WALCHAND COLLEGE OF ENGINEERING, SANGLI

Department of Information Technology
UNIX OPERATING SYSTEM LAB(IT 371)
Academic Year: 2023-24
Term: Semester-2

Class: TY - IT Batch : T4

Name: Aaditya Ravaso Khot
PRN.No: 21610051

Certificate

This is to certify that

Mr. Aaditya Ravaso Khot (21610051)

Of

TY B. Tech (IT) class has completed experiments satisfactorily in UNIX OPERATING SYSTEM LAB (3IT371) during the

Year 2023-24

Dr. A.J. Umbarkar
COURSE TEACHER

Dr.R.R.Rathod
HOD

UNIX Operating System Lab. IT 371

Course Objectives:

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

Percentage of objective achieved by students

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

Please tick the 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 to write effective programs.
f. Learn and understand the OS interaction with socket programming.
g. Learn about python as a scripting option.
h. Learn about openMP for better use of multicore systems.

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
Aaditya Ravaso Khot

Roll No
21610051

Assignment list

Note:

1. All programs on Linux using either in C/JAVA/Shell programming/python.
2. Write up should stick to program objective (title, subtitle, theory, application system call, system call used, variable/data dictionary flowcharts and conclusions). Sample template is given at the end of the assignment list. Use appropriate tool like BAKOMA, Lyx, winedit etc. (Lyx Template will be available on ftp/moodle)
3. Write up should stick to program objective (theory, system call used, variable dictionary and application system call)(soft copy submission allowed in latex only)
4. Submit any two from a, b, c… if you solve all, bonus mark will be given.
5. Assignment 10, 11 and 12 are optional.

Sr. no.	Assignments	Page No.
1.	**Processing Environment:** fork, vfork, wait, waitpid(),exec (all variations exec), and exit, 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. 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.html e. 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 OR	12 15 18 24 28

	System call fork/vfork search	
2.	**IPC**: Interrupts and Signals: signal(any fives 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 (say10 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 to process opened in two terminal using kill() and signal().(I) d. Write aapplication 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	32 35 41 43 45 49
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)	52 56 60

	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. <u>File locking system call</u> :** fnctl.h: flock/lockf **<u>(Optional)</u>** **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 fnctl 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.	64 67 69
4.	**<u>Thread concept</u>**: 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. Write a multithreaded program in java/c for chatting (multiuser and multi-terminal) using threads. (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 –	71 78 85

	monitor/serialize/semaphore of Java (In java only) (I) Write a program in Linux to use clone system call and show how it is different from fork system call. (I) 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 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 Write program to synchronize threads using construct – monitor/serialize/semaphore of Java (In java only) (E) 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	92 95 102 106 113
5.	**Shell programming**: shell scripts Objectives: 1.To learn shell programming and use it for write effective programs. Write a program to implement a shell script for calculator (B) Write a program to implement a digital clock using shell script. (B) Using shell sort the given 10 number in ascending order (use of array). (B) Shell script to print "Hello World" message, in Bold, Blink effect, and in different colors like red, brown etc. (B) Shell script to determine whether given file exists or not. (I) Ref: ftp://10.1013.3/pub/UOS..../ Import python script in shell script. (E)	119 123 126 128 130 132
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	136

	communicate via semaphore. (B) Ref:http://www.cs.cf.ac.uk/Dave/C/node26.html#SECTION002600000000000000000 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 fromsem.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	142 145
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	154 161 165
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	175 180 189
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/iteraitve server to conntect multiple clients requests handled through concurrent/iterative logic	194 198

	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 chating 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 OR System call search	205 212 217 220
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 behard-coded. It should be obtained from the user. The pyramid should appear as close to the centreof the screen as possible. (B) 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	225 227 229 232

	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)	234 236 239 242
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	 247 249
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	 251

	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_Operating_System/OpenMP..... OR Search on internet	253
13.	**STREAMS message/PIPEs/FIFO**:pipe, popenand pcloseFunctions (Optional) 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) Ref: advanced network programming- Stevens .pdf g. Routines to let a parent and child synchronize. (E)	

Assignment 1A : Processing Environment

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.

