Linux. Create three threads to take odd, even and prime respectively and print their average respectively. (In C) (I)
Ref:<http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html#BASICS>
 - e. Write a program using p thread library of Linux. Create three threads to take numbers and use join to print their average. (in C) (E)
Ref:<http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html#BASICS>
 - f. Write program to synchronize threads using construct – monitor/serialize/semaphore of Java (In java only) (E)
 - g. Write program using semaphore to ensure that function f1 should executed after f2. (In java only) (E)
Ref: <ftp://10.1013.3/pub/UOS..../>
System call search

5. **Shell programming:** shell scripts

Objectives:

1. To learn shell programming and use it for write effective programs.
- a. Write a program to implement a shell script for calculator (B)
- b. Write a program to implement a digital clock using shell script. (B)
- c. Using shell sort the given 10 number in ascending order (use of array). (B)
- d. Shell script to print "Hello World" message, in Bold, Blink effect, and in different colors like red, brown etc. (B)
- e. Shell script to determine whether given file exists or not. (I)
Ref: ftp://10.1013.3/pub/UOS..../
- f. Import python script in shell script. (E)

6.

IPC: Semaphores: semaphore.h-semget, semctl, semop.

Objectives:

1. To learn about IPC through semaphore.
 2. Use of system call and IPC mechanism to write effective application programs.
-
- a. Write a program to illustrate the semaphore concept. Use fork so that 2 process running simultaneously and communicate via semaphore. (B)
Ref: <http://www.cs.cf.ac.uk/Dave/C/node26.html#SECTION00260000000000000000>
Ref: ftp://10.1013.3/pub/UOS..../ OR
Ref: www.cs.cf.ac.uk/Dave/C/CE.html
OR System call search
 - b. Write a program to implement producer consumer problem. (Use suitable synchronization system calls from sem.h/semaphore.h or semaphore of Java) (I)
 - c. Write two programs that will communicate both ways (i.e each process can read and write) when run concurrently via semaphores. (E)
 - d. Write 3 programs separately, 1st program will initialize the semaphore and display the semaphore ID. 2nd program will perform the P operation and print message accordingly. 3rd program will perform the V operation print the message accordingly for the same semaphore declared in the 1st program.
(E) Ref.: http://www.minek.com/files/unix_examples/semab.html

7.

IPC: Message Queues: msgget, msgsnd, msgrcv.

Objectives:

1. To learn about IPC through message queue.
 2. Use of system call and IPC mechanism to write effective application programs.
-

-
- a. Write a program to perform IPC using message and send did u get this? and then reply. (B)
 - b. Write a 2 programs that will both send and messages and construct the following dialog between them
 - (Process 1) Sends the message "Are you hearing me?"
 - (Process 2) Receives the message and replies "Loud and Clear".
 - (Process 1) Receives the reply and then says "I can hear you too". (I)
 - c. Write a *server* program and two *client* programs so that the *server* can communicate privately to *each client* individually via a *single* message queue. (E)
Ref: <ftp://10.1013.3/pub/UOS.../> OR
Ref: www.cs.cf.ac.uk/Dave/C/CE.html
OR System call search
-

8.

IPC: Shared Memory: (shmget, shmat, shmdt)

Objectives:

- 1. To learn about IPC through message queue.
 - 2. Use of system call and IPC mechanism to write effective application programs.
 - a. Write a program to perform IPC using shared memory to illustrate the passing of a simple piece of memory (a string) between the processes if running simultaneously. (B)
 - b. Write 2 programs that will communicate via shared memory and semaphores. Data will be exchanged via memory and semaphores will be used to synchronize and notify each process when operations such as memory loaded and memory read have been performed. (I)
 - c. Write 2 programs. 1st program will take small file from the user and write inside the shared memory. 2nd program will read from the shared memory and write into the file. (E)
Ref: <ftp://10.1013.3/pub/UOS.../> OR
Ref: www.cs.cf.ac.uk/Dave/C/CE.html
OR System call search
-

9.

IPC: Sockets: socket system call in C/socket programming of Java

Objectives:

- 1. To learn about fundamentals of IPC through C socket programming.
 - 2. Learn and understand the OS intraction with socket programming.
 - 3. Use of system call and IPC mechanism to write effective application programs.
 - 4. To know the port numbersing and process relation.
 - 5. To knows the iterative and concurrent server concept.
-

-
- a. Write two programs (server/client) and establish a socket to communicate. (In Java only) (B)

Ref:<http://www.prasannatech.net/2008/07/socket-programming-tutorial.html>

- b. Write programs (server and client) to implement concurrent/iterative server to connect multiple clients requests handled through concurrent/iterative logic using UDP/TCP socket connection. (C or Java only) (use process concept for c server and thread for java server) (B)
- c. Write two programs (server and client) to show how you can establish a TCP socket connection using the above functions. (in C only) (in exam allowed to do in java and python) (I)

Ref:<http://www.prasannatech.net/2008/07/socket-programming-tutorial.html>

- d. Write two programs (server and client) to show how you can establish a UDP socket connection using the above functions. (in C only) (I)

Ref:<http://www.prasannatech.net/2008/07/socket-programming-tutorial.html>

Ref: pdfbooks given

- e. Implement echo server using TCP/UDP in iterative/concurrent logic. (E)
- f. Implement chatting using TCP/UDP socket (between two or more users.) (E)

Other programs are at:

Ref: <ftp://10.1013.3/pub/UOS.../> OR

Ref: www.cs.cf.ac.uk/Dave/C/CE.html

10. **Python:** As a scripting language (Optional)

Objectives:

1. To learn about python as scripting option.

- a. Write a program to display the following pyramid. The number of lines in the pyramid should not be hard-coded. It should be obtained from the user. The pyramid should appear as close to the centre of the screen as possible. (B)

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

(Hint: Basics n loops)

- b. Write a python function for prime number input limit in as parameter to it. (B)

Ref: <ftp://10.1013.3/pub/UOS.../> python by AJU

-
- c. Take any txt file and count word frequencies in a file.(hint : file handling + basics) (I)
 - d. Generate frequency list of all the commands you have used, and show the top 5 commands along with their count. (Hint: history command will give you a list of all commands used.) (I)
 - e. Write a shell script that will take a filename as input and check if it is executable. 2. Modify the script in the previous question, to remove the execute permissions, if the file is executable. (E)
 - f. Generate a word frequency list for wonderland.txt. Hint: use grep, tr, sort, uniq (or anything else that you want) (E)
 - g. Write a bash script that takes 2 or more arguments, i)All arguments are filenames
ii)If fewer than two arguments are given, print an error message
iii)If the files do not exist, print error message
iv)Otherwise concatenate files (E)
 - h. Write a python function for merge/quick sort for integer list as parameter to it. (E)
-

11. **IPC:** MPI(C library for message passing between processes of different systems) Distributed memory programming. (Optional)

Objectives:

1. To learn about IPC through MPI.
 2. Use of IPC mechanism to write effective application programs.
- a. Implement the program IPC/IPS using MPI library. Communication in processes of users. (B)
 - b. Implement the program IPC/IPS using MPI library. Communication in between processes OS's: Unix. (I)
 - c. configure cluster and experiment MPI program on it. (E)
- Ref:
ftp://10.10.13.16/pub/Academic_Material/TYIT/Semister_2/Unix_Operating_System/OpenMP..... OR
Search on internet
-

12. **OpenMP:** (C library for Threading on multicore) shared memory programming. (Optional)

Objectives:

1. To learn about openMP for better use multicore system.
- a. Implement the program for threads using Open MP library. Print number of core. (B)
 - b. Implement the program OpenMP threads and print prime number task, odd number and Fibonacci series using three thread on core. Comment on performance CPU. (I)
 - c. Write a program for Matrix Multiplication in OpenMP. (E)
-

	Ref: ftp://10.10.13.16/pub/Academic_Material/TYIT/Semister_2/Unix_Op erating_System/OpenMP..... OR Search on internet	
13.	<p><u>STREAMS message/PIPEs/FIFO</u>:pipe, popenand pcloseFunctions <u>(Optional)</u></p> <ul style="list-style-type: none"> a. Send data from parent to child over a pipe. (B) b. Filter to convert uppercase characters to lowercase. (B) c. Simple filter to add two numbers. (B) d. Invoke uppercase/lowercase filter to read commands. (I) e. Filter to add two numbers, using standard I/O. (E) f. Client–Server Communication Using a FIFO. (E) <p>Ref: advanced network programming- Stevens .pdf</p> <ul style="list-style-type: none"> g. Routines to let a parent and child synchronize. (E) 	

Chapter 1

Processing Environment

1.1 - Write the application or program to open applications of Linux by creating new processes using fork system call. Comment on how various application's/command's process get created in linux.

Objectives:

1. To learn about Processing Environment.
2. To know the difference between fork/vfork and various execs variations.
3. Use of system call to write effective programs.

Theory:

How various application's/command's process get created in linux?

A new process is created because an existing process makes an exact copy of itself. This child process has the same environment as its parent, only the process ID number is different. This procedure is called *forking*.

After the forking process, the address space of the child process is overwritten with the new process data. This is done through an *exec* call to the system.

The *fork-and-exec* mechanism thus switches an old command with a new, while the environment in which the new program is executed remains the same, including configuration of input and output devices, environment variables and priority. This mechanism is used to create all UNIX processes, so it also applies to the Linux operating system. Even the first process, **init**, with process ID 1, is forked during the boot procedure in the so-called *bootstrapping* procedure.

There are a couple of cases in which **init** becomes the parent of a process, while the process was not started by **init**, as we already saw in the **pstree** example. Many programs, for instance, **daemonize** their child processes, so they can keep on running when the parent stops or is being stopped. A window manager is a typical example; it starts an **xterm** process that generates a shell that accepts commands. The window manager then denies any further responsibility and passes the child process to **init**. Using this mechanism, it is possible to change window managers without interrupting running applications.

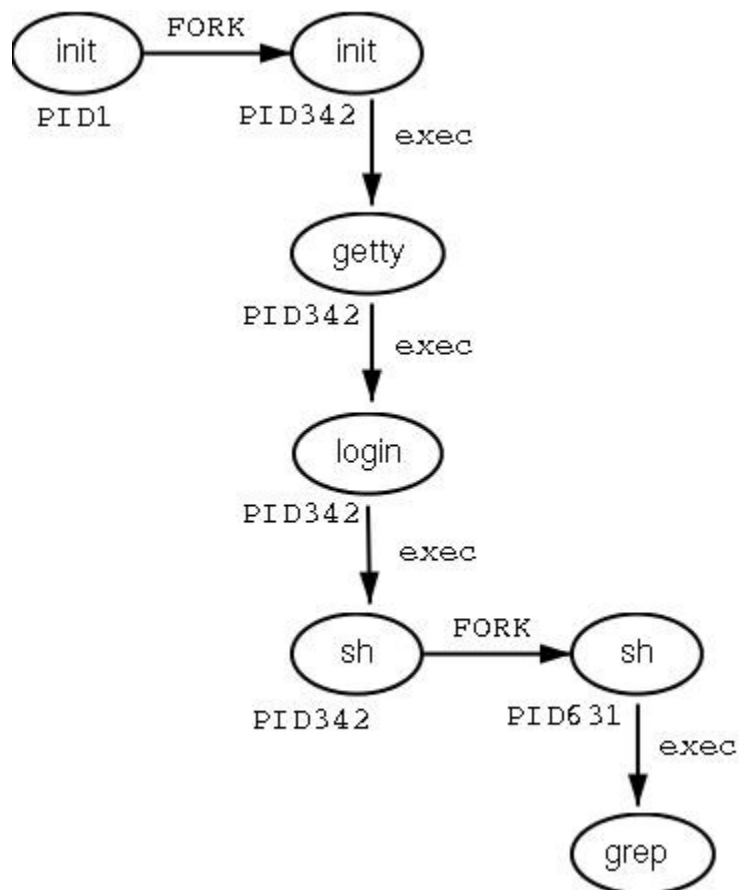
Flowchart:

Fig: 1.1 Flowchart of fork

Data Dictionary:

Sr Number	Variable/Function	Datatype	Use
1	Counter	int	Used to increment number of child and parent processes.
2	pid	int	Process ID

Fig:1.1 Data Dictionary

Program:

```
#include<stdio.h>
#include<unistd.h>
int main()
{
```

```

printf("Beginning\n");
    int counter = 0;
    int pid = fork();
    if(pid==0)
    {
        for(int i=0;i<5;i++)
        {
            printf("Child process = %d\n",++counter);
        }
        printf("Child Ended\n");
    }
    else if(pid>0)
    {
        for(int i=0;i<5;i++)
        {
            printf("Parent process = %d\n",++counter);
        }
        printf("Parent Ended\n");
    }
    else
    {
        printf("fork() failed\n");
        return 1;
    }
    return 0;
}

```

Output:

it@it-OptiPlex-3046:~/Vijay/UOS assignments\$ gcc 1c1.c

it@it-OptiPlex-3046:~/Vijay/UOS assignments\$./a.out

Beginning

Parent process = 1

Parent process = 2

Parent process = 3

Parent process = 4

Parent process = 5

Parent Ended

Child process = 1

Child process = 2

Child process = 3

Child process = 4

Child process = 5

Child Ended

Conclusion:

- Fork system call can be used to create processes from a running process.
- These processes can be made to execute different application programs using various exec statements.

References:

- [1] www.tutorialspoint.com/unix_system_calls/
[2] <https://users.cs.cf.ac.uk/Dave.Marshall/C/node22.html>

1.2- Write the application or program to create Childs assign the task to them by variation exec system calls.

Objectives:

1. To learn about Processing Environment.
2. To know the difference between fork/vfork and various execs variations.
3. Use of system call to write effective programs.

Theory:

fork vs exec:

The use of fork and exec exemplifies the spirit of UNIX in that it provides a very simple way to start new processes

The fork call basically makes a duplicate of the current process, identical in almost every way (not everything is copied over, for example, resource limits in some implementations but the idea is to create as close a copy as possible).

The new process (child) gets a different process ID (PID) and has the PID of the old process (parent) as its parent PID (PPID). Because the two processes are now running exactly the same code, they can tell which is which by the return code of fork - the child gets 0, the parent gets the PID of the child. This is all, of course, assuming the fork call works - if not, no child is created and the parent gets an error code.

The exec call is a way to basically replace the entire current process with a new program. It loads the program into the current process space and runs it from the entry point.

So, fork and exec are often used in sequence to get a new program running as a child of a current process. Shells typically do this whenever you try to run a program like find - the shell forks, then the child loads the find program into memory, setting up all command line arguments, standard I/O and so forth.

Processing Environment:

Process creation

Unless the system is being bootstrapped a process can only come into existence as the child of another process. This done by the fork system call.

The first process created is "hand tooled" by the boot process. This is the swapper process.

The swapper process creates the init process, which is the ancestor of all further processes. In particular, init forks off a process getty, which monitors terminal lines and allows users to log in.

Upon login, the command shell is run as the first process. The command shell for a given user is specified in the /etc/passwd file. From thereon, any process may fork to produce new processes, considered to be children of the forking process.

The process table and uarea

Information about processes is described in two data structures, the kernel process table and a "uarea" associated with each process.

The process table holds information required by the kernel. The uarea holds information required by the process itself.

Flowchart:

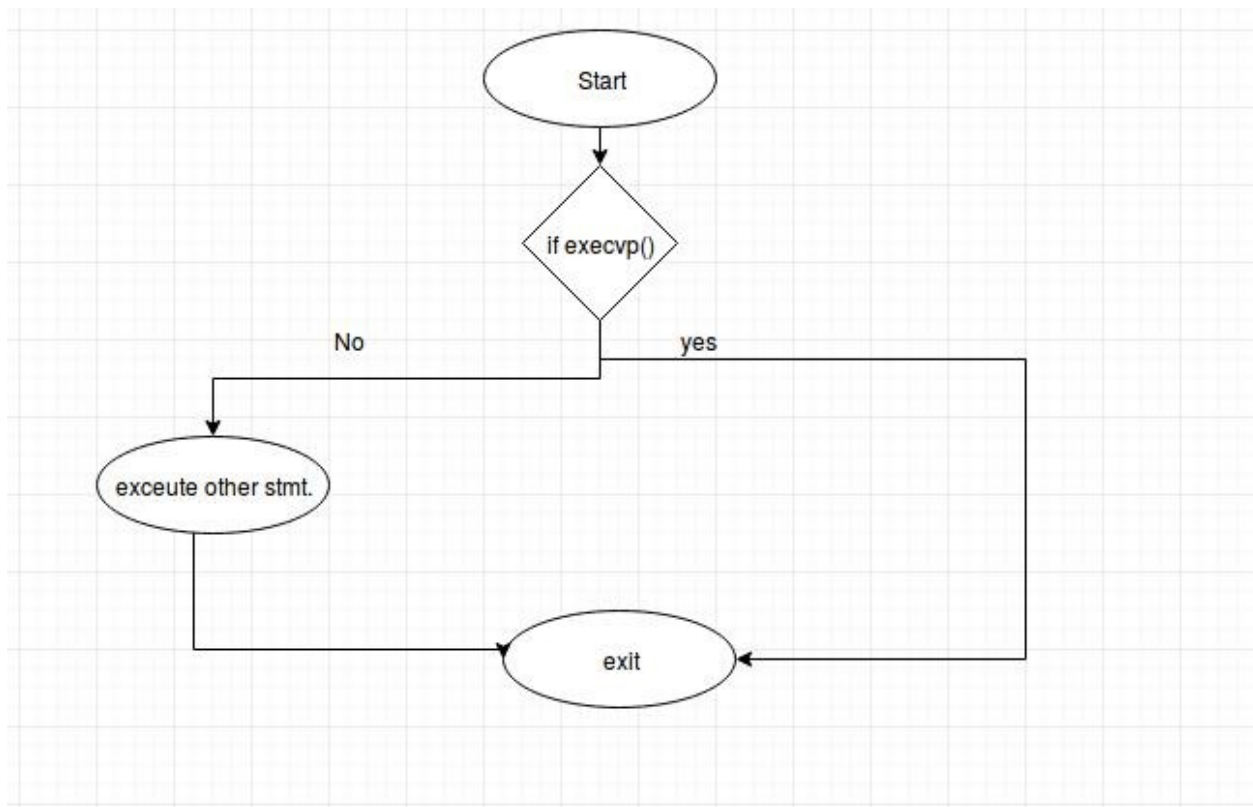


Fig: 1.2 Flowchart

Program:

File execDemo.c:

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main()
{
    char *args[]={"/EXEC",NULL};
    execvp(args[0],args);
    printf("Ending\n");
    return 0;
}
```

File EXEC.c:

```
#include<stdio.h>
#include<unistd.h>
int main()
```

```

{
    printf("I am EXEC.c called by execvp()\n");
    return 0;
}

```

Output:

```

it@it-OptiPlex-3046:~/Vijay/UOS assignments$ gcc EXEC.c -o EXEC it@it-
OptiPlex-3046:~/Vijay/UOS assignments$ gcc execDemo.c -o execDemo it@it-
OptiPlex-3046:~/Vijay/UOS assignments$ ./execDemo I am EXEC.c called by
execvp()

```

Conclusion:

- Learned how to use exec system call.
- Different versions of exec like execvp() can be used to assign task like running another program while fork just creates child which shares resources with parent and runs the same way as the parent.

References:

- [1] www.tutorialspoint.com/unix_system_calls/
 [2] <https://users.cs.cf.ac.uk/Dave.Marshall/C/node22.html>

1.3 - Write the program to use fork/vfork system call. Justify the difference by using suitable application of fork/vfork system calls.

Objectives:

1. To learn about Processing Environment.
2. To know the difference between fork/vfork and various execs variations.
3. Use of system call to write effective programs.

Theory:

fork():

The **fork()** is a system call use to create a **new process**. The new process created by the fork() call is the child process, of the process that invoked the fork() system call. The code of child process is identical to the code of its parent process. After the creation of child process, both process i.e. parent and child process start their execution from the next statement after fork() and both the processes get executed **simultaneously**.

vfork():

The modified version of fork() is vfork(). The **vfork()** system call is also used to create a new process. Similar to the fork(), here also the new process created is the child process, of the process that invoked vfork(). The child process code is also identical to the parent process code. Here, the child process **suspends the execution** of parent process till it completes its execution as both the process share the same address space to use.

Comparison Chart for fork and vfork:

Basis for Comparison	fork()	vfork()
Basic	Child process and parent process has separate address spaces.	Child process and parent process shares the same address space.
Execution	Parent and child process execute simultaneously.	Parent process remains suspended till child process completes its execution.
Modification	If the child process alters any page in the address space, it is invisible to the parent process as the address space are separate.	If child process alters any page in the address space, it is visible to the parent process as they share the same address space.
Copy-on-write	fork() uses copy-on-write as an alternative where the parent and child shares same pages until any one of them modifies the shared page.	vfork() does not use copy-on-write

Table: Comparison between fork and vfork

Flowchart:



Fig: 1.3 Flowchart

Data Dictionary:

Sr Number	Variable/Function	Datatype	Use
1	Counter	int	Used to increment number of child and parent processes.
2	pid	int	Process ID

Table: 1.3 Data Dictionary

Program 1 (fork()):

```
#include<stdio.h>
#include<unistd.h>
int main()
{
    printf("Beginning\n");
    int counter = 0;
    int pid = fork();
    if(pid==0)
    {
        for(int i=0;i<5;i++)
        {
            printf("Child process = %d\n",++counter);
        }
        printf("Child Ended\n");
    }
    else if(pid>0)
    {
        for(int i=0;i<5;i++)
        {
            printf("Parent process = %d\n",++counter);
        }
        printf("Parent Ended\n");
    }
    else
    {
        printf("fork() failed\n");
        return 1;
    }
    return 0;
}
```


Output:

Beginning

Parent process = 1

Parent process = 2

Parent process = 3

Parent process = 4

Parent process = 5

Parent Ended

Child process = 1

Child process = 2

Child process = 3

Child process = 4

Child process = 5

Child Ended

Program 2 (vfork()):

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

```
#include<stdlib.h>
```

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

```
void main()
```

```
{
```

```
    pid_t p = vfork();
```

```
    if(p < 0)
```

```
    {
```

```
        printf("vfork() failed\n");
```

```
    }
```

```
    else if(p == 0)
```

```
    {
```

```
        printf("In Child Process Started with pid = %d\n",getpid());
```

```
        for(int i=1;i<=5;i++)
```

```

        {
            printf("In Child : %d\n",i);
        }
        printf("Child Finished\n");
        exit(0);
    }
    else
    {
        printf("Parent Process Starded with pid = %d\n",getpid());
        for(int i=1;i<=5;i++)
        {
            printf("In Parent : %d\n",i);
        }
        printf("Parent Finished\n");
    }
}

```

Output:

In Child Process Starded with pid = 2501

In Child : 1

In Child : 2

In Child : 3

In Child : 4

In Child : 5

Child Finished

Parent Process Starded with pid = 2500

In Parent : 1

In Parent : 2

In Parent : 3

In Parent : 4

In Parent : 5

Conclusion:

- Learned how to use fork and vfork system calls.
- fork() and vfork() system calls have some differences which allows different type of execution of child processes.

References:

[1]www.tutorialspoint.com/unix_system_calls/

[2]<https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/14/lec.html>

1.4 - Write the program to use wait/waitpid system call and explain what it do when call in parent

Objectives:

1. To learn about Processing Environment.
2. To know the difference between fork/vfork and various execs variations.
3. Use of system call to write effective programs.

Theory:**Declarations:**

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

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

```
pid_t wait(int *statloc);
```

```
/* returns process ID if OK, or -1 on error */
```

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

```
/* returns process ID : if OK,
```

```
* 0 : if non-blocking option && no zombies around
```

```
* -1 : on error
```

```
*/
```

wait()	waitpid()
wait blocks the caller until a child process terminates	<p>waitpid can be either blocking or non-blocking:</p> <ul style="list-style-type: none"> • If <i>options</i> is 0, then it is blocking • If <i>options</i> is WNOHANG, then is it non-blocking
if more than one child is running then wait() returns the first time one of the parent's offspring exits	<p>waitpid is more flexible:</p> <ul style="list-style-type: none"> • If <i>pid</i> == -1, it waits for any child process. In this respect, waitpid is equivalent to wait • If <i>pid</i> > 0, it waits for the child whose process ID equals pid • If <i>pid</i> == 0, it waits for any child whose process group ID equals that of the calling process • If <i>pid</i> < -1, it waits for any child whose process group ID equals that absolute value of pid

Table: Comparison between wait and waitpid

Flowchart:

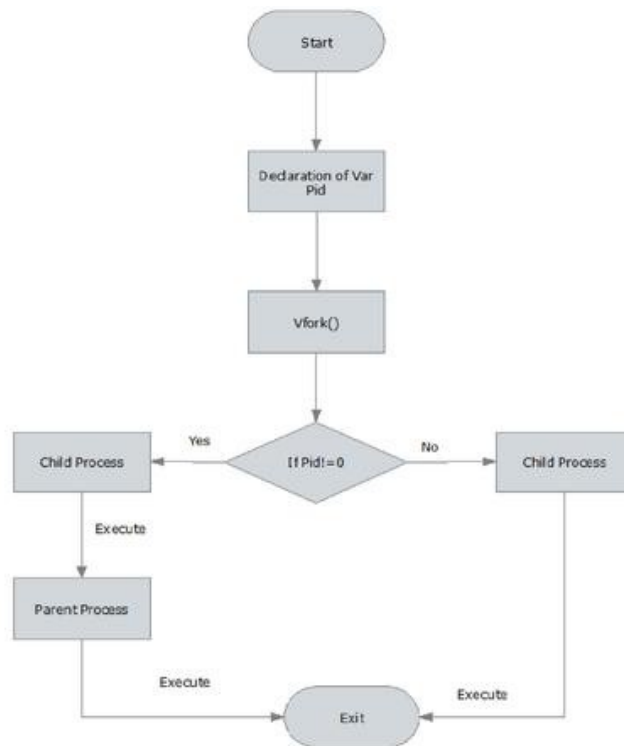


Fig: 1.4 Flowchart

Data Dictionary:

Sr Number	Variable/Function	Datatype	Use
1	id	pid_t	Process ID
2	i	int	Iterating for loop.

Table: 1.4 Data Dictionary

Program:

```
#include<stdlib.h>
#include<stdio.h>
#include<unistd.h>
#include<sys/wait.h>
void main()
{
    pid_t id=fork();
    if(id==0)
```

```

{
    printf("Child Process Started..ProcessID = %d\n", getpid());
    printf("In Child\n");
    for(int i=0;i<5;i++)
    {
        printf("In Child : %d\n",i);
    }
    printf("Child Finished\n");
    exit(0);
}
else
{
    printf("Parent Process Started..ProcessID = %d\n", getpid());
    printf("In Parent\n");
    printf("Parent waiting\n");
    wait(NULL);
    printf("Parent Resumed\n");
    for(int i=0;i<5;i++)
    {
        printf("In parent : %d\n",i);
    }
    printf("Parent Finished\n");
}
}

```

Output:

it@it-OptiPlex-3046:~/Vijay/UOS assignments\$ gcc 1d.c

it@it-OptiPlex-3046:~/Vijay/UOS assignments\$./a.out

Parent Process Started..ProcessID = 2530 In Parent

Parent waiting

Child Process Started..ProcessID = 2531

In Child

In Child : 0

In Child : 1

In Child : 2

In Child : 3

In Child : 4

Child Finished

Parent Resumed

In parent : 0

In parent : 1

In parent : 2

In parent : 3

In parent : 4

Parent Finished

Conclusion:

- Learned how to use wait and waitpid system call
- The waitpid() call is more flexible than wait() system call as wait() would block the parent until child processes complete, while waitpid() can be implemented in blocking or unblocking ways.

References:

[1]www.tutorialspoint.com/unix_system_calls/

[2]<https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/14/lec.html>

1.5 - Write the program to use fork/vfork system call and assign process to work as a shell. OR Read commands from standard input and execute them. Comment on the feature of this program

Objectives:

1. To learn about Processing Environment.
2. To know the difference between fork/vfork and various execs variations.
3. Use of system call to write effective programs.

Theory:

Name:

system - execute a shell command

Syntax:

```
#include <stdlib.h>
```

```
int system(const char *command);
```

Description:

system() executes a command specified in *command* by calling `/bin/sh -c command`, and returns after the command has been completed. During execution of the command, SIGCHLD will be blocked, and SIGINT and SIGQUIT will be ignored.

Return Value:

The value returned is -1 on error (e.g., *fork(2)* failed), and the return status of the command otherwise. This latter return status is in the format specified in *wait(2)*. Thus, the exit code of the command will be *WEXITSTATUS(status)*. In case `/bin/sh` could not be executed, the exit status will be that of a command that does *exit(127)*.

If the value of *command* is NULL, system() returns nonzero if the shell is available, and zero if not.

system() does not affect the wait status of any other children.

Data Dictionary:

Sr Number	Variable/Function	Datatype	Use
1	str	char[]	Input command.

Table: 1.5 Data Dictionary

Program:

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

```
#include<stdlib.h>
```

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

```
int main()
```

```
{
```



```
char str[100];  
printf("Input command\n");  
scanf("%s", str);  
printf("Thank you\n");  
}
```

Output:

Input command
history
sh: 1: history: not found
Thank you

Input command
sudo apt install fortune
[sudo] password for it:
Reading package lists... Done
Building dependency tree
Reading state information... Done
Note, selecting 'fortune-mod' instead of 'fortune'
fortune-mod is already the newest version (1:1.99.1-7).
0 upgraded, 0 newly installed, 0 to remove and 4 not upgraded.
Thank you

Input command
fortune
Your present plans will be successful.
Thank you

Conclusion:

- system() can be used to perform various shell commands when the commands are read from standard input. The output of shell is printed.

References:

- [1] www.tutorialspoint.com/unix_system_calls/
[2] <https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/14/lec.html>

Chapter 2

IPC: Interrupts and Signals

2.1 - Write application or program to use alarm and signal system calls such that, it will read input from user within mentioned time (say 10 seconds) ,otherwise terminate by printing message.

Objectives:

1. To learn about IPC through signal.
2. To know the process management of Unix/Linux OS
3. Use of system call to write effective application programs.

Theory:

1) Alarm:

Name:

alarm - set an alarm clock for delivery of a signal

Syntax:

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

```
unsigned int alarm(unsigned int seconds);
```

Description:

alarm() arranges for a SIGALRM signal to be delivered to the process in *seconds* seconds.

If *seconds* is zero, no new alarm() is scheduled.

In any event any previously set alarm() is cancelled.

Return Value:

alarm() returns the number of seconds remaining until any previously scheduled alarm was due to be delivered, or zero if there was no previously scheduled alarm.

Notes:

alarm() and setitimer() share the same timer; calls to one will interfere with use of the other.

sleep() may be implemented using SIGALRM; mixing calls to alarm() and sleep() is a bad idea.

Scheduling delays can, as ever, cause the execution of the process to be delayed by an arbitrary amount of time.

2) Signal:

Name:

signal - ANSI C signal handling

Syntax:

```
#include <signal.h>
```

```
typedef void (*sighandler_t)(int);
```

```
sighandler_t signal(int signum, sighandler_t handler);
```

Description:

The behavior of signal() varies across UNIX versions, and has also varied historically across different versions of Linux. Avoid its use: use [sigaction\(2\)](#) instead. See *Portability* below.

signal() sets the disposition of the signal *signum* to *handler*, which is either SIG_IGN, SIG_DFL, or the address of a programmer-defined function (a "signal handler"). If the signal *signum* is delivered to the process, then one of the following happens:

If the disposition is set to SIG_IGN, then the signal is ignored.

If the disposition is set to SIG_DFL, then the default action associated with the signal occurs.

If the disposition is set to a function, then first either the disposition is reset to SIG_DFL, or the signal is blocked (see *Portability* below), and then *handler* is called with argument *signum*. If invocation of the handler caused the signal to be blocked, then the signal is unblocked upon return from the handler. The signals SIGKILL and SIGSTOP cannot be caught or ignored.

Return Value:

signal() returns the previous value of the signal handler, or SIG_ERR on error. In the event of an error, errno is set to indicate the cause.

Data Dictionary:

Sr Number	Variable/Function	Datatype	Use
1	alarmhandle	void	Used to handle alarm.
2	A	int	Input.

Table: 2.1 Data Dictionary

Program:

```
#include<signal.h>
```

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

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

```
#include<stdbool.h>
```

```
#include<stdlib.h>
```

```
bool flag=false;
```

```
void alarmhandle(int sig)
```

```
{
```

```
    printf("Input time expired\n");
```

```
    exit(1);
```

```
}
```

```
int main()
```

```
{
```

```
    int a=0;
```

```
    printf("Input now in 10 seconds\n");
```

```
    sleep(1);
```

```
    alarm(10);
```

```
    signal(SIGALRM,alarmhandle);
```

```
    scanf("%d",&a);
```

```
    printf("You entered %d\n",a);  
}
```

Output:

Input now in 10 seconds

13

You entered 13

Input now in 10 seconds

Input time expired

Conclusion:

- alarm() signal can be used to raise alarm after particular time period.
- Signal() system call is evoked by alarm() which is further processed by signal handler.

References:

[1]www.tutorialspoint.com/unix_system_calls/

[2]<https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/14/lec.html>

2.2 - Write a application or program that communicates between child and parent processes using kill() and signal().**Objectives:**

1. To learn about IPC through signal.
2. To know the process management of Unix/Linux OS
3. Use of system call to write effective application programs.

Theory:

1) Kill:

Name:

kill - send signal to a process

Syntax:

```
#include <sys/types.h> #include  
<signal.h> int kill(pid_t pid, int  
sig);
```

Description:

The kill() system call can be used to send any signal to any process group or process.

If *pid* is positive, then signal *sig* is sent to *pid*.

If *pid* equals 0, then *sig* is sent to every process in the process group of the current process.

If *pid* equals -1, then *sig* is sent to every process for which the calling process has permission to send signals, except for process 1 (init), but see below.

If *pid* is less than -1, then *sig* is sent to every process in the process group *-pid*.

If *sig* is 0, then no signal is sent, but error checking is still performed.

For a process to have permission to send a signal it must either be privileged (under Linux: have the CAP_KILL capability), or the real or effective user ID of the sending process must equal the real or saved set-user-ID of the target process. In the case of SIGCONT it suffices when the sending and receiving processes belong to the same session.

Return Value:

On success (at least one signal was sent), zero is returned. On error, -1 is returned, and *errno* is set appropriately.

2) Signal:

Name:

signal - ANSI C signal handling

Syntax:

```
#include <signal.h>  
typedef void (*sighandler_t)(int);  
sighandler_t signal(int signum, sighandler_t handler);
```

Description:

The behavior of `signal()` varies across UNIX versions, and has also varied historically across different versions of Linux. Avoid its use: use `sigaction(2)` instead. See *Portability* below.

`signal()` sets the disposition of the signal *signum* to *handler*, which is either `SIG_IGN`, `SIG_DFL`, or the address of a programmer-defined function (a "signal handler"). If the signal *signum* is delivered to the process, then one of the following happens:

If the disposition is set to `SIG_IGN`, then the signal is ignored.

If the disposition is set to `SIG_DFL`, then the default action associated with the signal occurs.

If the disposition is set to a function, then first either the disposition is reset to `SIG_DFL`, or the signal is blocked (see *Portability* below), and then *handler* is called with argument *signum*. If invocation of the handler caused the signal to be blocked, then the signal is unblocked upon return from the handler. The signals `SIGKILL` and `SIGSTOP` cannot be caught or ignored.

Return Value:

`signal()` returns the previous value of the signal handler, or `SIG_ERR` on error. In the event of an error, *errno* is set to indicate the cause.

Flowchart:

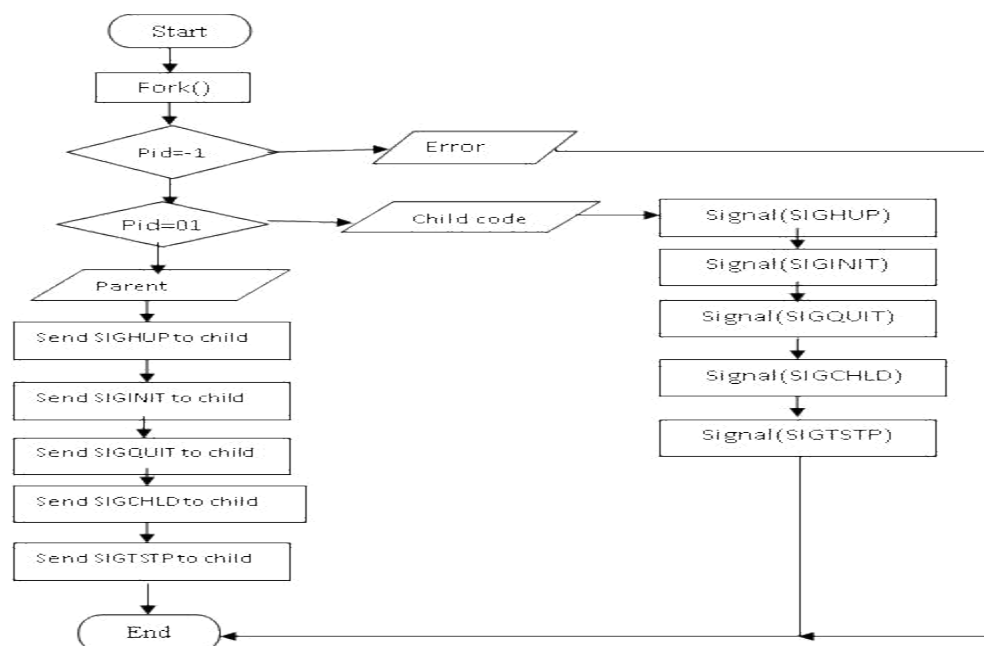


Fig: 2.2 Flowchart

Data Dictionary:

Sr Number	Variable/Function	Datatype	Use
1	inthandle	void	Used for handling interrupt signal.
2	quithandle	void	Used for dumped core handling.
3	huphandle	void	Used for handling signal hang up.
4	pid	pid_t	Used for child ID
5	ppid	pid_t	Used for parent ID

Table: 2.2 Data Dictionary

Program:

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<signal.h>
void inthandle(int sig)
{
    signal(SIGINT,inthandle);
    printf("SIGINT invoked by daughter\n");
}

void quithandle(int sig)
{
    signal(SIGQUIT,quithandle);
    printf("SIGQUIT invoked by son\n He killed me\n");
    exit(1);
}
```



```

void huphandle(int sig)
{
    signal(SIGHUP,huphandle);
    printf("SIGHUP invoked by child\n");
}

int main()
{
    pid_t ppid,pid;
    ppid=getpid();
    if((pid=vfork())<0)
    {
        printf("Fork Failed!!!\n");
        exit(1);
    }
    else if(pid==0)

    {
        printf("In Child!!!\n");
        signal(SIGINT,inthandle);
        signal(SIGHUP,huphandle);
        signal(SIGQUIT, quithandle);
        printf("Looping\n");
        for(;;);
    }
    else
    {
        printf("In Parent!!!\n");
        printf("kill SIGHUP\n");
        kill(pid,SIGHUP);
        sleep(2);
        printf("kill SIGINT\n");
        kill(pid,SIGINT);
        sleep(2);
        printf("kill SIGQUIT\n");
    }
}

```

```
        kill(pid,SIGQUIT);
        sleep(2);
        //exit(0);
    }
}
```

Output:

In Parent!!!!

kill SIGHUP

kill SIGINT

kill SIGQUIT

In Child!!!!

Looping

^C

SIGINT invoked by daughter

Conclusion:

- Various signal interrupts can be used in the form of signal handler and kill() can be used to evoke these signals to abort processes with different interrupts.

References:

[1]www.tutorialspoint.com/unix_system_calls/

[2]<https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/14/lec.html>

2.3 - Write an application or program that communicates between two processes opened in two terminals using kill() and signal().**Objectives:**

1. To learn about IPC through signals.
2. To know the process management of Unix/Linux OS
3. Use of system calls to write effective application programs.

Theory:

In Unix and Unix-like operating systems, kill is a command used to send a signal to a process. By default, the message sent is the termination signal, which requests that the process exit. But kill is something of a misnomer; the signal sent may have nothing to do with process killing. The kill command is a wrapper around the kill() system call, which sends signals to processes or process groups on the system, referenced by their numeric process IDs (PIDs) or process group IDs (PGIDs). kill is always provided as a standalone utility as defined by the POSIX standard. However, most shells have built-in kill commands that may slightly differ from it.

Data Variables:

Sr Number	Variable/Function	Datatype	Use
1	SIGINT_handler	void	Used for handling interrupt signal.
2	SIGQUIT_handler	void	Used for dumped core handling.
3	pid	pid_t	Process ID.
4	MyKey	key_t	Shared memory key.
5	ShmID	int	ID of shared memory.
6	ShmPTR	pid_t*	Pointer.
7	i	int	Iterating for loop.

Table: 2.3 Data Dictionary

Program:

```
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>
#include <sys/ipc.h>
#include <sys/shm.h>
void SIGINT_handler(int);
void SIGQUIT_handler(int);

int ShmID;
pid_t *ShmPTR;
```

```

void main(void)
{
    int i;
    pid_t pid = getpid();
    key_t MyKey;

    if (signal(SIGINT, SIGINT_handler) == SIG_ERR) {
        printf("SIGINT install error\n");
        exit(1);
    }
    if (signal(SIGQUIT, SIGQUIT_handler) == SIG_ERR)
    { printf("SIGQUIT install error\n");
      exit(2);
    }

    MyKey = ftok(".", 's');
    ShmID = shmget(MyKey, sizeof(pid_t), IPC_CREAT | 0666);
    ShmPTR = (pid_t *) shmat(ShmID, NULL, 0);
    *ShmPTR = pid;

    for (i = 0; ; i++) {

printf("From process %d: %d\n", pid, i);
        sleep(1);
    }
}

void SIGINT_handler(int sig)
{
    signal(sig, SIG_IGN);
    printf("From SIGINT: just got a %d (SIGINT ^C) signal\n",
sig); signal(sig, SIGINT_handler);
}

void SIGQUIT_handler(int sig)
{
    signal(sig, SIG_IGN);
    printf("From SIGQUIT: just got a %d (SIGQUIT ^\\) signal"
        " and is about to quit\n",
sig); shmdt(ShmPTR);
    shmctl(ShmID, IPC_RMID, NULL);

    exit(3);
}

```

Conclusion:

- Processes opened in two terminals can also be handled using signal handlers and kill() function calls.
- Shared memory can be used as a mode of IPC.

References:

[1] <http://www.csl.mtu.edu/cs4411.ck/www/NOTES/signal/kill.html>

[2] <https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/14/lec.html>

2.4 Write application or program to trap a ctrl-c but not quit on this signal.

Objectives:

1. To learn about IPC through signal.
2. To know the process management of Unix/Linux OS
3. Use of system call to write effective application programs.

Theory:

1. We can use signal handling in C for this. When Ctrl+C is pressed, SIGINT signal is generated, we can catch this signal and run our defined signal handler.
2. C standard defines following 6 signals in signal.h header file.

SIGABRT – abnormal termination.

SIGFPE – floating point exception.

SIGILL – invalid instruction.

SIGINT – interactive attention request sent to the program.

SIGSEGV – invalid memory access.

SIGTERM – termination request sent to the program.

Data Dictionary:

Sr Number	Variable/Function	Datatype	Use
1	sigproc	void	Function to disable Ctrl-c
2	quitproc	void	Function to quit using Ctrl-backslash.
3	quitprocess	void	Function to quit using Ctrl-z.

Table : 2.4 Data Dictionary

Program:

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

```

#include <unistd.h>

int ctrl_c_count = 0;
void (* old_handler)(int);
void ctrl_c(int);

int main()
{
    int c;
    old_handler = signal(SIGINT, ctrl_c);
    while ((c = getchar())!='\n');
        printf("ctrl-c count = %d\n",
            ctrl_c_count); (void) signal(SIGINT, ctrl_c); for
        (;;);
    return 0;
}
void ctrl_c(int signum)
{
    //(void) signal(SIGINT, ctrl_c);
    ++ctrl_c_count;
}

```

Conclusion:

- Signals like SIGINT which is generated by ctrl+c can be handled by replacing its old handler with the new behavior.

References:

- [1]<https://www.geeksforgeeks.org/write-a-c-program-that-doesnt-terminate-when-ctrlc-is-pressed/>
- [2]<https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/14/lec.html>

2.5- Write a program to send signal by five different signal sending system calls and identify the difference in working with example.

Objectives:

1. To learn about IPC through signal.
2. To know the process management of Unix/Linux OS
3. Use of system call to write effective application programs.

Theory:

INTERRUPTS AND SIGNALS:

Signals and interrupts behave in pretty similar ways. The difference is that signals happen to a process (which lives in a virtual environment), while exceptions are system-wide. Certain faults are flagged by the CPU as an exception, and then mapped to a signal that is delivered to the process by the kernel.

TYPES OF SIGNALS:

SIGHUP:-Hang up detected on controlling terminal or death of controlling process

SIGINT:-Issued if the user sends an interrupt signal (Ctrl + C)

SIGQUIT:-Issued if the user sends a quit signal (Ctrl + D)

SIGFPE:-Issued if an illegal mathematical operation is attempted

SIGKILL:-If a process gets this signal it must quit immediately and will not perform any clean-up operations

SIGALRM:-Alarm clock signal (used for timers)

Types of System Call:

1 .Alarm System Call:

alarm - set an alarm clock for delivery of a signal.alarm() arranges for a SIGALRM signal to be delivered to the process in seconds seconds.If seconds is zero, no new alarm() is scheduled.In any event any previously set alarm() is cancelled.alarm() returns

the number of seconds remaining until any previously scheduled alarm was due to be delivered, or zero if there was no previously scheduled alarm.