Objectives:

1. To learn about the 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?

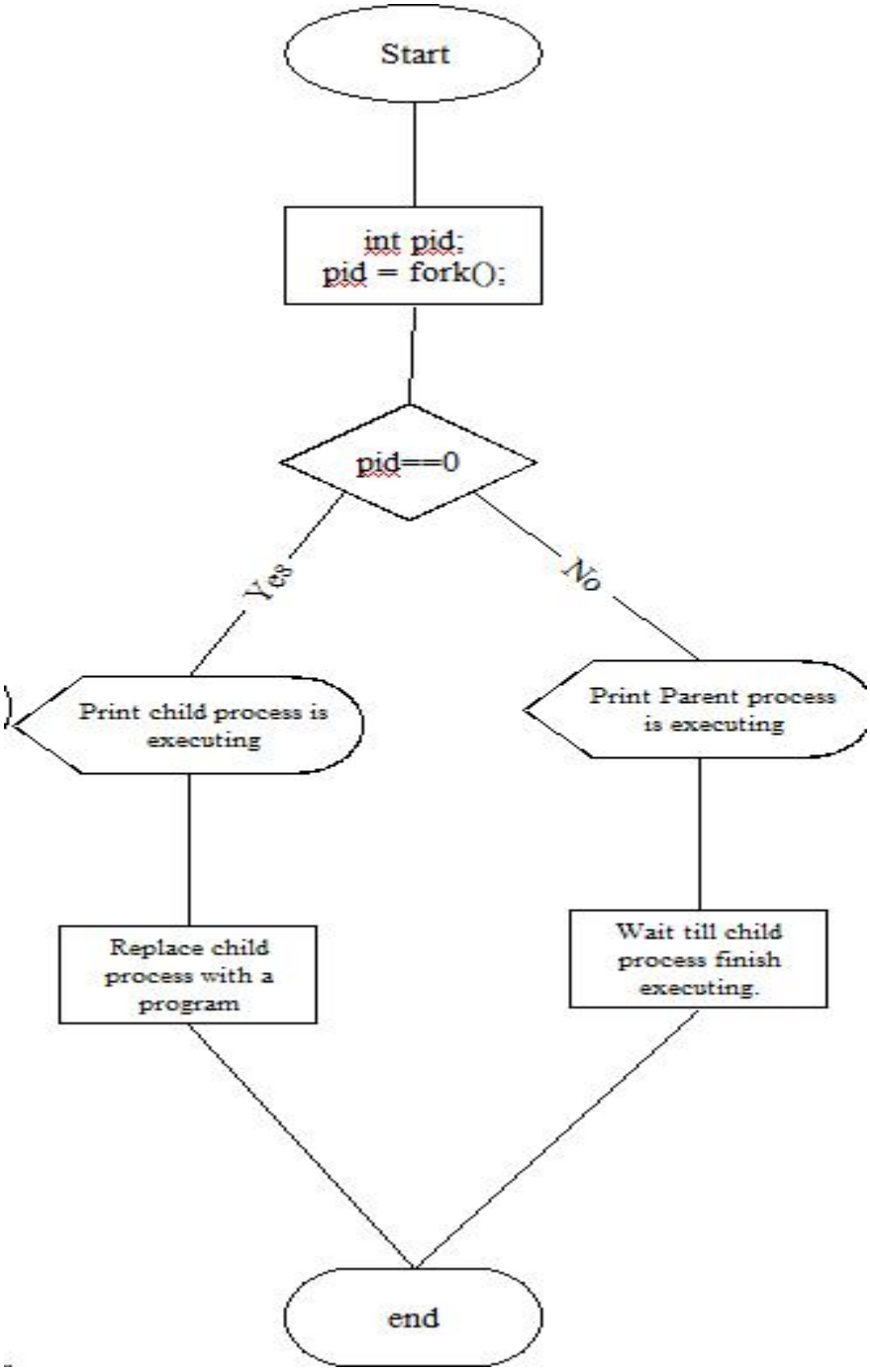
A new process is created because an existing process makes an exact copy of itself. This child process has the same environment as its parent, only the process ID number is different. This procedure is called *forking*.