2 .kill System Call:

The kill() system call can be used to send any signal to any process group or process.If pid is positive, then signal sig is sent to the process with the ID specified by pid.If pid

equals 0, then sig is sent to every process in the process group of the calling process. If pid equals -1, then sig is sent to every process for which the calling process has permission to send signals, except for process 1 (init). If pid is less than -1, then sig is sent to every process in the process group whose ID is -pid.

3. Raise System call

raise - send a signal to the caller The raise() function sends a signal to the calling process or thread. In a single-threaded program it is equivalent to kill(getpid(), sig); In a multithreaded program it is equivalent to pthread_kill(pthread_self(), sig); If the signal causes a handler to be called, raise() will return only after the signal handler has returned.

4. Killpg System Call:

send signal to a process group

5. Sigaction System Call:

examine and change a signal action

Data Dictionary:

Sr Number	Variable/Function	Datatype	Use
1	pid	int	Process ID.
2	sighup	void	Used for handling signal hang up.
3	sigint	void	Used for handling interrupt signal.
4	sigquit	void	Used for dumped core handling.
5	sighold	void	Function that add sig to the calling process' signal mask.
6	sigrelse	void	Function that remove sig from the calling process' signal mask.
7	sigpause	void	Function that restore the process' signal mask to its original state before returning.
8	sigignore	void	Function that set the disposition of sig to SIG_IGN.

Table : 2.5 Data Dictionary

Program:

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

```

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

void sighup();
void sigint();
void sigquit();
void sig_handler(int);

int main()
{
    int pid;

    if ((pid = fork()) < 0)
    {
        perror("fork");
        exit(1);
    }

    if (pid == 0)
    {
        signal(SIGHUP,sighup);
        signal(SIGINT,sigint);
        signal(SIGQUIT, sigquit);
        signal(SIGUSR1, sig_handler);
        signal(SIGSTOP, sig_handler) ;
        for(;;);
    }

    else
    {
        printf("\nPARENT: sending SIGHUP\n\n");
        kill(pid,SIGHUP);
        sleep(3);

        printf("\nPARENT: sending SIGINT\n\n");
        kill(pid,SIGINT);
        sleep(3);

        printf("\nPARENT: sending SIGQUIT\n\n");
        kill(pid,SIGQUIT);
        sleep(3);

        printf("\nPARENT: sending SIGUSR1\n\n");
        kill(pid,SIGUSR1);
        sleep(3);
    }
}

```

```

        printf("\nPARENT: sending SIGSTOP\n\n");
        kill(pid,SIGSTOP);
        sleep(3);
    }
}

void sighup()
{
    signal(SIGHUP,sighup);
    printf("CHILD: I have received a SIGHUP\n");
}

void sigint()
{
    signal(SIGINT,sigint);
    printf("CHILD: I have received a SIGINT\n");
}

void sigquit()
{
    printf("My DADDY has Killed me!!!\n");
    exit(0);
}

void sig_handler(int signo)
{
    if (signo == SIGUSR1)
    {
        signal(SIGUSR1, sig_handler);
        printf("received SIGUSR1\n");
    }
    else if (signo == SIGSTOP)
    {
        signal(SIGSTOP, sig_handler);
        printf("received SIGSTOP\n");
    }
}

```

Conclusion:

- The following assignment deals with providing the guideline about the various types of system call
- Their usage in IPC and management about process in windows and linux operating system and written the application using the signal and alarm system call.

References:

- [1]www.cs.cf.ac.uk/Dave/C/CE.html
 [2]<https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/14/lec.html>

2.6 Write application of signal handling in linux OS and program any one.

Objectives:

1. To learn about IPC through signal.
2. To know the process management of Unix/Linux OS
3. Use of system call to write effective application programs.

Theory:

What are signals?

1.Signal is a notification, a message sent by either operating system or some application to your program (or one of its threads).

2.Each signal identified by a number, from 1 to 31. Signals don't carry any argument and their names are mostly self explanatory. For instance SIGKILL or signal number 9 tells the program that someone tries to kill it.

3.An application program can specify a function called a signal handler to be invoked when a specific signal is received. When a signal handler is invoked on receipt of a signal, it is said to catch the signal. A process can deal with a signal in one of the following ways:

- The process can let the default action happen
- The process can block the signal (some signals cannot be ignored)
- The process can catch the signal with a handler.

4.Signal handlers usually execute on the current stack of the process. This lets the signal handler return to the point that execution was interrupted in the process. This can be changed on a per-signal basis so that a signal handler executes on a special stack. If a process must resume in a different context than the interrupted one, it must restore the previous context itself.

Data Dictionary:

Sr Number	Variable/Function	Datatype	Use
1	sig_handler	void	Catch interrupt signal.

Table: 2.6 Data Dictionary

Program:

```
#include <fcntl.h>

#include <stdio.h>

#include <string.h>
#include <unistd.h>
int main (int argc, char* argv[])
{
char* file = argv[1];
int fd;
struct flock lock;
printf ("opening %s\n", file);
/* Open a file descriptor to the file. */
fd = open (file, O_WRONLY);
printf ("locking\n");
/* Initialize the flock structure. */
memset (&lock, 0, sizeof(lock));
lock.l_type = F_WRLCK;
/* Place a write lock on the file. */
fcntl (fd, F_SETLKW, &lock);
printf ("locked; hit Enter to unlock... ");
/* Wait for the user to hit Enter. */
getchar ();
printf ("unlocking\n");
/* Release the lock. */
lock.l_type = F_UNLCK;
fcntl (fd, F_SETLKW, &lock);
close (fd);
return 0;
```

Conclusion:

- Signal handling is used for various applications in Linux based OS.
- File locking is one of the applications discussed above

References:

[1]www.cs.cf.ac.uk/Dave/C/CE.html

Chapter 3

File system Internals

3.1 - Write the program to show file statistics using the stat system call. Take the filename / directory name from user including path.

Objectives:

1. To learn about File system Internals.

Theory:

Name:

stat, fstat, lstat - get file status

Syntax:

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

int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

Description:

These functions return information about a file. No permissions are required on the file itself, but-in the case of stat() and lstat() - execute (search) permission is required on all of the directories in *path* that lead to the file.

stat() stats the file pointed to by *path* and fills in *buf*.

lstat() is identical to stat(), except that if *path* is a symbolic link, then the link itself is stat-ed, not the file that it refers to.

fstat() is identical to stat(), except that the file to be stat-ed is specified by the file descriptor *fd*.

All of these system calls return a *stat* structure, which contains the following fields:

```
struct stat {  
    dev_t    st_dev;    /* ID of device containing file */  
    ino_t    st_ino;    /* inode number */  
    mode_t   st_mode;   /* protection */  
    nlink_t  st_nlink;  /* number of hard links */  
    uid_t    st_uid;    /* user ID of owner */  
    gid_t    st_gid;    /* group ID of owner */  
    dev_t    st_rdev;   /* device ID (if special file) */  
    off_t    st_size;   /* total size, in bytes */  
    blksize_t st_blksize; /* blocksize for file system I/O */  
    blkcnt_t st_blocks; /* number of 512B blocks allocated */  
    time_t   st_atime;  /* time of last access */  
    time_t   st_mtime;  /* time of last modification */  
    time_t   st_ctime;  /* time of last status change */  
};
```

Data Dictionary:

Sr Number	Variable/Function	Datatype	Use
1	fileStat	struct stat	Store information about files.

Table :3.1 Data Dictionary

Flowchart:

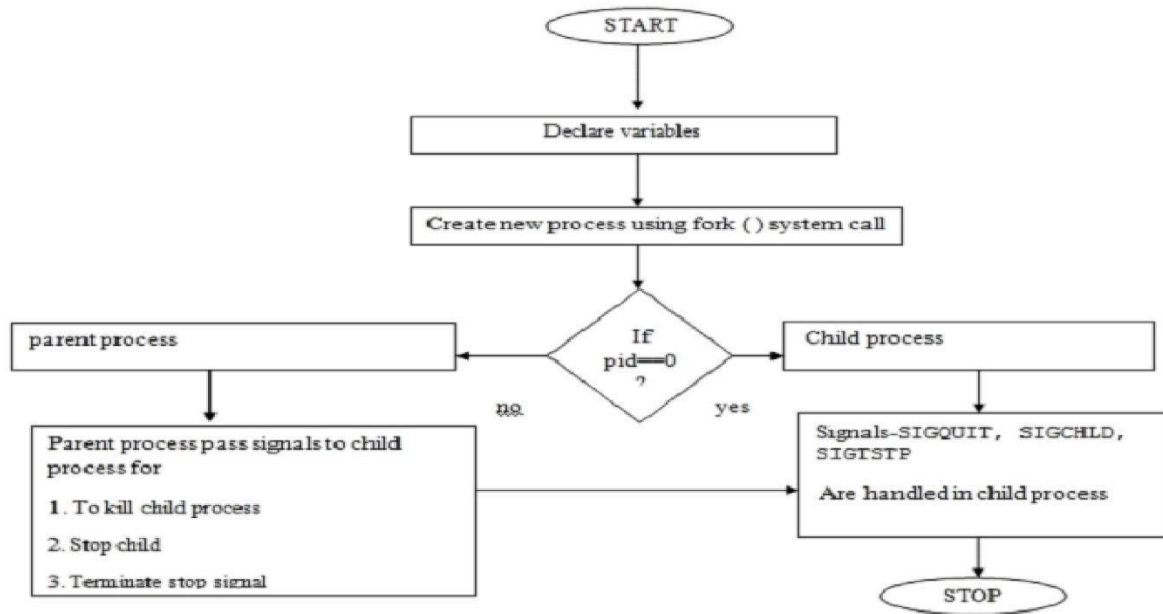


Fig: 3.1 Flowchart

Program:

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
int main(int argc, char **argv)
{
    struct stat fileStat;
    stat("/home/it/Gaurav/UOS/Demo.txt",&fileStat)
    if(stat("/home/it/Gaurav/UOS/Demo.txt",&fileStat) < 0)
    {
        printf("Failed\n");
        return 1;
    }
    printf("-----\n");
    printf("File Size: \t\t%ld bytes\n",(long)fileStat.st_size);
    printf("Number of Links: \t%ld\n",(long)fileStat.st_nlink);
```

```

printf("File inode: \t\t%d\n", (long)fileStat.st_ino);
printf("File Permissions: \t");
printf( (S_ISDIR(fileStat.st_mode)) ? "d" : "-");
printf( (fileStat.st_mode & S_IRUSR) ? "r" : "-");
printf( (fileStat.st_mode & S_IWUSR) ? "w" : "-");
printf( (fileStat.st_mode & S_IXUSR) ? "x" : "-");
printf( (fileStat.st_mode & S_IRGRP) ? "r" : "-");
printf( (fileStat.st_mode & S_IWGRP) ? "w" : "-");
printf( (fileStat.st_mode & S_IXGRP) ? "x" : "-");
printf( (fileStat.st_mode & S_IROTH) ? "r" : "-");
printf( (fileStat.st_mode & S_IWOTH) ? "w" :
"-"); printf( (fileStat.st_mode & S_IXOTH) ? "x" :
"-"); printf("\n\n");
printf("The file %s a symbolic link\n", (S_ISLNK(fileStat.st_mode)) ? "is" : "is
not"); return 0;
}

```

Output:

it@it-OptiPlex-3046:~/Vijay/UOS\$./a.out

File Size: 6299 bytes

Number of Links: 1

File inode: 15736649

File Permissions: -rw-rw-r--

The file is not a symbolic link

Conclusion:

- Stats of file like file size, links, permissions, inode number and type of link can be retrieved using stat() and stored in a structure.

References:

[1]<https://www.lix.polytechnique.fr/~liberti/public/computing/prog/c/C/FUNCTIONS/stat.html>

3.2- Write the program to show file statistics using the fstat system call. Take the file name / directory name from user including path. Print only inode no, UID, GID, FAP and File type only.

Objectives:

1. To learn about File system Internals.

Theory:

Name:

stat, fstat, lstat - get file status

Syntax:

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

int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

Description:

These functions return information about a file. No permissions are required on the file itself, but-in the case of stat() and lstat() - execute (search) permission is required on all of the directories in *path* that lead to the file.

stat() stats the file pointed to by *path* and fills in *buf*.

lstat() is identical to stat(), except that if *path* is a symbolic link, then the link itself is stat-ed, not the file that it refers to.

fstat() is identical to stat(), except that the file to be stat-ed is specified by the file descriptor *fd*.

All of these system calls return a *stat* structure, which contains the following fields:

```
struct stat {
    dev_t    st_dev;    /* ID of device containing file */

    ino_t    st_ino;    /* inode number */

    mode_t    st_mode;    /* protection */
```

```

nlink_t st_nlink; /* number of hard links */

uid_t st_uid; /* user ID of owner */

gid_t st_gid; /* group ID of owner */

dev_t st_rdev; /* device ID (if special file) */

off_t st_size; /* total size, in bytes */

blksize_t st_blksize; /* blocksize for file system I/O */

blkcnt_t st_blocks; /* number of 512B blocks allocated */

time_t st_atime; /* time of last access */

time_t st_mtime; /* time of last modification */

time_t st_ctime; /* time of last status change */

};

```

Data Dictionary:

Sr Number	Variable/Function	Datatype	Use
1	s	char[]	Get file name.
2	fp	FILE*	Pointer to file.
3	fn	int	File descriptor number.
4	sta	struct stat	Store information about files.

Table: 3.2 Data Dictionary

Program:

```

#include<stdio.h>
#include<stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int main()

```

```

{
char s[100];
gets(s);
//printf("%s",s);
FILE *fp;
if((fp=fopen(s,"r"))==NULL)
    return 1;
int fn=0;
fn=fileno(fp);
struct stat sta;
if(fstat(fn,&sta) < 0) return 1;
printf("File size : %ld\n",(long)sta.st_size);
printf("File INode Number : %ld\n",sta.st_ino);
printf("File UID : %ld\n",(long)sta.st_uid);
printf("File GID : %ld\n",(long)sta.st_gid);

printf("File Permissions: \t");
printf( (S_ISDIR(sta.st_mode)) ? "d" : "-");
printf( (sta.st_mode & S_IRUSR) ? "r" : "-");
printf( (sta.st_mode & S_IWUSR) ? "w" : "-");
printf( (sta.st_mode & S_IXUSR) ? "x" : "-");
printf( (sta.st_mode & S_IRGRP) ? "r" : "-");
printf( (sta.st_mode & S_IWGRP) ? "w" : "-");
printf( (sta.st_mode & S_IXGRP) ? "x" : "-");
printf( (sta.st_mode & S_IROTH) ? "r" : "-");
printf( (sta.st_mode & S_IWOTH) ? "w" : "-");
printf( (sta.st_mode & S_IXOTH) ? "x" : "-");
printf("\n\n");
printf("File type: ");
switch (sta.st_mode & S_IFMT)
{
    case S_IFBLK:
        printf("block device\n");

```

```

        break;
        case S_IFCHR:
            printf("character device\n");
            break;
        case S_IFDIR:
            printf("directory\n");
            break;
        case S_IFIFO:
            printf("FIFO/pipe\n");
            break;
        case S_IFLNK:
            printf("symlink\n");
            break;
        case S_IFREG:
            printf("regular file\n");
            break;
        case S_IFSOCK:
            printf("socket\n");
            break;
        default:
            printf("unknown?\n");
            break;
    }
    return 0;
}

```

Output:

```

it@it-OptiPlex-3046:~$ ./a.out
/home/Vijay/UOS
File size : 44
File INode Number : 975617
File UID : 1000

```

File GID : 1000

File Permissions: -rw-rw-r--

File type: regular file

Conclusion:

- Stats of file like UID, GID, file size, links, permissions, inode number and type of link can be retrieved using `stat()`, `fstat()` and `link()` and stored in a structure.

References:

References:

[1] <https://www.lix.polytechnique.fr/~liberti/public/computing/prog/c/C/FUNCTIONS/stat.html>

3.3 - Write a program to use link/unlink system call for creating logical link and identifying the difference using stat.

Objectives:

1. To learn about File system Internals.

Theory:

1) Link:

Name:

link - make a new name for a file

Syntax:

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

```
int link(const char *oldpath, const char *newpath);
```

Description:

`link()` creates a new link (also known as a hard link) to an existing file.

If *newpath* exists it will *not* be overwritten.

This new name may be used exactly as the old one for any operation; both names refer to the same file (and so have the same permissions and ownership) and it is impossible to tell which name was the 'original'.

Return Value:

On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

2) Unlink:

Name:

unlink - delete a name and possibly the file it refers to

Syntax:

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

```
int unlink(const char *pathname);
```

Description:

unlink() deletes a name from the filesystem. If that name was last link to a file and no processes have the file open the file is deleted and the space it was using is made available for reuse.

If the name was the last link to a file but any processes still have the file open the file will remain in existence until the last file descriptor referring to it is closed.

If the name referred to a symbolic link the link is removed. If the name referred to a socket, fifo or device the name for it is removed but processes which have the object open may continue to use it.

Return Value:

On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

Data Dictionary:

Sr Number	Variable/Function	Datatype	Use
1	old	char[]	Old pathname.
2	new	char[]	New pathname.
3	ch	char	Choice asking to unlink or not.

Table: 3.3 Data Dictionary

Program:

```
#include<stdio.h>
#include<unistd.h>
int main()
{
    char old[100];
    char new[100];
    char ch;
    printf("Enter the old and new pathname: \n");
    gets(old);
    gets(new);
    int n = link(old,new);
    if(n==0)
    {
        printf("Linked successfully\n");
    }
    else
    {
        printf("Linked unsuccessfully\n");
    }
    printf("Do you want to unlink the new file?\n1:Y\n2:N\n");
    scanf("%c",&ch);
    if(ch=='Y'||ch=='y')
    {
        int m = unlink(new);
        if(m==0)
        {
            printf("Unlinked successfully\n");
        }
        else
        {
            printf("Unlinked unsuccessfully\n");
        }
    }
}
```

```

    }
    else
    {
        printf("Not Unlinked\n");
    }
}

```

Output:

```

it@it-OptiPlex-3046:~/Vijay/UOS$ gcc 3A_c.c
it@it-OptiPlex-3046:~/Vijay/UOS$ ./a.out Enter
the old and new pathname:
/home/it/Vijay/UOS/Demo.txt
/home/it/Vijay/UOS/Demo1.txt Linked
successfully
Do you want to unlink the new file?
1:Y
2:N
Y
Unlinked successfully
[1]+  Killed                  gedit 3B_b.c

```

Conclusion:

- The concepts of creating link or shortcut to file and unlinking it understood through link and unlink function calls.
- The change in number of links takes place as we implement the program.

References:

[1]www.tutorialspoint.com/unix_system_calls

3.4- Implement a program to print the various types of file in Linux.

Objectives:

1. To learn about File system Internals.

Theory:

```
stat(pathname, &sb);  
if ((sb.st_mode & S_IFMT) == S_IFREG) {  
    /* Handle regular file */  
}
```

Because tests of the above form are common, additional macros are defined by POSIX to allow the test of the file type in `st_mode` to be written more concisely:

`S_ISREG(m)` is it a regular file?

`S_ISDIR(m)` directory? \newline

`S_ISCHR(m)` character device?

`S_ISBLK(m)` block device?

`S_ISFIFO(m)` FIFO (named pipe)?

`S_ISLNK(m)` symbolic link? (Not in POSIX.1-1996.)

`S_ISSOCK(m)` socket? (Not in POSIX.1-1996.)

Data Dictionary:

Sr Number	Variable/Function	Datatype	Use
1	file	int	Open a file and store file descriptor.
2	fileStat	struct stat	Store information about files.

Table: 3.4Data Dictionary

Program:

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
int main(int argc, char **argv)
{
    if(argc != 2)
        return 1;
    int file;
    if((file=open(argv[1],O_RDONLY)) < -1)
        return 1;
    struct stat fileStat;
    if(fstat(file,&fileStat) < 0)
        return 1;
    if(S_ISBLK(fileStat.st_mode))
    {
        printf("Block special file\n");
    }
    if(S_ISCHR(fileStat.st_mode))
    {
        printf("Character special file\n");
    }
    if(S_ISDIR(fileStat.st_mode))
    {
        printf("Is a directory\n");
    }
}
```

```

    if(S_ISFIFO(fileStat.st_mode))
    {
        printf("Pipes and FIFO\n");

    }
    if(S_ISLNK(fileStat.st_mode))
    {
        printf("Is a symbolic link\n");
    }
    if(S_ISREG(fileStat.st_mode))
    {
        printf("A Regular File\n");
    }
    if(S_ISSOCK(fileStat.st_mode))
    {
        printf("Integrated Sockets\n");
    }
    return 0;
}

```

Conclusion:

- Various types of files like block file, character file, regular file, FIFO and pipe can be studied and checked with the status of different flags.

References:

[1]<https://www.cyberciti.biz/faq/linuxunix-determine-file-type/>

3.5 -Write a program to lock the file using flock system call

Objectives:

1. To learn about File locking-mandatory and advisory locking.

Theory:

1.Name:-

flock - apply or remove an advisory lock on an open file

2. Syntax:-

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

int flock(int fd, int operation);
```

3.Description:-

Apply or remove an advisory lock on the open file specified by fd. The argument operation is one of the following:

1.LOCK_SH

Place a shared lock. More than one process may hold a shared lock for a given file at a given time.

2.LOCK_EX

Place an exclusive lock. Only one process may hold an exclusive lock for a given file at a given time.

3.LOCK_UN

Remove an existing lock held by this process.

A call to flock() may block if an incompatible lock is held by another process. To make a nonblocking request, include LOCK_NB (by ORing) with any of the above operations.

A single file may not simultaneously have both shared and exclusive locks.

4.Return Value:-

On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

Data Dictionary:

Sr Number	Variable/Function	Datatype	Use
1	fd	int	Open a file and store file descriptor.

Table: 3.5Data Dictionary

Program:

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
int main (int argc, char* argv[])
{
    char* file = argv[1];
    int fd;
    struct flock lock;
    printf ("opening %s\n", file);
    /* Open a file descriptor to the file. */
    fd = open (file, O_WRONLY);
    printf ("locking\n");
    /* Initialize the flock structure. */
    memset (&lock, 0, sizeof(lock));
    lock.l_type = F_WRLCK;
    /* Place a write lock on the file. */
    fcntl (fd, F_SETLKW, &lock);
    printf ("locked; hit Enter to unlock... ");
    /* Wait for the user to hit Enter. */
    getchar ();
    printf ("unlocking\n");
    /* Release the lock. */
    lock.l_type = F_UNLCK;
    fcntl (fd, F_SETLKW, &lock);
    close (fd);
    return 0;
}
```

Conclusion:

- Functions of flock() to use shared, exclusive locks and unlock them are studied.

References :

[1]www.tutorialspoint.com/unix_system_calls

3.6- write a program to lock file using fcntl system call

Objectives:

1. To learn about File locking-mandatory and advisory locking

Theory:

1.The fcntl system call is the access point for several advanced operations on file descriptors.

2.The first argument to fcntl is an open file descriptor, and the second is a value that indicates which operation is to be performed. For some operations, fcntl takes an additional argument.

3.The fcntl system call allows a program to place a read lock or a write lock on a file.

4.A read lock is placed on a readable file descriptor, and a write lock is placed on a writable file descriptor. More than one process may hold a read lock on the same file at the same time, but only one process may hold a write lock, and the same file may not be both locked for read and locked for write. Note that placing a lock does not actually

prevent other processes from opening the file, reading from it, or writing to it, unless they acquire locks with fcntl as well.

5.To place a lock on a file, first create and zero out a struct flock variable. Set the l_type field of the structure to F_RDLCK for a read lock or F_WRLCK for a write lock. Then call fcntl, passing a file descriptor to the file, the F_SETLCKW operation code, and a pointer to the struct flock variable. If another process holds a lock that prevents a new lock from being acquired, fcntl blocks until that lock is released.

6.It returns -1 if lock is not acquired on a file.

Data Dictionary:

Sr Number	Variable/Function	Datatype	Use
1	file	char*	File name.
2	fd	int	File descriptor.
3	lock	struct flock	Structure to describe a file lock.

Table: 3.6Data Dictionary

Program:

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/file.h>
int main()
{
    int fd = open("File1.txt", O_WRONLY);
    if(fd==-1)
        printf("open file fails\n");
}
else
{
    if(flock(fd, LOCK_EX)==0)
        printf("Lock is acquired on file\n");
    else
        printf("Lock is not acquired on a file\n");
}
close(fd);
return 0;
}
```

Conclusion:

- Functions of fcntl() to use read lock, write lock are studied.

References:

[1]<https://gavv.github.io/blog/file-locks/>

Chapter 4

Thread concept

4.1 - Write a multi-threaded program in Java/c for chatting (multiuser and multi-terminal) using threads

Objectives:

1. To learn about threading in Linux/Unix and Java and difference between them
2. Use of system call/library to write effective programs

Theory:

As normal, we will create two Java files, Server.java and Client.java. Server file contains two classes namely Server (public class for creating server) and ClientHandler (for handling any client using multi-threading). Client file contain only one public class Client (for creating a client). Below is the flow diagram of how these three classes interact with each other.

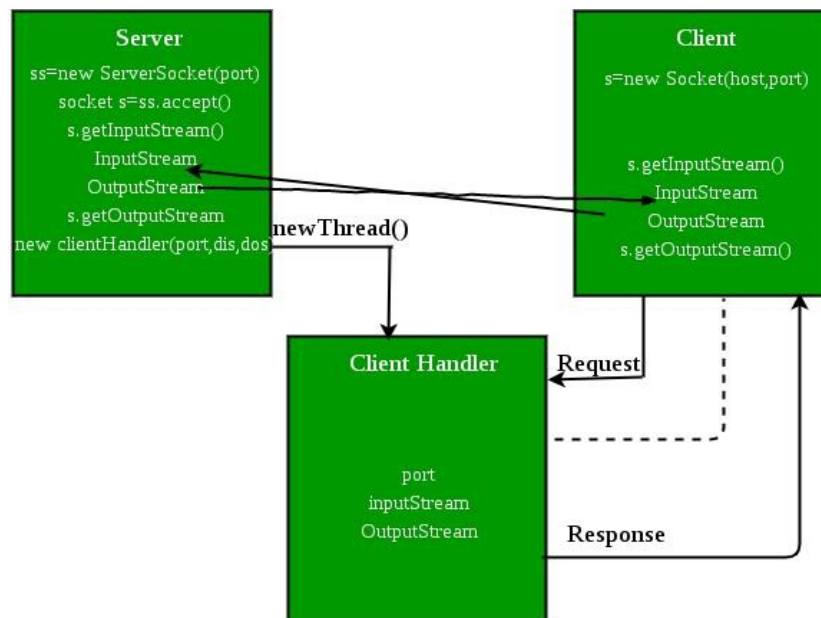


Fig: Flow of message between client and server

Server class : The steps involved on server side are similar to the article [Socket Programming In Java](#) with a slight change to create the thread object after obtaining the streams and port number.

1. Establishing the Connection: Server socket object is initialized and inside a while loop a socket object continuously accepts incoming connection.
2. Obtaining the Streams: The inputstream object and outputstream object is extracted from the current requests' socket object.
3. Creating a handler object: After obtaining the streams and port number, a new ClientHandler object (the above class) is created with these parameters.
4. Invoking the start() method : The start() method is invoked on this newly created thread object.

ClientHandler class : As we will be using separate threads for each request, lets understand the working and implementation of the ClientHandler class extending Threads. An object of this class will be instantiated each time a request comes.

5. First of all this class extends Thread so that its objects assumes all properties of Threads.
6. Secondly, the constructor of this class takes three parameters, which can uniquely identify any incoming request, i.e. a Socket, a DataInput Stream to read from and a DataOutputStream to write to. Whenever we receive any request of client, the server extracts its port number, the DataInputStream object and DataOutputStream object and creates a new thread object of this class and invokes start() method on it.

Note : Every request will always have a triplet of socket, input stream and output stream. This ensures that each object of this class writes on one specific stream rather than on multiple streams.

7. Inside the run() method of this class, it performs three operations: request the user to specify whether time or date needed, read the answer from input stream object and accordingly write the output on the output stream object.

Flowchart:

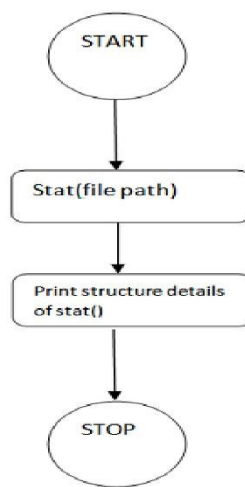


Fig: 4.1 Flowchart

Data Dictionary:

Sr Number	Variable/Function	Datatype	Use
Class Server			
1	Ar	Vector	Store which clients are there.
2	i	int	Count of clients.
3	ss	ServerSocket	Create a socket for server side communication.
4	s	Socket	Socket is created.
5	dis	DataInputStream	Input data.
6	t	Thread	Used to create new thread for each client.
7	mtch	ClientHandler	Object of ClientHandler type. Used to handle client.
Class ClientHandler			
1	scn	Scanner	Usedfor any input.
2	name	String	Store name of client.
3	dis	DataInputStream	Input a message.
4	dos	DataOutputStream	Output to standart ouput the message input.
5	received	String	Store message.
6	isloggedin	boolean	If the client is logged or not.

Table : 4.1 Data Dictionary

Program:**For Server:**

```
import java.io.*;
import java.util.*;
import java.net.*;
public class Server
{
    static Vector<ClientHandler> ar = new
    Vector<>(); static int i = 0;

    public static void main(String[] args) throws IOException
    {
        ServerSocket ss = new ServerSocket(1234);
        Socket s;
        while (true)
```

```

        {
            s = ss.accept();
            System.out.println("New client request received : " + s);
            DataInputStream dis = new DataInputStream(s.getInputStream());
            DataOutputStream dos = new DataOutputStream(s.getOutputStream());
            System.out.println("Creating a new handler for this client...");
            ClientHandler mtch = new ClientHandler(s,"client " + i, dis, dos);
            Thread t = new Thread(mtch);
            System.out.println("Adding this client to active client list");
            ar.add(mtch);
            t.start();
            i++;
        }
    }
}

```

class ClientHandler implements Runnable

```

{
    Scanner scn = new Scanner(System.in);
    private String name;
    final DataInputStream dis;
    final DataOutputStream dos;
    Socket s;
}

```

```
boolean isloggedin;
```

```
public ClientHandler(Socket s, String name, DataInputStream  
dis, DataOutputStream dos)
```

```
{  
  
    this.dis = dis;  
    this.dos = dos;  
    this.name = name;  
    this.s = s;  
    this.isloggedin=true;  
}
```

```
@Override
```

```
public void run()
```

```
{  
  
    String received;  
    while (true)  
    {  
        try  
        {  
            received = dis.readUTF();  
            System.out.println(received);  
            if(received.equals("logout"))  
            {  
                this.isloggedin=false;  
                this.s.close();  
                break;  
            }  
        }  
    }  
}
```

```

        StringTokenizer st = new StringTokenizer(received, "#");
        String MsgToSend = st.nextToken();
        String recipient = st.nextToken();
        for (ClientHandler mc : Server.ar)
        {
            if (mc.name.equals(recipient) && mc.isloggedin==true)
            {
                mc.dos.writeUTF(this.name+" : "+MsgToSend);
                break;
            }
        }
    } catch (IOException e){e.printStackTrace();}
}
try
{
    this.dis.close();
    this.dos.close();

} catch (IOException e){e.printStackTrace();}
}
}

```

For Client:

```

import java.io.*;
import java.net.*;
import java.util.Scanner;
public class Client
{

```



```

final static int ServerPort = 1234;

public static void main(String args[]) throws UnknownHostException, IOException
{
    final Scanner scn = new Scanner(System.in);
    InetAddress ip = InetAddress.getByName("localhost");
    Socket s = new Socket(ip, ServerPort);
    final DataInputStream dis = new DataInputStream(s.getInputStream());
    final DataOutputStream dos = new
    DataOutputStream(s.getOutputStream()); Thread sendMessage = new
    Thread(new Runnable() {
        @Override
        public void run()
        {
            while (true)
            {
                String msg = scn.nextLine();

                try
                {
                    dos.writeUTF(msg);
                } catch (IOException e){e.printStackTrace();}
            }
        }
    });

    Thread readMessage = new Thread(new Runnable()
    {
        @Override
        public void run()

```

```

    {
    while (true)
    {
    try
    {

        String msg = dis.readUTF();

        System.out.println(msg);

    } catch (IOException e){e.printStackTrace();}

    }

sendMessage.start();

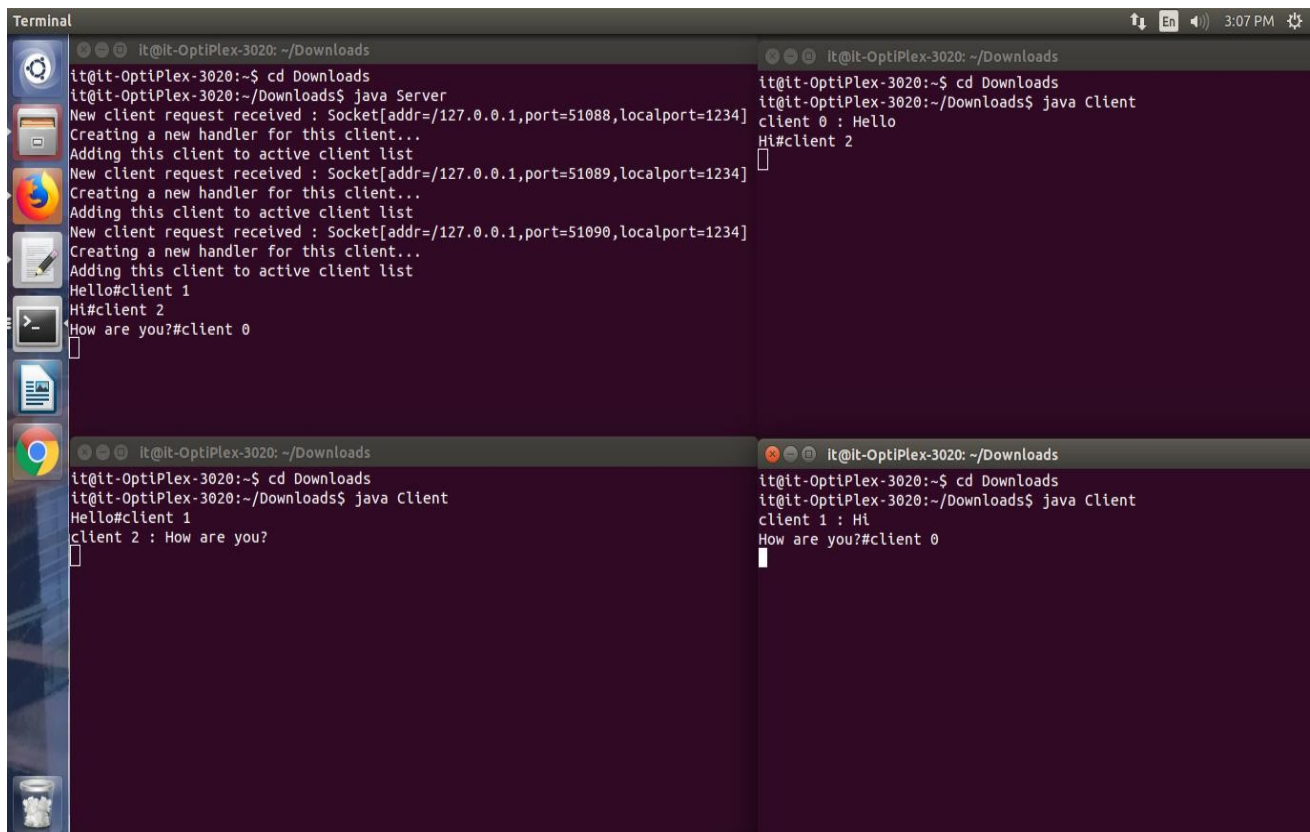
readMessage.start();

}

}

```

Output:



```

Terminal
it@it-OptiPlex-3020: ~/Downloads
it@it-OptiPlex-3020:~/Downloads$ cd Downloads
it@it-OptiPlex-3020:~/Downloads$ java Server
New client request received : Socket[addr=/127.0.0.1,port=51088,localport=1234]
Creating a new handler for this client...
Adding this client to active client list
New client request received : Socket[addr=/127.0.0.1,port=51089,localport=1234]
Creating a new handler for this client...
Adding this client to active client list
New client request received : Socket[addr=/127.0.0.1,port=51090,localport=1234]
Creating a new handler for this client...
Adding this client to active client list
Hello#client 1
Hi#client 2
How are you?#client 0

it@it-OptiPlex-3020:~/Downloads
it@it-OptiPlex-3020:~/Downloads$ cd Downloads
it@it-OptiPlex-3020:~/Downloads$ java Client
client 0 : Hello
Hi#client 2

it@it-OptiPlex-3020:~/Downloads
it@it-OptiPlex-3020:~/Downloads$ cd Downloads
it@it-OptiPlex-3020:~/Downloads$ java Client
Hello#client 1
client 2 : How are you?

it@it-OptiPlex-3020:~/Downloads
it@it-OptiPlex-3020:~/Downloads$ cd Downloads
it@it-OptiPlex-3020:~/Downloads$ java Client
client 1 : Hi
How are you?#client 0

```

Conclusion:

- Various concepts and effective programming in Java using threads and sockets was studied.
- The concept of threading and multithreading understood.

References:

[1]<https://www.geeksforgeeks.org/multithreading-in-java/>

4.2 - Write a program which creates 3 threads, first thread printing even number, second thread printing odd number and third thread printing prime number in terminals.**Objectives:**

1. To learn about threading in Linux/Unix and Java and difference between them
2. Use of system call/library to write effective programs

Theory:

Java is a *multi-threaded programming language* which means we can develop multi-threaded program using Java. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

By definition, multitasking is when multiple processes share common processing resources such as a CPU. Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

Multi-threading enables you to write in a way where multiple activities can proceed concurrently in the same program.

Life Cycle of a Thread:

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread.

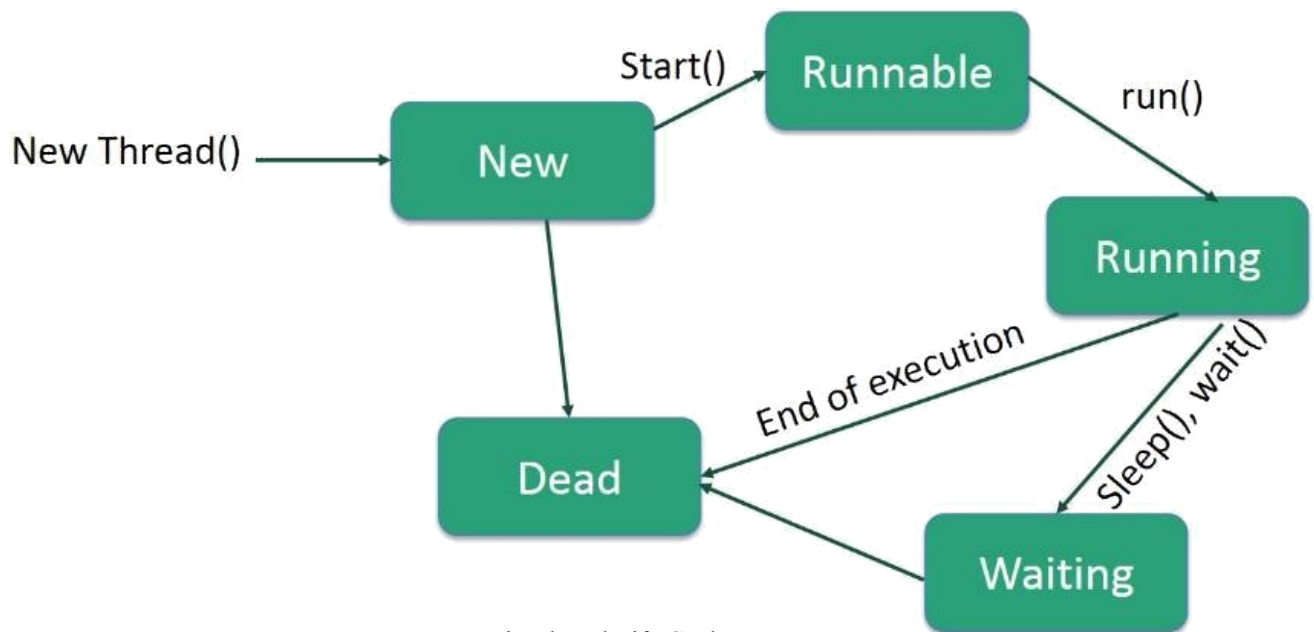


Fig:Thread Life Cycle

Following are the stages of the life cycle:

- **New** – A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
- **Runnable** – After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting** – Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed Waiting** – A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated (Dead)** – A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Flowchart:

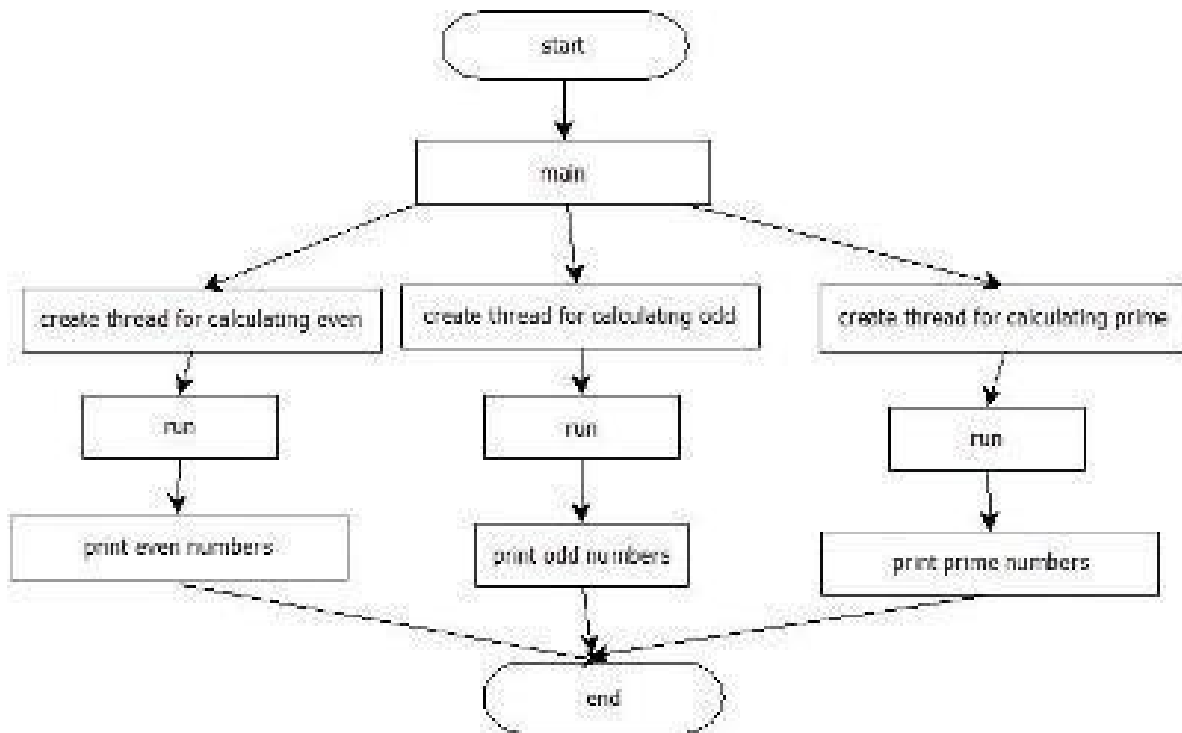


Fig: 4.2 Flowchart

Data Dictionary:

Sr Number	Variable/Function	Datatype	Use
1	t1	Thread1	Print odd numbers.
2	t2	Thread2	Print prime numbers.
3	tm	B4	Print even numbers.
4	i	int	Iterationg for loop.

Table: 4.2 Data Dictionary

Program:

```
public class B4 extends Thread
{
    public static void main(String []args)
    {
        Thread1 t1 = new Thread1();
        Thread2 t2 = new Thread2();
```

```

        B4 tm = new B4()

tm.start();

        t1.start();

        t2.start();

    }

    public void run()
    {

        for(int i=0;i<=30;i=i+2)

        {

            System.out.println("Even: "+i);

        }

    }

}

class Thread1 extends Thread

{

    public void run()

    {

        for(int i=1;i<=31;i=i+2)

        {

            System.out.println("Odd: "+i);

        }

    }

}

```

```
class Thread2 extends Thread
{
public void run()
{
for(int i=2;i<100;i++)
{
int count = 0;
for(int j=2;j<i;j++)
{
if(i%j==0)
{
count++;
}
}
if(count==0)
System.out.println("Prime: "+i);
}
}
}
```

Output:

```
it@it-OptiPlex-3020:~/Vijay$ javac B4.java
it@it-OptiPlex-3020:~/Vijay$ java B4 Prime: 2
Prime: 3
```


Prime: 5

Prime: 7

Even: 0

Even: 2

Odd: 1

Even: 4

Even: 6

Even: 8

Prime: 11

Even: 10

Odd: 3

Even: 12

Prime: 13

Even: 14

Odd: 5

Even: 16

Prime: 17

Even: 18

Odd: 7

Even: 20

Prime: 19

Even: 22

Odd: 9

Even: 24

Prime: 23

Even: 26

Odd: 11

Even: 28

Prime: 29

Even: 30

Odd: 13

Prime: 31

Odd: 15

Prime: 37

Odd: 17

Prime: 41

Odd: 19

Prime: 43

Odd: 21

Prime: 47

Odd: 23

Prime: 53

Odd: 25

Prime: 59

Odd: 27

Prime: 61

Odd: 29

Prime: 67

Odd: 31

Prime: 71

Prime: 73

Prime: 79

Prime: 83

Prime: 89

Prime: 97

Conclusion:

- Multiple threads and their execution pattern studied in details.
- Need for mechanism to synchronize recognized.

References:

[1]<https://www.geeksforgeeks.org/multithreading-in-java/>

4.3 - Write program to synchronize threads using construct – monitor/serialize/semaphore of Java.

Objectives:

1. To learn about threading in Linux/Unix and Java and difference between them
2. Use of system call/library to write effective programs

Theory:

A semaphore controls access to a shared resource through the use of a counter. If the counter is greater than zero, then access is allowed. If it is zero, then access is denied. What the counter is counting are permits that allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore.

Working of semaphore :

In general, to use a semaphore, the thread that wants access to the shared resource tries to acquire a permit.

- If the semaphore's count is greater than zero, then the thread acquires a permit, which causes the semaphore's count to be decremented.

- Otherwise, the thread will be blocked until a permit can be acquired.
- When the thread no longer needs an access to the shared resource, it releases the permit, which causes the semaphore's count to be incremented.
- If there is another thread waiting for a permit, then that thread will acquire a permit at that time.

Java provide **Semaphore** class in *java.util.concurrent* package that implements this mechanism, so you don't have to implement your own semaphores.

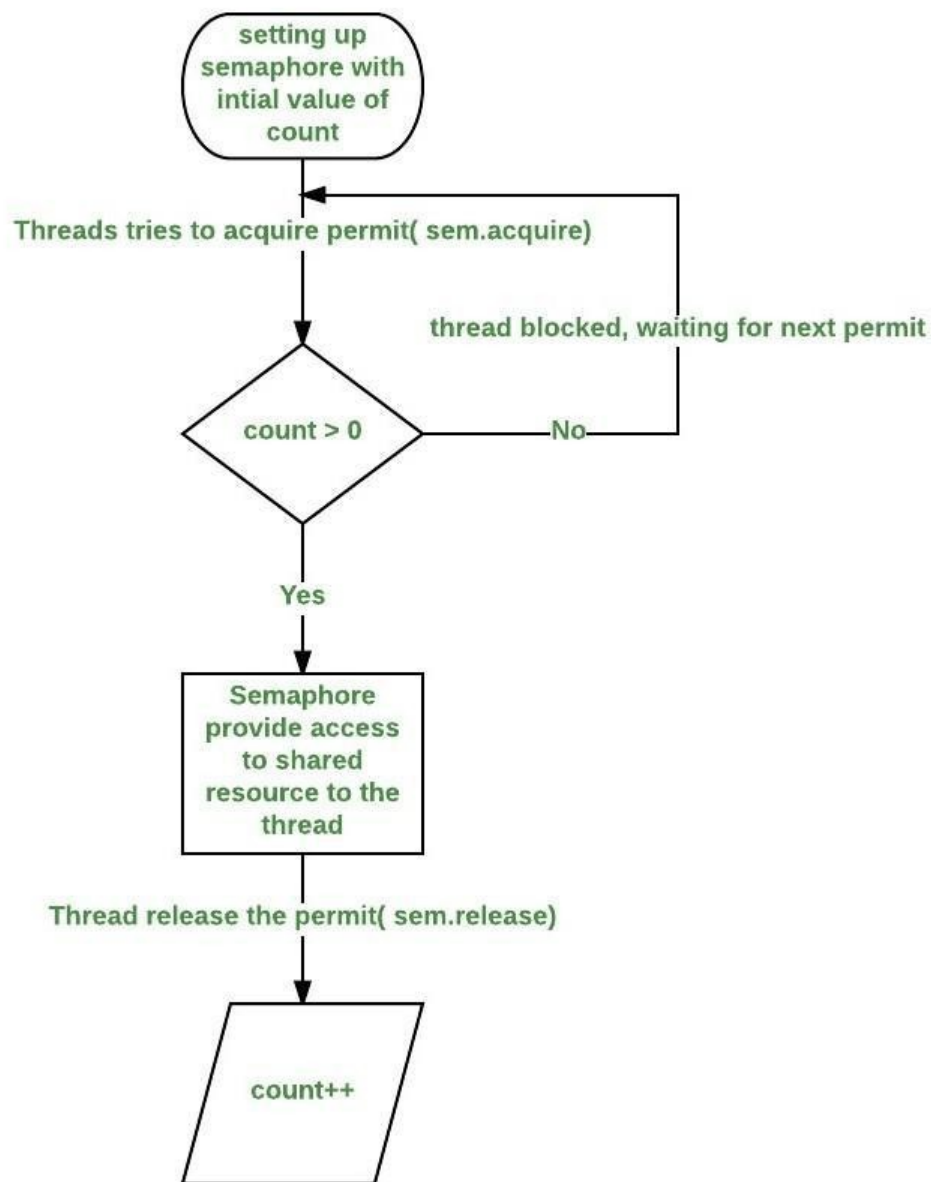


Fig:Working of Semaphore

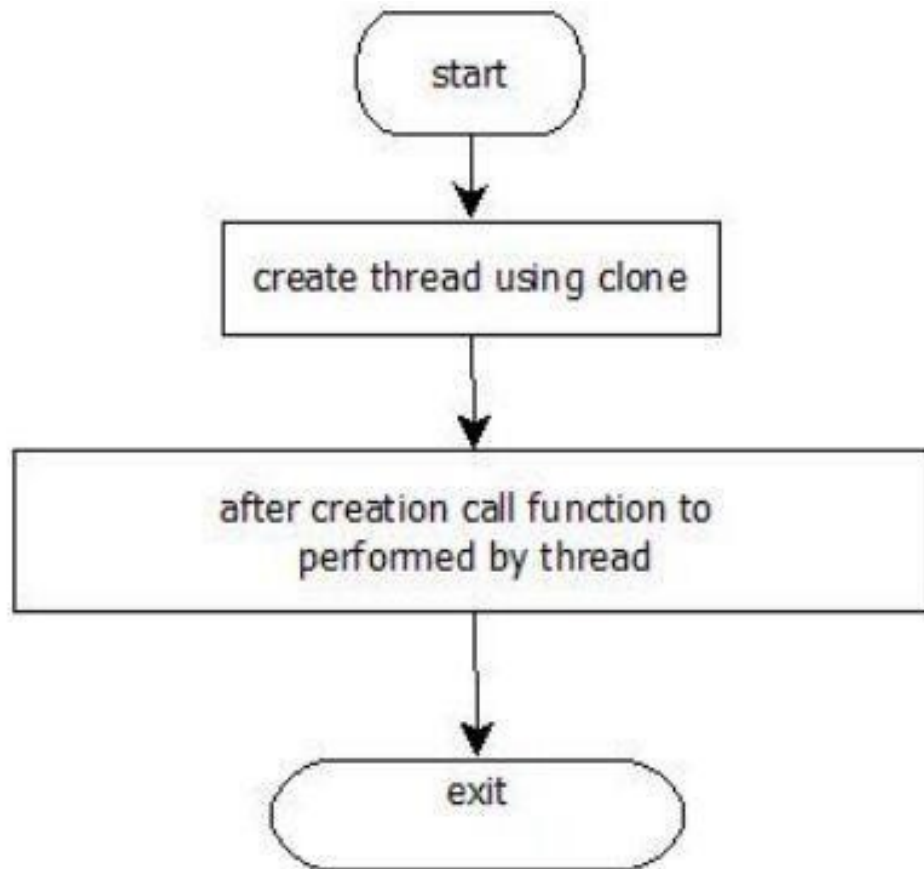
Flowchart:

Fig: 4.3 Flowchart

Data Dictionary:

Sr Number	Variable/Function	Datatype	Use
1	semaphore	Semaphore	Used to synchronize between the threads.
2	t1	Thread1	Print odd numbers.
3	t2	Thread2	Print prime numbers.
4	tm	G4	Print even numbers.

Program:

```
import java.util.concurrent.*;  
public class G4 extends Thread
```

```

public static Semaphore semaphore = new
Semaphore(1); public static void main(String args[])
throws Exception {
    Thread1 t1 = new Thread1();
    Thread2 t2 = new Thread2();
    G4 tm = new G4();
    tm.start();
    t1.start();
    t2.start();
}
public void run()
{
    try
    {
        semaphore.acquire();
        for(int i=0;i<=20;i=i+2)
        {
            System.out.println("Even: "+i);
        }
        semaphore.release();
    }
    catch(InterruptedException exc){}
}
static class Thread1 extends Thread
{

```

```

public void run()
{
try
{
semaphore.acquire();
for(int i=1;i<=21;i=i+2)
{
System.out.println("Odd: "+i);
}
semaphore.release();
}
catch(InterruptedException exc){}
}
}

```

```

static class Thread2 extends Thread
{
public void run()
{
try
{
semaphore.acquire();
for(int i=2;i<50;i++)
{
int count = 0;

```

```

for(int j=2;j<i;j++)
{
if(i%j==0)
{
count++;
}
}
if(count==0)
System.out.println("Prime: "+i);
}
semaphore.release();
}
catch(InterruptedException exc){}
}
}
}

```

Output:

it@it-OptiPlex-3020:~/Vijay\$ javac C4.java

^[[Ait@it-OptiPlex-3020:~/Vijay\$ java C4

Even: 0

Even: 2

Even: 4

Even: 6

Even: 8

Even: 10

Even: 12

Even: 14

Even: 16

Even: 18

Even: 20

Odd: 1

Odd: 3

Odd: 5

Odd: 7

Odd: 9

Odd: 11

Odd: 13

Odd: 15

Odd: 17

Odd: 19

Odd: 21

Prime: 2

Prime: 3

Prime: 5

Prime: 7

Prime: 11

Prime: 13

Prime: 17

Prime: 19

Prime: 23

Prime: 29

Prime: 31

Prime: 37

Prime: 41

Prime: 43

Prime: 47

Conclusion:

- Synchronization of multiple threads using semaphore to let threads work synchronously to produce desirable outputs learned and implemented in Java

References:

[1]<https://www.geeksforgeeks.org/multithreading-in-java/>

4.4 - Write a program in Linux to use clone system call and show how it is different from fork system call.

Objectives:

1. To learn about threading in Linux/Unix and Java and difference between them
2. Use of system call/library to write effective programs

Theory:

Syntax:

```
#define _GNU_SOURCE
```

```
#include <sched.h>
```

```
int clone(int (*fn)(void *), void *child_stack,
```

```
int flags, void *arg, ...
/* pid_t *ptid, void *newtls, pid_t *ctid */ );
/* For the prototype of the raw system call, see NOTES */
```

Description:

clone() creates a new process, in a manner similar to fork(2).

This page describes both the glibc clone() wrapper function and the underlying system call on which it is based. The main text describes the wrapper function; the differences for the raw system call are described toward the end of this page.

Difference between fork and clone:

Unlike fork(2), clone() allows the child process to share parts of its execution context with the calling process, such as the virtual address space, the table of file descriptors, and the table of signal handlers. (Note that on this manual page, "calling process" normally corresponds to "parent process". But see the description of CLONE_PARENT below.)

Data Dictionary:

Sr Number	Variable/Function	Datatype	Use
1	fn	Int (function)	Run code under child process.
2	pchild_stack	void*	Allocating memory.
3	pid	int	ID of cloned process.

Table: 4.4Data Dictionary:

Program:

```
#include <stdio.h>

#include <sched.h>

#include <stdlib.h>

#include <sys/wait.h>

int fn()
{
printf("\nThis code is running under child process.\n\n");
```

```

int i = 0;

int n = 7;

for ( i = 1 ; i <= 8 ; i++ )
{
printf("%d * %d = %d\n", n, i, (n*i));
}

return 0;
}

void main(int argc, char *argv[])
{
printf("We are in Parent\n\n");

void *pchild_stack = malloc(1024 * 1024);

if ( pchild_stack == NULL )
{
printf("ERROR: Unable to allocate memory.\n");
exit(EXIT_FAILURE);
}

printf("Creating Child\n");

int pid = clone(fn, pchild_stack + (1024 * 1024), SIGCHLD, argv[1]);

if ( pid < 0 )
{
printf("ERROR: Unable to create the child process.\n");
exit(EXIT_FAILURE);
}

wait(NULL);

```

```
free(pchild_stack);  
printf("\nChild process terminated.\n");  
}
```

Output:

```
it@it-OptiPlex-3020:~/Vijay$ gcc D4.c
```

```
it@it-OptiPlex-3020:~/Vijay$ ./a.out
```

We are in Parent

Creating Child

This code is running under child process.

$7 * 1 = 7$

$7 * 2 = 14$

$7 * 3 = 21$

$7 * 4 = 28$

$7 * 5 = 35$

$7 * 6 = 42$

$7 * 7 = 49$

$7 * 8 = 56$

Child process terminated.

Conclusion:

- clone() can be used to produce a child which could share some information of its parent and can be further modified to have its own form.

References:

[1] <https://www.geeksforgeeks.org/clone-method-in-java-2/>

4.5 - Write a program using p-thread library of Linux. Create three threads to take odd, even and prime respectively and print their average respectively.

Objectives:

1. To learn about threading in Linux/Unix and Java and difference between them..
2. Use of system call/library to write effective programs

Theory:

Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.

- In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required.
- For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995).
- Implementations adhering to this standard are referred to as POSIX threads, or Pthreads.
- Most hardware vendors now offer Pthreads in addition to their proprietary API's.
- The POSIX standard has continued to evolve and undergo revisions, including the Pthreads specification.

Pthreads defines a set of C programming language types, functions and constants. It is implemented with a pthread.h header and a thread library. Pthread programs are compiled using gcc -pthread.

There are around 100 threads procedures, all prefixed 'pthread_' and they can be categorized into four groups:

- Thread management - creating, joining threads etc.
- Mutexes.
- Condition variables.
- Synchronization between threads using read/write locks and barriers.

On Linux, both fork() and pthreads use the same system call clone(), which creates a new process. The difference between them is simply the parameters they send to clone(), when creating a new thread, it simply makes both processes use the same memory mappings.

Pthreads are created using `pthread_create()`. Pthreads terminate when the function returns, or the thread can call `pthread_exit()` which terminates the calling thread explicitly.

- `int pthread_create(pthread_t *thread_id, const pthread_attr_t *attributes, void *(*thread_function)(void *), void *arguments);`

When an attribute object is not specified, it is NULL, and the *default* thread is created with the following attributes:

- It is unbounded and nondetached.
 - It has a default stack and stack size.
 - It inherits the parent's priority.
- `int pthread_exit (void *status);`

Advantages:

- **Light Weight:** When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.
- **Efficient Communications/Data Exchange:** The primary motivation for considering the use of Pthreads in a high performance computing environment is to achieve optimum performance. In particular, if an application is using MPI for on-node communications, there is a potential that performance could be improved by using Pthreads instead. For Pthreads there is no intermediate memory copy required because threads share the same address space within a single process. There is no data transfer, per se. It can be as efficient as simply passing a pointer.
- A perfect example is the typical web browser, where many interleaved tasks can be happening at the same time, and where tasks can vary in priority.

Data Dictionary:

Sr Number	Variable/Function	Datatype	Use
1	t1	pthread_t	Create thread in c.
2	t2	pthread_t	Create thread in c.
3	t3	pthread_t	Create thread in c.
4	even	void *	Function for thread t1. Prints even numbers.
5	odd	void *	Function for thread t2. Prints odd numbers.
6	prime	void *	Function for thread t3. Prints prime numbers.

Program:

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

void *even(void *mid)
{
    int count = 0, sum = 0;
    float avg = 0.0;
    int* id = (int*)mid;
    for(int i=1;i<=20;i++)
    {
        if(i%2==0)
        {
            count++;
            sum = sum + i;
            printf("Even %d: %d\n",count,i);
        }
    }
    avg = (float)sum/count;
    printf("Average is %f\n", avg);
}

void *odd(void *mid)
{
    int count = 0, sum = 0;
```

```

float avg = 0.0;
for(int i=1;i<=20;i++)
{
    if(i%2!=0)
    {
        count++;
        sum = sum + i;
        printf("Odd %d: %d\n",count,i);
    }
}
avg = (float)sum/count;
printf("Average is %f\n", avg);
}

```

```

void *prime(void *mid)
{
    int count = 1, sum = 2, c = 0;
    float avg = 0.0;
    printf("Prime %d: %d\n",count,2);
    for(int i=3;i<=20;i++)
    {
        c = 0;
        for(int j=2;j<i;j++)
        {
            if(i%j==0)

```

```

{
c++;
break;
}
}
if(c == 0)
{
count++;
sum = sum + i;
printf("Prime %d: %d\n",count,i);
}
}
avg = (float)sum/count;
printf("Average is %f\n", avg);
}
int main()
{
pthread_t t1,t2,t3;

// Let us create three threads

pthread_create(&t1, NULL, &odd, NULL);
pthread_create(&t2, NULL, &even, NULL);
pthread_create(&t3, NULL, &prime, NULL);
pthread_exit(NULL);

return 0;
}

```

Output:

Odd 1: 1

Odd 2: 3

Odd 3: 5

Odd 4: 7

Odd 5: 9

Odd 6: 11

Odd 7: 13

Odd 8: 15

Odd 9: 17

Odd 10: 19

Average is 10.000000

Prime 1: 2

Prime 2: 3

Prime 3: 5

Prime 4: 7

Prime 5: 11

Prime 6: 13

Prime 7: 17

Prime 8: 19

Average is 9.625000

Even 1: 2

Even 2: 4

Even 3: 6

Even 4: 8

Even 5: 10

Even 6: 12

Even 7: 14

Even 8: 16

Even 9: 18

Even 10: 20

Average is 11.000000

Conclusion:

- Use of p_threads in C language to create and synchronize using semaphore can be done.

References:

[1]<https://www.cs.nmsu.edu/~jcook/Tools/pthreads/library.html>

4.6 - Write a program using p-thread library of Linux. Create three threads to take odd, even and prime respectively and print their average respectively.

Objectives:

1. To learn about threading in Linux/Unix and Java and difference between them..
2. Use of system call/library to write effective programs

Theory:

Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.

- In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required.
- For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995).
- Implementations adhering to this standard are referred to as POSIX threads, or Pthreads.
- Most hardware vendors now offer Pthreads in addition to their proprietary API's.
- The POSIX standard has continued to evolve and undergo revisions, including the Pthreads specification.

Pthreads defines a set of C programming language types, functions and constants. It is implemented with a pthread.h header and a thread library. Pthread programs are compiled using gcc -pthread.

There are around 100 threads procedures, all prefixed 'pthread_' and they can be categorized into four groups:

- Thread management - creating, joining threads etc.
- Mutexes.
- Condition variables.
- Synchronization between threads using read/write locks and barriers.

On Linux, both fork() and pthreads use the same system call clone(), which creates a new process. The difference between them is simply the parameters they send to clone(), when creating a new thread, it simply makes both processes use the same memory mappings.

Pthreads are created using pthread_create(). Pthreads terminate when the function returns, or the thread can call pthread_exit() which terminates the calling thread explicitly. One Thread can wait on the termination of another by using pthread_join(). The exit status is returned in status_ptr.

- int pthread_create(pthread_t *thread_id, const pthread_attr_t *attributes, void *(*thread_function)(void *), void *arguments);

When an attribute object is not specified, it is NULL, and the *default* thread is created with the following attributes:

- It is unbounded.
- It is nondetached.
- It has a default stack and stack size.
- It inherits the parent's priority.
- Int pthread_exit (void *status);

```
int pthread_join (pthread_t thread, void **status_ptr);
```

Advantages:

- **Light Weight:** When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.
- **Efficient Communications/Data Exchange:** The primary motivation for considering the use of Pthreads in a high performance computing environment is to achieve optimum performance. In particular, if an application is using MPI for on-node communications, there is a potential that performance could be improved by using Pthreads instead. For Pthreads there is no intermediate memory copy required because threads share the same address space within a single process. There is no data transfer, per se. It can be as efficient as simply passing a pointer.
- A perfect example is the typical web browser, where many interleaved tasks can be happening at the same time, and where tasks can vary in priority.

Data Dictionary:

Sr Number	Variable/Function	Datatype	Use
1	t	int	Input variable.
2	tid	pthread_t	Creating a thread in c.
3	st1	void*	Number 1
4	st2	void*	Number 2
5	st3	void*	Number 3
6	st4	void*	Number 4
7	avg	float	Average of above 4 numbers.

Table: 4.6 Data Dictionary

Program:

```
#include <pthread.h>
#include <stdio.h>
void *myThread(){
int t;
scanf("%d",&t);
```

```

return (void *)t;
}

int main()
{
pthread_t tid;
void *st1,*st2,*st3,*st4;
printf("First number\n");
pthread_create(&tid, NULL, myThread, NULL);
pthread_join(tid, &st1);
printf("Second number\n");
pthread_create(&tid, NULL, myThread, NULL);
pthread_join(tid, &st2);
printf("Third number\n");
pthread_create(&tid, NULL, myThread, NULL);
pthread_join(tid, &st3);
printf("Fourth number\n");
pthread_create(&tid, NULL, myThread,
NULL); pthread_join(tid, &st4);
float avg = ((int)st1+(int)st2+(int)st3+(int)st4)/4;
printf("%f\n",avg);
return 0;
}

```

Output:

```
$ ./a.out
```

First number

12

Second number

23

Third number

34

Fourth number

45

28.000000

\$./a.out

First number

100

Second number

200

Third number

400

Fourth number

500

300.000000

Conclusion:

- Use of p_threads in C language to create and synchronize using semaphore can be done.

References:

[1] <https://www.cs.nmsu.edu/~jcook/Tools/pthreads/library.html>

4.7 - Write program to synchronize threads using construct – monitor/serialize/semaphore of Java

Objectives:

1. To learn about threading in Linux/Unix and Java and difference between them
2. Use of system call/library to write effective programs

Theory:

A semaphore controls access to a shared resource through the use of a counter. If the counter is greater than zero, then access is allowed. If it is zero, then access is denied. What the counter is counting are permits that allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore.

Working of semaphore :

In general, to use a semaphore, the thread that wants access to the shared resource tries to acquire a permit.

- If the semaphore's count is greater than zero, then the thread acquires a permit, which causes the semaphore's count to be decremented.
- Otherwise, the thread will be blocked until a permit can be acquired.
- When the thread no longer needs an access to the shared resource, it releases the permit, which causes the semaphore's count to be incremented.
- If there is another thread waiting for a permit, then that thread will acquire a permit at that time.
- Java provide Semaphore class in *java.util.concurrent* package that implements this mechanism, so you don't have to implement your own semaphores.

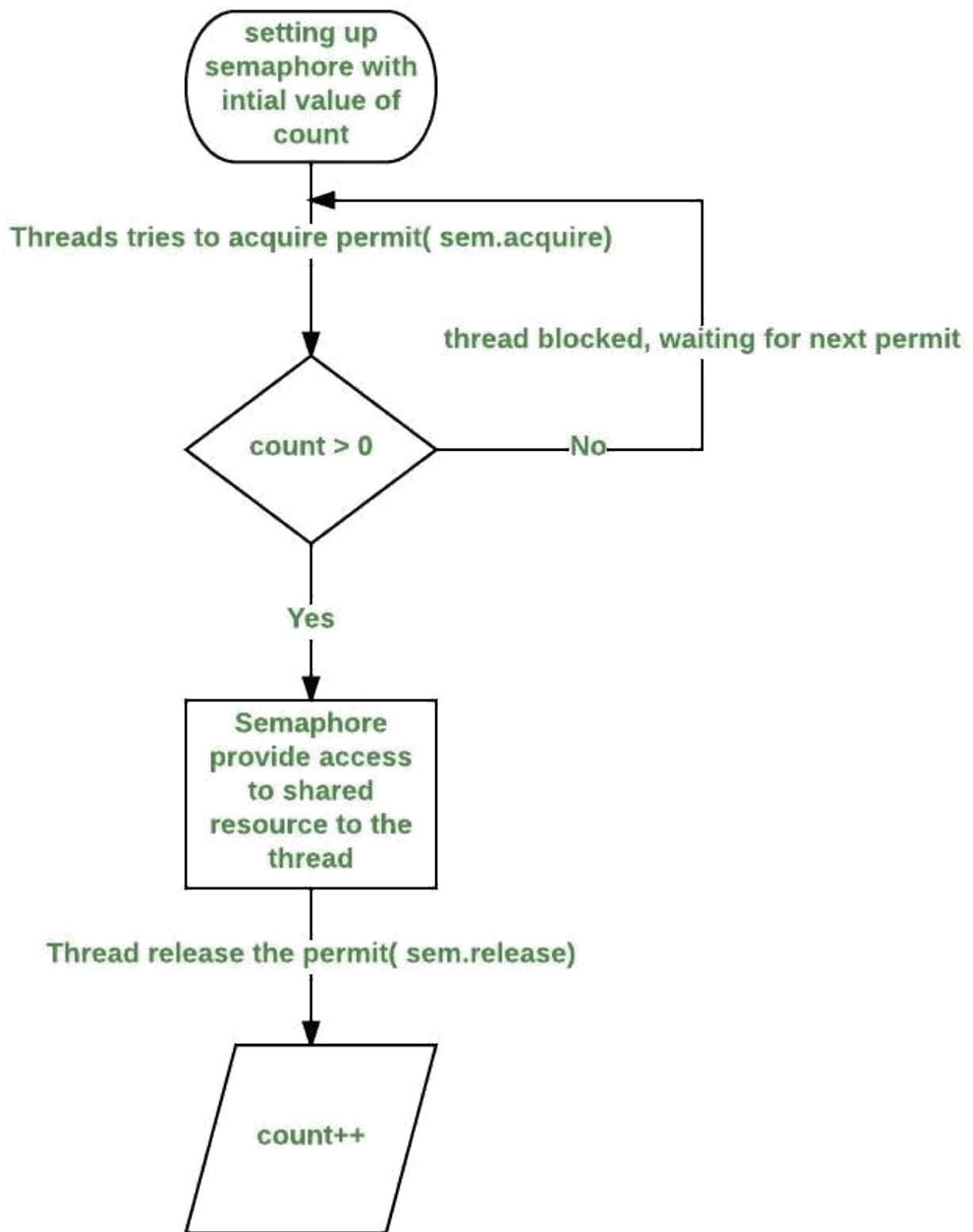


Fig:Working of Semaphore

Data Dictionary:

Sr Number	Variable/Function	Datatype	Use
1	Semaphore	Semaphore	Used to synchronize between the threads.
2	t1	Thread1	Print odd numbers.
3	t2	Thread2	Print prime numbers.
4	tm	G4	Print even numbers.

Table: 4.7 Data Dictionary

Program:

```
import java.util.concurrent.*;
public class G4 extends Thread
{
    public static Semaphore semaphore = new
    Semaphore(1); public static void main(String args[])
    throws Exception {
        Thread1 t1 = new Thread1();
        Thread2 t2 = new Thread2();
        G4 tm = new G4();
        tm.start();
        t1.start();
        t2.start();
    }
    public void run()
    {
        try
        {
```

```

semaphore.acquire();
for(int i=0;i<=30;i=i+2)

{
System.out.println("Even: "+i);
}
semaphore.release();
}
catch(InterruptedException exc){}
}

static class Thread1 extends Thread
{
public void run()
{
try
{
semaphore.acquire();
for(int i=1;i<=31;i=i+2)
{
System.out.println("Odd: "+i);
}
semaphore.release();
}
catch(InterruptedException exc){}
}
}

static class Thread2 extends Thread

```

```

{

public void run()
{
try
{
semaphore.acquire();
for(int i=2;i<100;i++)
{
int count = 0;
for(int j=2;j<i;j++)
{
if(i%j==0)
{
count++;
}
}
if(count==0)
System.out.println("Prime: "+i);
}
semaphore.release();
}
catch(InterruptedException exc){}
}
}
}

```

Output:

```
it@it-OptiPlex-3020:~/Vijay$ javac C4.java
```

```
^[[Ait@it-OptiPlex-3020:~/Vijay$ java C4
```

```
Even: 0
```

```
Even: 2
```

```
Even: 4
```

```
Even: 6
```

```
Even: 8
```

```
Even: 10
```

```
Even: 12
```

```
Even: 14
```

```
Even: 16
```

```
Even: 18
```

```
Even: 20
```

```
Even: 22
```

```
Even: 24
```

```
Even: 26
```

```
Even: 28
```

```
Even: 30
```

```
Odd: 1
```

```
Odd: 3
```

```
Odd: 5
```

```
Odd: 7
```

```
Odd: 9
```

```
Odd: 11
```

Odd: 13

Odd: 15

Odd: 17

Odd: 19

Odd: 21

Odd: 23

Odd: 25

Odd: 27

Odd: 29

Odd: 31

Prime: 2

Prime: 3

Prime: 5

Prime: 7

Prime: 11

Prime: 13

Prime: 17

Prime: 19

Prime: 23

Prime: 29

Prime: 31

Prime: 37

Prime: 41

Prime: 43

Prime: 47

Prime: 53

Prime: 59

Prime: 61

Prime: 67

Prime: 71

Prime: 73

Prime: 79

Prime: 83

Prime: 89

Prime: 97

Conclusion:

- Synchronization of multiple threads using semaphore to let threads work synchronously to produce desirable outputs learned and implemented in Java

References:

[1] <https://www.geeksforgeeks.org/multithreading-in-java/>

4.8 - Write program using semaphore to ensure that function f1 should executed after f2 in java.

Objectives:

1. To learn about threading in Linux/Unix and Java and difference between them..
2. Use of system call/library to write effective programs

Theory:

A Semaphore is a thread synchronization construct that can be used either to send signals between threads to avoid missed signals, or to guard a critical section like you would with a lock.

Methods in Semaphore Class:

- void acquire() : This method acquires a permit, if one is available and returns immediately, reducing the number of available permits by one.
- void release() : This method releases a permit, increasing the number of available permits by one. If any threads are trying to acquire a permit, then one is selected and given the permit that was just released.

Working of semaphore :

- In general, to use a semaphore, the thread that wants access to the shared resource tries to acquire a permit.
- If the semaphore's count is greater than zero, then the thread acquires a permit, which causes the semaphore's count to be decremented.
- Otherwise, the thread will be blocked until a permit can be acquired.
- When the thread no longer needs an access to the shared resource, it releases the permit, which causes the semaphore's count to be incremented.
- If there is another thread waiting for a permit, then that thread will acquire a permit at that time.

Java provide Semaphore class in *java.util.concurrent* package that implements this mechanism, so you don't have to implement your own semaphores.

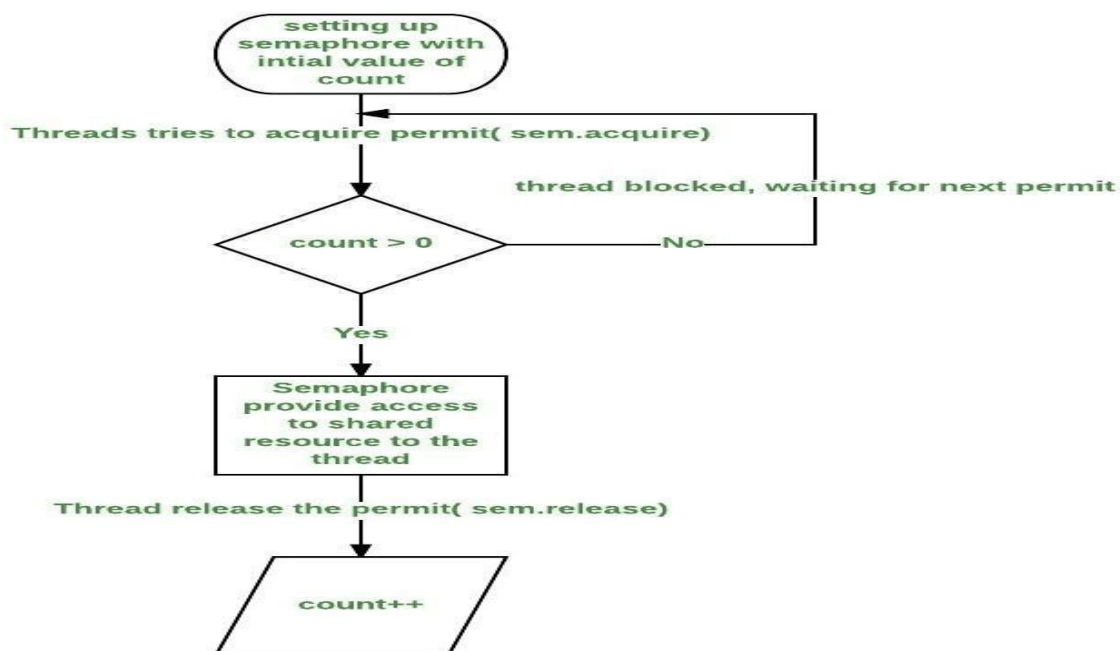


Fig:Working of Semaphore

Data Dictionary:

Sr Number	Variable/Function	Datatype	Use
1	sem	Semaphore	Used to synchronize between the threads.
2	mt1	MyThread	Thread 1.
3	mt2	MyThread	Thread 2.
4	threadName	String	Name of the thread.

Table : 4.8 Data Dictionary

Program:

```
import java.util.concurrent.*;
class Shared
{
    static int count = 0;
}
class MyThread extends Thread
{
    Semaphore sem;
    String threadName;
    public MyThread(Semaphore sem, String threadName)
    {
        super(threadName);
        this.sem = sem;
        this.threadName = threadName;
    }
```

@Override

```

public void run() {

    if(this.getName().equals("A"))
    {
        System.out.println("Starting " + threadName);
        try
        {
            System.out.println(threadName + " is waiting for a permit.");
            sem.acquire();

            System.out.println(threadName + " gets a permit.");
            for(int i=0; i < 5; i++)
            {
                Shared.count++;
                System.out.println(threadName + ": " + Shared.count);
                Thread.sleep(1000);
            }
        } catch (InterruptedException exc)
        { System.out.println(exc);
        }

        System.out.println(threadName + " releases the
        permit."); sem.release();
    }
    else
    {

```

```

System.out.println("Starting " +
threadName); try

{
System.out.println(threadName + " is waiting for a permit.");
sem.acquire();
System.out.println(threadName + " gets a permit.");
for(int i=0; i < 5; i++)
{
Shared.count--;
System.out.println(threadName + ": " + Shared.count);
Thread.sleep(1000);
}
} catch (InterruptedException exc)
{ System.out.println(exc);
}
System.out.println(threadName + " releases the
permit."); sem.release();
}
}
}

public class H4
{
public static void main(String args[]) throws InterruptedException
{

```

```
Semaphore sem = new Semaphore(1);  
MyThread mt1 = new MyThread(sem, "A");  
  
MyThread mt2 = new MyThread(sem, "B");  
mt1.start();  
mt2.start();  
mt1.join();  
mt2.join();  
System.out.println("count: " + Shared.count);  
}  
}
```

Output:

Starting B

B is waiting for a permit.

Starting A

A is waiting for a permit.

B gets a permit.

B: -1

B: -2

B: -3

B: -4

B: -5

B releases the permit.

A gets a permit.

A: -4

A: -3

A: -2

A: -1

A: 0

A releases the permit.

count: 0

Conclusion:

- Synchronization of functions using semaphores in Java by acquire and release of semaphores studied and implemented.

References:

[1] <https://www.geeksforgeeks.org/multithreading-in-java/>

Chapter 5

Shell Programming: Shell Scripts

5.1 Write a program to implement a shell script for calculator

Objectives:

To learn shell programming and use it for write effective programs.

Theory:

Theory: The shell provides you with an interface to the UNIX system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

A shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of shells, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.

Flowchart:

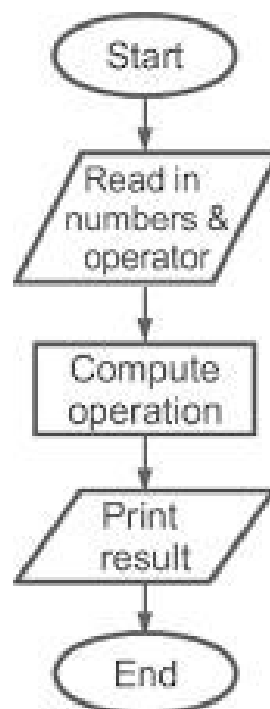


Fig : 5.1 Flowchart

Data Variables:

no	Variable	Type	use
1	i	int	do while loop
2	n1	int	calculation
3	n2	int	calculation

Table : 5.1 Data Dictionary

Program:

```
clear
sum=0
i="y"
echo " Enter one no."
read n1
echo "Enter second no."
read n2
while [ $i = "y" ]
do
    echo "1.Addition"
    echo "2.Subtraction"
    echo "3.Multiplication"
    echo "4.Division"
    echo "Enter your choice"
    read ch
    case $ch in
        1)sum=`expr $n1 + $n2`
        echo "Sum ="$sum;;
        2)sum=`expr $n1 - $n2`
        echo "Sub = "$sum;;
        3)sum=`expr $n1 \* $n2`
        echo "Mul = "$sum;;
```

```
4)sum=`expr $n1 / $n2`  
    echo "Div = "$sum;;  
    *)echo "Invalid choice";;  
    esac  
    echo "Do u want to continue ?"  
    read i  
    if [ $i != "y" ]  
    then  
        exit  
    fi  
done
```

Output:

it@it-OptiPlex-3sk046:~/Vijay\$ sh ./5a.sh

Enter any no.

121

Enter one no.

21

Enter second no.

58

1.Addition

2.Subtraction

3.Multiplication

4.Division

Enter your choice

1

Sum =79

Do u want to continue ?

y

1.Addition

2.Subtraction

3.Multiplication

4.Division

Enter your choice

2

Sub = -37

Do u want to continue ?

y

1.Addition

2.Subtraction

3.Multiplication

4.Division

Enter your choice

3

Mul = 1218

Do u want to continue ?

y

1.Addition

2.Subtraction

3.Multiplication

4.Division

Enter your choice

4

Div = 0

Do u want to continue ?

n

Conclusion:

- Calculator constructed using shell programming.

References:

[1] <https://www.tutorialspoint.com/unix/unix-what-is-shell.htm/>

5.2 - Write a program to implement a digital clock using shell script.

Objectives:

To learn shell programming and use it for write effective programs

Theory:

The shell provides you with an interface to the UNIX system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

A shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of shells, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.

Flowchart:

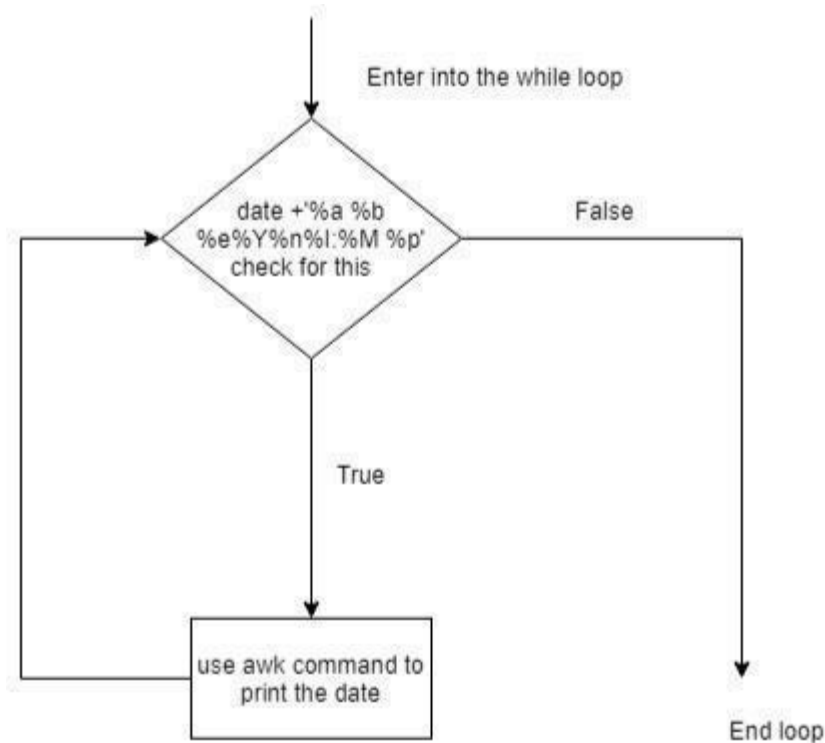


Fig: 5.2 Flowchart

Program:

```
while true do
    date +%a %b %e %Y%n%I:%M %p
done | awk '!seen[$0]++'
```

Output:

\$Vijay@victorykumar :~/Downloads/5b\$ sh 5b.sh Sat Apr 23 2019 05:31 PM ^C \$

Conclusion:

- Array sorted constructed using shell programming.

References:

[1] <http://www.tutorialspoint.com/unix/unix-shell.htm/>

5.3 Using shell sort the given 10 number in ascending order (use of array).

Objectives:

To learn shell programming and use it for write effective programs.

Theory:

The shell provides you with an interface to the UNIX system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

A shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of shells, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.

Flowchart:

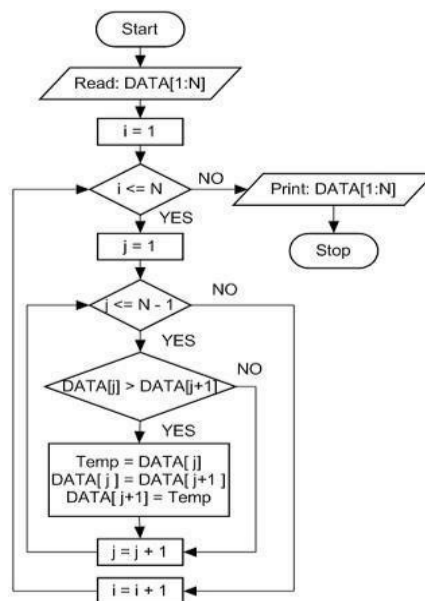


Fig: 5.3 Flowchart

Data Variables:

no	Variable	Type	use
1	a[i]	array	array
2	i	int	outer loop
3	j	int	inner loop

Table : 5.3 Data Dictionary

Program:

```
for i in {0..9}
do
    echo "Enter the element"
    read a[i]
done
#array display before sorting
echo "The array elements are:"
for i in {0..9}
do
    echo ${a[i]}
done

#algorithm to sort the array..bubble sort
for ((i=0;i<10;i++))
do
    for (( j = $i; j < 10; j++ ))
    do
        if [ ${a[$i]} -gt ${a[$j]} ]; then
            t=${a[$i]}
            a[$i]=${a[$j]}
            a[$j]=$t
        fi
    done
done
```

```
done
#array display after sorting
echo "The array elements are:"
for i in {0..9}
do
    echo ${a[i]}
done
```

Output:

```
it@it-Vijay$ ./5c.sh
```

Enter the element

1

Enter the element

5

Enter the element

9

Enter the element

2

Enter the element

3

Enter the element

0

Enter the element

11

Enter the element

67

Enter the element

54

Enter the element

90

The array elements are:

1
5
9
2
3
0
11
67
54

The array elements are:

0
1
2
3
5
9
11
54
67

Conclusion:

- Array sorted constructed using shell programming.

References:

[1]<http://www.tutorialspoint.com/unix/unix-shell.htm/>

5.4 - Shell script to print "Hello World" message, in Bold, Blink effect, and in different colors like red, brown etc.

Objectives:

To learn shell programming and use it for write effective programs.

Theory:

The shell provides you with an interface to the UNIX system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

A shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of shells, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.