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**. 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 applicatios.

Flowchart:



Flowchart 1.a.1

Data Dictionary:

SR+	Variable/Function	Datatype	Use
1	Counter	int	Used to increment number of child and parent processes.

2	pid	int	Process ID
3	Fork()		System call to create a new process (child process) by duplicating the calling

Fig:1.1 Data Dictionary

Program:

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

int main() {
   pid_t pid;

   // Fork a child process
   pid = fork();

   if (pid < 0) {
      // Fork failed
      perror("Fork failed");
      exit(EXIT_FAILURE);
   } else if (pid == 0) {
      // Child process
      printf("Child process is executing...\n");

      // Use execvp to replace the child process with a new program
      char *args[] = {"ls", "-l", NULL};
      execvp(args[0], args);

      // If execvp returns, it means an error occurred
      perror("Execvp failed");
      exit(EXIT_FAILURE);
   } else {
      // Parent process
      printf("Parent process is waiting for the child...\n");

      // Wait for the child process to finish
      wait(NULL);
      printf("Child process finished execution.\n");
   }
```

```
    return 0;
}
```

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

Assignment 1B

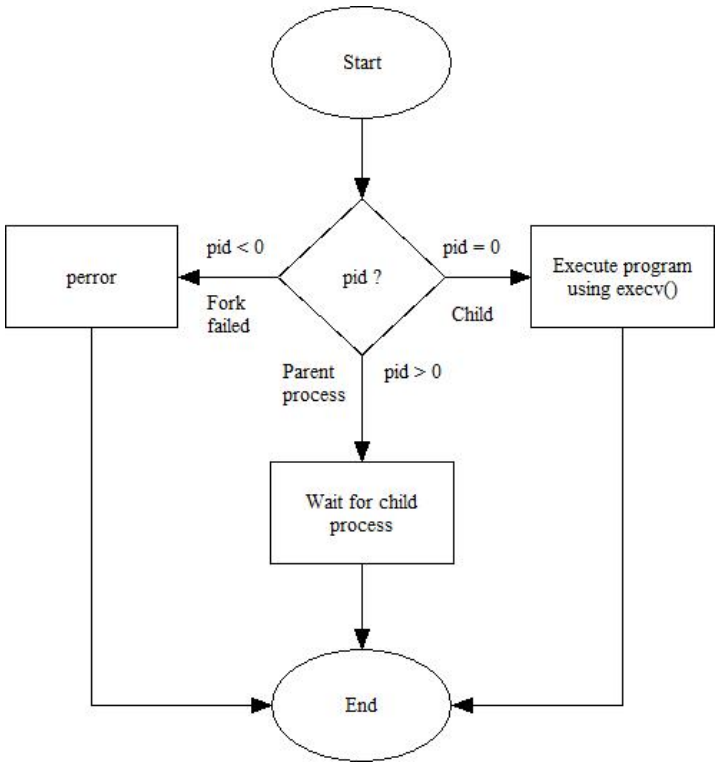
Title:

Write the application or program to create Childs and assign the task to them by variation exec system calls. (B)

Objectives:

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

Theory:



Code:

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

int main(){
    char *args[] = {"./hello", NULL};
   pid_t pid = fork();
   if(pid < 0){
      perror("Fork failed!\n");
   }
   if(pid==0){
      printf("PID of child process = %d\n", getpid());
      execv(args[0], args);
   } else {
      wait(NULL);
      printf("PID of parent process = %d\n", getpid());
      printf("Child has finished execution\n");
   }
   return 0;
}
```

hello.c:

```
#include<stdlib.h>

#include<stdio.h>

#include<unistd.h>

int main(){

   printf("I am in hello.c\n");

   printf("PID of hello.c is %d\n", getpid());

   return 0;

}
```

Output:

PID of child process = 8077

I am in hello.c

PID of hello.c is 8077

PID of parent process = 8076

Child has finished execution

Conclusion:

We can observe that the pid of hello.c (program to execute) is the same as that of the child process - the child process itself is being replaced by hello.c. Different versions of exec like execv() can be used to assign tasks like running another program.