Flowchart:

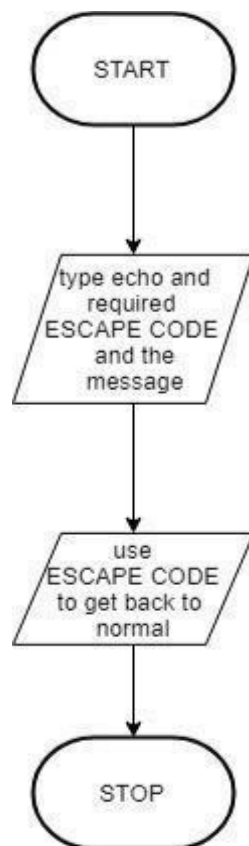


Fig: 5.4 Flowchart

Program:

```
clear
```

```
echo -e "\033[1m Hello World"
```

```
# bold effect
```

```
echo -e "\033[5m Hello World"
```

```
# blink effect
```

```
echo -e "\033[0m Hello World"
```

```
# back to normal
```

```
echo -e "\033[31m Hello World"
```

```
# Red color
```

```
echo -e "\033[32m Hello World"
```

```
# Green color
```

```
echo -e "\033[33m Hello World"
```

```
# See remaining on screen
```

```
echo -e "\033[34m Hello World"
```

```
echo -e "\033[35m Hello World"
```

```
echo -e "\033[36m Hello World"
```

```
echo -e -n "\033[0m "
```

```
# back to normal
```

```
echo -e "\033[41m Hello World"
```

```
echo -e "\033[42m Hello World"
```

```
echo -e "\033[43m Hello World"
```

```
echo -e "\033[44m Hello World"
```

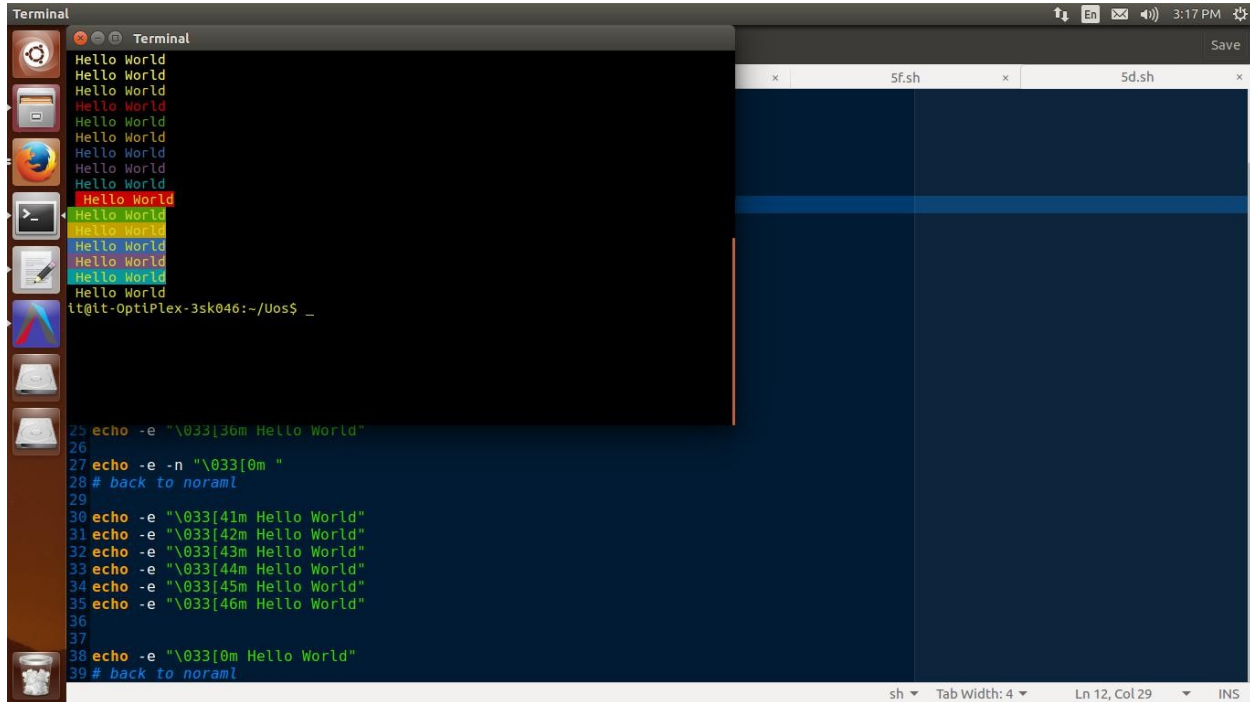
```
echo -e "\033[45m Hello World"
```

```
echo -e "\033[46m Hello World"
```

```
echo -e "\033[0m Hello World"
```

```
# back to normal
```

Output:



```
tt@it-OptiPlex-3sk046:~/Uos$ _
25 echo -e "\033[36m Hello World"
26
27 echo -e -n "\033[0m "
28 # back to nora ml
29
30 echo -e "\033[41m Hello World"
31 echo -e "\033[42m Hello World"
32 echo -e "\033[43m Hello World"
33 echo -e "\033[44m Hello World"
34 echo -e "\033[45m Hello World"
35 echo -e "\033[46m Hello World"
36
37
38 echo -e "\033[0m Hello World"
39 # back to nora ml
```

Conclusion:

- Given text effectively styled in various formats
- Learned aspects of shell scripting.

References:

[1] <http://www.tutorialspoint.com/unix/unix-shell.htm/>

5.5 Shell script to determine whether given file exists or not.

Objectives:

To learn shell programming and use it for write effective programs.

Theory:

The shell provides you with an interface to the UNIX system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

A shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of shells, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.

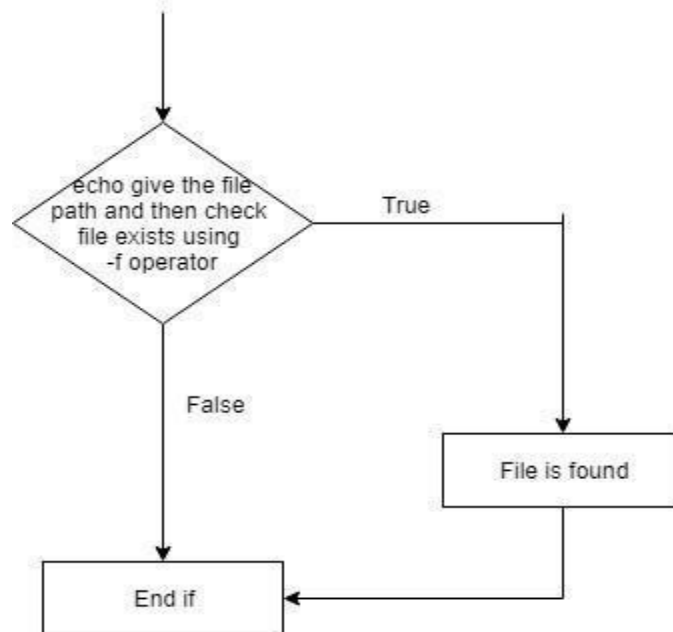
Flowchart:

Fig: 5.5 Flowchart

Data Variables:

no	Variable	Type	Use
1	file	file	Pathname

Table : 5.5 Data Dictionary

Program:

```
echo "Enter file path to check it is available or not"
read file
if [ -f "$file" ]
then
    echo "$file found."
else
    echo "$file not found."
fi
```

Output:

```
it@it-Vijay Sakhare $ ./5e.sh
```

Enter file path to check it is available or not

/home/it/Uos/5atheory

3/home/it/Uos/5atheory found.

it@it-OptiPlex-Vijay-Sakahre\$./5e.sh

Enter file path to check it is available or not

/home/it/5atheory

/home/it/5atheory not found.

Conclusion:

File path verified using shell programming

References:

<http://www.tutorialspoint.com/unix/unix-shell.htm/>

5.6 - Import python script in shell script.

Objectives:

To learn shell programming and use it for write effective programs.

Theory:

The shell provides you with an interface to the UNIX system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

A shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of shells, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.

Flowchart:

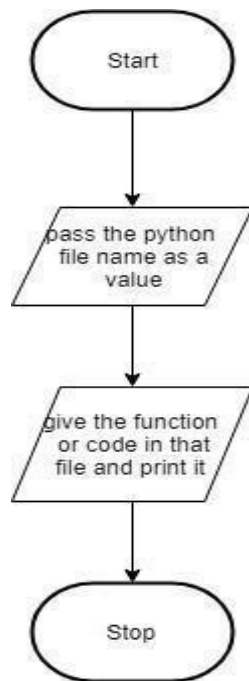


Fig: 5.6 Flowchart

Data Variables:

no	Variable	Type	use
1	value	any	import python

Table: 5.6 Data Dictionary

Program:

5f.sh

```
echo "This is shell script"
value=$(python p1.py)
echo $value
```

p1.py

```
def function():
    return "This is python script"
if __name__ == "__main__":
    print(function())
```

Output:

```
it@it-OptiPlex-Vijay-Sakhre$ ./5f.sh
```

```
This is shell script
```

```
This is python script
```

```
it@it-OptiPlex-Vijay-Sakahre$
```

Conclusion:

- Python script is imported using shell script

References:

[1] <http://www.tutorialspoint.com/unix/unix-shell.htm/>

Chapter 6

IPC:Semaphores

6.1 Write a program to illustrate the semaphore concept. Use fork so that 2 process running simultaneously and communicate via semaphore.

Objectives:

1. To learn about IPC through semaphore.
2. Use of system call and IPC mechanism to write effective application programs.

Theory:

A semaphore controls access to a shared resource through the use of a counter. If the counter is greater than zero, then access is allowed. If it is zero, then access is denied. What the counter is counting are permits that allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore.

Working of semaphore :

In general, to use a semaphore, the thread that wants access to the shared resource tries to acquire a permit. If the semaphore's count is greater than zero, then the thread acquires a permit, which causes the semaphore's count to be decremented. Otherwise, the thread will be blocked until a permit can be acquired. When the thread no longer needs an access to the shared resource, it releases the permit, which causes the semaphore's count to be incremented. If there is another thread waiting for a permit, then that thread will acquire a permit at that time. The function `semget()` initializes or gains access to a semaphore.

It is prototyped by: `int semget(key_t key, int nsems, int semflg);`

When the call succeeds, it returns the semaphore ID (`semid`). The `key` argument is a access value associated with the semaphore ID. The `nsems` argument specifies the number of elements in a semaphore array. The call fails when `nsems` is greater than the number of elements in an existing array; when the correct count is not known, supplying 0 for this argument ensures that it will succeed. POSIX Semaphores: `<semaphore.h>`

`sem_open()` -- Connects to, and optionally creates, a named semaphore

`sem_init()` -- Initializes a semaphore structure (internal to the calling program, so not a named semaphore).

`sem_close()` -- Ends the connection to an open semaphore.

`sem_unlink()` -- Ends the connection to an open semaphore and causes the semaphore to be removed when the last process closes it.

`sem_destroy()` -- Initializes a semaphore structure (internal to the calling program, so not a named semaphore).

`sem_getvalue()` -- Copies the value of the semaphore into the specified integer.

`sem_wait()`, `sem_trywait()` -- Blocks while the semaphore is held by other processes or returns an error if the semaphore is held by another process.

`sem_post()` -- Increments the count of the semaphore.

Data Dictionary:

Number	Variable/function	Data Type	Use
1	pid	int	Get Process ID
2	semflg	int	Flag to pass to semget
3	semid	int	Id of semaphore
4	key	key_t	Key to pass to semget
5	nops	int	Number of Operations
6	sops	Struct sembuf	Pointer to operations to perform

Table: 6.1 Data Dictionary

Program:-

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

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

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

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

```
union semun {
```

```
int val;
```

```

struct semid_ds *buf;

ushort *array;

};

main()

{ int i,j;

int pid;

int semid; /* semid of semaphore set */

key_t key = 1234; /* key to pass to semget() */

int semflg = IPC_CREAT | 0666; /* semflg to pass to semget()

*/ int nsems = 1; /* nsems to pass to semget() */

int nsops; /* number of operations to do */

struct sembuf *sops = (struct sembuf *) malloc(2*sizeof(struct sembuf));

/* ptr to operations to perform */

/* set up semaphore */

(void) fprintf(stderr, "\nsemget: Setting up semaphore: semget(%#lx,

%\ %#o)\n",key, nsems, semflg);

if ((semid = semget(key, nsems, semflg)) == -1)

{ perror("semget: semget failed");

exit(1)

; } else

(void) fprintf(stderr, "semget: semget succeeded: semid =\

%d\n", semid)

/* get child process */

```

```

if ((pid = fork()) < 0) {
    perror("fork");
    exit(1);
}
if (pid == 0)
{ /* child
    */ i = 0;

    while (i < 3) { /* allow for 3 semaphore sets */
        nsops = 2;

        /* wait for semaphore to reach zero */ sops[0].sem_num = 0; /* We only
        use one track */ sops[0].sem_op = 0; /* wait for semaphore flag to
        become zero */ sops[0].sem_flg = SEM_UNDO; /* take off semaphore
        asynchronous */ sops[1].sem_num = 0;

        sops[1].sem_op = 1; /* increment semaphore -- take control of track */
        sops[1].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore */

        /* Recap the call to be made. */

        (void) fprintf(stderr, "\nsemop:Child Calling semop(%d, &sops, %d) with:", semid, nsops);
        for (j = 0; j < nsops; j++)
        {
            (void) fprintf(stderr, "\n\t sops[%d].sem_num = %d, ", j, sops[j].sem_num);
            (void) fprintf(stderr, "sem_op = %d, ", sops[j].sem_op);

            (void) fprintf(stderr, "sem_flg = %#o\n", sops[j].sem_flg);

```

```

}

/* Make the semop() call and report the results.

*/ if ((j = semop(semid, sops, nsops)) == -1)

{ perror("semop: semop failed"); }

else

{

(void) fprintf(stderr, "\tsemop: semop returned %d\n", j);


(void) fprintf(stderr, "\n\nChild Process Taking Control of Track: %d/3 times\n", i+1);

sleep(5); /* DO Nothing for 5 seconds */

nsops = 1;

/* wait for semaphore to reach zero */

sops[0].sem_num = 0;

sops[0].sem_op = -1; /* Give UP Control of track */

sops[0].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore, asynchronous */

if ((j = semop(semid, sops, nsops)) == -1) {

perror("semop: semop failed");

}

else

(void) fprintf(stderr, "Child Process Giving up Control of Track: %d/3 times\n", i+1);


sleep(5); /* halt process to allow parent to catch semaphor change first */

```

```

}
++i;
}
}

else /* parent */

{ /* pid hold id of child */

i = 0;

while (i < 3) { /* allow for 3 semaphore sets */

nsops = 2;

/* wait for semaphore to reach zero

*/ sops[0].sem_num = 0;

sops[0].sem_op = 0; /* wait for semaphore flag to become zero */

sops[0].sem_flg = SEM_UNDO; /* take off semaphore asynchronous */

sops[1].sem_num = 0;

sops[1].sem_op = 1; /* increment semaphore -- take control of track */

sops[1].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore */

/* Recap the call to be made. */

(void) fprintf(stderr, "\nsemop:Parent Calling semop(%d, &sops, %d) with:", semid,
nsops); for (j = 0; j < nsops; j++)

{

(void) fprintf(stderr, "\n\tsops[%d].sem_num = %d, ", j, sops[j].sem_num);

(void) fprintf(stderr, "sem_op = %d, ", sops[j].sem_op);

```

```

(void) fprintf(stderr, "sem_flg = %#o\n", sops[j].sem_flg);

}

/* Make the semop() call and report the results.

*/ if ((j = semop(semid, sops, nsops)) == -1)

{ perror("semop: semop failed"); }

else

{

(void) fprintf(stderr, "semop: semop returned %d\n", j);

(void) fprintf(stderr, "Parent Process Taking Control of Track: %d/3 times\n", i+1);

sleep(5); /* Do nothing for 5 seconds */

nsops = 1;

/* wait for semaphore to reach zero */

sops[0].sem_num = 0;

sops[0].sem_op = -1; /* Give UP COntrol of track

*/ if ((j = semop(semid, sops, nsops)) == -1)

{ perror("semop: semop failed"); }

else

(void) fprintf(stderr, "Parent Process Giving up Control of Track: %d/3 times\n", i+1);

sleep(5); /* halt process to allow child to catch semaphor change first */ }

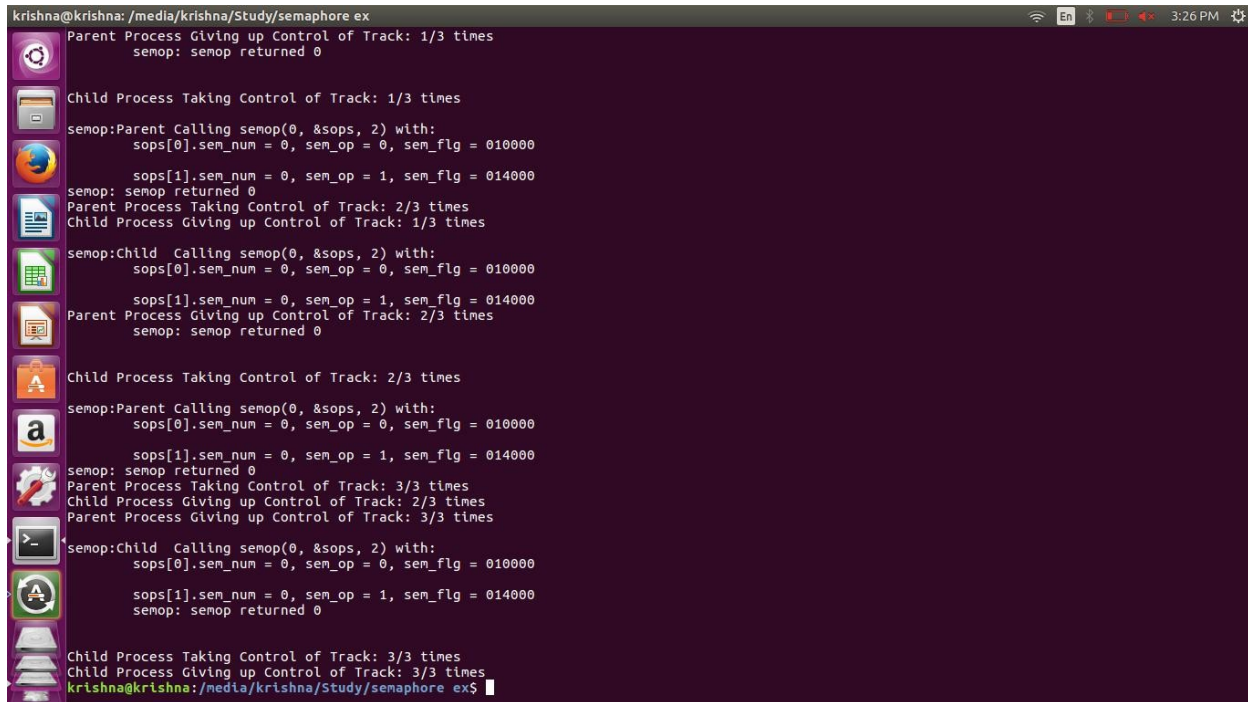
++i;

}

```

```
}  
  
}
```

Output:



```
krishna@krishna: /media/krishna/Study/semaphore ex
Parent Process Giving up Control of Track: 1/3 times
semop: semop returned 0

Child Process Taking Control of Track: 1/3 times
semop:Parent Calling semop(0, &sops, 2) with:
sops[0].sem_num = 0, sem_op = 0, sem_flg = 010000
sops[1].sem_num = 0, sem_op = 1, sem_flg = 014000
semop: semop returned 0
Parent Process Taking Control of Track: 2/3 times
Child Process Giving up Control of Track: 1/3 times
semop:Child Calling semop(0, &sops, 2) with:
sops[0].sem_num = 0, sem_op = 0, sem_flg = 010000
sops[1].sem_num = 0, sem_op = 1, sem_flg = 014000
Parent Process Giving up Control of Track: 2/3 times
semop: semop returned 0

Child Process Taking Control of Track: 2/3 times
semop:Parent Calling semop(0, &sops, 2) with:
sops[0].sem_num = 0, sem_op = 0, sem_flg = 010000
sops[1].sem_num = 0, sem_op = 1, sem_flg = 014000
semop: semop returned 0
Parent Process Taking Control of Track: 3/3 times
Child Process Giving up Control of Track: 2/3 times
Parent Process Giving up Control of Track: 3/3 times
semop:Child Calling semop(0, &sops, 2) with:
sops[0].sem_num = 0, sem_op = 0, sem_flg = 010000
sops[1].sem_num = 0, sem_op = 1, sem_flg = 014000
semop: semop returned 0

Child Process Taking Control of Track: 3/3 times
Child Process Giving up Control of Track: 3/3 times
krishna@krishna: /media/krishna/Study/semaphore ex$
```

Conclusion:

- Use of semaphore for IPC where one process is child of other and in same program using various system calls like semget,semctl is studied.

References:

[1]<https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/14/lec.html>

6.2 - Write a program to illustrate the semaphore concept. Use fork so that 2 process running simultaneously and communicate via semaphore.

Objectives:

1. To learn about IPC through semaphore.
2. Use of system call and IPC mechanism to write effective application programs.

Theory:

The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Data Dictionary:

Number	Variable/function	Data Type	Use
1	Producers	pthread_t	Process Thread of Producer
2	Consumer	pthread_t	Process Thread of Consumer
3	buf_mutex	sem_t	To process wait condition
4	empty_count	sem_t	Keeps track of empty count
5	fill_count	sem_t	Keeps track of fill count
6	consumer	void	Used to regulate consumer action
7	producer	void	Used to regulate producer action

Table: 6.2 Data Dictionary

Program :

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

/* use the pthread flag with gcc to compile this code
~$ gcc -pthread producer_consumer.c -o producer_consumer */

pthread_t *producers;
pthread_t *consumers;
```



```

sem_t buf_mutex,empty_count,fill_count;
int *buf,buf_pos=-1,prod_count,con_count,buf_len;

int produce(pthread_t self){ int i = 0;

int p = 1 + rand()%40;
while(!pthread_equal(*(producers+i),self) && i < prod_count){
i++;
}
printf("Producer %d produced %d \n",i+1,p);
return p;
}

void consume(int p,thread_t self){
int i = 0;
while(!pthread_equal(*(consumers+i),self) && i < con_count){
i++;
}
printf("Buffer:");
for(i=0;i<=buf_pos;++i)
printf("%d ",*(buf+i));
printf("\nConsumer %d consumed %d \nCurrent buffer len: %d\n",i+1,p,buf_pos);
}

void* producer(void *args){

while(1){
int p = produce(pthread_self());
sem_wait(&empty_count);
sem_wait(&buf_mutex);
++buf_pos;                // critical section
*(buf + buf_pos) = p;
sem_post(&buf_mutex);
}
}

```

```

sem_post(&fill_count);
sleep(1 + rand()%3);
}
return NULL;
}

void* consumer(void *args){
int c;
while(1){
sem_wait(&fill_count);
sem_wait(&buf_mutex);
c = *(buf+buf_pos);
consume(c, pthread_self());
--buf_pos;
sem_post(&buf_mutex);
sem_post(&empty_count);
sleep(1+rand()%5);
}
return NULL;
}

int main(void){
int i,err;
srand(time(NULL));
sem_init(&buf_mutex,0,1);
sem_init(&fill_count,0,0);
printf("Enter the number of Producers:");
scanf("%d",&prod_count);
producers = (pthread_t*)
malloc(prod_count*sizeof(pthread_t)); printf("Enter the number
of Consumers:"); scanf("%d",&con_count);
consumers = (pthread_t*) malloc(con_count*sizeof(pthread_t));

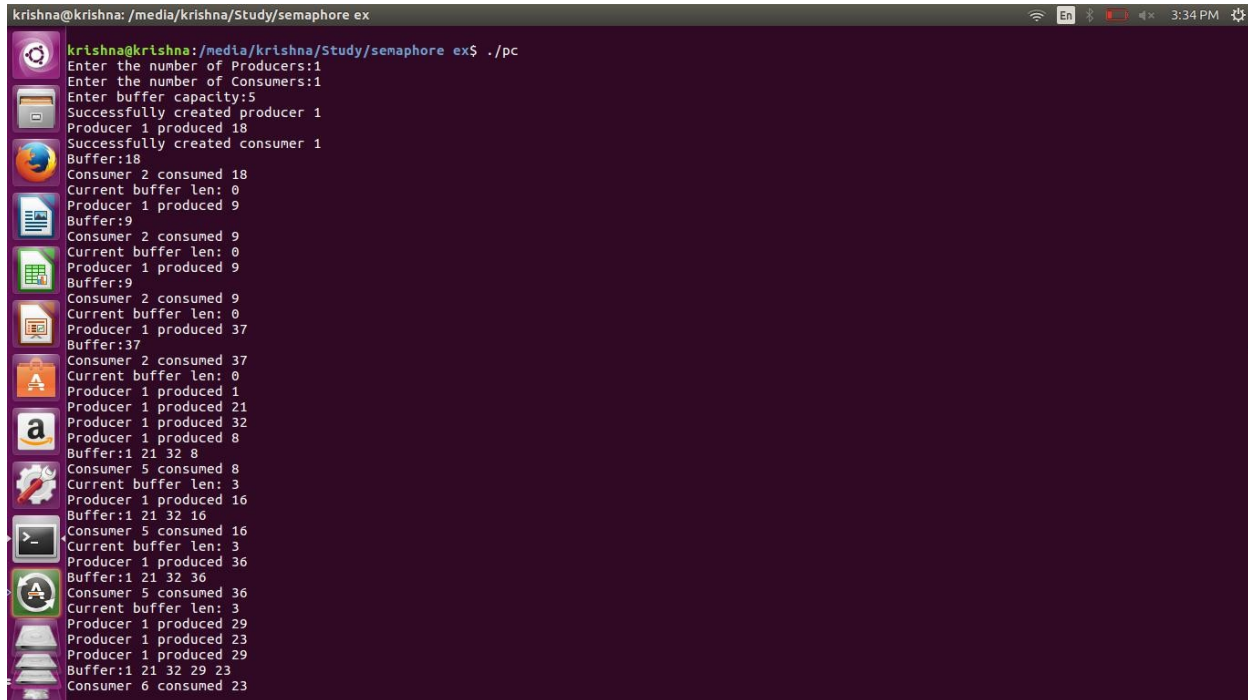
```

```

printf("Enter buffer capacity:");
scanf("%d",&buf_len);
buf = (int*) malloc(buf_len*sizeof(int));
sem_init(&empty_count,0,buf_len);
for(i=0;i<prod_count;i++){
err =
pthread_create(producers+i,NULL,&producer,NULL);
if(err != 0){
printf("Error creating producer %d: %s\
n",i+1,strerror(err)); } else {
printf("Successfully created producer %d\n",i+1);
}
}
for(i=0;i<con_count;i++){
err =
pthread_create(consumers+i,NULL,&consumer,NULL);
if(err != 0){
printf("Error creating consumer %d: %s\
n",i+1,strerror(err)); } else {
printf("Successfully created consumer %d\
n",i+1); }
}
for(i=0;i<prod_count;i++){
pthread_join(*(producers+i),NULL);
}
for(i=0;i<con_count;i++){
pthread_join(*(consumers+i),NULL);
}
return 0;
}

```

Output:



```
krishna@krishna: /media/krishna/Study/semaphore ex
krishna@krishna: /media/krishna/Study/semaphore ex$ ./pc
Enter the number of Producers:1
Enter the number of Consumers:1
Enter buffer capacity:5
Successfully created producer 1
Producer 1 produced 18
Successfully created consumer 1
Buffer:18
Consumer 2 consumed 18
Current buffer len: 0
Producer 1 produced 9
Buffer:9
Consumer 2 consumed 9
Current buffer len: 0
Producer 1 produced 9
Buffer:9
Consumer 2 consumed 9
Current buffer len: 0
Producer 1 produced 37
Buffer:37
Consumer 2 consumed 37
Current buffer len: 0
Producer 1 produced 1
Producer 1 produced 21
Producer 1 produced 32
Producer 1 produced 8
Buffer:1 21 32 8
Consumer 5 consumed 8
Current buffer len: 3
Producer 1 produced 16
Buffer:1 21 32 16
Consumer 5 consumed 16
Current buffer len: 3
Producer 1 produced 36
Buffer:1 21 32 36
Consumer 5 consumed 36
Current buffer len: 3
Producer 1 produced 29
Producer 1 produced 23
Producer 1 produced 29
Buffer:1 21 32 29 23
Consumer 6 consumed 23
```

Conclusion:

- Synchronization using IPC semaphores done to implement and study Producer-Consumer problem.

References:

[1] <https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/14/lec.html>

6.4 Write a program to illustrate the semaphore concept. Use fork so that 2 process running simultaneously and communicate via semaphore.

Objectives:

1. To learn about IPC through semaphore.
2. Use of system call and IPC mechanism to write effective application programs.

Theory:

A semaphore controls access to a shared resource through the use of a counter. If the counter is greater than zero, then access is allowed. If it is zero, then access is denied. What the counter is counting are permits that allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore.

Working of semaphore :

In general, to use a semaphore, the thread that wants access to the shared resource tries to acquire

a permit.

- If the semaphore's count is greater than zero, then the thread acquires a permit, which causes the semaphore's count to be decremented.
- Otherwise, the thread will be blocked until a permit can be acquired.
- When the thread no longer needs an access to the shared resource, it releases the permit, which causes the semaphore's count to be incremented.
- If there is another thread waiting for a permit, then that thread will acquire a permit at that time.

The function `semget()` initializes or gains access to a semaphore. It is prototyped by:

`int semget(key_t key, int nsems, int semflg);` When the call succeeds, it returns the semaphore ID (`semid`).

The `key` argument is a access value associated with the semaphore ID.

The `nsems` argument specifies the number of elements in a semaphore array. The call fails when `nsems` is greater than the number of elements in an existing array; when the correct count is not known, supplying 0 for this argument ensures that it will succeed.

POSIX Semaphores: `<semaphore.h>`

`sem_open()` -- Connects to, and optionally creates, a named semaphore

`sem_init()` -- Initializes a semaphore structure (internal to the calling program, so not a named semaphore).

`sem_close()` -- Ends the connection to an open semaphore.

`sem_unlink()` -- Ends the connection to an open semaphore and causes the semaphore to be removed when the last process closes it.

`sem_destroy()` -- Destroys a semaphore structure (internal to the calling program, so not a named semaphore).

`sem_getvalue()` -- Copies the value of the semaphore into the specified integer.

`sem_wait()`, `sem_trywait()` -- Blocks while the semaphore is held by other processes or returns an error if the semaphore is held by another process.

`sem_post()` -- Increments the count of the semaphore.

Data Dictionary:

Number	Variable/function	Data Type	Use
1	KEY	Long int	External identifier for the program
2	id	int	Number by which semaphore is known within a program
3	argument	Union semun	To pass arguments to the semctl function
4	retval	int	Store return value of semop function

Table: 6.3 Data Dictionary

Program 1: Initialization of Semaphore

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>

/* The semaphore key is an arbitrary long integer which serves as an external identifier by
which the semaphore is known to any program that wishes to use it. */
#define KEY (1492)

void main()
{
    int id; /* Number by which the semaphore is known within a program */

    /* The next thing is an argument to the semctl() function. Semctl() does various things to the
    semaphore depending on which arguments are passed. We will use it to make sure that the value
    of the semaphore is initially 0. */
    union semun {
        int val;
        struct semid_ds *buf;
```

```

ushort * array;
} argument;
argument.val =
0;

/* Create the semaphore with external key KEY if it doesn't already exists. Give permissions
to the world. */
id = semget(KEY, 1, 0666 | IPC_CREAT);
/* Always check system returns. */
if(id < 0)
{
fprintf(stderr, "Unable to obtain semaphore.\n");
exit(0);
}

/* What we actually get is an array of semaphores. The second argument to semget() was the
array dimension - in our case 1. */

/* Set the value of the number 0 semaphore in semaphore array # id to the value 0. */
if( semctl(id, 0, SETVAL, argument) < 0)
{
fprintf( stderr, "Cannot set semaphore value.\n");
}
else
{
fprintf(stderr, "Semaphore %d initialized.\n", KEY);
}
}

```

Program 2: Semaphore A

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#define KEY (1492)

```

```

/* This is the external name by which the semaphore is known to any program that wishes to
access it. */

void main()

{
int id; /* Internal identifier of the semaphore. */
struct sembuf operations[1];

/* An "array" of one operation to perform on the semaphore.
*/ int retval; /* Return value from semop() */

/* Get the index for the semaphore with external name KEY. */
id = semget(KEY, 1, 0666);
if(id < 0)
/* Semaphore does not exist. */
{
fprintf(stderr, "Program sema cannot find semaphore, exiting.\n");
exit(0);
}

/* Do a semaphore V-operation. */
printf("Program sema about to do a V-operation. \
n"); /* Set up the sembuf structure. */

/* Which semaphore in the semaphore array : */
operations[0].sem_num = 0;

/* Which operation? Add 1 to semaphore value : */

operations[0].sem_op = 1;
/* Set the flag so we will wait : */
operations[0].sem_flg = 0;
/* So do the operation! */
retval = semop(id, operations, 1);
if(retval == 0)
{

```



```

printf("Successful V-operation by program sema.\n");
}
else
{
printf("sema: V-operation did not succeed.\n");
perror("REASON");
}
}

```

Program 3: Semaphore B

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define KEY (1492)

/* This is the external name by which the semaphore is known to any program that wishes to
access it. */

void main()
{
int id; /* Internal identifier of the semaphore. */
struct sembuf operations[1];

/* An "array" of one operation to perform on the semaphore. */


int retval; /* Return value from semop() */

/* Get the index for the semaphore with external name KEY. */
id = semget(KEY, 1, 0666);
if(id < 0)
/* Semaphore does not exist. */
{
fprintf(stderr, "Program semb cannot find semaphore, exiting.\n");
exit(0);
}

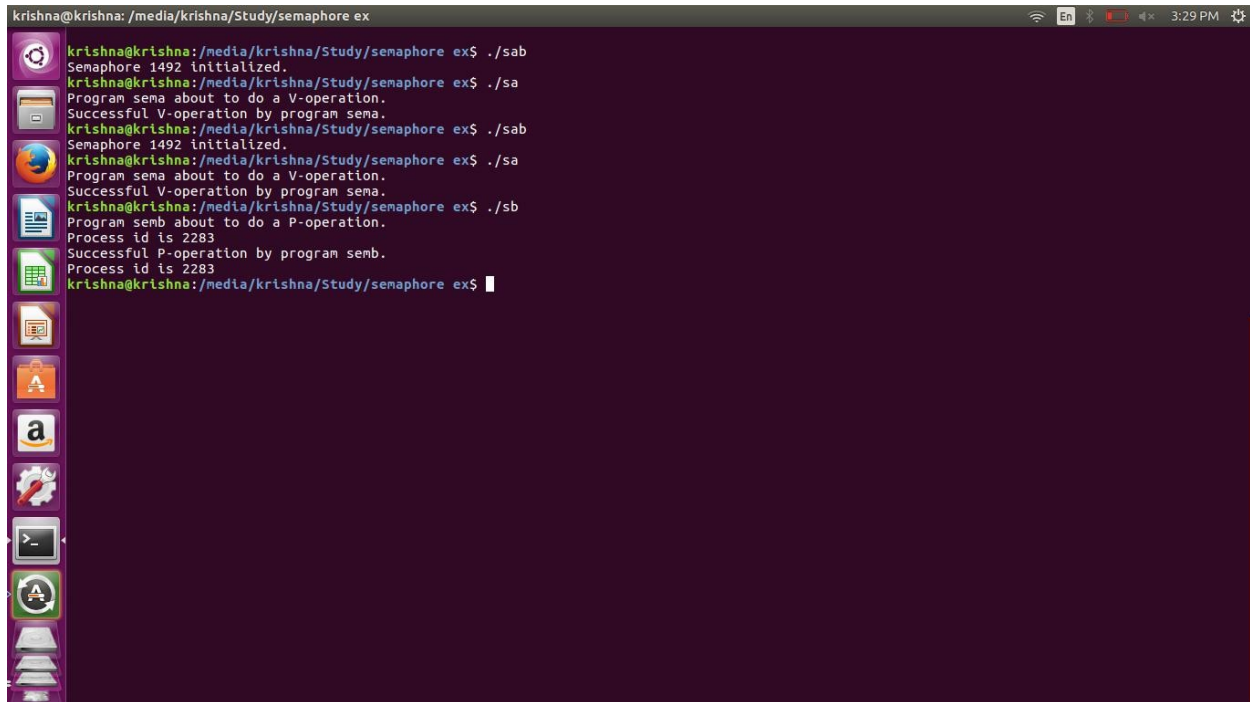
```

```

}
/* Do a semaphore P-operation. */
printf("Program semb about to do a P-operation. \
n"); printf("Process id is %d\n", getpid()); /* Set up
the sembuf structure. */
/* Which semaphore in the semaphore array : */
operations[0].sem_num = 0;
/* Which operation? Subtract 1 from semaphore value : */
operations[0].sem_op = -1;
/* Set the flag so we will wait : */
operations[0].sem_flg = 0;
/* So do the operation! */
retval = semop(id, operations, 1);
if(retval == 0)
{
printf("Successful P-operation by program semb.\
n"); printf("Process id is %d\n", getpid());
else
{
printf("semb: P-operation did not succeed.\n");
}
}
}

```

Output:

A terminal window titled 'krishna@krishna: /media/krishna/Study/semaphore ex' with a dark purple background. The window shows a series of commands and their outputs. The commands are: './sab', './sa', './sab', './sa', and './sb'. The outputs are: 'Semaphore 1492 initialized.', 'Program sema about to do a V-operation.', 'Successful V-operation by program sema.', 'Semaphore 1492 initialized.', 'Program sema about to do a V-operation.', 'Successful V-operation by program sema.', 'Program semb about to do a P-operation.', 'Process id is 2283', 'Successful P-operation by program semb.', and 'Process id is 2283'. The terminal has a sidebar on the left with various application icons and a top status bar showing 'En', signal icons, and the time '3:29 PM'.

```
krishna@krishna: /media/krishna/Study/semaphore ex
krishna@krishna:/media/krishna/Study/semaphore ex$ ./sab
Semaphore 1492 initialized.
krishna@krishna:/media/krishna/Study/semaphore ex$ ./sa
Program sema about to do a V-operation.
Successful V-operation by program sema.
krishna@krishna:/media/krishna/Study/semaphore ex$ ./sab
Semaphore 1492 initialized.
krishna@krishna:/media/krishna/Study/semaphore ex$ ./sa
Program sema about to do a V-operation.
Successful V-operation by program sema.
krishna@krishna:/media/krishna/Study/semaphore ex$ ./sb
Program semb about to do a P-operation.
Process id is 2283
Successful P-operation by program semb.
Process id is 2283
krishna@krishna:/media/krishna/Study/semaphore ex$
```

Conclusion:

- Use of semaphore for two different processes where one has to wait for other to release is studied using IPC Semaphore

References:

[1] <https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/14/lec.html>

Chapter 7

IPC: Message Queue

7.1 Write a program to perform IPC using message and send “ did u get this?”

Theory:

Msgget() -

The msgget() system call returns the System V message queue identifier associated with the value of the *key* argument. A new message queue is created if *key* has the value IPC_PRIVATE or *key* isn't IPC_PRIVATE, no message queue with the given *key* exists, and IPC_CREAT is specified in *msgflg*. If *msgflg* specifies both IPC_CREAT and IPC_EXCL and a message queue already exists for *key*, then msgget() fails with *errno* set to EEXIST. (This is analogous to the effect of the combination O_CREAT | O_EXCL for open(2).) Upon creation, the least significant bits of the argument *msgflg* define the permissions of the message queue.

Msgrcv() -

The *msgrcv()* function reads a message from the queue associated with the message queue identifier specified by *msqid* and places it in the user-defined buffer pointed to by *msgp*.

The argument *msgp* points to a user-defined buffer that must contain first a field of type long int that will specify the type of the message, and then a data portion that will hold the data bytes of the message. The structure below is an example of what this user-defined buffer might look like:

```
struct mymsg { long int mtype; /* message type */ char mtext[1]; /* message text */}
```

The structure member *mtype* is the received message's type as specified by the sending process.

The structure member *mtext* is the text of the message.

Msgsnd() -

The msgsnd() and msgrcv() system calls are used, respectively, to send messages to, and receive messages from, a message queue. The calling process must have write permission on the message queue in order to send a message, and read permission to receive a message.

The *msgp* argument is a pointer to caller-defined structure of the following general form:

```

struct msgbuf    long mtype;    /* message type, must be > 0    char mtext[1];    /* message
{
    data */};

```

The *mtext* field is an array (or other structure) whose size is specified by *msgsz*, a nonnegative integer value. Messages of zero length (i.e., no *mtext* field) are permitted. The *mtype* field must have a strictly positive integer value. This value can be used by the receiving process for message selection (see the description of *msgrcv()* below).

Data/Variables dictionary for sender:

Sr.No	NAME OF VARIABLE/FUNCTION/STRUCT	TYPE	USE	WHERE THE VARIABLE IS CALLED
1	Msqid	int	FOR SOCKET TUPLE	In if statement for the result of msgget method
2	Msgflg	int	FOR SEMAPHORE	As a parameter to msgflg
3	Key	key_t	Semaphore id	Initialising it with some constant value
4	Sbuf	struct msgbuf		For composing message
5	buf_length	size_t		As a parameter to msgsnd function

Table: 7.1 Data Dictionary

Data/Variables dictionary for Receiver:

Sr. No.	NAME OF VARIABLE/FUNCTION/STRUCT	TYPE	USE	WHERE THE VARIABLE IS CALLED
1	msqid	int	FOR SOCKET TUPLE	In if statement for the result of msgget method
3	Key	key_t	Semaphore id	Initialising it with some constant value
4	rbuf	struct msgbuf	To store the message	For composing message
5	buf_length	size_t	Length of message to be sent	As a parameter to msgrcv function

Table: 7.1 Data Dictionary

Algorithm / Flowchart :

Algorithm : send(msqid, msgflg, key, sbuf, buf_length)

1. Initialise key with 1234
2. Assign result of msgget(key, msgflg) to msqid
3. If msqid is less than 0
 - a. Print error
 - b. Exit
4. End if
5. Set message type to in sbuf.mtype
6. Copy string “did you get this?” to sbuf.mtext
7. Assign length of sbuf.mtext to buf_length
8. Apply msgsnd(msqid, &sbuf, buf_length)
 - a. If returned –ve
 - i. exit
 - b. End if

- c. Else
 - i. Print “message sent”
- 9. End send()

Algorithm : receive()

- 1. Initialise key with 1234
- 2. Assign result of msgget(key, msgflg) to msqid
- 3. If msqid is less than 0
 - a. Print error
 - b. Exit
- 4. End if
- 5. Receive message using msgrcv(msqid, &rbuf, MSGSZ, 1, 0)
 - a. If returned –ve result
 - i. Print error
 - ii. Exit
 - b. End if
 - c. Else
 - i. Print rbuf.mtext
- 6. End receive()

Programs

Sender :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <string.h>
#define MSGSZ 128
/* Declare the message structure.*/
```

```

typedef struct msgbuf {
    long  mtype;
    char  mtext[MSGSZ];
} message_buf;

main()
{
    int msqid;
    int msgflg = IPC_CREAT |
0666; key_t key;
    message_buf sbuf;
    size_t
    buf_length; /*
 * Get the message queue id for the
 * "name" 1234, which was created by
 * the server.
 */
    key = 1234;
    (void) fprintf(stderr, "\nmsgget: Calling
msgget(%#lx,\ %#o)\n",
key, msgflg);
    if ((msqid = msgget(key, msgflg )) < 0)
    { perror("msgget");
    exit(1);
    }
    else
    (void) fprintf(stderr, "msgget: msgget succeeded: msqid = %d\n", msqid);

    /*
 * We'll send message type
1 */
    sbuf.mtype = 1;

```



```

(void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);
(void) strcpy(sbuf.mtext, "Did you get this?");
(void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n",
msqid); buf_length = strlen(sbuf.mtext) + 1 ;
/*
* Send a message.
*/
if (msgsnd(msqid, &sbuf, buf_length, IPC_NOWAIT) < 0) {
printf ("%d, %d, %s, %d\n", msqid, sbuf.mtype, sbuf.mtext, buf_length);
perror("msgsnd");
exit(1);
}
else
printf("Message: \"%s\" Sent\n", sbuf.mtext);
exit(0);
}

```

Receiver :

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#define MSGSZ 128
/*
* Declare the message structure.
*/
typedef struct msgbuf {
long mtype;
char mtext[MSGSZ];
} message_buf;
main()

```

```

{
int msqid; key_t
key; message_buf
rbuf; /*

* Get the message queue id for the
* "name" 1234, which was created by
* the server.
*/

key = 1234;
if ((msqid = msgget(key, 0666)) < 0)
{ perror("msgget");
exit(1);
}
/*

* Receive an answer of message type
1. */
if (msgrcv(msqid, &rbuf, MSGSZ, 1, 0) < 0) {
perror("msgrcv");
exit(1);
}
/* Print the answer.*/

printf("%s\n", rbuf.mtext);
exit(0);
}

```

Output Sender:

```

it@it-OptiPlex-3046:~/Vijay-Sakhare$ ./send msgget: Calling
msgget(0x4d2,01666) msgget: msgget succeeded: msqid = 0
msgget: msgget succeeded: msqid = 0

```

msgget: msgget succeeded: msqid = 0

Message: "Did you get this?" Sent

Output Reciever:

it@it-OptiPlex-3046:~/Vijay-Sakhare\$./rec Did you get this?

7. b) IPC: Message Queues: msgget, msgsnd,

msgrcv. Objectives:

To learn about IPC through message queue.

Use of system call and IPC mechanism to write effective application programs.

Conclusions:

- Use of message queue functions like msgget, msgsend, and msgrcv to implement message passing mechanism between server and client studied.

References:

[1] <https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/14/lec.html>

7.2 Write a 2 programs that will both send and messages and construct the following dialog between them

IPC:Message Queues:<sys/msg.h>

Theory:

Two (or more) processes can exchange information via access to a common system message queue. The sending process places via some (OS) message-passing module a message onto a queue which can be read by another process (Figure 24.1). Each message is given an identification or type so that processes can select the appropriate message. Process must share a common key in order to gain access to the queue in the first place.

Basic Message Passing IPC messaging lets processes send and receive messages, and queue messages for processing in an arbitrary order. Unlike the file byte-stream data flow of pipes, each IPC message has an explicit length. Messages can be assigned a specific type. Because of this, a server process can direct message traffic between clients on its queue by using the client

process PID as the message type. For single-message transactions, multiple server processes can work in parallel on transactions sent to a shared message queue.

Before a process can send or receive a message, the queue must be initialized

Operations to send and receive messages are performed by the `msgsnd()` and `msgrcv()` functions, respectively.

When a message is sent, its text is copied to the message queue. The `msgsnd()` and `msgrcv()` functions can be performed as either blocking or non-blocking operations. Non-blocking operations allow for asynchronous message transfer -- the process is not suspended as a result of sending or receiving a message. In blocking or synchronous message passing the sending process cannot continue until the message has been transferred or has even been acknowledged by a receiver. IPC signal and other mechanisms can be employed to implement such transfer. A blocked message operation remains suspended until one of the following three conditions occurs:

- 1.The call succeeds.
- 2.The process receives a signal.
- 3.The queue is removed.

1. Initialising the Message Queue

The `msgget()` function initializes a new message queue:

```
int msgget(key_t key, int msgflg)
```

It can also return the message queue ID (`msqid`) of the queue corresponding to the key argument. The value passed as the `msgflg` argument must be an octal integer with settings for the queue's permissions and control flags.

2. Controlling message queues

The `msgctl()` function alters the permissions and other characteristics of a message queue. The owner or creator of a queue can change its ownership or permissions using `msgctl()`. Also, any process with permission to do so can use `msgctl()` for control operations.

The `msgctl()` function is prototypes as follows:

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf )
```

3. Sending and Receiving Messages

The `msgsnd()` and `msgrcv()` functions send and receive messages, respectively:

```
int msgsnd(int msqid, const void *msgp, size_t msgsz,  
int msgflg);  
  
int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,  
int msgflg);
```

The `msqid` argument must be the ID of an existing message queue. The `msgp` argument is a pointer to a structure that contains the type of the message and its text. The structure below is an example of what this user-defined buffer might look like:

```
struct mymsg {  
    long    mtype; /* message type */  
    char mtext[MSGSZ]; /* message text of length MSGSZ */  
}
```

The `msgsz` argument specifies the length of the message in bytes.

The structure member `msgtype` is the received message's type as specified by the sending process.

The argument `msgflg` specifies the action to be taken if one or more of the following are true:

The number of bytes already on the queue is equal to `msg_qbytes`.

The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are as follows:

If `(msgflg & IPC_NOWAIT)` is non-zero, the message will not be sent and the calling process will return immediately.

If `(msgflg & IPC_NOWAIT)` is 0, the calling process will suspend execution until one of the following occurs:

The condition responsible for the suspension no longer exists, in which case the message is sent.

The message queue identifier `msqid` is removed from the system; when this occurs, `errno` is set equal to `EIDRM` and -1 is returned.

The calling process receives a signal that is to be caught; in this case the message is not sent and the calling process resumes execution.

Upon successful completion, the following actions are taken with respect to the data structure associated with `msqid`:

1. `msg_qnum` is incremented by 1.
2. `msg_lspid` is set equal to the process ID of the calling process.

3.msg_stime is set equal to the current time.

Data Dictionary:

Sr. No.	NAME OF VARIABLE /FUNCTION/STRUCT	TYPE	USE
1	Msqid	int	FOR SOCKET TUPLE
2	Msgflg	int	FOR SEMAPHORE
3	Key	Key_t	Semaphore id
4	Sbuf	Struct msgbuf	
5	buf_length	size_t	

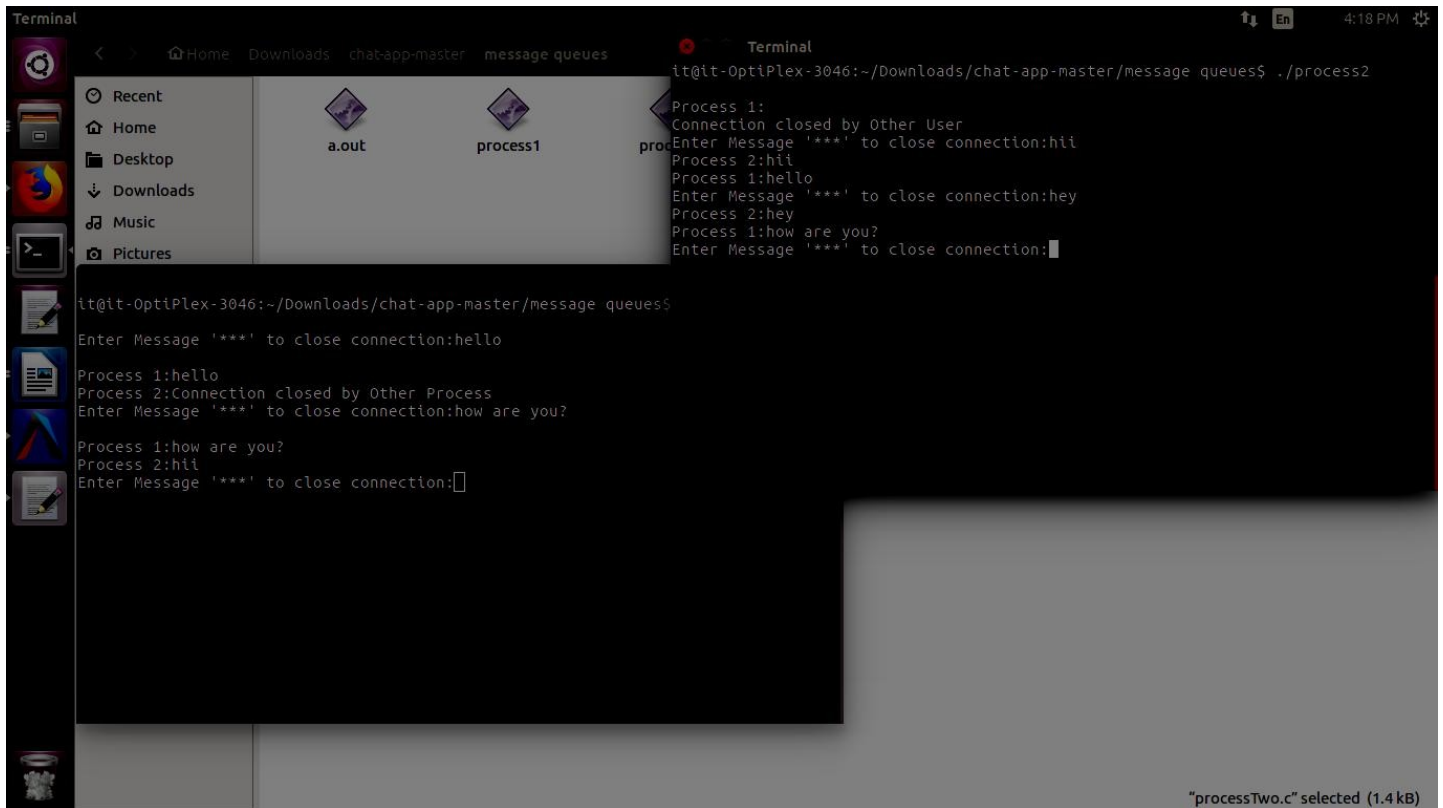
Table: 7.2 Data Dictionary

Data/Variables dictionary for Receiver:

SN.	NAME OF VARIABLE/ FUNCTION/ STRUCT	TYPE	USE
1	msqid	int	FOR SOCKET TUPLE
3	Key	key_t	Semaphore id
4	rbuf	Struct msgbuf	To store the message
5	buf_length	size_t	Length of message to be sent

Table: 7.2 Data Dictionary

Output:



```
Terminal
it@it-OptiPlex-3046:~/Downloads/chat-app-master/message queues$ ./process2
Process 1:
Connection closed by Other User
Enter Message '***' to close connection:hi!
Process 2:hi!
Process 1:hello
Enter Message '***' to close connection:hey
Process 2:hey
Process 1:how are you?
Enter Message '***' to close connection:

it@it-OptiPlex-3046:~/Downloads/chat-app-master/message queues$
Enter Message '***' to close connection:hello
Process 1:hello
Process 2:Connection closed by Other Process
Enter Message '***' to close connection:how are you?
Process 1:how are you?
Process 2:hi!
Enter Message '***' to close connection:

"processTwo.c" selected (1.4 kB)
```

Conclusion:

- Use of message queue functions like msgget, msgsnd, and msgrcv to implement message passing mechanism between server and client studied and implemented it to introduce concept of chatting.

References:

[1] <https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/14/lec.html>

7.3 Write a *server* program and two *client* programs so that the *server* can communicate privately to *each client* individually via a *single* message queue.

IPC: Message Queues: msgget, msgsnd, msgrcv

Message queues are one of the interprocess communication mechanisms available under Linux. Message queues, shared memory and semaphores are normally listed as the three

interprocess communication mechanisms under Linux. Semaphores, though, are really for process

synchronization. In practice, shared memory, aided by semaphores, makes an interprocess communication mechanism. Message queues is the other interprocess communication mechanism.

1. msgget():

In order to create a new message queue, or access an existing queue, the msgget() system call is used.

SYSTEM CALL: msgget();

PROTOTYPE: int msgget (key_t key, int msgflg);

RETURNS: message queue identifier on success

-1 on error: errno = EACCESS (permission denied)

EEXIST (Queue exists, cannot create)

EIDRM (Queue is marked for deletion)

ENOENT (Queue does not exist)

ENOMEM (Not enough memory to create queue)

ENOSPC (Maximum queue limit exceeded)

2. msgsnd():

The msgsnd() function is used to send a message to the queue associated with the message queue identifier specified by msgid.

The argument msgp points to a user-defined buffer that must contain first a field of type long int that will specify the type of the message, and then a data portion that will hold the data bytes of the message. The structure below is an example of what this user-defined buffer should look like:

struct message

{

long int mtype; Message type

int mtext[n]; Message text

}

3. msgrcv():

The msgrcv() function reads a message from the queue associated with the message queue

identifier that msqid specifies and places it in the user-defined structure that msgp points to. When

successfully completed, the following actions are taken with respect to the data structure associated with msqid:

- msg_lrpid is set to the process ID of the calling process.
- msg_rtime is set to the current time.

Program:

Write a server program and two client programs so that the server can communicate privately to each client individually via a single message queue

- (Process 1) Sends the message "Are you hearing me?"
- (Process 2) Receives the message and replies "Loud and Clear"
- (Process 1) Receives the reply and then says "I can hear you too"

Server.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
extern void exit();
extern void perror();
main()
{
    key_t key; /* key to be passed to msgget() */
    int msgflg, /* msgflg to be passed to msgget() */
    msqid; /* return value from msgget() */
    (void) fprintf(stderr,
        "All numeric input is expected to follow C conventions:\n"
        "n"); (void) fprintf(stderr,
        "\t0x... is interpreted as hexadecimal,\n");
    (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
```

```

(void) fprintf(stderr, "\totherwise, decimal.\n");
(void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
(void) fprintf(stderr, "Enter key: ");
(void) scanf("%li", &key);
(void) fprintf(stderr, "\nExpected flags for msgflg
argument are:\n");
(void) fprintf(stderr, "\tIPC_EXCL =\t%#8.8o\n", IPC_EXCL);
(void) fprintf(stderr, "\tIPC_CREAT =\t%#8.8o\n", IPC_CREAT);
(void) fprintf(stderr, "\towner read =\t%#8.8o\n", 0400);
(void) fprintf(stderr, "\towner write =\t%#8.8o\n", 0200);
(void) fprintf(stderr, "\tgroup read =\t%#8.8o\n", 040);

(void) fprintf(stderr, "\tgroup write =\t%#8.8o\n", 020);
(void) fprintf(stderr, "\tother read =\t%#8.8o\n", 04);
(void) fprintf(stderr, "\tother write =\t%#8.8o\n", 02);
(void) fprintf(stderr, "Enter msgflg value: ");
(void) scanf("%i", &msgflg);
(void) fprintf(stderr, "\nmsgget: Calling
msgget(%#lx, %#o)\n",
key, msgflg);if ((msqid = msgget(key, msgflg)) == -1)
{
perror("msgget: msgget failed");
exit(1);
} else {
(void) fprintf(stderr,
"msgget: msgget succeeded: msqid = %d\n", msqid);
exit(0);
}
}

Client1.c
#include <stdio.h>

```

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <time.h>
static void do_msgctl();
extern void exit();
extern void perror();

static char warning_message[] = "If you remove read permission
for \
yourself, this program will fail frequently!";

main()
{
    struct msqid_ds buf; /* queue descriptor buffer for
IPC_STAT and IP_SET commands */
    int cmd, /* command to be given to msgctl() */
    msqid; /* queue ID to be given to msgctl() */
    (void) fprintf(stderr,
    "All numeric input is expected to follow C conventions:\n");
    (void) fprintf(stderr,
    "\t0x... is interpreted as hexadecimal,\n");
    (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
    (void) fprintf(stderr, "\totherwise, decimal.\n");
    /* Get the msqid and cmd arguments for the msgctl() call. */
    (void) fprintf(stderr,
    "Please enter arguments for msgctl() as requested.");
    (void) fprintf(stderr, "\nEnter the msqid: ");
    (void) scanf("%i", &msqid);
    (void) fprintf(stderr, "\tIPC_RMID = %d\n", IPC_RMID);
    (void) fprintf(stderr, "\tIPC_SET = %d\n", IPC_SET);
    (void) fprintf(stderr, "\tIPC_STAT = %d\n", IPC_STAT);

```

```

(void) fprintf(stderr, "\nEnter the value for the command: ");
(void) scanf("%i", &cmd);
switch (cmd) {
case IPC_SET:/* Modify settings in the message queue control structure.
*/
(void) fprintf(stderr, "Before IPC_SET, get
current values:");

/* fall through to IPC_STAT processing */
case IPC_STAT:
/* Get a copy of the current message queue control
* structure and show it to the user. */
do_msgctl(msqid,      IPC_STAT,
&buf); (void) fprintf(stderr, ]
"msg_perm.uid = %d\n",
buf.msg_perm.uid); (void) fprintf(stderr,
"msg_perm.gid = %d\n",
buf.msg_perm.gid); (void) fprintf(stderr,
"msg_perm.cuid = %d\n", buf.msg_perm.cuid);
(void) fprintf(stderr, "msg_perm.cgid = %d\n", buf.msg_perm.cgid);
(void) fprintf(stderr, "msg_perm.mode = %#o, ", buf.msg_perm.mode);
(void) fprintf(stderr, "access permissions = %#o\n", buf.msg_perm.mode & 0777);
(void) fprintf(stderr, "msg_cbytes = %d\n", buf.msg_cbytes);
(void) fprintf(stderr, "msg_qbytes = %d\n", buf.msg_qbytes);
(void) fprintf(stderr, "msg_qnum = %d\n", buf.msg_qnum);
(void) fprintf(stderr, "msg_lspid = %d\n", buf.msg_lspid);
(void) fprintf(stderr, "msg_lrpid = %d\n", buf.msg_lrpid);
(void) fprintf(stderr, "msg_stime = %s",
buf.msg_stime ? ctime(&buf.msg_stime) : "Not Set\n");
(void) fprintf(stderr, "msg_rtime = %s",
buf.msg_rtime ? ctime(&buf.msg_rtime) : "Not Set\n");

```

```

(void) fprintf(stderr, "msg_ctime = %s",
ctime(&buf.msg_ctime)); if (cmd == IPC_STAT)
break;

/* Now continue with IPC_SET. */

(void) fprintf(stderr, "Enter msg_perm.uid: ")
(void) scanf ("%hi", &buf.msg_perm.uid);

(void) fprintf(stderr, "Enter msg_perm.gid: ");
(void) scanf ("%hi", &buf.msg_perm.gid);

(void) fprintf(stderr, "%s\n", warning_message);

(void) fprintf(stderr, "Enter msg_perm.mode: ");
(void) scanf ("%hi", &buf.msg_perm.mode);

(void) fprintf(stderr, "Enter msg_qbytes: ");
(void) scanf ("%hi", &buf.msg_qbytes);

do_msgctl(msqid, IPC_SET, &buf);

break;

case IPC_RMID:

default:/* Remove the message queue or try an unknown command. */

do_msgctl(msqid, cmd, (struct msqid_ds *)NULL); break;

}
exit(0);
}

```

```

{
register int rtn; /* hold area for return value from msgctl()
*/

(void) fprintf(stderr, "\nmsgctl: Calling msgctl(%d,
%d, %s)\n",
msqid, cmd, buf ? "&buf" : "(struct msqid_ds
*)NULL"); rtn = msgctl(msqid, cmd, buf); if (rtn == -
1) {
perror("msgctl: msgctl failed");
exit(1);
} else {
(void) fprintf(stderr, "msgctl: msgctl returned %d\n",
rtn);
}
} Client2.c

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <string.h>
#define MSGSZ
128
/*
* Declare the message structure.
*/

typedef struct msgbuf
{ long mtype;
char mtext[MSGSZ];

```

```

} message_buf;

main()
{
int msqid;
int msgflg = IPC_CREAT |
0666; key_t key;
message_buf sbuf;
size_t
buf_length; /*
* Get the message queue id for the
* "name" 1234, which was created by
* the server.
*/
key = 1234;
(void) fprintf(stderr, "\nmsgget: Calling
msgget(%#lx, \ %#o)\n",
key, msgflg);
if ((msqid = msgget(key, msgflg )) < 0)
{ perror("msgget");
exit(1);
}
else
(void) fprintf(stderr, "msgget: msgget succeeded: msqid = %d\n",
msqid); /* We'll send message type 1*/
sbuf.mtype = 1; (void) fprintf(stderr, "msgget: msgget succeeded: msqid = %d\n",
msqid); (void) strcpy(sbuf.mtext, "Did you get this?");
(void) fprintf(stderr, "msgget: msgget succeeded: msqid = %d\n",
msqid); buf_length = strlen(sbuf.mtext) + 1 ;

```

```
if (msgsnd(msqid, &sbuf, buf_length, IPC_NOWAIT) < 0) {
    printf ("%d, %d, %s, %d\n", msqid, sbuf.mtype, sbuf.mtext, buf_length);
    perror("msgsnd");
    exit(1);
}
else
    printf("Message: \"%s\" Sent\n", sbuf.mtext);
    exit(0);
}
```

```
Terminal
```

```
user1@bagpipe:~/src$ # Server
user1@bagpipe:~/src$ gcc server.c -o server -lrt
user1@bagpipe:~/src$ ./server
Server: Hello, World!
Server: message received.
Server: response sent to client.
Server: message received.
Server: response sent to client.
Server: message received.
Server: response sent to client.
Server: message received.
Server: response sent to client.
Server: message received.
Server: response sent to client.
Server
[ ] user1@bagpipe:~/src$ #client
user1@bagpipe:~/src$ gcc client.c -o client -lrt
user1@bagpipe:~/src$ ./client
Ask for a token (Press <ENTER>):
Client: Token received from server: 2

Ask for a token (Press ):
Client: Token received from server: 3

Ask for a token (Press ):
Client: Token received from server: 5

Ask for a token (Press ): [ ]
```


Chapter 8

IPC: Shared Memory

8.1 Write a program to perform IPC using shared memory to illustrate the passing of a simple piece of memory (a string) between the processes if running simultaneously.

Objectives:

1. To learn about IPC through message queue.
2. Use of system call and IPC mechanism to write effective application programs.

Theory:

Shared Memory is an efficient means of passing data between programs. One program will create a memory portion which other processes (if permitted) can access.

Communication between processes using shared memory requires processes to share some variable and it completely depends on how programmer will implement it. One way of communication using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously and they share some resources or use some information from other process, process1 generate information about certain computations or resources being used and keeps it as a record in shared memory. When process2 need to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from other process as well as for delivering any specific information to other process.

The server maps a shared memory in its address space and also gets access to a synchronization mechanism. The server obtains exclusive access to the memory using the synchronization mechanism and copies the file to memory. The client maps the shared memory in its address space. Waits until the server releases the exclusive access and uses the data.

To use shared memory, we have to perform 2 basic steps:

- Request to the operating system a memory segment that can be shared between processes. The user can create/destroy/open this memory using a shared memory object: *An object that represents memory that can be mapped concurrently into the address space of more than one process..*
- Associate a part of that memory or the whole memory with the address space of the calling process. The operating system looks for a big enough memory address range in the calling process' address space and marks that address range as an special range.

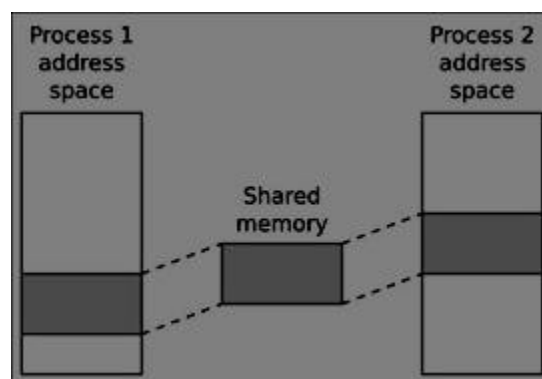
A process creates a shared memory segment using `shmget()`

The original owner of a shared memory segment can assign ownership to another user with `shmctl()`.

A shared segment can be attached to a process address space using `shmat()`.

It can be detached using `shmdt()`.

Once attached, the process can read or write to the segment, as allowed by the permission requested in the attach operation. A shared segment can be attached multiple times by the same process. A shared memory segment is described by a control structure with a unique ID that points to an area of physical memory.



Data Dictionary:

Number	Variable/function	Data Type	Use
1	shmid	int	Store value of identifier of System V shared memory
2	key	key_t	Used to pass the key to shmget

Server:

Number	Variable/function	Data Type	Use
1	shmid	int	Store value of identifier of System V shared memory
2	key	key_t	Used to pass the key to shmget

3	c	char	Used to check character
---	---	------	-------------------------

Table: 8.1 Data Dictionary

Program:

```
//client side program
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#define SHMSZ 30
void main()
{
    int shmid;
    key_t key;
    char *shm, *s;
    key = 5858;
    if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
        perror("shmget");
        exit(1);
    }
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }
    for (s = shm; *s != NULL; s++)
        putchar(*s);
    putchar('\n');
    shm = '#';
    exit(0);
}
```

```

}
//server side program
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define SHMSZ 30
void main()
{
char c;
int shmid;
key_t key;
char *shm, *s;
key = 5858;
if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
perror("shmget");
exit(1);
}
if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
perror("shmat");

exit(1);
}
s = shm;
for (c = 'a'; c <= 'z'; c++)
*s++ = c;
*s = NULL;
while (*shm != '#')
sleep(1);

```

```
exit(0);  
}
```

Output:

```
it@it-OptiPlex-3046:~/uos25$ gcc assign8a_server.c -o 8as  
assign8a_server.c: In function 'main':  
assign8a_server.c:33:8: warning: assignment makes integer from pointer  
without a cast [-Wint-conversion]  
    *s = NULL;  
    ^  
it@it-OptiPlex-3046:~/uos25$ ./8as  
it@it-OptiPlex-3046:~/uos25$
```

```
it@it-OptiPlex-3046:~/uos25$ gcc assign8a_client.c -o 8ac  
assign8a_client.c: In function 'main':  
assign8a_client.c:27:22: warning: comparison between pointer and  
er  
    for (s = shm; *s != NULL; s++)  
                ^  
it@it-OptiPlex-3046:~/uos25$ ./8ac  
abcdefghijklmnopqrstuvwxyz  
it@it-OptiPlex-3046:~/uos25$
```

Conclusion:

- Memory shared between client and server using IPC-SHM functions. The data placed can be accessed by both.

References:

[1] <https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/14/lec.html>

8.2 Write 2 programs that will communicate via shared memory and semaphores. Data will be exchanged via memory and semaphores will be used to synchronize and notify each process when operations such as memory loaded and memory read have been performed.

Objectives:

1. To learn about IPC through message queue.
2. Use of system call and IPC mechanism to write effective application programs.

Theory:

Shared Memory is an efficient means of passing data between programs. One program will create a memory portion which other processes (if permitted) can access.

Communication between processes using shared memory requires processes to share some variable and it completely depends on how programmer will implement it. One way of communication using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously and they share some resources or use some information from other process, process1 generate information about certain computations or resources being used and keeps it as a record in shared memory. When process2 need to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from other process as well as for delivering any specific information to other process.

The server maps a shared memory in its address space and also gets access to a synchronization mechanism. The server obtains exclusive access to the memory using the synchronization mechanism and copies the file to memory. The client maps the shared memory in its address space. Waits until the server releases the exclusive access and uses the data.

To use shared memory, we have to perform 2 basic steps:

- Request to the operating system a memory segment that can be shared between processes. The user can create/destroy/open this memory using a shared memory object: *An object that represents memory that can be mapped concurrently into the address space of more than one process..*
- Associate a part of that memory or the whole memory with the address space of the calling process. The operating system looks for a big enough memory address range in the calling process' address space and marks that address range as an special range.

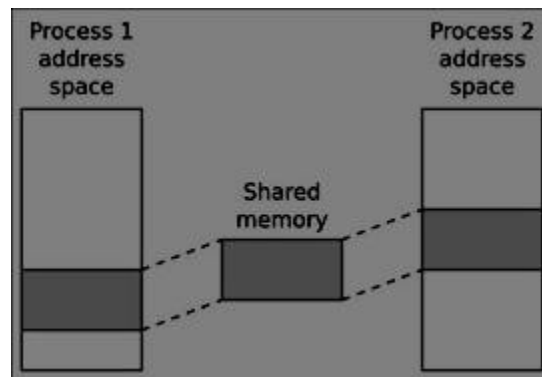
A process creates a shared memory segment using shmget()

The original owner of a shared memory segment can assign ownership to another user with `shmctl()`.

A shared segment can be attached to a process address space using `shmat()`.

It can be detached using `shmdt()`.

Once attached, the process can read or write to the segment, as allowed by the permission requested in the attach operation. A shared segment can be attached multiple times by the same process. A shared memory segment is described by a control structure with a unique ID that points to an area of physical memory.



Data Dictionary:

Number	Variable/function	Data Type	Use
1	operations	Struct sembuf	Used to Store Operations
2	shm_address	void	Shared memory address
3	semid	int	Used to store semaphore id
4	shmid	int	Used to store Shared memory id

5	semkey	key_t	Store key of semaphores
6	shmkey	key_t	Store key of SHM

Table: 8.2 Data Dictionary

Program:

```

//client side program
#include <stdio.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#define SEMKEYPATH "/dev/null"
#define SEMKEYID 1
#define SHMKEYPATH "/dev/null"
#define SHMKEYID 1
#define NUMSEMS 2
#define SIZEOFshmSEG 50
int main(int argc, char *argv[])
{
    struct sembuf operations[2];
    void      *shm_address;
    int semid, shmid, rc;
    key_t semkey, shmkey;
    semkey = ftok(SEMKEYPATH,SEMKEYID);
    if ( semkey == (key_t)-1 )
    {
        printf("main: ftok() for sem failed\n");
        return -1;
    }
    semid = semget( semkey, NUMSEMS, 0666);
    if ( semid == -1 )
    {
        printf("main: semget() failed\n");
        return -1;
    }
}

```



```

shm_address = shmat(shmid, NULL, 0);
if ( shm_address==NULL )
{
printf("main: shmat() failed\n");
return -1;
}

operations[0].sem_num = 0;
operations[0].sem_op = 0;
operations[0].sem_flg = 0;
operations[1].sem_num = 0;
operations[1].sem_op = 1;
operations[1].sem_flg = 0;
rc = semop( semid, operations, 2 );
if (rc == -1)
{
printf("main: semop() failed\n");
return -1;
}

strcpy((char *) shm_address, "Hello from
Client"); operations[0].sem_num = 0;
operations[0].sem_op = -1; operations[0].sem_flg
= 0; operations[1].sem_num = 1;
operations[1].sem_op = 1; operations[1].sem_flg
= 0;

rc = semop( semid, operations, 2 );
if (rc == -1)
{
printf("main: semop() failed\n");
return -1;
}

```

```

    }
    rc = shmdt(shm_address);
    if (rc == -1)
    {
        printf("main: shmdt() failed\n");
        return -1;
    }
    if(operations[0].sem_op == -1)
        printf("\nread performed by server");
    return 0;
}

//server side program
#include <stdio.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>

#define SEMKEYPATH "/dev/null"
#define SEMKEYID 1
#define SHMKEYPATH "/dev/null"
#define SHMKEYID 1
#define NUMSEMS 2
#define SIZEOFshmSEG 50
#define NUMMSG 2

int main(int argc, char *argv[])
{
    int rc, semid, shmid, i;
    key_t semkey, shmkey;
    void *shm_address;
    struct sembuf operations[2];

```

```

struct shmid_ds shmid_struct;
short sarray[NUMSEMS];
semkey = ftok(SEMKEYPATH,SEMKEYID);
if ( semkey == (key_t)-1 )
{
printf("main: ftok() for sem failed\n");
return -1;
}
shmkey = ftok(SHMKEYPATH,SHMKEYID);
if ( shmkey == (key_t)-1 )
{
printf("main: ftok() for shm failed\n");
return -1;
}
semid = semget( semkey, NUMSEMS, 0666 | IPC_CREAT | IPC_EXCL );
if ( semid == -1 )
{
printf("main: semget() failed\n");
return -1;
}
sarray[0] = 0;
sarray[1] = 0;
rc = semctl( semid, 1, SETALL, sarray);
if(rc == -1)
{
printf("main: semctl() initialization failed\n");
return -1;
}
shmid = shmget(shmkey, SIZEOFSHMSEG, 0666 | IPC_CREAT | IPC_EXCL);
if (shmid == -1)

```

```

{
printf("main: shmget() failed\n");
return -1;
}
shm_address = shmat(shmid, NULL, 0);
if ( shm_address==NULL )
{
printf("main: shmat() failed\n");
return -1;
}
printf("Ready for client jobs\n");
for (i=0; i < NUMMSG; i++)
{
operations[0].sem_num = 1;
operations[0].sem_op = -1;

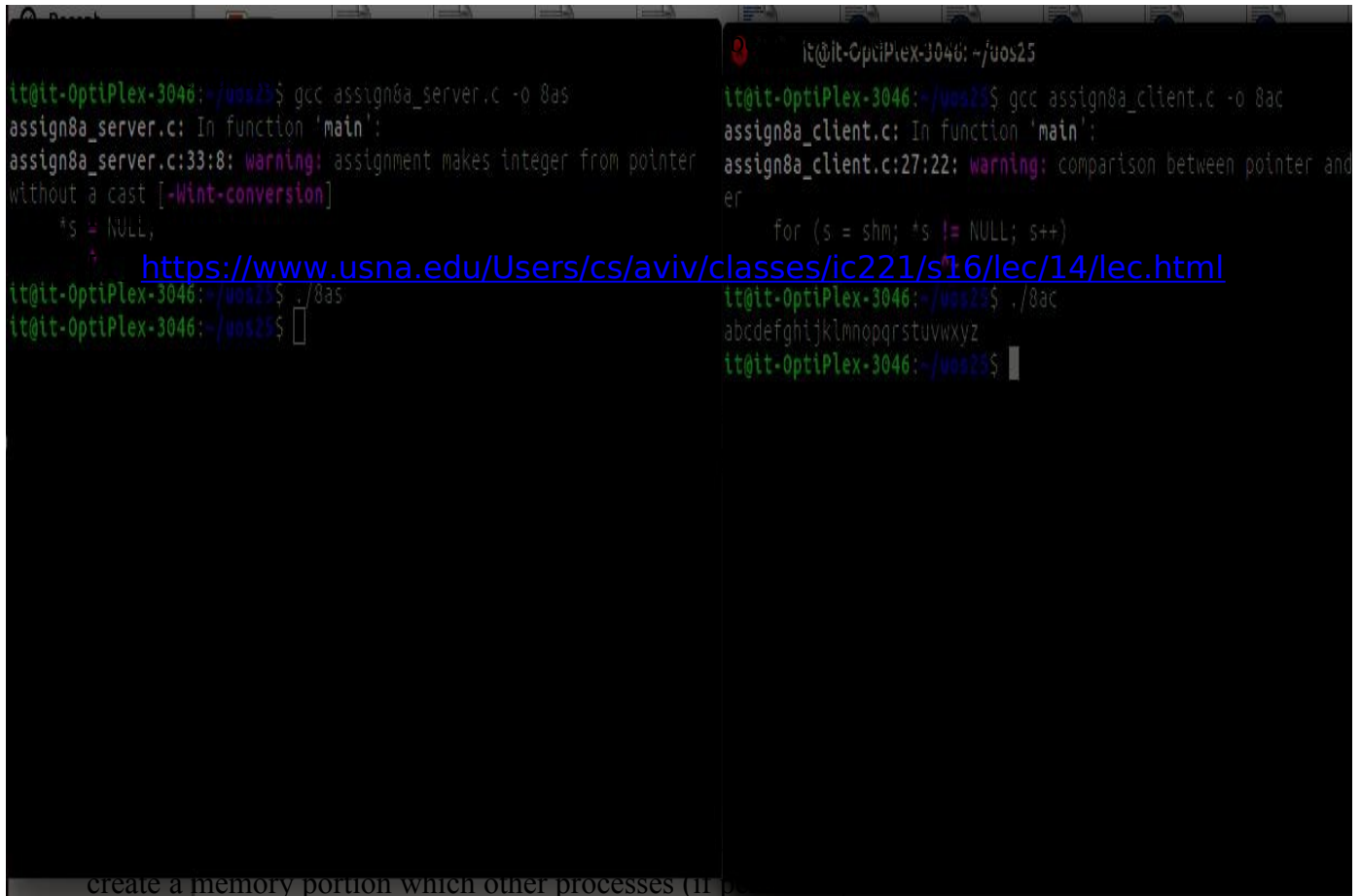
operations[0].sem_flg = 0;
operations[1].sem_num = 0;
operations[1].sem_op = 1;
operations[1].sem_flg = IPC_NOWAIT;
rc = semop( semid, operations, 2 );
if (rc == -1)
{
printf("main: semop() failed\n");
return -1;
}
printf("Server Received : \"\n", (char *) shm_address);
operations[0].sem_num = 0;
operations[0].sem_op = -1;
operations[0].sem_flg = IPC_NOWAIT;
rc = semop( semid, operations, 1 );

```

```
if (rc == -1)
{
printf("main: semop() failed\n");
return -1;
}
rc = semctl( semid, 1, IPC_RMID );
if (rc==-1)
{
printf("main: semctl() remove id failed\n");
return -1;
}
rc = shmdt(shm_address);
if (rc==-1)
{
printf("main: shmdt() failed\n");
return -1;
}
rc = shmctl(shmid, IPC_RMID, &shmid_struct);
if (rc==-1)
{
printf("main: shmctl() failed\n");
return -1;
}
return 0;
}
```

Output:

Conclusion:



```
it@it-OptiPlex-3046:~/uos25$ gcc assign8a_server.c -o 8as
assign8a_server.c: In function 'main':
assign8a_server.c:33:8: warning: assignment makes integer from pointer
without a cast [-Wint-conversion]
    *s = NULL,
    ^
https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/14/lec.html
it@it-OptiPlex-3046:~/uos25$ ./8as
it@it-OptiPlex-3046:~/uos25$

it@it-OptiPlex-3046:~/uos25$ gcc assign8a_client.c -o 8ac
assign8a_client.c: In function 'main':
assign8a_client.c:27:22: warning: comparison between pointer and
er
    for (s = shm; *s != NULL; s++)
                   ^
it@it-OptiPlex-3046:~/uos25$ ./8ac
abcdefghijklmnopqrstuvwxyz
it@it-OptiPlex-3046:~/uos25$
```

Communication between processes using shared memory requires processes to share some variable and it completely depends on how programmer will implement it. One way of communication using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously and they share some resources or use some information from other process, process1 generate information about certain computations or resources being used and keeps it as a record in shared memory. When process2 need to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from other process as well as for delivering any specific information to other process.

The server maps a shared memory in its address space and also gets access to a synchronization mechanism. The server obtains exclusive access to the memory using the synchronization mechanism and copies the file to memory. The client maps the shared memory in its address space. Waits until the server releases the exclusive access and uses the data.

To use shared memory, we have to perform 2 basic steps:

- Request to the operating system a memory segment that can be shared between processes. The user can create/destroy/open this memory using a shared memory object: *An object that represents memory that can be mapped concurrently into the address space of more than one process..*

- Associate a part of that memory or the whole memory with the address space of the calling process. The operating system looks for a big enough memory address range in the calling process' address space and marks that address range as a special range.

A process creates a shared memory segment using `shmget()`

The original owner of a shared memory segment can assign ownership to another user with `shmctl()`.

A shared segment can be attached to a process address space using `shmat()`.

It can be detached using `shmdt()`.

Once attached, the process can read or write to the segment, as allowed by the permission requested in the attach operation. A shared segment can be attached multiple times by the same process. A shared memory segment is described by a control structure with a unique ID that points to an area of physical memory.

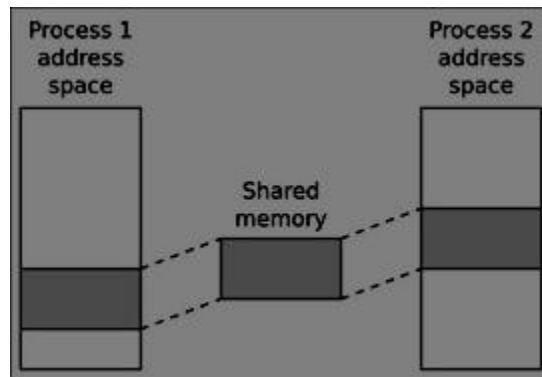


Fig: 8.3 Shared Memory Architecture

Data Dictionary:

Number	Variable/function	Data Type	Use
1	shm	char	Shared memory address
2	shmid	int	Used to store Shared memory id
3	key	key_t	Store key of SHM

Server:

Number	Variable/function	Data Type	Use
1	shm	char	Shared memory address

2	shmid	int	Used to store Shared memory id
3	key	key_t	Store key of SHM

Table: 8.3 Data Dictionary

Program:

```
//client side program
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#define SHMSZ 30
void main()
{
int shmid;
key_t key;
char *shm, *s;
key = 5858;
if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
perror("shmget");
exit(1);
}
if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
perror("shmat");
exit(1);
}
```

```

for (s = shm; *s != NULL; s++)
    putchar(*s);
    putchar('\n');
    *shm = '#';
    exit(0);
}

//server side program
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define SHMSZ 30
void main()
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;
    key = 5858;
    if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        exit(1);
    }
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }
}

```

```

s = shm;

FILE *fptr;

fptr=fopen("test.txt","r");

while(c!=EOF)
{
c=fgetc(fptr);
*s++=c;
}
*s=NULL;

/* for (c = 'a'; c <= 'z'; c++)

*s++ = c;

*s = NULL; */

while (*shm != '#')

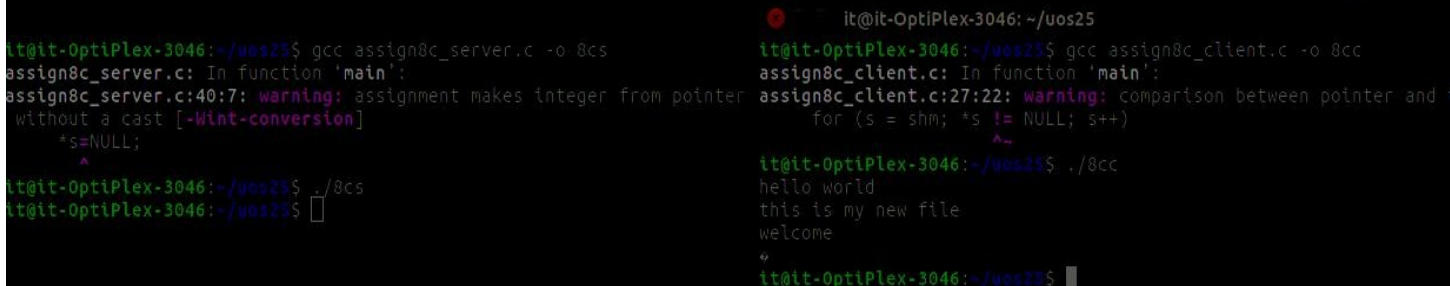
sleep(1);

exit(0);

}

```

Output:



```

it@it-OptiPlex-3046:~/uos25$ gcc assign8c_server.c -o 8cs
assign8c_server.c: In function 'main':
assign8c_server.c:40:7: warning: assignment makes integer from pointer
without a cast [-Wint-conversion]
    *s=NULL;
    ^
it@it-OptiPlex-3046:~/uos25$ ./8cs
it@it-OptiPlex-3046:~/uos25$ █

it@it-OptiPlex-3046:~/uos25$ gcc assign8c_client.c -o 8cc
assign8c_client.c: In function 'main':
assign8c_client.c:27:22: warning: comparison between pointer and
for (s = shm; *s != NULL; s++)
                    ^
it@it-OptiPlex-3046:~/uos25$ ./8cc
hello world
this is my new file
welcome
^
it@it-OptiPlex-3046:~/uos25$ █

```

Conclusion:

- Sharing of files between client and server by using shared memory IPC was implemented.

References:

[1] <https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/14/lec.html>

Chapter 9

IPC: Sockets

9.1 Write two programs (server/client) and establish a socket to communicate.

Objectives:

1. To learn about fundamentals of IPC through C socket programming.
2. Learn and understand the OS interaction with socket programming.
3. Use of system call and IPC mechanism to write effective application programs.
4. To know the port numbering and process relation.
5. To know the iterative and concurrent server concept.

Theory:

JAVA SOCKET PROGRAMMING

A very basic one-way Client and Server setup where a Client connects, sends messages to server and the server shows them using socket connection. Java API networking package (java.net) takes care of all of that, making network programming very easy for programmers.

CLIENT SIDE PROGRAMMING:

Establish a Socket Connection

To connect to other machine we need a socket connection. A socket connection means the two machines have information about each other's network location (IP Address) and TCP port. The java.net.Socket class represents a Socket. To open a socket:

```
Socket socket = new Socket("127.0.0.1", 5000)
```

- First argument – IP address of Server. (127.0.0.1 is the IP address of localhost, where code will run on single stand-alone machine).
- Second argument – TCP Port. (Just a number representing which application to run on a server. For example, HTTP runs on port 80. Port number can be from 0 to 65535)

To communicate over a socket connection, streams are used to both input and output the data.

Closing the connection

The socket connection is closed explicitly once the message to server is sent.

SERVER SIDE PROGRAMMING:

Establish a Socket Connection

To write a server application two sockets are needed.

- A ServerSocket which waits for the client requests (when a client makes a new Socket())
- A plain old Socket socket to use for communication with the client.

getOutputStream() method is used to send the output through the socket.

Close the Connection

After finishing, it is important to close the connection by closing the socket as well as input/output streams.

Data Dictionary:

Sr Number	Variable/Function	Datatype	Use
1	Ss	ServerSocket	Create a socket for server side communication.
2	S	Socket	Socket is created.
3	Dos	DataOutputStream	Output Stream.
4	Dis	DataInputStream	Input Stream.
5	Str	String	String to display message from clients.

Table: 9.1 Data Dictionary

Program:

SERVER:

```
import java.net.*;

import java.io.*;

class Server1

{

public static void main(String []args)throws Exception

{

ServerSocket ss=new ServerSocket(5050);//5050 is port no.
```

```

System.out.println("Server is Waiting.....");

Socket s=ss.accept();//waiting for client

DataOutputStream dos=new DataOutputStream(s.getOutputStream());

DataInputStream dis=new DataInputStream(s.getInputStream());

String str="Welcomes you are connected \n";

dos.writeUTF(str); //sends msg to client

str=dis.readUTF(); //reads msg send by client

System.out.println("From client"+" "+str);

ss.close();

s.close();

dos.close();

dis.close();

}

}

```

CLIENT:

```

import java.net.*;

import java.io.*;

class Client1

{

public static void main(String []args)throws Exception

{

Socket s=new Socket("localhost",5050);

DataOutputStream dos=new DataOutputStream(s.getOutputStream());

```

```

DataInputStream dis=new DataInputStream(s.getInputStream());

String str=dis.readUTF(); //receives msg send by server

System.out.println("From server"+" "+str);

str="Thank u for connecting";

dos.writeUTF(str); //writes to server

s.close();

dos.close();

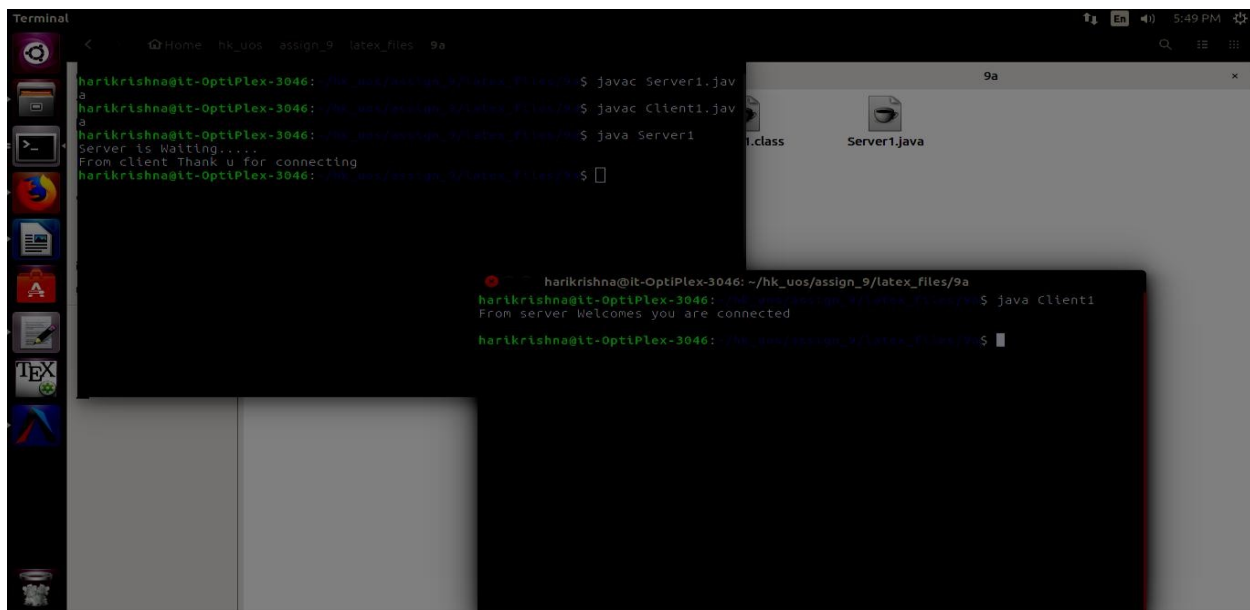
dis.close();

}

}

```

OUTPUT:



```

Terminal
harikrishna@lt-OptiPlex-3046: ~/hk_uos/assign_9/latex_files/9a
harikrishna@lt-OptiPlex-3046: ~/hk_uos/assign_9/latex_files/9a $ javac Server1.java
harikrishna@lt-OptiPlex-3046: ~/hk_uos/assign_9/latex_files/9a $ javac Client1.java
harikrishna@lt-OptiPlex-3046: ~/hk_uos/assign_9/latex_files/9a $ java Server1
Server is Waiting....
From client Thank u for connecting
harikrishna@lt-OptiPlex-3046: ~/hk_uos/assign_9/latex_files/9a $

```

```

harikrishna@lt-OptiPlex-3046: ~/hk_uos/assign_9/latex_files/9a
harikrishna@lt-OptiPlex-3046: ~/hk_uos/assign_9/latex_files/9a $ java Client1
From server welcomes you are connected
harikrishna@lt-OptiPlex-3046: ~/hk_uos/assign_9/latex_files/9a $

```

Conclusion:

- Java can be used to establish communication between two programs on remote or same machine using sockets and system calls.

References:

[1] <http://www.prasannatech.net/2008/07/socket-programming-tutorial.html>

9.2 Write programs (server and client) to implement concurrent/iterative server to connect multiple clients requests handled through concurrent/iterative logic using UDP/TCP socket connection.

Objectives:

1. To learn about fundamentals of IPC through C socket programming.
2. Learn and understand the OS interaction with socket programming.
3. Use of system call and IPC mechanism to write effective application programs.
4. To know the port numbering and process relation.
5. To know the iterative and concurrent server concept.

Theory:

JAVA SOCKET PROGRAMMING

A very basic one-way Client and Server setup where a Client connects, sends messages to server and the server shows them using socket connection. Java API networking package (java.net) takes care of all of that, making network programming very easy for programmers.

CLIENT SIDE PROGRAMMING:

Establish a Socket Connection

To connect to other machine we need a socket connection. A socket connection means the two machines have information about each other's network location (IP Address) and TCP port. The java.net.Socket class represents a Socket. To open a socket:

```
Socket socket = new Socket("127.0.0.1", 5000)
```

3. First argument – IP address of Server. (127.0.0.1 is the IP address of localhost, where code will run on single stand-alone machine).

Second argument – TCP Port. (Just a number representing which application to run on a server. For example, HTTP runs on port 80. Port number can be from 0 to 65535)

To communicate over a socket connection, streams are used to both input and output the data.

Closing the connection

The socket connection is closed explicitly once the message to server is sent.

SERVER SIDE PROGRAMMING:

Establish a Socket Connection

To write a server application two sockets are needed.

- A ServerSocket which waits for the client requests (when a client makes a new Socket())
- A plain old Socket socket to use for communication with the client.

getOutputStream() method is used to send the output through the socket.

Close the Connection

After finishing, it is important to close the connection by closing the socket as well as input/output streams.

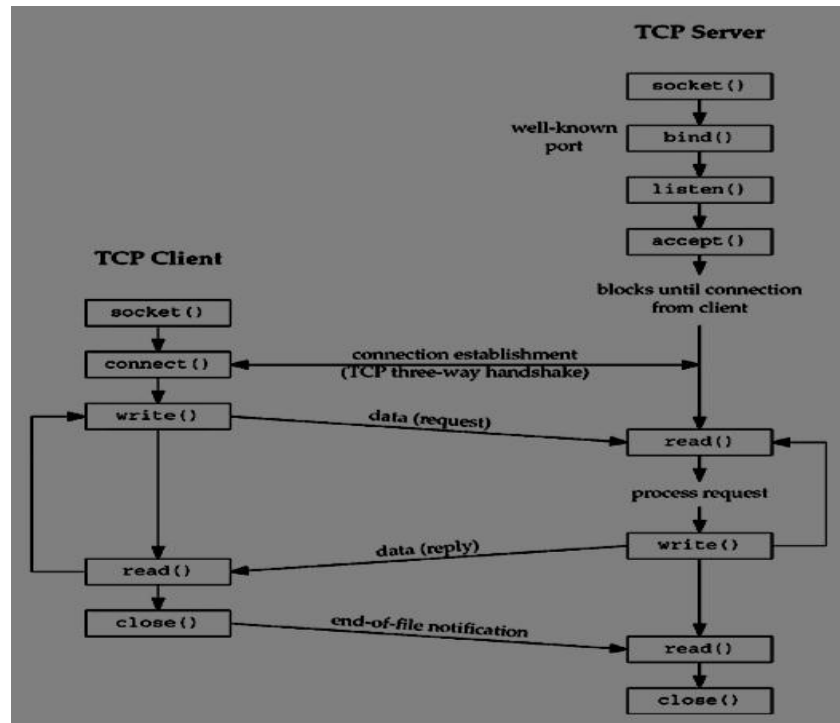


Fig:9.1 TCP Architecture

Data Dictionary:

Sr Number	Variable/Function	Datatype	Use
1	ss	ServerSocket	Create a socket for server side communication.
2	s	Socket	Socket is created.
3	dos	DataOutputStream	Output Stream.
4	dis	DataInputStream	Input Stream.
5	cnm	String	String to display message from clients.
6	br	BufferedReader	Input data.

Table: 9.2 Data Dictionary

Program:

SERVER:

```
import java.net.*;

import java.io.*;

class Server5

{

public static void main(String []args)throws Exception

{

ServerSocket ss=new ServerSocket(5060);

while(true)

{

Socket s=ss.accept();

DataOutputStream dos=new DataOutputStream(s.getOutputStream());

DataInputStream dis=new DataInputStream(s.getInputStream());

dos.writeUTF("Welcomes u");

String cnm=dis.readUTF();

ThrdComm a=new ThrdComm(dos,dis,cnm);

}}

class ThrdComm extends Thread

{

DataOutputStream dos;

DataInputStream dis;

BufferedReader br;

String str,cnm;
```

ThrdComm(DataOutputStream dos,DataInputStream dis,String cnm)throws Exception

```
{
super(cnm);
this.dos=dos;
this.dis=dis;
this.cnm=cnm;
br=new BufferedReader(new InputStreamReader(System.in));
start();
}

public void run()
{
while(true)
{
try
{
talk();
}
catch(Exception e){}
}
}

synchronized void talk()throws Exception
{
System.out.println("Message To"+" "+cnm+":");
str=br.readLine();
dos.writeUTF(str);//sends msg to Client
```

```
str=dis.readUTF();//reads msg from client  
System.out.println("From Client:"+" "+str); } }
```

CLIENT:

```
import java.net.*;  
import java.io.*;  
class Client5 extends Thread  
{  
public static void main(String []args)throws Exception  
{  
if(args.length!=1)  
return;  
Client5 a=new Client5(args[0]);  
  
}  
Client5(String s1)throws Exception  
{  
super(s1);//naming to thread  
s=new Socket("localhost",5060);  
dos=new DataOutputStream(s.getOutputStream());  
dis=new DataInputStream(s.getInputStream());  
br=new BufferedReader(new  
InputStreamReader(System.in)); cnm=s1;//set argument as  
client name str="";  
str=dis.readUTF();//reads msg send by server  
System.out.println("From Server:"+" "+str);  
dos.writeUTF(cnm);//client sends its name to server  
start();}  
public void run()  
{
```

```

while(true)
{
try
{
talk();
}
catch(Exception e){}
}
}
synchronized void talk()throws Exception
{
str=dis.readUTF();//msg from server
System.out.println("From Server:"+" "+str);
System.out.println("Message to Server:");
str=br.readLine();
dos.writeUTF(str);}

Socket s;

String str,cnm;

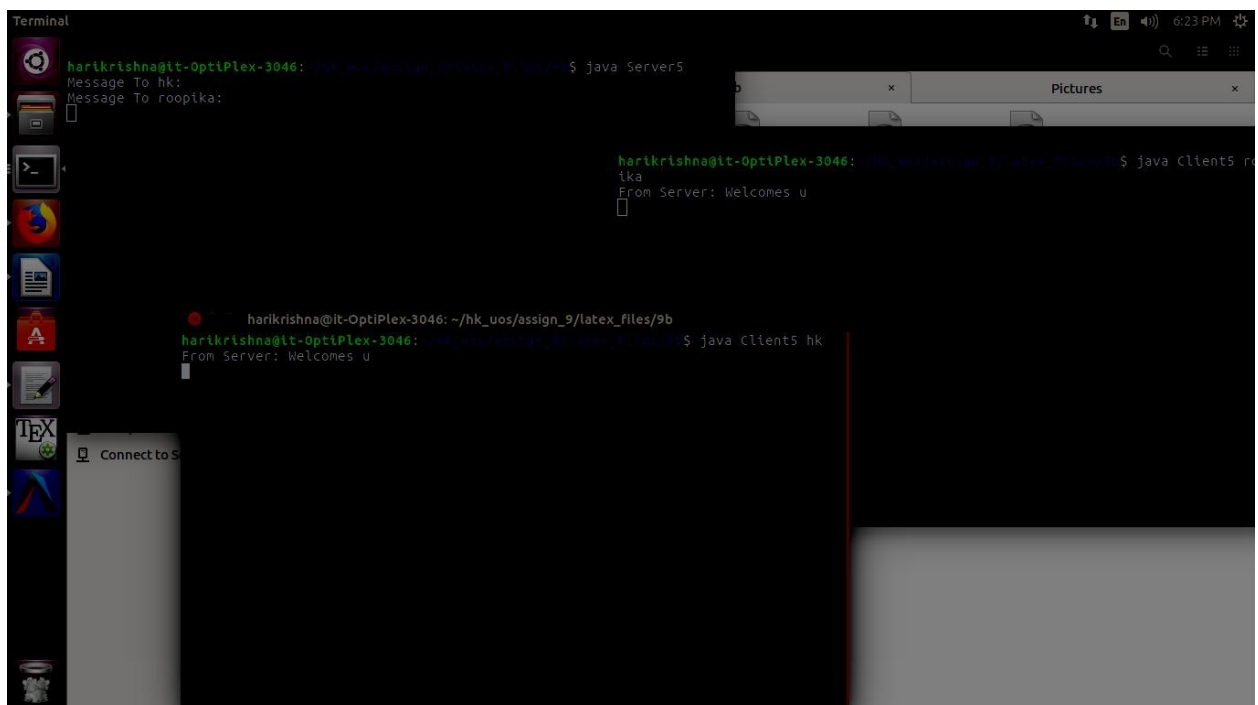
DataOutputStream dos;

DataInputStream dis;

BufferedReader br;}

```

Output:



```

Terminal
harikrishna@it-OptiPlex-3046: ~/hk_uos/assign_9/latex_files/9b$ java Server5
Message To hk:
Message To roopika:
From Server: Welcomes u
harikrishna@it-OptiPlex-3046: ~/hk_uos/assign_9/latex_files/9b$ java Client5 hk
From Server: Welcomes u

```

Conclusion:

- Various communication protocols like TCP/UDP can be implemented using socket programming in Java to serve requests from multiple clients.

References:

[1]<http://www.prasannatech.net/2008/07/socket-programming-tutorial.html>

9.3 Write two programs (server and client) to show how you can establish a TCP socket connection using the above functions.**Objectives:**

1. To learn about fundamentals of IPC through C socket programming.
2. Learn and understand the OS interaction with socket programming.
3. Use of system call and IPC mechanism to write effective application programs.
4. To know the port numbering and process relation.
5. To know the iterative and concurrent server concept.

Theory:**JAVA SOCKET PROGRAMMING**

A very basic one-way Client and Server setup where a Client connects, sends messages to server and the server shows them using socket connection. Java API networking package (java.net) takes care of all of that, making network programming very easy for programmers.

CLIENT SIDE PROGRAMMING:**Establish a Socket Connection**

To connect to other machine we need a socket connection. A socket connection means the two machines have information about each other's network location (IP Address) and TCP port. The java.net.Socket class represents a Socket. To open a socket:

```
Socket socket = new Socket("127.0.0.1", 5000)
```

- First argument – IP address of Server. (127.0.0.1 is the IP address of localhost, where code will run on single stand-alone machine).
- Second argument – TCP Port. (Just a number representing which application to run on a server. For example, HTTP runs on port 80. Port number can be from 0 to 65535)

To communicate over a socket connection, streams are used to both input and output the data.

Closing the connection

The socket connection is closed explicitly once the message to server is sent.

SERVER SIDE PROGRAMMING:

Establish a Socket Connection

To write a server application two sockets are needed.

- A ServerSocket which waits for the client requests (when a client makes a new Socket())
- A plain old Socket socket to use for communication with the client.

getOutputStream() method is used to send the output through the socket.

Close the Connection

After finishing, it is important to close the connection by closing the socket as well as input/output streams.

Program:

SERVER:

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <errno.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#define SERV_TCP_PORT 8000 /* server's port number
*/ #define MAX_SIZE 80

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

int sockfd, newsockfd, clien;
```

```

struct sockaddr_in cli_addr, serv_addr;

int port;

char string[MAX_SIZE];

int len;

/* command line: server [port_number] */

if(argc == 2)

sscanf(argv[1], "%d", &port); /* read the port number if provided

*/ else

port = SERV_TCP_PORT;

/* open a TCP socket (an Internet stream socket) */

if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {

perror("can't open stream socket"); exit(1);

}

/* bind the local address, so that the cliend can send to server */

bzero((char *) &serv_addr, sizeof(serv_addr));

serv_addr.sin_family = AF_INET;

serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);

serv_addr.sin_port = htons(port);

if(bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {

perror("can't bind local address");

exit(1);

}

/* listen to the socket */

listen(sockfd, 5);

```



```

for(;;) {

/* wait for a connection from a client; this is an iterative server

*/ clilen = sizeof(cli_addr);

newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
&clilen); if(newsockfd < 0) {

perror("can't bind local address");

}

/* read a message from the client */

len = read(newsockfd, string, MAX_SIZE);

/* make sure it's a proper string */

string[len] = 0;

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

close(newsockfd);

}

}

```

CLIENT:

```

#include <string.h>

#include <stdio.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <netinet/in.h>

#include <netinet/tcp.h>

#include <netdb.h>

```

```

#include <stdlib.h>

#include <unistd.h>

#include <errno.h>

#define SERV_TCP_PORT 8000 /* server's port

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

    int sockfd;

    struct sockaddr_in serv_addr;

    char *serv_host = "localhost";

    struct hostent *host_ptr;

    int port;

    int buff_size = 0;

    /* command line: client [host [port]]*/

    if(argc >= 2)

        serv_host = argv[1]; /* read the host if provided

    */ if(argc == 3)

        sscanf(argv[2], "%d", &port); /* read the port if provided

    */ else

        port = SERV_TCP_PORT;

    /* get the address of the host */

    if((host_ptr = gethostbyname(serv_host)) == NULL)

        { perror("gethostbyname error"); exit(1);

    }

    if(host_ptr->h_addrtype != AF_INET) {

```

```

perror("unknown address type");

exit(1);

}

bzero((char *) &serv_addr, sizeof(serv_addr));

serv_addr.sin_family = AF_INET;

serv_addr.sin_addr.s_addr =

((struct in_addr *)host_ptr->h_addr_list[0])->s_addr;

serv_addr.sin_port = htons(port); /* open a TCP

socket */

if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {

perror("can't open stream socket"); exit(1);


/* connect to the server */

if(connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {

perror("can't connect to server");

exit(1);

}

/* write a message to the server */

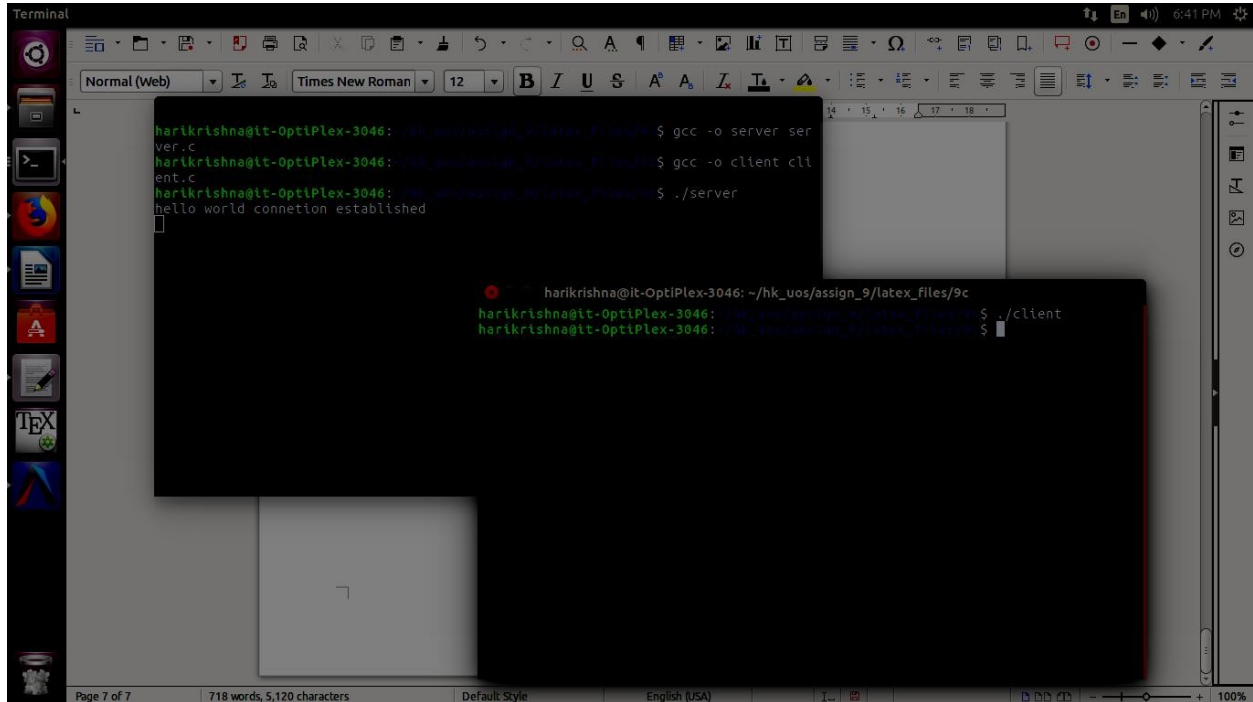
write(sockfd, "hello world connetion established", sizeof("hello world connetion established"));

close(sockfd);

}

```

OUTPUT:-



```
harikrishna@it-OptiPlex-3046: ~/hk_uos/assign_9/latex_files/9c $ gcc -o server server.c
harikrishna@it-OptiPlex-3046: ~/hk_uos/assign_9/latex_files/9c $ gcc -o client client.c
harikrishna@it-OptiPlex-3046: ~/hk_uos/assign_9/latex_files/9c $ ./server
hello world connection established
harikrishna@it-OptiPlex-3046: ~/hk_uos/assign_9/latex_files/9c $ ./client
```

Conclusion:

- TCP socket connection using system calls in C studied and client server connection established.

References:

[1] www.cs.cf.ac.uk/Dave/C/CE.html

9.4 Write two programs (server and client) to show how you can establish a UDP socket connection using the above functions.

Objectives:

1. To learn about fundamentals of IPC through C socket programming.
2. Learn and understand the OS interaction with socket programming.
3. Use of system call and IPC mechanism to write effective application programs.
4. To know the port numbering and process relation.
5. To know the iterative and concurrent server concept.

Theory:

JAVA SOCKET PROGRAMMING

TCP is a connection-oriented protocol layered on the top of IP of the TCP/IP stack with the ability to acknowledge receipt of packets at both ends. Acknowledgement ensures that the lost/corrupt packets can be retransmitted upon request. It also maintains a sequence in the sense that packets can be put back in the same order at the receiving end as they were transmitted. Although everything seems fair and advantageous at first look, it is also its weakness on occasion, because maintaining a guaranteed data transmission carries a fair amount of overhead (the header size of TCP packet is 20 bit whereas UDP header is 8 bit). In a situation where the order of the data is not that important or say, loss of a few packets does not matter to the verge of completely corrupting the data, TCP can be a real bottleneck. UDP is an unreliable connectionless protocol that neither guarantees that the packets will ever reach the destination nor that they will arrive in the same order they were sent. But, it works and surprisingly reaches the destination, without the slightest aura of "guarantee" or "reliability." TCP can be best suited for file transfer or the like where loss of bits is unacceptable. UDP, on the other hand, is best suited where a little loss in the transmission bits does not matter. For example, a few lost bits in video or audio signals are less severe without much quality degradation. Further, error correction in UDP can be built into data streams at the application level to account for missing information. So, UDP is not a total loss, after all.

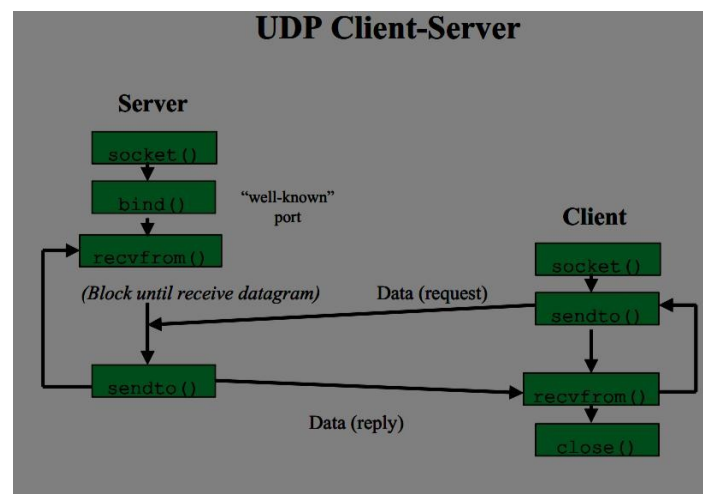


Fig9.2: UDP Architecture for Client-Server

Program:

SERVER:

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>

```

```

#include <netinet/in.h>
#include <arpa/inet.h>
#define MYPOR 4950
#define MAXBUFL 500
int main(int argc, char *argv[])
{
    int sockfd;
    struct sockaddr_in my_addr;
    struct sockaddr_in their_addr;
    int addr_len, numbytes;
    char buf[MAXBUFL];
    if((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
    {
        perror("Server-socket() sockfd error lol!");
        exit(1);
    }
    else
        printf("Server-socket() sockfd is OK...\n");
    /* host byte order */
    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(MYPOR);
    my_addr.sin_addr.s_addr = INADDR_ANY;
    memset(&(my_addr.sin_zero), '\0', 8);
    if(bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) == -1)
    {
        perror("Server-bind() error lol!");
        exit(1);
    }
    else
        printf("Server-bind() is OK...\n");

```

```

addr_len = sizeof(struct sockaddr);

if((numbytes = recvfrom(sockfd, buf, MAXBUFLen-1, 0, (struct sockaddr
*)&their_addr, &addr_len)) == -1)
{
perror("Server-recvfrom() error lol!");
exit(1);
}
else
{
printf("Server-Waiting and listening...\n");
printf("Server-recvfrom() is OK...\n");
}

printf("Server-Got packet from %s\n", inet_ntoa(their_addr.sin_addr));
printf("Server-Packet is %d bytes long\n", numbytes);
buf[numbytes] = '\0';
printf("Server-Packet contains \"%s\"\n", buf);
if(close(sockfd) != 0)
printf("Server-sockfd closing failed!\n");
else
printf("Server-sockfd successfully closed!\n");
return 0;
}

```

CLIENT:

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#define MYPORT 4950

```

```

int main(int argc, char *argv[ ])
{
    int sockfd;
    struct sockaddr_in their_addr;
    struct hostent *he;
    int numbytes;
    if (argc != 3)
    {
        fprintf(stderr, "Client-Usage: %s <hostname> <message>\n", argv[0]);
        exit(1);
    }
    if ((he = gethostbyname(argv[1])) == NULL)
    {
        perror("Client-gethostbyname() error lol!");
        exit(1);
    }
    else
        printf("Client-gethostname() is OK...\n");
    if((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
    {
        perror("Client-socket() error lol!");
        exit(1);
    }
    else
        printf("Client-socket() sockfd is OK...\n");
    their_addr.sin_family = AF_INET;
    printf("Using port: 4950\n");
    their_addr.sin_port = htons(MYPORT);
    their_addr.sin_addr = *((struct in_addr *)he-
>h_addr); memset(&(their_addr.sin_zero), '\0', 8);

```



```

if((numbytes = sendto(sockfd, argv[2], strlen(argv[2]), 0, (struct sockaddr
*)&their_addr, sizeof(struct sockaddr))) == -1)
{
perror("Client-sendto() error lol!");
exit(1);
}
else
printf("Client-sendto() is OK...\n");
printf("sent %d bytes to %s\n", numbytes, inet_ntoa(their_addr.sin_addr));

```

```

Terminal
harikrishna@lt-OptiPlex-3046: ~/hk_uos/assign_9/latex_files/9d$ ./server
Server-socket(): sockfd is OK...
Server-bind() is OK...
Server-Waiting and listening...
Server-recvfrom() is OK...
Server-Got packet from 127.0.0.1
Server-Packet is 5 bytes long
Server-Packet contains "hello"
Server-sockfd successfully closed!
harikrishna@lt-OptiPlex-3046: ~/hk_uos/assign_9/latex_files/9d$

harikrishna@lt-OptiPlex-3046: ~/hk_uos/assign_9/latex_files/9d$ ./client localhost
t hello
Client-gethostname() is OK...
Client-socket() sockfd is OK...
Using port: 4950
Client-sendto() is OK...
sent 5 bytes to 127.0.0.1
Client-sockfd successfully closed!
harikrishna@lt-OptiPlex-3046: ~/hk_uos/assign_9/latex_files/9d$

Pictures
C
t.c
udpserv.c

```

Conclusion:

- UDP socket connection using system calls in C studied and client server connection established.

References:

[1]www.cs.cf.ac.uk/Dave/C/CE.html

9.5 Implement echo server using TCP/UDP in iterative/concurrent logic.

Objectives:

1. To learn about fundamentals of IPC through C socket programming.
2. Learn and understand the OS interaction with socket programming.
3. Use of system call and IPC mechanism to write effective application programs.
4. To know the port numbering and process relation.
5. To know the iterative and concurrent server concept.

Theory:

JAVA SOCKET PROGRAMMING

1. TCP Echo Client

- In the TCP Echo client a socket is created.
- Using the socket a connection is made to the server using the connect() function.
- After a connection is established, we send messages input from the user and display the data received from the server using send() and read() functions.

2. UDP Echo Client

- In the UDP Echo client a socket is created.
- Then we bind the socket.

After the binding is successful, we send messages input from the user and display the data received from the server using sendto() and recvfrom() functions

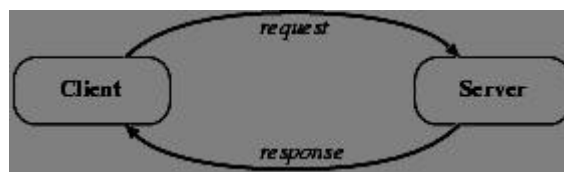


Fig9.5: Client-Server Architecture

3. TCP Echo Server

- In the TCP Echo server, we create a socket and bind to an advertised port number.
- After binding, the process listens for incoming connections.
- Then an infinite loop is started to process the client requests for connections.
- After a connection is requested, it accepts the connection from the client machine and forks a new process.
- The new process receives data from the client using recv() function and echoes the same data using the send() function.
- Please note that this server is capable of handling multiple clients as it forks a new process for every client trying to connect to the server.

3. UDP Echo Server

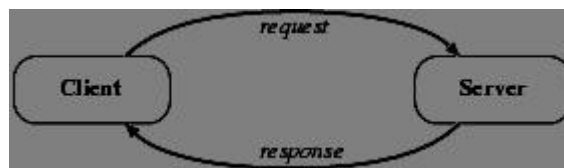


Fig9.6: Client-Server Architecture

- In the UDP Echo server, we create a socket and bind to an advertised port number.
- Then an infinite loop is started to process the client requests for connections.
- The process receives data from the client using `recvfrom ()` function and echoes the same data using the `sendto()` function.
- Please note that this server is capable of handles multiple clients automatically as UDP is a datagram based protocol hence no exclusive connection is required to a client in this case.

Program:

SERVER:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
    char str[100];
    int listen_fd, comm_fd;
    struct sockaddr_in servaddr;
    listen_fd = socket(AF_INET, SOCK_STREAM, 0);
    bzero( &servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(22000);
    bind(listen_fd, (struct sockaddr *) &servaddr,
    sizeof(servaddr)); listen(listen_fd, 10);
    comm_fd = accept(listen_fd, (struct sockaddr*) NULL, NULL);
    while(1)
    {
        bzero( str, 100);
        read(comm_fd,str,100);
        printf("Echoing back - %s",str);
        write(comm_fd, str, strlen(str)+1);
    }
}
```

CLIENT:

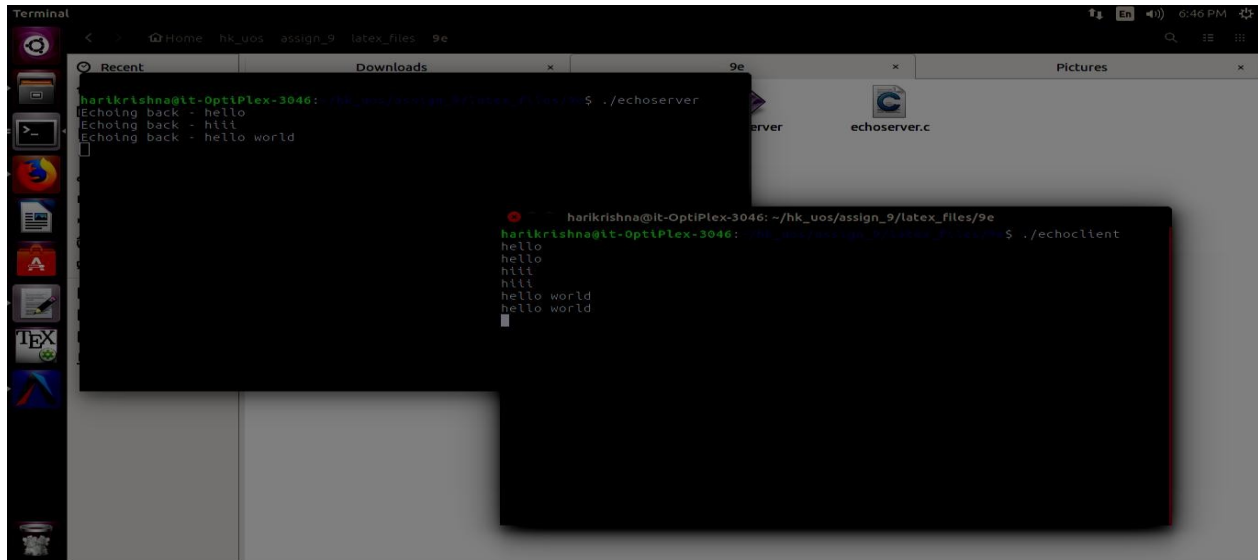
```
#include <stdio.h>
```

```

#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
int main(int argc,char **argv)
{
    int sockfd,n;
    char sendline[100];
    char recvline[100];
    struct sockaddr_in servaddr;
    sockfd=socket(AF_INET,SOCK_STREAM,0);
    bzero(&servaddr,sizeof servaddr);
    servaddr.sin_family=AF_INET;
    servaddr.sin_port=htons(22000);
    inet_pton(AF_INET,"127.0.0.1",&(servaddr.sin_addr));
    connect(sockfd,(struct sockaddr *)&servaddr,sizeof(servaddr));
    while(1)
    {
        bzero( sendline, 100);
        bzero( recvline, 100);
        fgets(sendline,100,stdin); /*stdin = 0 , for standard input */
        write(sockfd,sendline,strlen(sendline)+1);
        read(sockfd,recvline,100);
        printf("%s",recvline);
    }
}

```

Output:



The screenshot shows a Linux desktop environment. In the background, a file manager window displays the contents of a directory named '9e', which includes a file named 'echoserver.c'. In the foreground, two terminal windows are open. The top terminal window shows the execution of the './echoserver' program, which outputs 'Echoing back - hello', 'Echoing back - hltt', and 'Echoing back - hello world'. The bottom terminal window shows the execution of the './echoclient' program, which inputs 'hello', 'hello', 'hltt', 'hltt', 'hello world', and 'hello world'.

```
harikrishna@it-OptiPlex-3046: ~/hk_uos/assign_9/latex_files/9e$ ./echoserver
Echoing back - hello
Echoing back - hltt
Echoing back - hello world

harikrishna@it-OptiPlex-3046: ~/hk_uos/assign_9/latex_files/9e$ ./echoclient
hello
hello
hltt
hltt
hello world
hello world
```

Conclusion:

- Using TCP connection between Client and server, the echo server mechanism was implemented using iterative logic.

Reference:

[1]www.cs.cf.ac.uk/Dave/C/CE.html

9.6 Implement chatting using TCP/UDP socket (between two or more users.)

Objectives:

1. To learn about fundamentals of IPC through C socket programming.
2. Learn and understand the OS interaction with socket programming.
3. Use of system call and IPC mechanism to write effective application programs.
4. To know the port numbering and process relation.
5. To know the iterative and concurrent server concept.

Theory:

JAVA SOCKET PROGRAMMING

TCP is a connection-oriented protocol layered on the top of IP of the TCP/IP stack with the ability to acknowledge receipt of packets at both ends. Acknowledgement ensures that the lost/corrupt packets can be retransmitted upon request. It also maintains a sequence in the sense that packets can be put back in the same order at the receiving end as they were transmitted. Although everything seems fair and advantageous at first look, it is also its weakness on occasion, because maintaining a guaranteed data transmission carries a fair amount of overhead (the header size of TCP packet is 20 bit whereas UDP header is 8 bit). In a situation where the order of the data is not that important or say, loss of a few packets does not matter to the verge of completely corrupting the data, TCP can be a real bottleneck. UDP is an unreliable connectionless protocol that neither guarantees that the packets will ever reach the destination nor that they will arrive in the same order they were sent. But, it works and surprisingly reaches the destination, without the slightest aura of "guarantee" or "reliability." TCP can be best suited for file transfer or the like where loss of bits is unacceptable. UDP, on the other hand, is best suited where a little loss in the transmission bits does not matter. For example, a few lost bits in video or audio signals are less severe without much quality degradation. Further, error correction in UDP can be built into data streams at the application level to account for missing information. So, UDP is not a total loss, after all.

Program:

SERVER:

```
import java.net.*;
import java.io.*;
import java.util.*;

class Server2
{
    public static void main(String []args)throws Exception
    {
```

```

Scanner sc=new Scanner(System.in);
ServerSocket ss=new ServerSocket(5050);//5050 is port no.
System.out.println("Server is Waiting.....");
Socket s=ss.accept();//waiting for client
DataOutputStream dos=new DataOutputStream(s.getOutputStream());
DataInputStream dis=new DataInputStream(s.getInputStream());
String str="Welcomes u";
dos.writeUTF(str); //sends msg to client
str=dis.readUTF(); //reads msg send by client
System.out.println("From client:"+" "+str);
while(true)
{
System.out.print("\nEnter Message(from Server to client):");
str=sc.nextLine();
dos.writeUTF(str);
if(str.equals("bye"))
break;
str=dis.readUTF();
System.out.print("From Client:"+" "+str);
if(str.equals("bye"))
break;
}
try
{
ss.close();
s.close();
dos.close();
dis.close();
}
catch(Exception e){}

```

```

}
}
CLIENT:
import java.net.*;
import java.io.*;
import java.util.*;

class Client2
{
public static void main(String []args)throws Exception
{
Scanner sc=new Scanner(System.in);
Socket s=new Socket("localhost",5050);
DataOutputStream dos=new DataOutputStream(s.getOutputStream());
DataInputStream dis=new DataInputStream(s.getInputStream());
String str=dis.readUTF(); //receives msg send by server
System.out.println("From server:"+" "+str);
str="Thank u";
dos.writeUTF(str); //writes to server
while(true)
{
str=dis.readUTF();
System.out.println("From Server:"+" "+str);
if(str.equals("bye"))
break;
System.out.print("Enter Message(from Client to Server):");
str=sc.nextLine();
dos.writeUTF(str);
if(str.equals("bye"))
break;
}
}

```



```

try
{
s.close();
dos.close();
dis.close();
}
catch(Exception e){}
}
}

```

Output:

The screenshot displays two side-by-side Windows command prompt windows. The left window, titled 'Windows PowerShell', shows the server's execution. It starts with 'Server is waiting...', followed by 'From client: Thank u'. Then, it prompts 'Enter Message(from server to client):' and receives 'hello sir how r u?'. This is followed by 'From client: a fine i need some help', 'Enter Message(from server to client):', and 'yes sir, how can i help you?'. Next, it receives 'From client: i want to know about gmail.com', prompts 'Enter Message(from server to client):', and receives 'oh sure sir it is used for sending mails'. Finally, it receives 'From client: oh! ok thank you' and prompts 'Enter Message(from server to client):', receiving 'welcome sir'. The right window, titled 'Windows PowerShell', shows the client's execution. It starts with 'PS C:\Users\Kopika\Desktop> java client1.java', followed by 'PS C:\Users\Kopika\Desktop> java client2'. It then receives 'From server: welcome u', prompts 'Enter Message(from client to server):', and receives 'a fine i need some help'. This is followed by 'From server: yes sir, how can i help you?', 'Enter Message(from client to server):', and 'i want to know about gmail.com'. Next, it receives 'From server: oh sure sir! it is used for sending mails', prompts 'Enter Message(from client to server):', and receives 'oh! ok thank you'. Finally, it receives 'From server: welcome sir' and prompts 'Enter Message(from client to server):', receiving an empty line.

Conclusion:

- Multiway communication from client to server and vice versa can be implemented in the form of chatting by using TCP connection and socket programming in C.

References :

[1]www.cs.cf.ac.uk/Dave/C/CE.html

Chapter 10

Python:As a scripting language

10.1 Write a program to display the following pyramid. The number of lines in the pyramid should not be hard-coded. It should be obtained from the user. The pyramid should appear as close to the center of the screen as possible.

Objectives:

1. To learn about python as scripting option.

Theory:

Subprocess Management:

The subprocess module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes. This module intends to replace several older modules and functions:

`os.system`

`os.spawn*`

`os.popen*`

`popen2.*`

`commands.*`

Using the subprocess module:

The recommended way to launch subprocesses is to use the following convenience functions. For more advanced use cases when these do not meet your needs, use the underlying Popen interface.

`subprocess.call(args, *, stdin=None, stdout=None, stderr=None, shell=False)`

Run the command described by *args*. Wait for command to complete, then return the returncode attribute.

The arguments shown above are merely the most common ones, described below in Frequently Used Arguments (hence the slightly odd notation in the abbreviated signature). The full function

signature is the same as that of the Popen constructor - this functions passes all supplied arguments directly through to that interface.

Program:

```
import os
rows, columns = os.popen('stty size', 'r').read().split()
r=int(rows)
c=int(columns)
n = int(input("Enter number of rows:"))
for i in range(int(r/2-n/2)):
    print()
    for i in range(n):
        for k in range(int(c/2)-int(n/2)):
            print(" ",end="")
            for k in range(n-i-1):
                print(" ",end="")
                for k in range(2*i+1):
                    print("*",end="")
            print("\n",end="")
        for i in range(int(r/2-n/2)):
            print()
```

Output:

Enter number of rows:6

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

Conclusion:

- Basics of python like the concept of loops and conditional statements learnt.

References:

[1]<https://docs.python.org/3/>

10.2 Write a python function for prime number input limit in as parameter to it.**Objectives:**

1. To learn about python as scripting option.

Theory:

Prime Number:

A prime number is a whole number greater than 1 whose only factors are 1 and itself. A factor is a whole numbers that can be divided evenly into another number. The first few prime numbers are 2, 3, 5, 7, 11, 13, 17, 19, 23 and 29.

How do for loops work?

Many languages have conditions in the syntax of their *for* loop, such as a relational expression to determine if the loop is done, and an increment expression to determine the next loop value. In Python this is controlled instead by generating the appropriate sequence. Basically, any object with an iterable method can be used in a *for* loop. Even strings, despite not having an iterable method - but we'll not get on to that here. Having an iterable method basically means that the data can be presented in list form, where there are multiple values in an orderly fashion. You can define your own iterables by creating an object with `next()` and `iter()` methods.

Nested loops:

When you have a block of code you want to run x number of times, then a block of code within that code which you want to run y number of times, you use what is known as a "nested loop". In Python, these are heavily used whenever someone has a list of lists - an iterable object within an iterable object.

for x in xrange(1, 11):

```
for y in xrange(1, 11):  
    print '%d * %d = %d' % (x, y, x*y)
```

Early Exits:

Like the *while* loop, the *for* loop can be made to exit before the given object is finished. This is done using the *break* statement, which will immediately drop out of the loop and continue execution at the first statement after the block. You can also have an optional *else* clause, which will run should the *for* loop exit cleanly - that is, without breaking.

```
for x in xrange(3):  
    if x == 1:  
        break
```

Program:

```
n=int(input("Enter number:"))
```

```
for i in range(1,n+1):  
    c=0  
    #    print(i)  
    #    print()  
    if(i==1)  
    :  
    continu  
    e  
    if(i==2)  
    :  
    print(2)  
    continu  
    e  
    for j in range(2,i-1):  
        if(i%j==0): c=c+1  
    if(c==0):  
        print(i)
```

Output:

```
it@it:~$ python 10b.py
```

```
Enter number:20
```

```
2
```

```
3
```

```
5
```

```
7
```

```
11
```

```
13
```

```
17
```

```
19
```

Conclusion:

- Basics of python like the concept of loops and conditional statements learnt.

References:

[1]<https://docs.python.org/3/>

10.3 Take any txt file and count word frequencies in a file. (hint: file handling+basics)**Objectives:**

1. To learn about python as scripting option.

Theory:

File handling:

A file is some information or data which stays in the computer storage devices. You already know about different kinds of file , like your music files, video files, text files. Python gives you easy ways to manipulate these files. Generally we divide files in two categories, text file and binary file. Text files are simple text where as the binary files contain binary data which is only readable by computer.

File opening:

To open a file we use *open()* function. It requires two arguments, first the file path or file name, second which mode it should open. Modes are like

- “r” -> open read only, you can read the file but can not edit / delete anything inside
- “w” -> open with write power, means if the file exists then delete all content and open it to write
- “a” -> open in append mode

The default mode is read only, ie if you do not provide any mode it will open the file as read only. Let us open a file

```
> fobj = open("love.txt")
```

```
> fobj
```

```
<_io.TextIOWrapper name='love.txt' mode='r' encoding='UTF-8'>
```

Closing a file:

After opening a file one should always close the opened file. We use method *close()* for this.

```
> fobj = open("love.txt")
```

```
> fobj
```

```
<_io.TextIOWrapper name='love.txt' mode='r' encoding='UTF-8'>
```

```
>>> fobj.close()
```

Reading a file:

To read the whole file at once use the *read()* method.

```
> fobj = open("sample.txt")
```

```
> fobj.read()
```

```
'I love Python\nPradepto loves KDE\nSankarshan loves Openoffice\n'
```

Program:

```
f=open("10cfile.txt")
```

```
d={}
```

```
for line in f:
```

```
l=line.split()
```

```
#print(line)
```

```
#print(l)
```

```
for word in l:
```

```
#print(word)
```

```
if word in d:
d[word]=d[word]+1
else:
d[word]=1
for key in d:
print key," :",d[key]
```

Output:

it@it-OptiPlex-3046:~\$ python uox10.py

a : 2

A : 1

Peter : 4

of : 4

Piper : 4

pickled : 4

Where's : 1

picked : 4

peppers : 4

the : 1

peck : 4

If : 1

Conclusion:

- File handling and manipulation of data using list and dictionary learnt.

References:

[1]<https://docs.python.org/3/>

10.4 Generate frequency list of all the commands you have used, and show the top 5 commands along with their count. (Hint: history command will give you a list of all commands used.)

Objectives:

1. To learn about python as scripting option.

Theory:

Overview:

All of our services are currently running on Linux. In Linux, there is a very useful command to show you all of the last commands that have been recently used. The command is simply called history, but can also be accessed by looking at your .bash_history in your home folder. By default, the history command will show you the last five hundred commands you have entered.

history

You should now see a list quickly go by with the last 500 commands used, like the example below. If you like, you can just use the up arrow and down arrow to browse for any particular command you may have used recently.

```
496 ls -la
497 ls
498 history
499 ls
500 cd domains
501 cd ..
502 ls
503 history
504 cd ls
505 ls
506 cd data
507 ls
508 cd ..
509 cd domains
```

510 ls

511 cd ..

512 history

Program:

```
import operator
from subprocess import Popen, PIPE, STDOUT
shell_command = 'bash -i -c "history -r;
history"'
event = Popen(shell_command, shell=True, stdin=PIPE,
stdout=PIPE, stderr=STDOUT)
output = event.communicate()
l=list(output)
l1=l[0].split("\n")
d={}
for key in l1:
key=key.split(" ")
if key[len(key)-1] in d:
d[key[len(key)-1]]=d[key[len(key)-1]]+1
else:
d[key[len(key)-1]]=1
l1=[]
for key,value in d.iteritems():
temp=[key,value]
l1.append(temp)
l1=sorted(l1,key=operator.itemgetter(1))
l1.reverse()
for i in range(5):
print(l1[i])
```

Output:

```
it@it-OptiPlex-3046:~$ python temp2.py
['./a.out', 136]
```

```
['g++ inher.cpp', 60]  
['gedit inher.cpp', 60]  
['packettracer', 29]  
['cd Desktop', 20]
```

Conclusion:

- Implementing of bash command using os library and some basics to find frequency of bash commands used.

References:

[1]<https://docs.python.org/3/>

10.5 Write a shell script that will take a filename as input and check if it is executable. 2. Modify the script in the previous question, to remove the execute permissions, if the file is executable.

Objectives:

1. To learn about python as scripting option.

Theory:

Executable File:

An executable file is a file that is used to perform various functions or operations on a computer. Unlike a data file, an executable file cannot be read because it has been compiled. On an IBM compatible computer, common executable files are .BAT, .COM, .EXE, and .BIN. On an Apple mac computer running macOS the .DMG and .APP files are executable files. Depending on the operating system and its setup, there can also be other executable files.

access():

Description:

The method access() uses the real uid/gid to test for access to path. Most operations will use the effective uid/gid, therefore this routine can be used in a suid/sgid environment to test if the invoking user has the specified access to path. It returns True if access is allowed, False if not.

Syntax:

Following is the syntax for access() method –

```
os.access(path, mode);
```

Parameters:

- path – This is the path which would be tested for existence or any access.
- mode – This should be F_OK to test the existence of path, or it can be the inclusive OR of one or more of R_OK, W_OK, and X_OK to test permissions.
 - os.F_OK – Value to pass as the mode parameter of access() to test the existence of path.
 - os.R_OK – Value to include in the mode parameter of access() to test the readability of path.
 - os.W_OK Value to include in the mode parameter of access() to test the writability of path.
 - os.X_OK Value to include in the mode parameter of access() to determine if path can be executed.

Return Value:

This method returns True if access is allowed, False if not.

Program:

```
import os
print "Enter file name:";
f = raw_input();
if os.access(f,os.X_OK):
print "Executable.....changing mode";
os.chmod(f,666)
if os.access(f,os.X_OK):
print "Mode Change error";
else:
print "Mode changed to non executable";
else:
print "Not Executable";
```

Output:

```
it@it-OptiPlex-3046:~/Gaurav/UOS assignments/Assignment No. 10$ python 10e.pyEnter
file name:
10h.docx
Executable.....changing mode
Mode changed to non executable
```

Conclusion:

- Various system calls can be invoked using OS library of python to check permissions on files and modify them.

References:

[1]<https://docs.python.org/3/>

10.6 Generate a word frequency list for wonderland.txt. Hint: use grep, tr, sort, uniq (or anything else that you want)

Objectives:

1. To learn about python as scripting option.

Theory:

File handling:

A file is some information or data which stays in the computer storage devices. You already know about different kinds of file, like your music files, video files, text files. Python gives you easy ways to manipulate these files. Generally we divide files in two categories, text file and binary file. Text files are simple text whereas the binary files contain binary data which is only readable by computer.

File opening:

To open a file we use *open()* function. It requires two arguments, first the file path or file name, second which mode it should open. Modes are like

- “r” -> open read only, you can read the file but can not edit / delete anything inside
- “w” -> open with write power, means if the file exists then delete all content and open it to write
- “a” -> open in append mode

The default mode is read only, ie if you do not provide any mode it will open the file as read only. Let us open a file

```
> fobj = open("love.txt")
```

```
> fobj
```

```
<_io.TextIOWrapper name='love.txt' mode='r' encoding='UTF-8'>
```

Closing a file:

After opening a file one should always close the opened file. We use method *close()* for this.

```
> fobj = open("love.txt")
> fobj
<_io.TextIOWrapper name='love.txt' mode='r' encoding='UTF-8'>
>>> fobj.close()
```

Reading a file:

To read the whole file at once use the *read()* method.

```
> fobj = open("sample.txt")
> fobj.read()
```

Program:

```
f=open("wonderland.txt")
d={}
for line in f:
    l=line.split()
    #print(line)
    #print(l)
    for word in l:
        #print(word)
        if word in d:
            d[word]=d[word]+1
        else:
            d[word]=1
for key in d:
    print key," : ",d[key]
```

Output:

```
it@it-OptiPlex-3046:~/Desktop$ python 10c.py
"--and : 2
figure!" : 1
four : 6
attending!" : 1
hanging : 3
```

ringlets : 1
story!" : 2
(And : 1
Foundation : 14
IX. : 1
cake, : 2
_Who : 2
dear!" : 3
dear!" : 1
joined): : 1
wood, : 1
wood. : 3
leisurely : 1
screaming : 1
prize : 1
wooden : 1
solid : 1
persisted : 1
and so on...

Conclusion:

- File handling and manipulation of data using list and dictionary learnt.

References:

[1]<https://docs.python.org/3>

10.7 Write a bash script that takes 2 or more arguments

Objectives:

1. To learn about python as scripting option.

Theory:

File handling:

A file is some information or data which stays in the computer storage devices. You already know about different kinds of file , like your music files, video files, text files. Python gives you easy ways to manipulate these files. Generally we divide files in two categories, text file and binary file. Text files are simple text where as the binary files contain binary data which is only readable by computer.

File opening:

To open a file we use *open()* function. It requires two arguments, first the file path or file name, second which mode it should open. Modes are like

- “r” -> open read only, you can read the file but can not edit / delete anything inside
- “w” -> open with write power, means if the file exists then delete all content and open it to write
- “a” -> open in append mode

The default mode is read only, ie if you do not provide any mode it will open the file as read only. Let us open a file

```
> fobj = open("love.txt")
```

```
> fobj
```

```
<_io.TextIOWrapper name='love.txt' mode='r' encoding='UTF-8'>
```

Closing a file:

After opening a file one should always close the opened file. We use method *close()* for this.

```
> fobj = open("love.txt")
```

```
> fobj
```

```
<_io.TextIOWrapper name='love.txt' mode='r' encoding='UTF-8'>
```

```
>>> fobj.close()
```

Reading a file:

To read the whole file at once use the *read()* method.

```
> fobj = open("sample.txt")
```



```
> fobj.read()
```

Program:

```
import sys
```

```
l=sys.argv
```

```
for i in range(1,len(l)):
    #print("witing ",l[i])
    with open(str(l[i])) as f:
        with open("test1.txt", "a") as fl:
            for line in f:
                fl.write(line)
            #print(line)
            fl=open("test1.txt")
            print("New file")
            for line in fl:

                print(line)
```

Output:

New file

The stty command sets certain I/O options for the device that is the current standard input. This command writes output to the device that is the current standard output.

This version of the operating system uses the standard X/Open Portability Guide Issue 4 interface to control the terminals, maintaining a compatibility with POSIX and BSD interfaces. The stty command supports both POSIX and BSD compliant options, but the usage of POSIX options is strongly recommended. A list of obsolete BSD options, with the corresponding POSIX options, is also provided.

The open() function is used to open files in our system, the filename is the name of the file to be opened.

The mode indicates, how the file is going to be opened "r" for reading, "w" for writing and "a" for a appending.

The open function takes two arguments, the name of the file and the mode for which we would like to open the file.

By default, when only the filename is passed, the open function opens the file in read mode.

Conclusion:

- File handling, command line arguments and checking of file using bash commands implemented in python.

References:

[1]<https://docs.python.org/3/>

10.8 Write a python function for merge/quick sort for integer list as parameter to it.

Objectives:

1. To learn about python as scripting option.

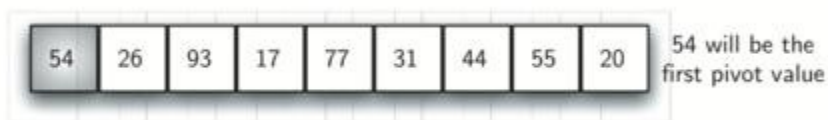
Theory:

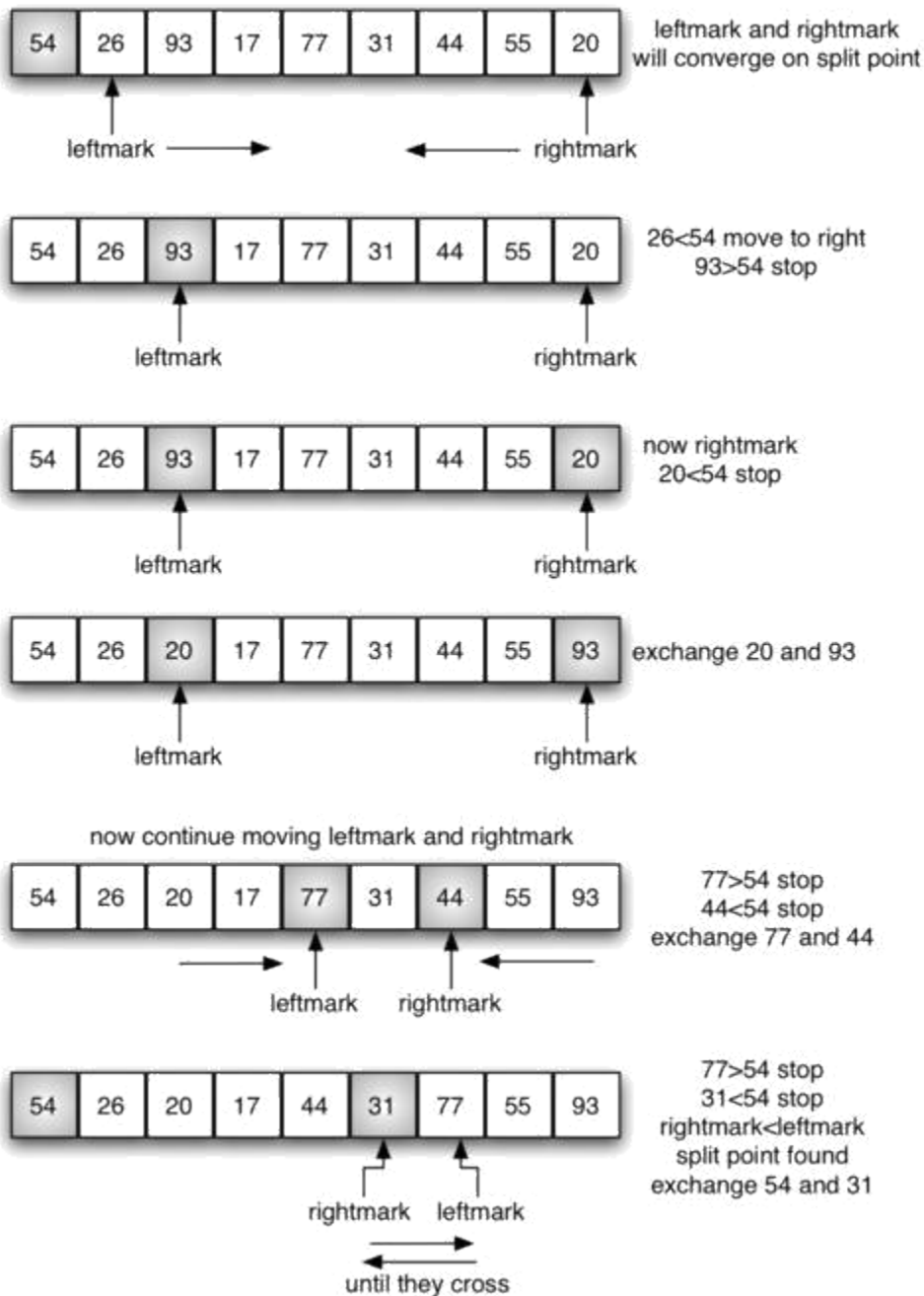
Quick Sort:

The quick sort uses divide and conquer to gain the same advantages as the merge sort, while not using additional storage. As a trade-off, however, it is possible that the list may not be divided in half. When this happens, we will see that performance is diminished.

A quick sort first selects a value, which is called the pivot value. Although there are many different ways to choose the pivot value, we will simply use the first item in the list. The role of the pivot value is to assist with splitting the list. The actual position where the pivot value belongs in the final sorted list, commonly called the split point, will be used to divide the list for subsequent calls to the quick sort.

Figure shows that 54 will serve as our first pivot value. Since we have looked at this example a few times already, we know that 54 will eventually end up in the position currently holding 31. The partition process will happen next. It will find the split point and at the same time move other items to the appropriate side of the list, either less than or greater than the pivot value.





Program:

```
def quickSort(alist):
    quickSortHelper(alist,0,len(alist)-1)

def quickSortHelper(alist,first,last):
    if first<last:

        splitpoint = partition(alist,first,last)
```

```
quickSortHelper(alist,first,splitpoint-1)
quickSortHelper(alist,splitpoint+1,last)
```

```
def partition(alist,first,last):
    pivotvalue = alist[first]
```

```
    leftmark = first+1
    rightmark = last
```

```
    done = False
    while not done:
```

```
        while leftmark <= rightmark and alist[leftmark] <= pivotvalue:
            leftmark = leftmark + 1
```

```
        while alist[rightmark] >= pivotvalue and rightmark >= leftmark:
            rightmark = rightmark - 1
```

```
    if rightmark < leftmark:
        done = True
    else:
        temp = alist[leftmark]
        alist[leftmark] = alist[rightmark]
        alist[rightmark] = temp
```

```
    temp = alist[first]
    alist[first] = alist[rightmark]
    alist[rightmark] = temp
```

```
    return rightmark
```

```
alist = list(map(int, input("Enter a list: ").split()))
```

```
quickSort(alist)  
print(alist)
```

Output:

Enter a list: 485 982745 9827 9274
[485, 9274, 9827, 982745]

Conclusion:

- Sorting a list using efficient algorithms like merge and quick sort was implemented to enhance the application of basics of python

References:

[1]<https://docs.python.org/3/>

Chapter 11

IPC

11.1 - Implement the program IPC/IPS using MPI library. Communication in processes of users.

Objectives:

1. To learn about IPC through MPI.
2. Use of IPC mechanism to write effective application programs.

Theory:

Concurrent programming using the extremely popular Boost libraries is a lot of fun. Boost has several libraries within the concurrent programming space—the Interprocess library (IPC) for shared memory, memory-mapped I/O, and message queue; the Thread library for portable multi-threading; the Message Passing Interface (MPI) library for message passing, which finds use in distributed computing; and the Asio library for portable networking using sockets and other low-level functions, to name just a few.

In computer science, inter-process communication or interprocess communication (IPC) refers specifically to the mechanisms an operating system provides to allow the processes to manage shared data. Typically, applications can use IPC, categorized as clients and servers, where the client requests data and the server responds to client requests.

Program:

```
#include <omp.h>
#include "mpi.h"
#include <stdio.h>
#define _NUM_THREADS 4
/* Each MPI process spawns a distinct OpenMP
 * master thread; so limit the number of MPI
 * processes to one per node
 */
int main (int argc, char *argv[])
{
    int p, my_rank, c;
    /* set number of threads to spawn */
    omp_set_num_threads(_NUM_THREADS);
    /* initialize MPI stu */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    /* the following is a parallel OpenMP
     * executed by each MPI process
     */
    #pragma omp parallel reduction(+:c)
    {
        c = omp_get_num_threads();
    }
}
```

```

/* expect a number to get printed for each MPI process */
printf("%d\n",c);
/* finalize MPI */
MPI_Finalize();
return 0;
}

```

Conclusion:

- We learned about the concepts of parallel programming using MPI.
- Use of MPI in Inter-process Communication
- Efficient use of processor and thereby reducing time by using MPI

References:

- [1]https://www.ibm.com/developerworks/aix/library/au-concurrent_boost/index.html
 [2]<https://ieeexplore.ieee.org/document/8082077>

11.2 - Implement the program IPC/IPS using MPI library. Communication in processes of OS's: Unix.

Objectives:

1. To learn about IPC through MPI.
2. Use of IPC mechanism to write effective application programs.

Theory:

What is MPI ? (Message Passing Interface)

- MPI is a *specification* for the developers and users of message passing libraries. By itself, it is NOT a library - but rather the specification of what such a library should be.
- MPI primarily addresses the *message-passing parallel programming model*: data is moved from the address space of one process to that of another process through cooperative operations on each process.
- Simply stated, the goal of the Message Passing Interface is to provide a widely used standard for writing message passing programs.

► Programming Model:

- Originally, MPI was designed for distributed memory architectures, which were becoming increasingly popular at that time (1980s - early 1990s).

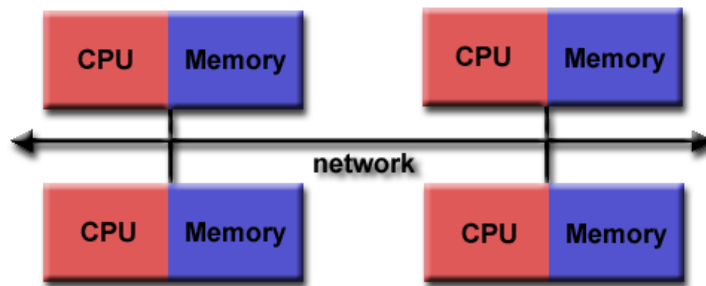
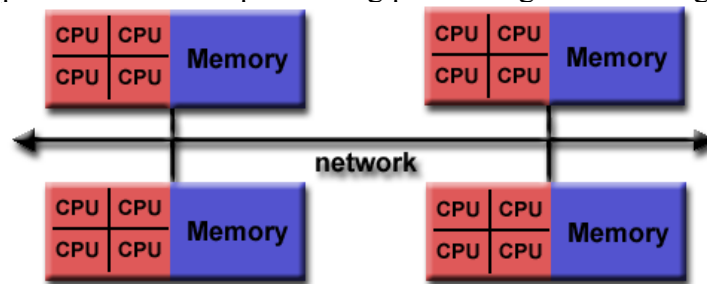


Fig:11.1 SMPs Architecture

- As architecture trends changed, shared memory SMPs were combined over networks creating hybrid distributed memory / shared memory systems.
- MPI implementors adapted their libraries to handle both types of underlying memory architectures seamlessly. They also adapted/developed ways of handling different interconnects and protocols.
- Today, MPI runs on virtually any hardware platform:
 - Distributed Memory
 - Shared Memory
 - Hybrid
- The programming model clearly remains a distributed memory model however, regardless of the underlying physical architecture of the machine.
 - All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI



• Fig:11.2 SMPs Architecture

► Reasons for Using MPI:

- Standardization
- Portability
- Performance Opportunities
- Functionality
- Availability

Program:

```
#include <boost/mpi.hpp>
#include <iostream>
```



```

int main(int argc, char* argv[])
{
    boost::mpi::environment env(argc, argv);
    boost::mpi::communicator world;

    if (world.rank() == 0) {
        world.send(1, 9, 32);
        world.send(2, 9, 33);
    } else {
        int data;
        world.recv(0, 9, data);
        std::cout << "In process " << world.rank() << "with data " << data
                    << std::endl;
    }
    return 0;
}

```

Output:

```

    In process 1 with data 32
1
2 In process 2 with data 33

```

Conclusion:

- We learned about the concepts of parallel programming using MPI.
- Use of MPI in Inter-process Communication
- From a usability standpoint, IPC is easy to use. However, the MPI library is dependant on native MPI implementations,

References:

- [1]https://www.ibm.com/developerworks/aix/library/au-concurrent_boost/index.html
 [2]<https://ieeexplore.ieee.org/document/8082077>

11.3 - configure cluster and experiment MPI program on it.

Objectives:

1. To learn about IPC through MPI.
2. Use of IPC mechanism to write effective application programs.

Theory:

The first step to building an MPI program is including the MPI header files with `#include <mpi.h>`. After this, the MPI environment must be initialized with:

```
MPI_Init(  
  
int* argc,  
  
char*** argv)
```

During MPI_Init, all of MPI's global and internal variables are constructed. For example, a communicator is formed around all of the processes that were spawned, and unique ranks are assigned to each process. Currently, MPI_Init takes two arguments that are not necessary, and the extra parameters are simply left as extra space in case future implementations might need them.(many times in practice the MPI code is the whole executable so it would make sense to pass all the command line arguments.)

After MPI_Init, there are two main functions that are called. These two functions are used in almost every single MPI program that you will write.

```
MPI_Comm_size(  
  
MPI_Comm communicator,  
  
int* size)
```

MPI_Comm_size returns the size of a communicator. In our example,

MPI_COMM_WORLD (which is constructed for us by MPI) encloses all of the processes in the job, so this call should return the amount of processes that were requested for the job.

```
MPI_Comm_rank(  
  
MPI_Comm communicator,  
  
int* rank)
```

MPI_Comm_rank returns the rank of a process in a communicator. Each process inside of a communicator is assigned an incremental rank starting from zero. The ranks of the processes are primarily used for identification purposes when sending and receiving messages.

MPI_Comm_rank returns the rank of a process in a communicator. Each process inside of a communicator is assigned an incremental rank starting from zero. The ranks of the processes are primarily used for identification purposes when sending and receiving messages.

```
MPI_Get_processor_name(  
  
char* name,  
  
int* name_length)
```

MPI_Get_processor_name obtains the actual name of the processor on which the process is executing. The final call in this program is:

MPI_Finalize()

MPI_Finalize is used to clean up the MPI environment. No more MPI calls can be made after this one.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>
#define N 1E8
#define d 1E-8

int main (int argc, char* argv[])
{
    int rank, size, error, i, result=0, sum=0;
    double pi=0.0, begin=0.0, end=0.0, x, y;
    error=MPI_Init (&argc, &argv);

    //Get process ID
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);

    //Get processes Number
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    //Synchronize all processes and get the begin time MPI_Barrier(MPI_COMM_WORLD); begin
    = MPI_Wtime();
    srand((int)time(0));

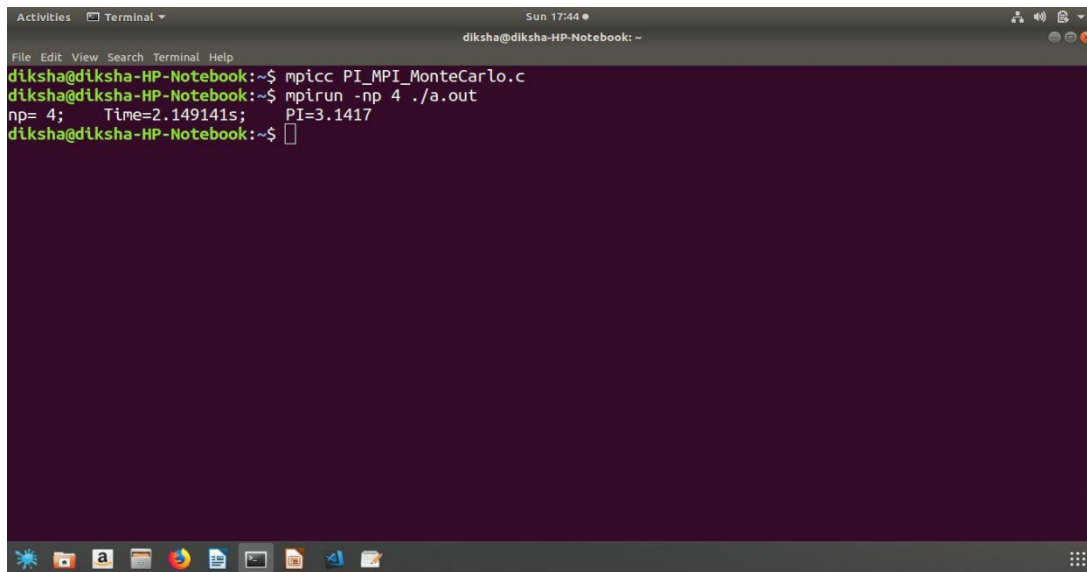
    //Each process will caculate a part of the sum
    for (i=rank; i<N; i+=size)
    {
        x=rand()/(RAND_MAX+1.0);
        y=rand()/(RAND_MAX+1.0);
        if(x*x+y*y<1.0)
            result++;
    }

    //Sum up all results
    MPI_Reduce(&result, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    //Synchronize all processes and get the end time
    MPI_Barrier(MPI_COMM_WORLD);
    end = MPI_Wtime();
    //Caculate and print PI
    if (rank==0)
    {
        pi=4*d*sum;
```

```
printf("np=%2d; Time=%fs; PI=%0.4f\n", size, end-begin, pi);
}
error=MPI_Finalize();
return 0;
}
```

Output:



```
File Edit View Search Terminal Help
diksha@diksha-HP-Notebook:~$ mpicc PI_MPI_MonteCarlo.c
diksha@diksha-HP-Notebook:~$ mpirun -np 4 ./a.out
np= 4; Time=2.149141s; PI=3.1417
diksha@diksha-HP-Notebook:~$
```

Conclusion:

- We learned about the concepts of parallel programming using MPI.
- Use of MPI in Inter-process Communication
- Efficient use of processor and thereby reducing time by using MPI

References:

- [1]https://www.ibm.com/developerworks/aix/library/au-concurrent_boost/index.html
[2]<https://ieeexplore.ieee.org/document/8082077>

12.1 - Implement the program for threads using Open MP library. Print number of core.

Objectives:

1. To learn about openMP for better use of multicore system.

Theory:

An OpenMP program has sections that are sequential and sections that are parallel. In general an OpenMP program starts with a sequential section in which it sets up the environment, initializes the variables, and so on.

When run, an OpenMP program will use one thread (in the sequential sections), and several threads (in the parallel sections).

There is one thread that runs from the beginning to the end, and it's called the *master thread*. The parallel sections of the program will cause additional threads to fork. These are called the *slave threads*.

A section of code that is to be executed in parallel is marked by a special directive (omp pragma). When the execution reaches a parallel section (marked by omp pragma), this directive will cause slave threads to form. Each thread executes the parallel section of the code independently. When a thread finishes, it joins the master. When all threads finish, the master continues with code following the parallel section.

Each thread has an ID attached to it that can be obtained using a runtime library function (called `omp_get_thread_num()`). The ID of the master thread is 0.

Program:

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
#define CHUNKSIZE 10
#define N 100

int main (int argc, char *argv[])
{
    int nthreads, tid, i, chunk;
    float a[N], b[N], c[N];

    /* Some initializations */
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;

    #pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
    {
        tid = omp_get_thread_num();
        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    }
```

```

printf("Thread %d starting...\n",tid);

#pragma omp for schedule(dynamic,chunk)
for (i=0; i < N; i++)
    c[i] = a[i] + b[i];
    printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
}

} /* end of parallel section */

}

```

Output:

```

Number of threads = 2
Thread 1 starting...
Thread 0 starting...
Thread 0: c[10]= 20.000000
Thread 1: c[0]= 0.000000
Thread 0: c[11]= 22.000000
Thread 1: c[1]= 2.000000
Thread 0: c[12]= 24.000000
Thread 1: c[2]= 4.000000
Thread 0: c[13]= 26.000000
Thread 1: c[3]= 6.000000
Thread 0: c[14]= 28.000000
Thread 1: c[4]= 8.000000
Thread 0: c[15]= 30.000000
Thread 1: c[5]= 10.000000
Thread 0: c[16]= 32.000000
Thread 1: c[6]= 12.000000
Thread 0: c[17]= 34.000000
Thread 1: c[7]= 14.000000
Thread 0: c[18]= 36.000000
Thread 1: c[8]= 16.000000
Thread 0: c[19]= 38.000000
Thread 1: c[9]= 18.000000
Thread 0: c[20]= 40.000000
Thread 1: c[30]= 60.000000
Thread 0: c[21]= 42.000000
Thread 1: c[31]= 62.000000
Thread 0: c[22]= 44.000000
Thread 1: c[32]= 64.000000
Thread 0: c[23]= 46.000000
Thread 1: c[33]= 66.000000

```

Conclusion:

- We learned about the concepts of parallel programming using OpenMP.
- Use of OpenMP in Shared memory programming
- Efficient use of processor and thereby reducing time by using OpenMP.

References:

[1]<https://www.codeproject.com/Articles/60176/A-Beginner-s-Primer-to-OpenMP>

[2][https://www.researchgate.net/post/](https://www.researchgate.net/post/Is_there_a_way_to_specify_how_many_cores_a_program_should_run_in_other_words_can_I_control_where_the_threads_are_mapped)

Is_there_a_way_to_specify_how_many_cores_a_program_should_run-
in_other_words_can_I_control_where_the_threads_are_mapped

[3]<https://www.embedded.com/design/mcus-processors-and-socs/4007155/Using-OpenMP-for-programming-parallel-threads-in-multicore-applications-Part-2>

[4]<http://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-basics.html>

12.2 - Implement the program OpenMP threads and print prime number task, odd number and Fibonacci series using three thread on core. Comment on performance CPU.

Objectives:

1. To learn about openMP for better use of the multicore system.

Theory:

► OpenMP Is:

- An Application Program Interface (API) that may be used to explicitly direct *multi-threaded, shared memory* parallelism.
- Comprised of three primary API components:
 - Compiler Directives
 - Runtime Library Routines
 - Environment Variables
- An abbreviation for: Open Multi-Processing

► Shared Memory Model:

- OpenMP is designed for multi-processor/core, shared memory machines. The underlying architecture can be shared memory UMA or NUMA.

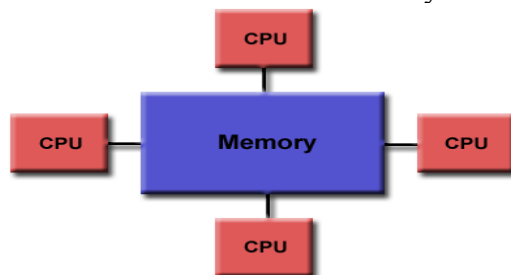
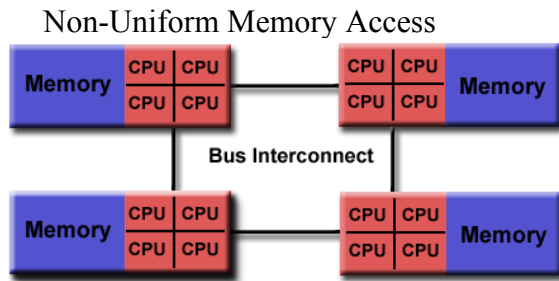


Fig:11.3 Shared Memory Model



Uniform Memory Access

Program:

```
#include<stdio.h>

#include<stdlib.h>

#include<omp.h>

int main()
{
    int i, n, t1 = 0, t2 = 1, nextTerm=0,j=0,counter,i,a,count;
    #pragma omp parallel private(i,counter,a,count)
    In ID = omp_get_thread_num();
    if(ID==1)
    {
        for (i = 1; i <= n; ++i)
        {
            printf("%dth Fib no: =%d \n", nextTerm);
            nextTerm = t1 + t2;
            t1 = t2;
            t2 = nextTerm;
        }
    }
    if(ID==2)
    {
        for(counter = 1; counter <= 100; counter++) {
```



```

        if(counter%2 == 1) {
printf("%dth Odd no: =%d \n", j,counter);

        }
    }
if(ID==3)
{
    for (i=100;i<500;i++)
    {
        count=0;
        for (a=1;a<=i;a++)
        {
            if (i%a==0)
                count++;
        }
        if (count==2)

printf("%dth Prime no: =%d \n", j,i);

        j++;
    }
}
}

```

Output:

```

1th Fib no: = 0
2th Fib no: = 1
3th Fib no: = 1
4th Fib no: = 2
5th Fib no: = 3
1th Prime no: = 2
6th Fib no: = 5
1th Odd no: = 1
2th Odd no: = 3
2th Prime no: = 3

```

7th Fib no: = 8

3th Odd no: = 5

3th Prime no: = 5

8th Fib no: = 13

4th Prime no: = 7

9th Fib no: = 21

4th Odd no: = 7

5th Prime no: = 11

10th Fib no: = 34

6th Prime no: = 13

Conclusion:

- We learned about the concepts of parallel programming using OpenMP.
- Use of OpenMP in Shared memory programming
- Efficient use of processor and thereby reducing time by using OpenMP.

References:

[1]<https://www.codeproject.com/Articles/60176/A-Beginner-s-Primer-to-OpenMP>

[2]<https://www.embedded.com/design/mcus-processors-and-socs/4007155/Using-OpenMP-for-programming-parallel-threads-in-multicore-applications-Part-2>

[3]<http://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-basics.html>

12.1 - Write a program for Matrix Multiplication in OpenMP.

Objectives:

1. To learn about openMP for better use of the multicore system.

Theory:

OpenMP is an implementation of multithreading, a method of parallelizing whereby a master thread (a series of instructions executed consecutively) forks a specified number of slave threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

The section of code that is meant to run in parallel is marked accordingly, with a compiler directive that will cause the threads to form before the section is executed. Each thread has an id attached to it which can be obtained using a function (called **omp_get_thread_num()**). The thread id is an integer, and the master thread has an id of 0. After the execution of the parallelized code, the threads join back into the master thread, which continues onward to the end of the program.

By default, each thread executes the parallelized section of code independently. Work-sharing constructs can be used to divide a task among the threads so that each thread executes its allocated part of the code. Both task parallelism and data parallelism can be achieved using OpenMP in this way.

The runtime environment allocates threads to processors depending on usage, machine load and other factors. The runtime environment can assign the number of threads based on environment variables, or the code can do so using functions. The OpenMP functions are included in a header file labelled `omp.h` in C/C++.

Program:

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

int main()
{
    clock_t begin = clock();
    int a[3][3],b[3][3],c[3][3],i,j,k,l;

    int time;
    for(i=0;i < 3 ;i++)
    {
        for(j=0;j<3;j++)
        {
            a[i][j]=5;
            b[i][j]=5;
            c[i][j]=0;
        }
    }

    #pragma OMP parallel for private(l,k)
    for(i=0;i<3;i++)
```

```

{
for(l=0;l<3;l++)
{
for(k=0;k<3;k++)
c[i][l]=a[i][k]*b[k][l];
time=omp_get_wtime();
//printf("time :%d\n",time);
}
}
printf("Matrix Multiplication\n");
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
printf(" %d ",c[i][j]);
}
printf("\n");
}
clock_t end = clock();
printf("CPU time required: %.5lf\n",(double)(end - begin) / CLOCKS_PER_SEC);
return 0;
}

```

Output:

```

it@it-OptiPlex-3046: ~
it@it-OptiPlex-3046:~$ gcc -fopenmp MatrixMul.c
it@it-OptiPlex-3046:~$ ./a.out
Matrix Multiplication
75 75 75
75 75 75
75 75 75
CPU time required: 0.00006
it@it-OptiPlex-3046:~$

```

Conclusion:

- We learned about the concepts of parallel programming using OpenMP.
- Use of OpenMP in Shared memory programming
- Implementation of matrix multiplication using OpenMP takes minimum time than serial execution.

References:

[1]ftp://10.10.13.16/pub/Academic_Material/TYIT/Semister_2/Unix_Operating_System/
OpenMP

[2]<https://www.codeproject.com/Articles/60176/A-Beginner-s-Primer-to-OpenMP>

[3]<https://www.embedded.com/design/mcus-processors-and-socs/4007155/Using-OpenMP-for-programming-parallel-threads-in-multicore-applications-Part-2>

[4]<http://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-basics.html>

STREAMS message/PIPEs/FIFO:pipe, popen and pclose Functions

Assignment No: 13_a

Title:

Send data from parent to child over a pipe

Objectives:

5. To learn about STREAMS message/PIPEs/FIFO:pipe, popen and pclose Functions

Theory:

Pipes

Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems. Pipes have two limitations:

b Historically, they have been half duplex (data flows in only one direction). Some systems now provide full-duplex pipes, but for maximum portability, we should never assume that this is the case.

c Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child.

FIFOs (Section 15.5) get around the second limitation, and that UNIX domain sockets (Section 17.2) get around both limitations.

Despite these limitations, half-duplex pipes are still the most commonly used form of IPC. Every time you type a sequence of commands in a pipeline for the shell to execute, the shell creates a separate process for each command and links the standard output of one process to the standard input of the next using a pipe.

A pipe is created by calling the `pipe` function.

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

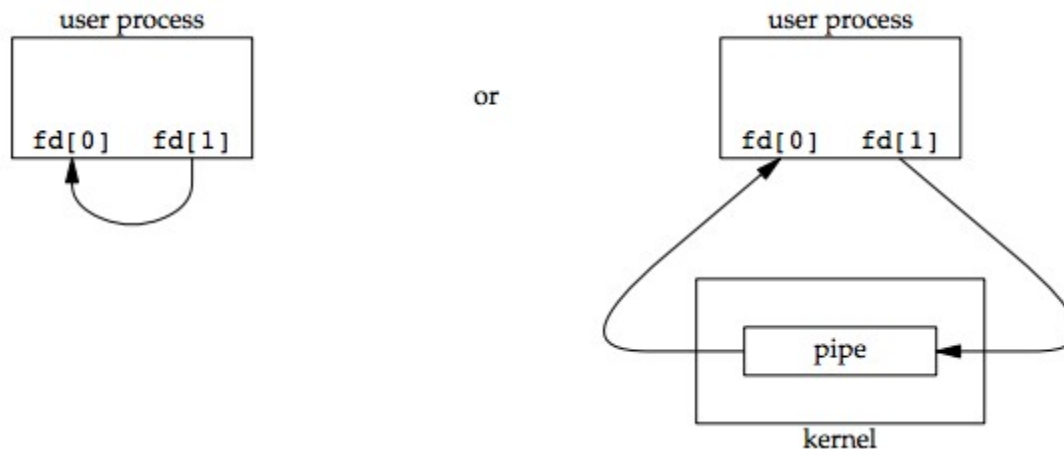
```
int pipe(int fd[2]);
```

```
/* Returns: 0 if OK, -1 on error */
```

Two file descriptors are returned through the *fd* argument: *fd[0]* is open for reading, and *fd[1]* is open for writing. The output of *fd[1]* is the input for *fd[0]*.

POSIX.1 allows for implementations to support full-duplex pipes. For these implementations, *fd[0]* and *fd[1]* are open for both reading and writing.

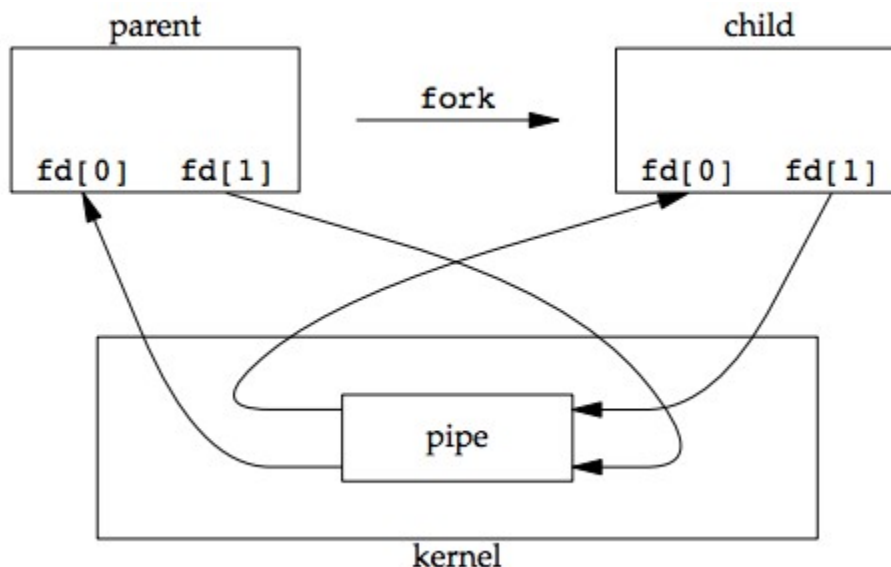
Two ways to picture a half-duplex pipe are shown in the figure below. The left half of the figure shows the two ends of the pipe connected in a single process. The right half of the figure emphasizes that the data in the pipe flows through the kernel.



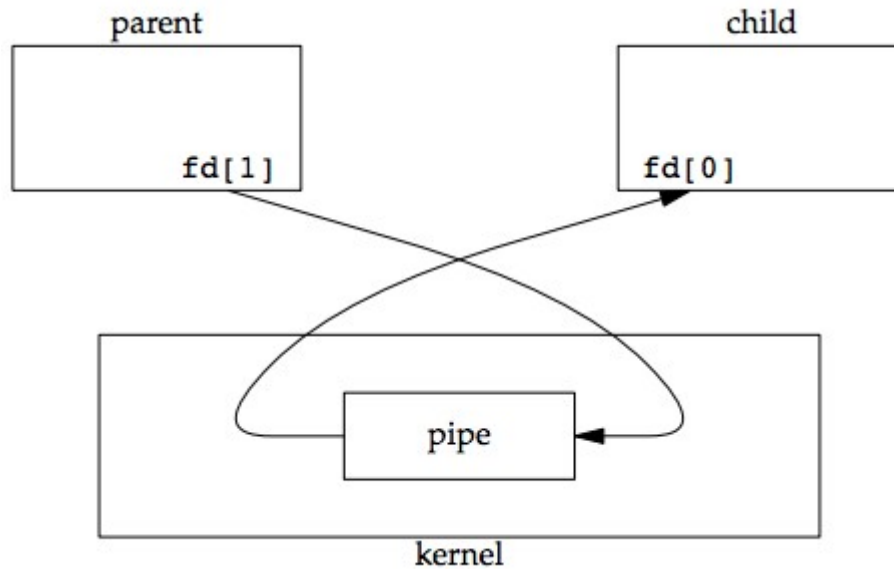
The `fstat` function returns a file type of FIFO for the file descriptor of either end of a pipe. We can test for a pipe with the `S_ISFIFO` macro.

POSIX.1 states that the `st_size` member of the `stat` structure is undefined for pipes. But when the `fstat` function is applied to the file descriptor for the read end of the pipe, many systems store in `st_size` the number of bytes available for reading in the pipe, which is nonportable.

A pipe in a single process is next to useless. Normally, the process that calls `pipe` then calls `fork`, creating an IPC channel from the parent to the child, or vice versa. The following figure shows this scenario:



What happens after the `fork` depends on which direction of data flow we want. For a pipe from the parent to the child, the parent closes the read end of the pipe (`fd[0]`), and the child closes the write end (`fd[1]`). The following figure shows the resulting arrangement of descriptors.



For a pipe from the child to the parent, the parent closes `fd[1]`, and the child closes `fd[0]`.

When one end of a pipe is closed, two rules apply:

- i. If we read from a pipe whose write end has been closed, read returns 0 to indicate an end of file after all the data has been read.
 - a. Technically, we should say that this end of file is not generated until there are no more writers for the pipe.
 - b. It's possible to duplicate a pipe descriptor so that multiple processes have the pipe open for writing.
 - c. Normally, there is a single reader and a single writer for a pipe. (The FIFOs in the next section dicusses that there are multiple writers for a single FIFO.)
- j. If we write to a pipe whose read end has been closed, the signal `SIGPIPE` is generated. If we either ignore the signal or catch it and return from the signal handler, write returns `-1` with `errno` set to `EPIPE`.

When we're writing to a pipe (or FIFO), the constant `PIPE_BUF` specifies the kernel's pipe buffer size. A write of `PIPE_BUF` bytes or less will not be interleaved with the writes from other processes to the same pipe (or FIFO). But if multiple processes are writing to a pipe (or FIFO), and if we write more than `PIPE_BUF` bytes, the data might be interleaved with the data from the other writers. We can determine the value of `PIPE_BUF` by using `pathconf` or `fpathconf`.

Example: creating a pipe between a parent and its child

`ipc1/pipe1.c`

```
#include "apue.h"
```



```

int
main(void)
{
    int    n;
    int    fd[2];
    pid_t  pid;
    char   line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("pipe error");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {    /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else {                /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}

```

Code:

```

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

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

    int fd[2];

    int childID = 0;

```

```

// create pipe descriptors
pipe(fd);

// fork() returns 0 for child process, child-pid for parent process.
if (fork() != 0) {
    // parent: writing only, so close read-descriptor.
    close(fd[0]);

    // send the childID on the write-descriptor.
    childID = 1;
    write(fd[1], &childID, sizeof(childID));
    printf("Parent(%d) send childID: %d\n", getpid(), childID);

    // close the write descriptor
    close(fd[1]);
} else {
    // child: reading only, so close the write-descriptor
    close(fd[1]);

    // now read the data (will block until it succeeds)
    read(fd[0], &childID, sizeof(childID));
    printf("Child(%d) received childID: %d\n", getpid(), childID);

    // close the read-descriptor
    close(fd[0]);
}
return 0;
}

```

Output:

```
sarita@hp:~/Desktop/UOS/13$ gedit A13_a.c
^C
sarita@hp:~/Desktop/UOS/13$ gcc A13_a.c
sarita@hp:~/Desktop/UOS/13$ ./a.out
Parent(23699) send childID: 1
Child(23700) received childID: 1
sarita@hp:~/Desktop/UOS/13$ ^C
sarita@hp:~/Desktop/UOS/13$
```

Conclusion:

The concept of pipe has been learned.

References:

2. <https://bytefreaks.net/programming-2/c-programming-2/cc-pass-value-from-parent-to-child-after-fork-via-a-pipe>
3. <https://notes.shichao.io/apue/ch15/>

STREAMS message/PIPEs/FIFO:pipe, popen and pclose Functions

Assignment No: 13_b

Title:

Filter to convert uppercase characters to lowercase.

Objectives:

d To learn about STREAMS message/PIPEs/FIFO:pipe, popen and pclose Functions

Theory:

Pipes

Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems. Pipes have two limitations:

4. Historically, they have been half duplex (data flows in only one direction). Some systems now provide full-duplex pipes, but for maximum portability, we should never assume that this is the case.
5. Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child.

FIFOs (Section 15.5) get around the second limitation, and that UNIX domain sockets (Section 17.2) get around both limitations.

Despite these limitations, half-duplex pipes are still the most commonly used form of IPC. Every time you type a sequence of commands in a pipeline for the shell to execute, the shell creates a separate process for each command and links the standard output of one process to the standard input of the next using a pipe.

A pipe is created by calling the `pipe` function.

```
#include <unistd.h>

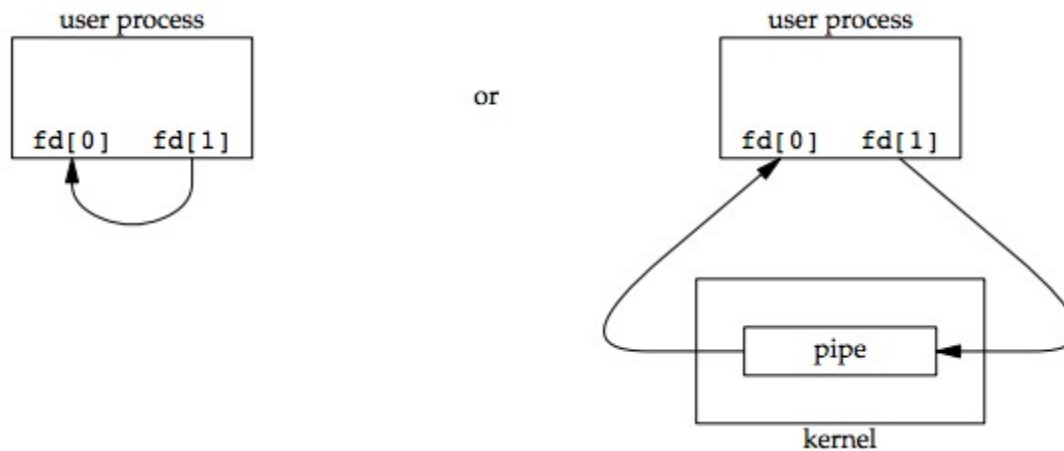
int pipe(int fd[2]);

/* Returns: 0 if OK, -1 on error */
```

Two file descriptors are returned through the *fd* argument: *fd[0]* is open for reading, and *fd[1]* is open for writing. The output of *fd[1]* is the input for *fd[0]*.

POSIX.1 allows for implementations to support full-duplex pipes. For these implementations, *fd[0]* and *fd[1]* are open for both reading and writing.

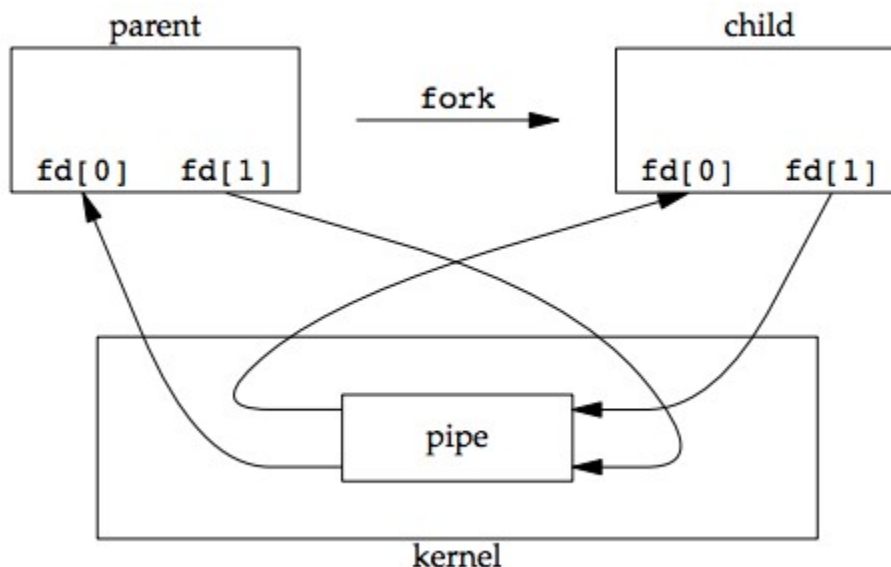
Two ways to picture a half-duplex pipe are shown in the figure below. The left half of the figure shows the two ends of the pipe connected in a single process. The right half of the figure emphasizes that the data in the pipe flows through the kernel.



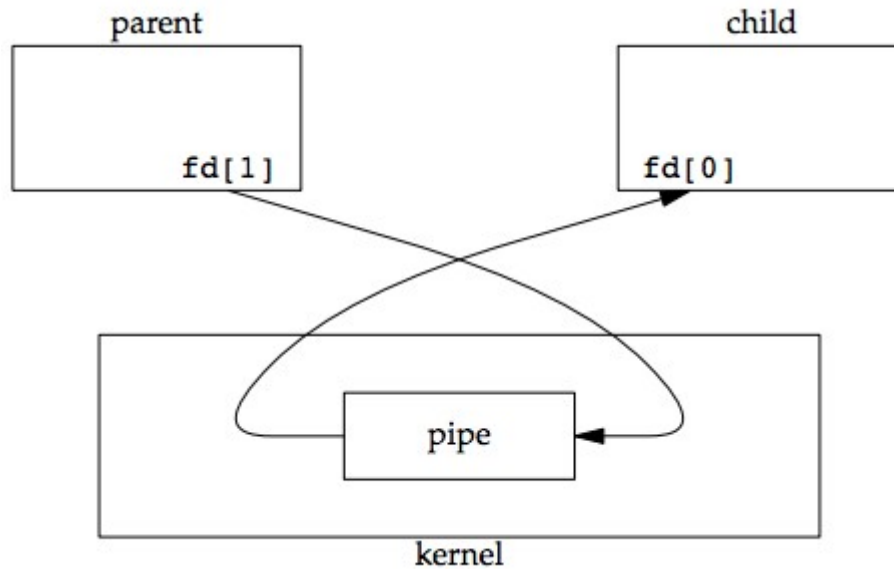
The `fstat` function returns a file type of FIFO for the file descriptor of either end of a pipe. We can test for a pipe with the `S_ISFIFO` macro.

POSIX.1 states that the `st_size` member of the `stat` structure is undefined for pipes. But when the `fstat` function is applied to the file descriptor for the read end of the pipe, many systems store in `st_size` the number of bytes available for reading in the pipe, which is nonportable.

A pipe in a single process is next to useless. Normally, the process that calls `pipe` then calls `fork`, creating an IPC channel from the parent to the child, or vice versa. The following figure shows this scenario:



What happens after the `fork` depends on which direction of data flow we want. For a pipe from the parent to the child, the parent closes the read end of the pipe (`fd[0]`), and the child closes the write end (`fd[1]`). The following figure shows the resulting arrangement of descriptors.



For a pipe from the child to the parent, the parent closes `fd[1]`, and the child closes `fd[0]`.

When one end of a pipe is closed, two rules apply:

3. If we read from a pipe whose write end has been closed, read returns 0 to indicate an end of file after all the data has been read.
0. Technically, we should say that this end of file is not generated until there are no more writers for the pipe.
 - a. It's possible to duplicate a pipe descriptor so that multiple processes have the pipe open for writing.
 - b. Normally, there is a single reader and a single writer for a pipe. (The FIFOs in the next section dicusses that there are multiple writers for a single FIFO.)
4. If we write to a pipe whose read end has been closed, the signal `SIGPIPE` is generated. If we either ignore the signal or catch it and return from the signal handler, write returns `-1` with `errno` set to `EPIPE`.

When we're writing to a pipe (or FIFO), the constant `PIPE_BUF` specifies the kernel's pipe buffer size. A write of `PIPE_BUF` bytes or less will not be interleaved with the writes from other processes to the same pipe (or FIFO). But if multiple processes are writing to a pipe (or FIFO), and if we write more than `PIPE_BUF` bytes, the data might be interleaved with the data from the other writers. We can determine the value of `PIPE_BUF` by using `pathconf` or `fpathconf`.

Example: creating a pipe between a parent and its child

`ipc1/pipe1.c`

```
#include "apue.h"
```

```

int
main(void)
{
    int    n;
    int    fd[2];
    pid_t  pid;
    char    line[MAXLINE];

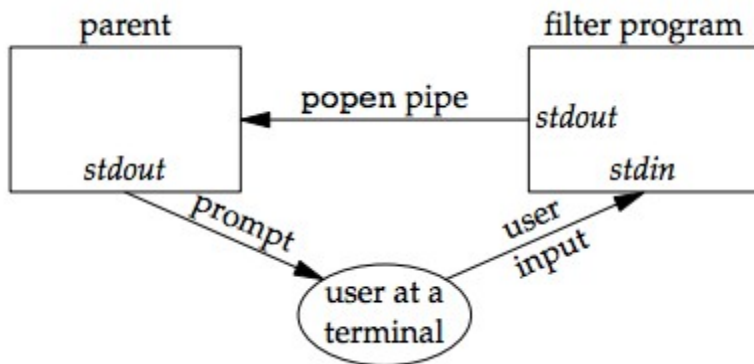
    if (pipe(fd) < 0)
        err_sys("pipe error");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {    /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else {                /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}

```

Example: transforming input using popen

One thing that `popen` is especially well suited for is executing simple filters to transform the input or output of the running command. Such is the case when a command wants to build its own pipeline.

Consider an application that writes a prompt to standard output and reads a line from standard input. With the `popen` function, we can interpose a program between the application and its input to transform the input. The following figure shows the arrangement of processes in this situation.



The following program is a simple filter to demonstrate this operation:

Code:

```

//file apue.h

/*
 * Our own header, to be included before all standard system headers.
 */
#ifndef _APUE_H
#define _APUE_H

#define _POSIX_C_SOURCE 200809L

#if defined(SOLARIS) /* Solaris 10 */
#define _XOPEN_SOURCE 600
#else
#define _XOPEN_SOURCE 700
#endif

#include <sys/types.h> /* some systems still require this */
#include <sys/stat.h>
#include <sys/termios.h> /* for winsize */
#if defined(MACOS) || !defined(TIOCGWINSZ)
#include <sys/ioctl.h>
#endif

#include <stdio.h> /* for convenience */
#include <stdlib.h> /* for convenience */
#include <stddef.h> /* for offsetof */
#include <string.h> /* for convenience */
#include <unistd.h> /* for convenience */
#include <signal.h> /* for SIG_ERR */

#define MAXLINE 4096 /* max line length */

/*
 * Default file access permissions for new files.
 */
#define FILE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

/*
 * Default permissions for new directories.
 */
#define DIR_MODE (FILE_MODE | S_IXUSR | S_IXGRP | S_IXOTH)

typedef void Sigfunc(int); /* for signal handlers */

#define min(a,b) ((a) < (b) ? (a) : (b))
#define max(a,b) ((a) > (b) ? (a) : (b))

```

```

/*
 * Prototypes for our own functions.
 */
char    *path_alloc(size_t *);          /* {Prog pathalloc} */
long    open_max(void);                 /* {Prog openmax} */

int      set_cloexec(int);               /* {Prog setfd} */
void     clr_fl(int, int);
void     set_fl(int, int);               /* {Prog setfl} */

void     pr_exit(int);                   /* {Prog prexit} */

void     pr_mask(const char *);           /* {Prog prmask} */
Sigfunc *signal_intr(int, Sigfunc *);    /* {Prog signal_intr_function} */

void     daemonize(const char *);         /* {Prog daemoninit} */

void     sleep_us(unsigned int);          /* {Ex sleepus} */
ssize_t  readn(int, void *, size_t);      /* {Prog readn_writen} */
ssize_t  writen(int, const void *, size_t); /* {Prog readn_writen} */

int      fd_pipe(int *);                  /* {Prog sock_fdpipe} */
int      recv_fd(int, ssize_t (*func)(int,
        const void *, size_t));          /* {Prog recvfd_sockets} */
int      send_fd(int, int);               /* {Prog sendfd_sockets} */
int      send_err(int, int,
        const char *);                   /* {Prog senderr} */
int      serv_listen(const char *);        /* {Prog servlisten_sockets} */
int      serv_accept(int, uid_t *);        /* {Prog servaccept_sockets} */
int      cli_conn(const char *);           /* {Prog cliconn_sockets} */
int      buf_args(char *, int (*func)(int,
        char **));                       /* {Prog bufargs} */

int      tty_cbreak(int);                  /* {Prog raw} */
int      tty_raw(int);                     /* {Prog raw} */
int      tty_reset(int);                   /* {Prog raw} */
void     tty_atexit(void);                 /* {Prog raw} */
struct termios *tty_termios(void);         /* {Prog raw} */

int      ptym_open(char *, int);           /* {Prog ptyopen} */
int      ptys_open(char *);               /* {Prog ptyopen} */
#ifdef TIOCGWINSZ
pid_t    pty_fork(int *, char *, int, const struct termios *,
        const struct winsize *);          /* {Prog ptyfork} */
#endif

int      lock_reg(int, int, int, off_t, int, off_t); /* {Prog lockreg} */

#define read_lock(fd, offset, whence, len) \
        lock_reg((fd), F_SETLK, F_RDLCK, (offset), (whence), (len))
#define readw_lock(fd, offset, whence, len) \
        lock_reg((fd), F_SETLKW, F_RDLCK, (offset), (whence), (len))
#define write_lock(fd, offset, whence, len) \
        lock_reg((fd), F_SETLK, F_WRLCK, (offset), (whence), (len))
#define writew_lock(fd, offset, whence, len) \
        lock_reg((fd), F_SETLKW, F_WRLCK, (offset), (whence), (len))
#define un_lock(fd, offset, whence, len) \
        lock_reg((fd), F_SETLK, F_UNLCK, (offset), (whence), (len))

pid_t    lock_test(int, int, off_t, int, off_t); /* {Prog locktest} */

#define is_read_lockable(fd, offset, whence, len) \
        (lock_test((fd), F_RDLCK, (offset), (whence), (len)) == 0)
#define is_write_lockable(fd, offset, whence, len) \
        (lock_test((fd), F_WRLCK, (offset), (whence), (len)) == 0)

void     err_msg(const char *, ...);        /* {App misc_source} */
void     err_dump(const char *, ...) __attribute__((noreturn));
void     err_quit(const char *, ...) __attribute__((noreturn));

```

```

void    err_cont(int, const char *, ...);
void    err_exit(int, const char *, ...) __attribute__((noreturn));
void    err_ret(const char *, ...);
void    err_sys(const char *, ...) __attribute__((noreturn));

void    log_msg(const char *, ...);           /* {App misc_source} */
void    log_open(const char *, int, int);
void    log_quit(const char *, ...) __attribute__((noreturn));
void    log_ret(const char *, ...);
void    log_sys(const char *, ...) __attribute__((noreturn));
void    log_exit(int, const char *, ...) __attribute__((noreturn));

void    TELL_WAIT(void);                     /* parent/child from {Sec race_conditions} */
void    TELL_PARENT(pid_t);
void    TELL_CHILD(pid_t);
void    WAIT_PARENT(void);
void    WAIT_CHILD(void);

#endif  /* _APUE_H */

```

//file myucl.c

```

#include "apue.h"
#include <ctype.h>

int
main(void)
{
    int    c;

    while ((c = getchar()) != EOF) {
        if (isupper(c))
            c = tolower(c);
        if (putchar(c) == EOF)
            err_sys("output error");
        if (c == '\n')
            fflush(stdout);
    }
    exit(0);
}

```

The filter copies standard input to standard output, converting any uppercase character to lowercase. The reason we're careful to `fflush` standard output after writing a newline is discussed in the next section when we talk about coprocesses.

We compile this filter into the executable file `myucl`, which we then invoke from the program in the following code using `popen`:

//file popen1.c

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

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

    if ((fpin = popen("myucl", "r")) == NULL)
        err_sys("popen error");
    for ( ; ; ) {
        fputs("prompt> ", stdout);
        fflush(stdout);
        if (fgets(line, MAXLINE, fpin) == NULL) /* read from pipe */
            break;
        if (fputs(line, stdout) == EOF)
            err_sys("fputs error to pipe");
    }
    if (pclose(fpin) == -1)
        err_sys("pclose error");
    putchar('\n');
    exit(0);
}
```

Conclusion:

The concept of pipe has been learned. File has converted from lowercase to upper case using filter.

References:

4. <https://notes.shichao.io/apue/ch15/>

STREAMS message/PIPEs/FIFO:pipe, popen and pclose Functions

Assignment No: 13c

Objectives:

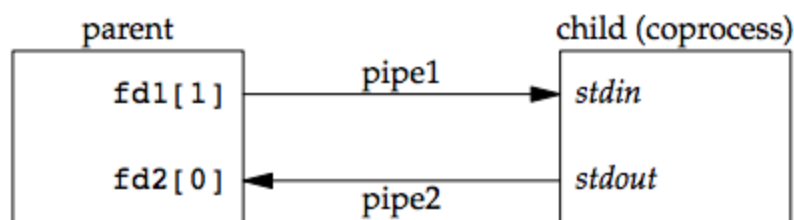
1.Simple filter to add two numbers. (B)

Theory:

A UNIX system filter is a program that reads from standard input and writes to standard output. Filters are normally connected linearly in shell pipelines. A filter becomes a coprocess when the same program generates the filter's input and reads the filter's output.

The Korn shell provides coprocesses. The Bourne shell, the Bourne-again shell, and the C shell don't provide a way to connect processes together as coprocesses. A coprocess normally runs in the background from a shell, and its standard input and standard output are connected to another program using a pipe. Although the shell syntax required to initiate a coprocess and connect its input and output to other processes is quite contorted, coprocesses are also useful from a C program.

Whereas popen gives us a one-way pipe to the standard input or from the standard output of another process, with a coprocess we have two one-way pipes to the other process: one to its standard input and one from its standard output. We want to write to its standard input, let it operate on the data, and then read from its standard output.



Example:

invoking add2 as a coprocess

For example, the process creates two pipes: one is the standard input of the coprocess and the other is the standard output of the coprocess. The figure below shows this arrangement:

Figure 15.16 Driving a coprocess by writing its standard input and reading its standard output

The following program is a simple coprocess that reads two numbers from its standard input, computes their sum, and writes the sum to its standard output.

Program:

```
#include "apue.h"
int
main(void)
{
    int    n, int1, int2;
    char   line[MAXLINE];

    while ((n = read(STDIN_FILENO, line, MAXLINE)) > 0) {
        line[n] = 0;        /* null terminate */
        if (sscanf(line, "%d%d", &int1, &int2) == 2) {
            sprintf(line, "%d\n", int1 + int2);
            n = strlen(line);
            if (write(STDOUT_FILENO, line, n) != n)
                err_sys("write error");
        } else {
            if (write(STDOUT_FILENO, "invalid args\n", 13) != 13)
                err_sys("write error");
        }
    }
    exit(0);
}
```

Input:

10 12

Output:

22

Conclusion:

In this way two numbers are added.

References:

1.[<https://notes.shichao.io/apue/ch15/>]

STREAMS message/PIPEs/FIFO:pipe, popen and pclose Functions

Assignment No: 13d

Objectives:

1. Invoke uppercase/lowercase filter to read commands. (I)

Theory:

One thing that popen is especially well suited for is executing simple filters to transform the input or output of the running command. Such is the case when a command wants to build its own pipeline.

Consider an application that writes a prompt to standard output and reads a line from standard input. With the popen function, we can interpose a program between the application and its input to transform the input. The following figure shows the arrangement of processes in this situation.

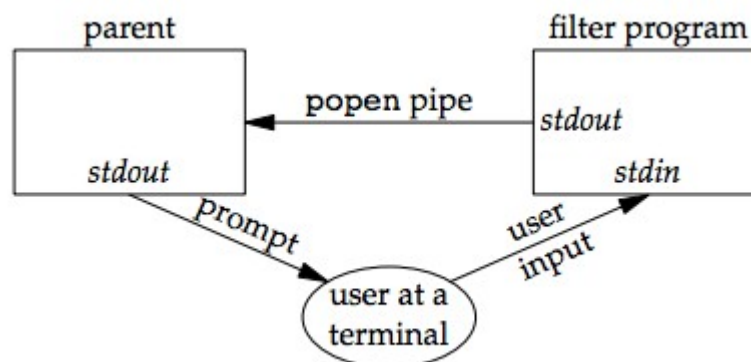


Figure 15.13

Transforming input using popen

The following program is a simple filter to demonstrate this operation:

The filter copies standard input to standard output, converting any uppercase character to lowercase. The reason we're careful to fflush standard output after writing a newline is discussed in the next section when we talk about coprocesses.

We compile this filter into the executable file `myucl`, which we then invoke from the program in the following code using `popen`:

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

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

    if ((fpin = popen("myucl", "r")) == NULL)
        err_sys("popen error");
    for ( ; ; ) {
        fputs("prompt> ", stdout);
        fflush(stdout);
        if (fgets(line, MAXLINE, fpin) == NULL) /* read from pipe */
            break;
        if (fputs(line, stdout) == EOF)
            err_sys("fputs error to pipe");
    }
    if (pclose(fpin) == -1)
        err_sys("pclose error");
    putchar('\n');
    exit(0);
}
```

We need to call `fflush` after writing the prompt, because the standard output is normally line buffered, and the prompt does not contain a newline.

Program:

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

int
main(void)
```

```

{
    int    c;

    while ((c = getchar()) != EOF) {
        if (isupper(c))
            c = tolower(c);
        if (putchar(c) == EOF)
            err_sys("output error");
        if (c == '\n')
            fflush(stdout);
    }
    exit(0);
}

```

Input:

INPut

Output:

input

Conclusion:

In this way lowercase text is formed.

References:

1.[<https://notes.shichao.io/apue/ch15/>]

13. e) STREAMS message/PIPEs/FIFO:pipe, popen and pclose Functions

Filter to add two numbers, using standard I/O.

Objectives:

1. To learn about STREAMS message/PIPEs/FIFO:pipe, popen and pclose Functions.

Theory:

Coprocesses-

A UNIX system filter is a program that reads from standard input and writes to standard output. Filters are normally connected linearly in shell pipelines. A filter becomes a *coprocess* when the same program generates the filter's input and reads the filter's output.

The Korn shell provides coprocesses. The Bourne shell, the Bourne-again shell, and the C shell don't provide a way to connect processes together as coprocesses. A coprocess normally runs in the background from a shell, and its standard input and standard output are connected to another program using a pipe. Although the shell syntax required to initiate a coprocess and connect its input and output to other processes is quite contorted, coprocesses are also useful from a C program.

Whereas popen gives us a one-way pipe to the standard input or from the standard output of another process, with a coprocess, we have two one-way pipes to the other process: one to its standard input and one from its standard output. We want to write to its standard input, let it operate on the data, and then read from its standard output.

Example

Let's look at coprocesses with an example. The process creates two pipes: one is the standard input of the coprocess, and the other is the standard output of the coprocess. shows this arrangement.

The program in [Figure 15.17](#) is a simple coprocess that reads two numbers from its standard input, computes their sum, and writes the sum to its standard output. (Coprocesses usually do more interesting work than we illustrate here. This example is admittedly contrived so that we can study the plumbing needed to connect the processes.)

We compile this program and leave the executable in the file add2.

The program in [Figure 15.18](#) invokes the add2 coprocess after reading two numbers from its standard input. The value from the coprocess is written to its standard output.

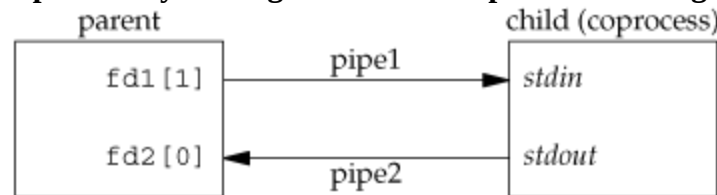
Here, we create two pipes, with the parent and the child closing the ends they don't need. We have to use two pipes: one for the standard input of the coprocess and one for its

standard output. The child then calls `dup2` to move the pipe descriptors onto its standard input and standard output, before calling `execl`.

If we compile and run the program in [Figure 15.18](#), it works as expected. Furthermore, if we kill the `add2` coprocess while the program in [Figure 15.18](#) is waiting for our input and then enter two numbers, the signal handler is invoked when the program writes to the pipe that has no reader.

Recall from [Figure 15.1](#) that not all systems provide full-duplex pipes using the `pipe` function. In [Figure 17.4](#), we provide another version of this example using a single full-duplex pipe instead of two half-duplex pipes, for those systems that support full-duplex pipes.

Figure Driving a coprocess by writing its standard input and reading its standard output



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

```
int main()
{
    int status = 0;
    int pfd1[2];
    int pfd2[2];
    int pfd3[2];
    int pfd4[2];
    int val = 0;
    int val2 = 0;
    pipe(pfd1);
    pipe(pfd2);
    pipe(pfd3);
    pipe(pfd4);
```

```
printf("Enter FIRST Number: ");
scanf("%i", &val);
printf("Enter SECOND Number: ");
scanf("%i", &val2);
if (fork() == 0)
{
//Child
    printf("I'm Child 1 and I'm Calculating Sum Of Numbers\n");
    read(pfds[0], &val, sizeof(val));
    read(pfds2[0], &val2, sizeof(val));
    int sum = val + val2;
```

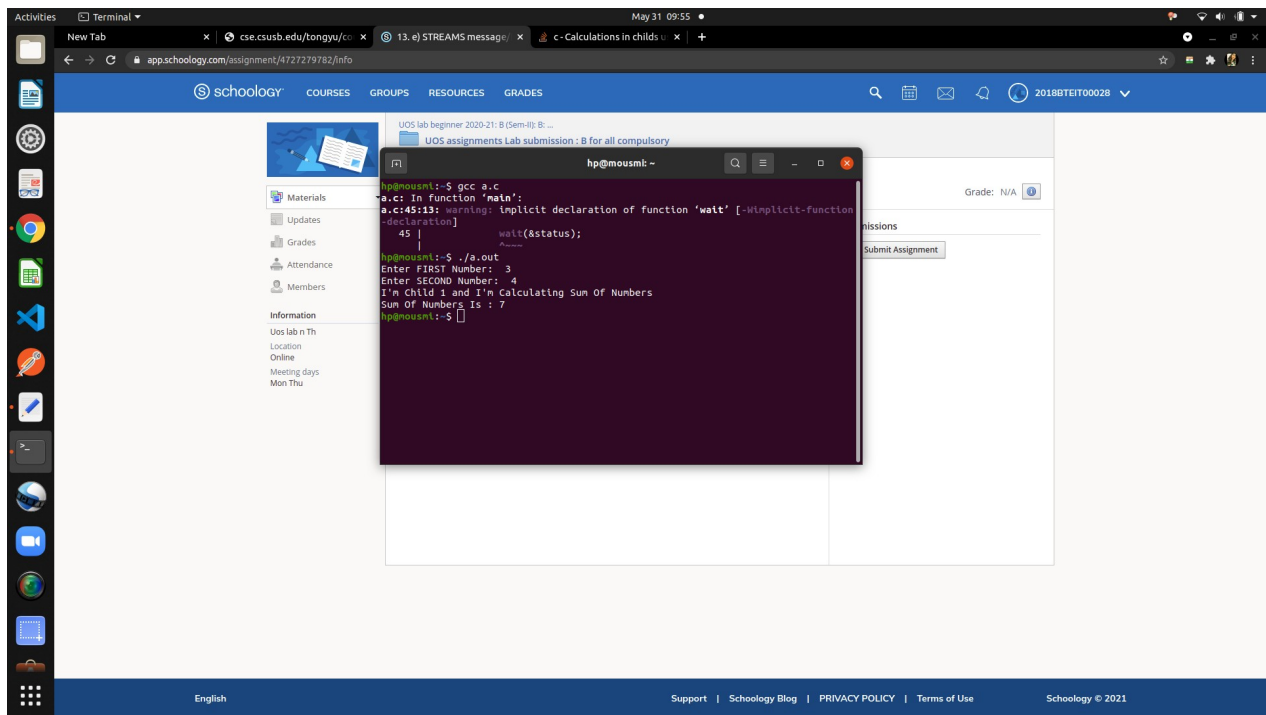
```

int sub = val - val2;
write(pfds3[1], &sum, sizeof(val));
write(pfds4[1], &sub, sizeof(val));
if (fork() == 0)
{
    read(pfds3[0], &val, sizeof(val));
    printf("Sum Of Numbers Is : %i\n", val);
    read(pfds4[0], &val, sizeof(val));

    exit(1);
}
else
{
    wait(&status);
}
exit(1);
}
else
{

//wait(&status);
    write(pfds[1], &val, sizeof(val));
    write(pfds2[1], &val2, sizeof(val));
    wait(&status);
}
}

```

Conclusion:

Filter to add two numbers, using standard I/O using STREAMS message/PIPEs/FIFO:pipe, popenand pcloseFunctions implemented

References :

<http://cse.csusb.edu/tongyu/courses/cs460/labs/lab4.php>

STREAMS message/PIPEs/FIFO:pipe, popenand pcloseFunctions

Assignment No: 13_f

Title:

Client–Server Communication Using a FIFO.

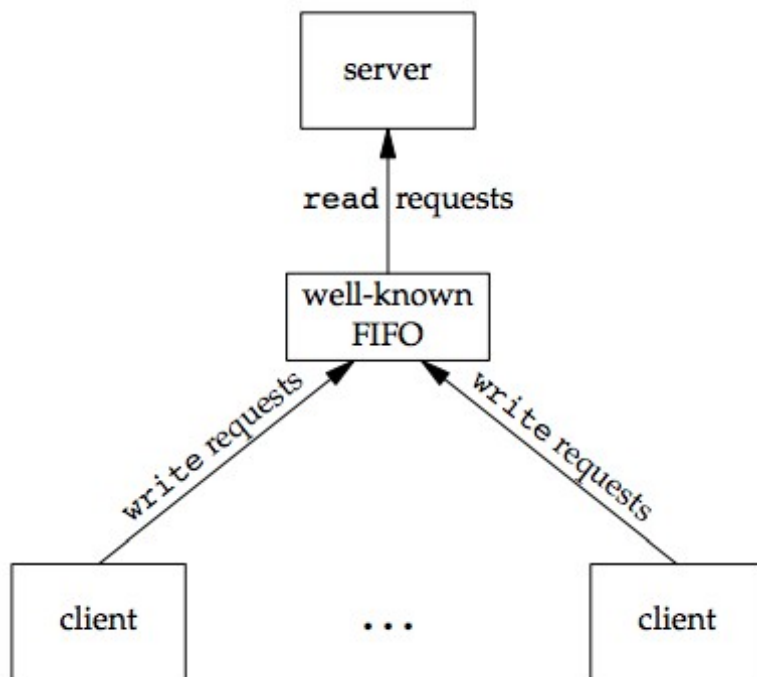
Objectives:

6. To learn about STREAMS message/PIPEs/FIFO:pipe, popenand pcloseFunctions.

Theory:

Client–Server Communication Using a FIFO-

Another use for FIFOs is to send data between a client and a server. If we have a server that is contacted by numerous clients, each client can write its request to a well-known FIFO that the server creates. "well-known" means that the pathname of the FIFO is known to all the clients that need to contact the server.

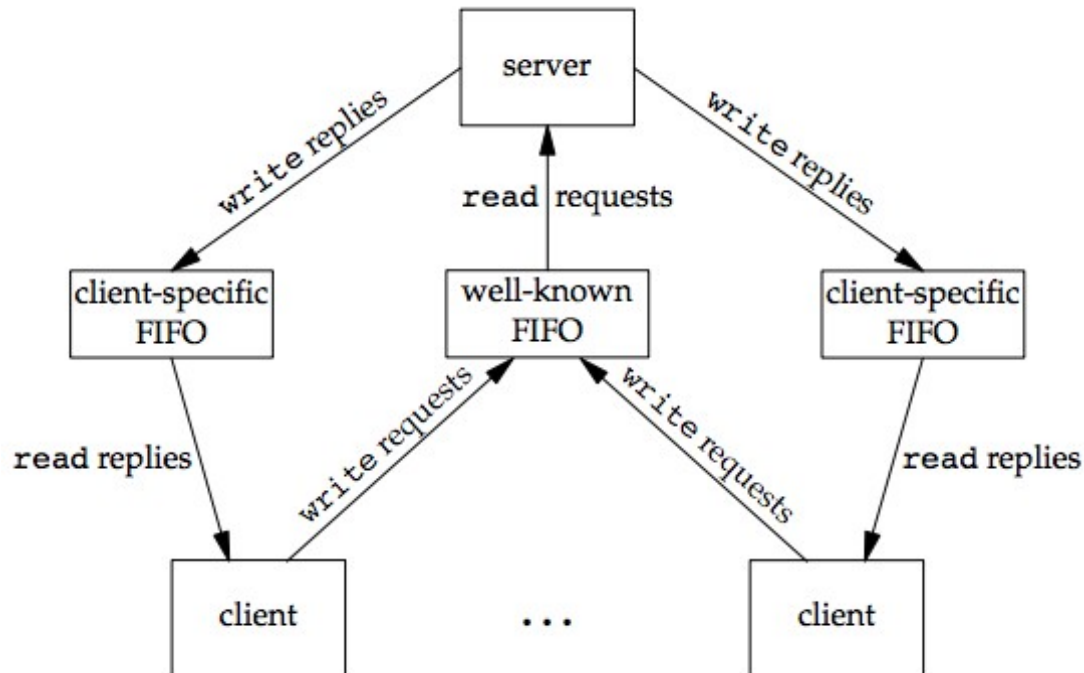


Since there are multiple writers for the FIFO, the requests sent by the clients to the server need to be less than PIPE_BUF bytes in size. This prevents any interleaving of the client writes.

The problem in using FIFOs for this type of client-server communication is how to send replies back from the server to each client. A single FIFO can't be used, as the clients would never know when to read their response versus responses for other clients. One solution is for each client to send its process ID with the request. The server then creates a unique FIFO for each client, using a pathname based on the client's process ID. For example, the server can create a

FIFO with the name `/tmp/serv1.XXXXX`, where `XXXXX` is replaced with the client's process ID. This arrangement is shown the figure below.

This arrangement works, although it is impossible for the server to tell whether a client crashes. A client crash leaves the client-specific FIFO in the file system. The server also must catch `SIGPIPE`, since it's possible for a client to send a request and terminate before reading the response, leaving the client-specific FIFO with one writer (the server) and no reader.



With the arrangement shown in the figure above, if the server opens its well-known FIFO read-only (since it only reads from it) each time the number of clients goes from 1 to 0, the server will read an end of file on the FIFO. To prevent the server from having to handle this case, a common trick is just to have the server open its well-known FIFO for read-write.

The following is a client/server application that makes use FIFOs to communicate. The server program **server.cpp** creates and opens the server pipe which is set to read-only, with blocking. After sleeping (for demonstration purposes), the server reads in any data sent by a client, which has the **data_to_pass_st** structure. In the next stage, it performs some processing on the data just read from the client, converting all characters recieved to uppercase and combine the **CLIENT_FIFO_NAME** with the received **client_pid**. Finally, it sends the data back, opening the client pipe in write-only, blocking mode, and then shut down the FIFO server by closing the file and unlinking the FIFO.

Code:

```
//server.cpp
#include <ctype.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>

#define SERVER_FIFO_NAME "/tmp/serv_fifo"
#define CLIENT_FIFO_NAME "/tmp/client_fifo"

#define BUFFER_SIZE 20

struct data_to_pass_st {
    pid_t  client_pid;
    char   some_data[BUFFER_SIZE - 1];
};

int main()
{
    int server_fifo_fd, client_fifo_fd;
    struct data_to_pass_st my_data;
    int read_res;
    char client_fifo[256];
    char *tmp_char_ptr;

    mkfifo(SERVER_FIFO_NAME, 0777);
    server_fifo_fd = open(SERVER_FIFO_NAME, O_RDONLY);
    if (server_fifo_fd == -1) {
        fprintf(stderr, "Server fifo failure\n");
        exit(EXIT_FAILURE);
    }

    sleep(10); /* lets clients queue for demo purposes */

    do {
        read_res = read(server_fifo_fd, &my_data, sizeof(my_data));
        if (read_res > 0) {

            // In this next stage, we perform some processing on the data just read from the cli
            // We convert all the characters in some_data to uppercase and combine the CLIENT_FI
            // with the received client_pid.

            tmp_char_ptr = my_data.some_data;
            while (*tmp_char_ptr) {
                *tmp_char_ptr = toupper(*tmp_char_ptr);
                tmp_char_ptr++;
            }
            sprintf(client_fifo, CLIENT_FIFO_NAME, my_data.client_pid);
```

```

// Then we send the processed data back, opening the client pipe in write-only,
// Finally, we shut down the server FIFO by closing the file and then unlinking the

        client_fifo_fd = open(client_fifo, O_WRONLY);
        if (client_fifo_fd != -1) {
            write(client_fifo_fd, &my_data, sizeof(my_data));
            close(client_fifo_fd);
        }
    }
} while (read_res > 0);
close(server_fifo_fd);
unlink(SERVER_FIFO_NAME);
exit(EXIT_SUCCESS);
}

```

The client program **client.cpp** opens the server FIFO, if it already exists, as a file. It then gets its own process ID, which forms some of the data that will be sent to the server. The client FIFO is also created and opened (read-only, blocking mode for reading back data.

```

//client.cpp
#include <ctype.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>

#define SERVER_FIFO_NAME "/tmp/serv_fifo"
#define CLIENT_FIFO_NAME "/tmp/client_fifo"

#define BUFFER_SIZE 20
struct data_to_pass_st {
    pid_t client_pid;
    char some_data[BUFFER_SIZE - 1];
};

int main()
{
    int server_fifo_fd, client_fifo_fd;
    struct data_to_pass_st my_data;
    int times_to_send;
    char client_fifo[256];

    server_fifo_fd = open(SERVER_FIFO_NAME, O_WRONLY);
    if (server_fifo_fd == -1) {
        fprintf(stderr, "Sorry, no server\n");
        exit(EXIT_FAILURE);
    }

    my_data.client_pid = getpid();
    //sprintf(client_fifo, CLIENT_FIFO_NAME, my_data.client_pid);
    sprintf(client_fifo, CLIENT_FIFO_NAME );
}

```

```

    if (mkfifo(client_fifo, 0777) == -1) {
        fprintf(stderr, "Sorry, can't make %s\n", client_fifo);
        exit(EXIT_FAILURE);
    }

    // For each of the five loops, the client data is sent to the server.
    // Then the client FIFO is opened (read-only, blocking mode) and the data read b ack
    // Finally, the server FIFO is closed and the client FIFO removed from memory.

    for (times_to_send = 0; times_to_send < 5; times_to_send++) {
        sprintf(my_data.some_data, "Hello from %d", my_data.client_pid);
        printf("%d sent %s, ", my_data.client_pid, my_data.some_data);
        write(server_fifo_fd, &my_data, sizeof(my_data));
        client_fifo_fd = open(client_fifo, O_RDONLY);
        if (client_fifo_fd != -1) {
            if (read(client_fifo_fd, &my_data, sizeof(my_data)) > 0) {
                printf("received: %s\n", my_data.some_data);
            }
            close(client_fifo_fd);
        }
    }
    close(server_fifo_fd);
    unlink(client_fifo);
    exit(EXIT_SUCCESS);
}

```

Compile with the commands:

```

$ g++ -o server server.cpp
$ g++ -o client client.cpp

```

To test this, run *server* in one X-Term. Open another X-term and run *client*. You should see something like the following:

Output:

```

5398 sent Hello from 5398, received: HELLO from 5398
5398 sent Hello from 5398, received: HELLO from 5398
5398 sent Hello from 5398, received: HELLO from 5398
5398 sent Hello from 5398, received: HELLO from 5398
5398 sent Hello from 5398, received: HELLO from 5398

```

Conclusion:

The concept of client-server communication using FIFO has been learned.

References:

6. <http://cse.csusb.edu/tongyu/courses/cs460/labs/lab4.php>
7. <https://notes.shichao.io/apue/ch15/>

13. g) STREAMS message/PIPEs/FIFO:pipe, popen and pclose Functions

Routines to let a parent and child synchronize. .

Objectives:

1. To learn about STREAMS message/PIPEs/FIFO:pipe, popen and pclose Functions.

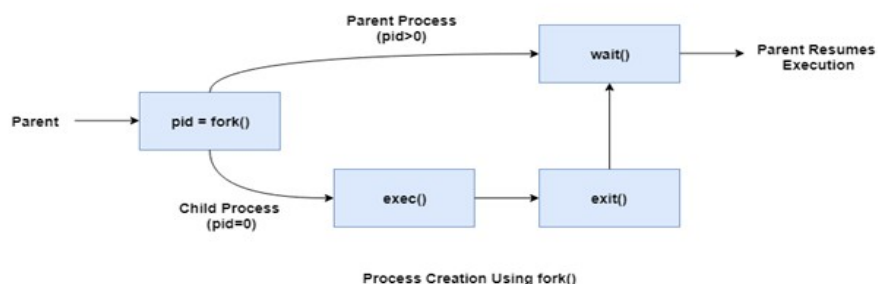
Theory:

Process synchronization in Linux involves providing a time slice for each process so that they get the required time for execution.

The process can be created using the fork() command in Linux. The creating process is called the parent process and the created process is the child process. A child process can have only one parent but a parent process may have many children. Both the parent and child processes have the same memory image, open files and environment strings. However, they have distinct address spaces.

If any process has more than one child processes, then after calling wait(), parent process has to be in wait state if no child terminates. If only one child process is terminated, then return a wait() returns process ID of the terminated child process. If more than one child processes are terminated than wait() reap any *arbitrarily child* and return a process ID of that child process. When wait() returns they also define exit status (which tells our, a process why terminated) via pointer, If status are not NULL. If any process has no child process then wait() returns immediately “-1”.

Figure



```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
static int pipefd1[2],pipefd2[2];
/*Create two pipes*/
void TELL_WAIT(void){
    if(pipe(pipefd1)<0 || pipe(pipefd2)<0){
        perror("pipe");
        exit(EXIT_FAILURE);
    }
}

void TELL_PARENT(void){
    /* send parent a message through pipe, need you to complete*/
    printf("Child send message to parent!\n");
}

void WAIT_PARENT(void){
    /* read message sent by parent from pipe, need you to complete*/
    printf("Child receive message from parent!\n");
}

void TELL_CHILD(void){
    /* send child a message through pipe, need you to complete*/
    printf("Parent send message to child!\n");
}

void WAIT_CHILD(void){
    /* read the message sent by child from pipe, need you to complete*/
    printf("Parent receive message from child!\n");
}

int main(int argc, char* argv[]){
    TELL_WAIT();
    pid_t pid;
    pid = fork();
    //set a timer, process will end after 1 second.
    alarm(10);
    if(pid==0){
        while(1){
            sleep(rand()%2+1);
            TELL_CHILD();
            WAIT_CHILD();
        }
    }else{

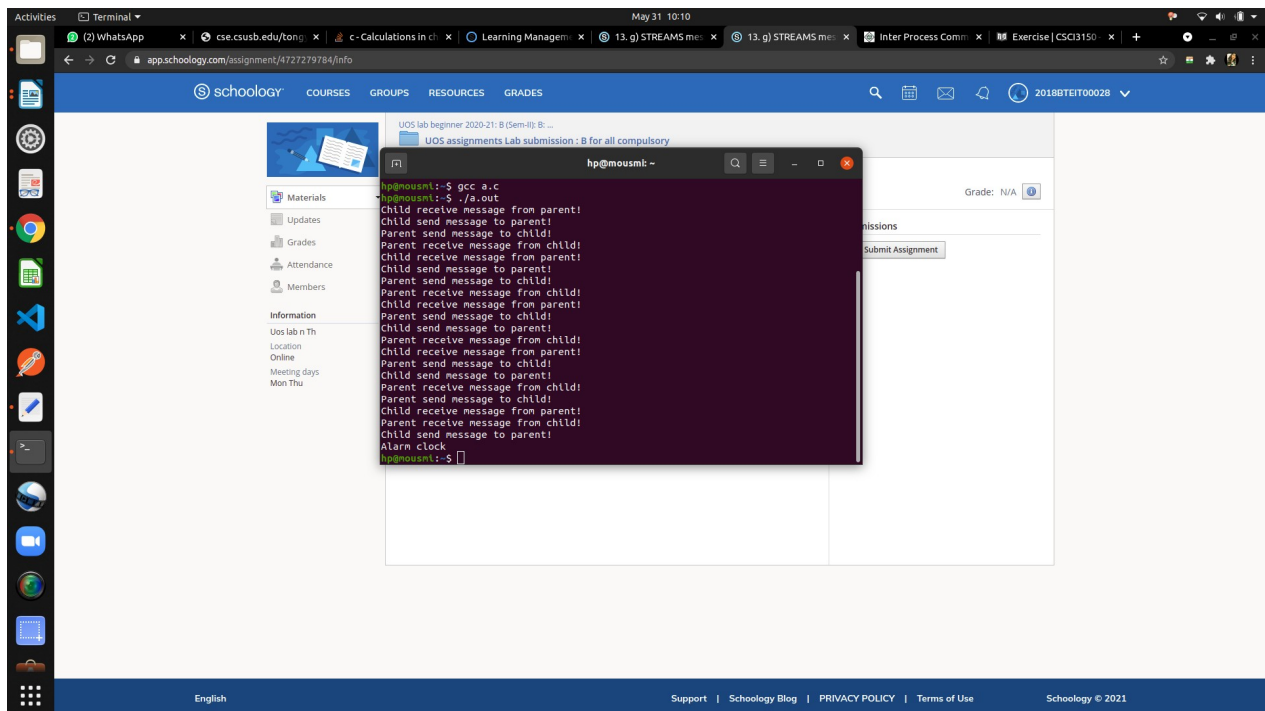
```



```

while(1){
    sleep(rand()%2+1);
    WAIT_PARENT();
    TELL_PARENT();
}
}
return 0;
}

```



Conclusion:

Routines to let a parent and child synchronize implemented

References :

<http://www.cse.cuhk.edu.hk/~ericlo/teaching/os/lab/6-IPC1/exercise-1.html>