Assignment 1C

1.C 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:

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 1.3 fork and vfork

Flowchart:

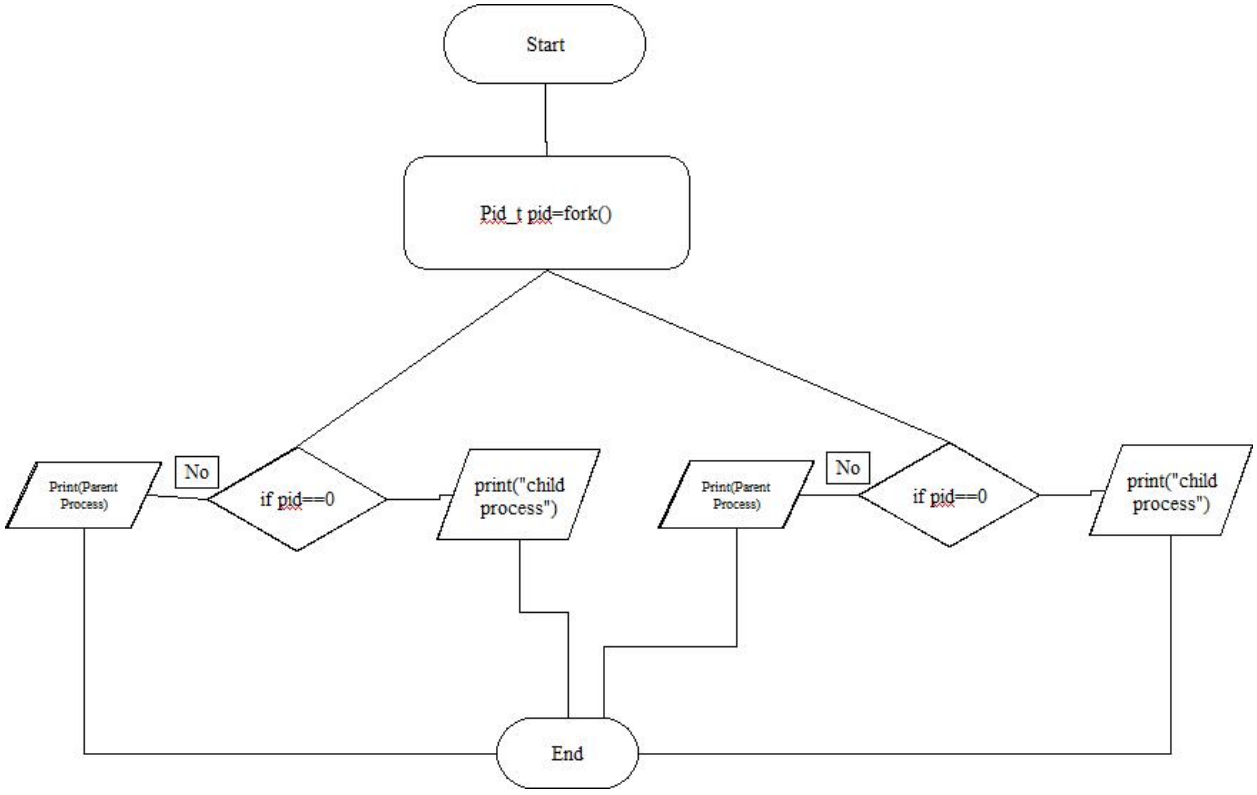
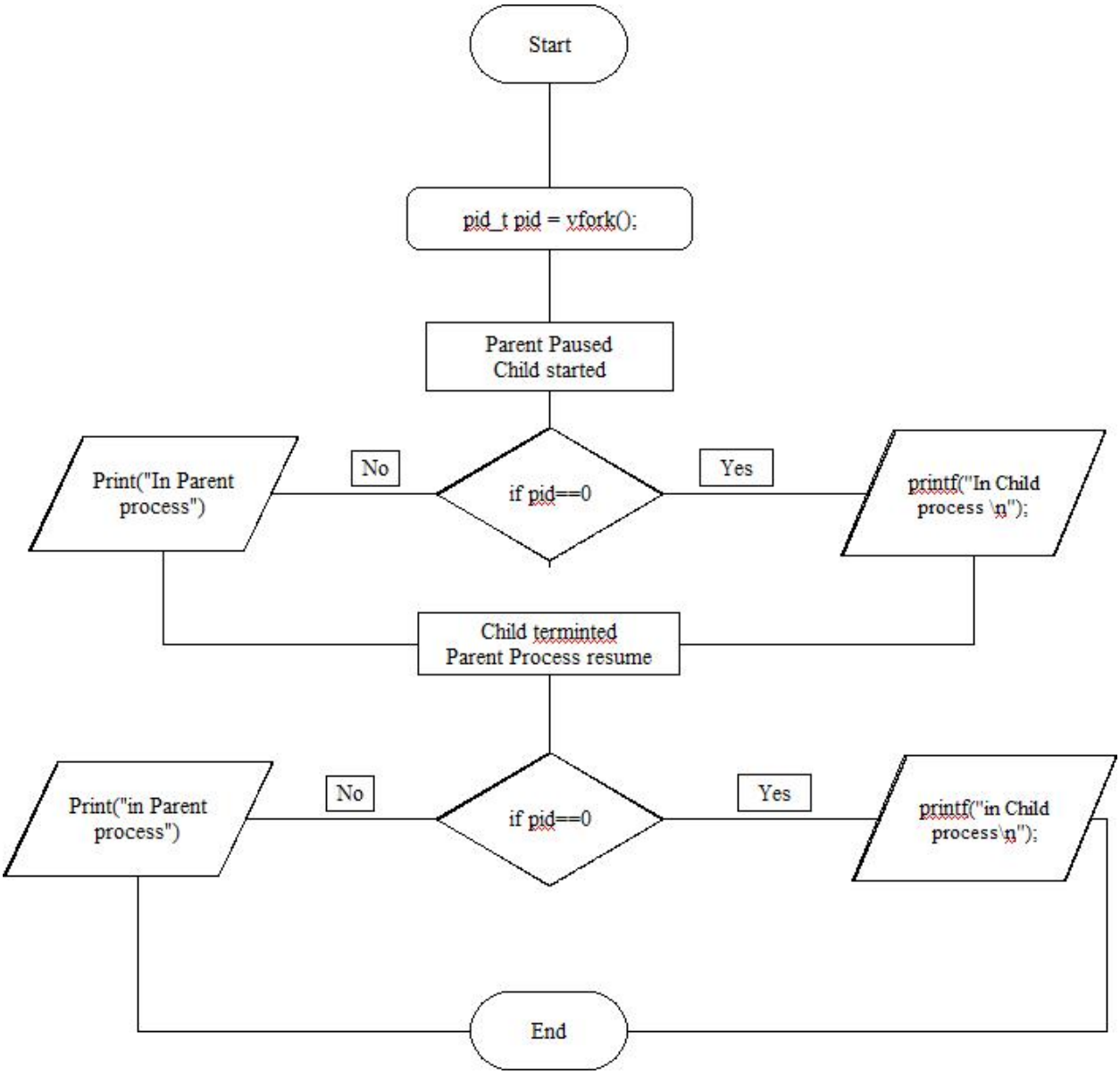

Start
Pid_t pid=fork()
Print(Parent Process)
No
if pid==0
print("child process")
Print(Parent Process)
No
if pid==0
print("child process")
End

Fig 1.4:Fork



1.5. Vfork

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

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 Started 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:

1. fork() and vfork() system calls have some differences which allows different type of execution of child processes.

2. learn wait and waitpid system calls.

References:

[1] www.tutorialspoint.com/unix_system_calls/

Assignment 1D

Title: Write the program to use wait/ waitpid system call and explain what it does when call in parent and called in child (). Justify the difference by using suitable application.

Objectives:

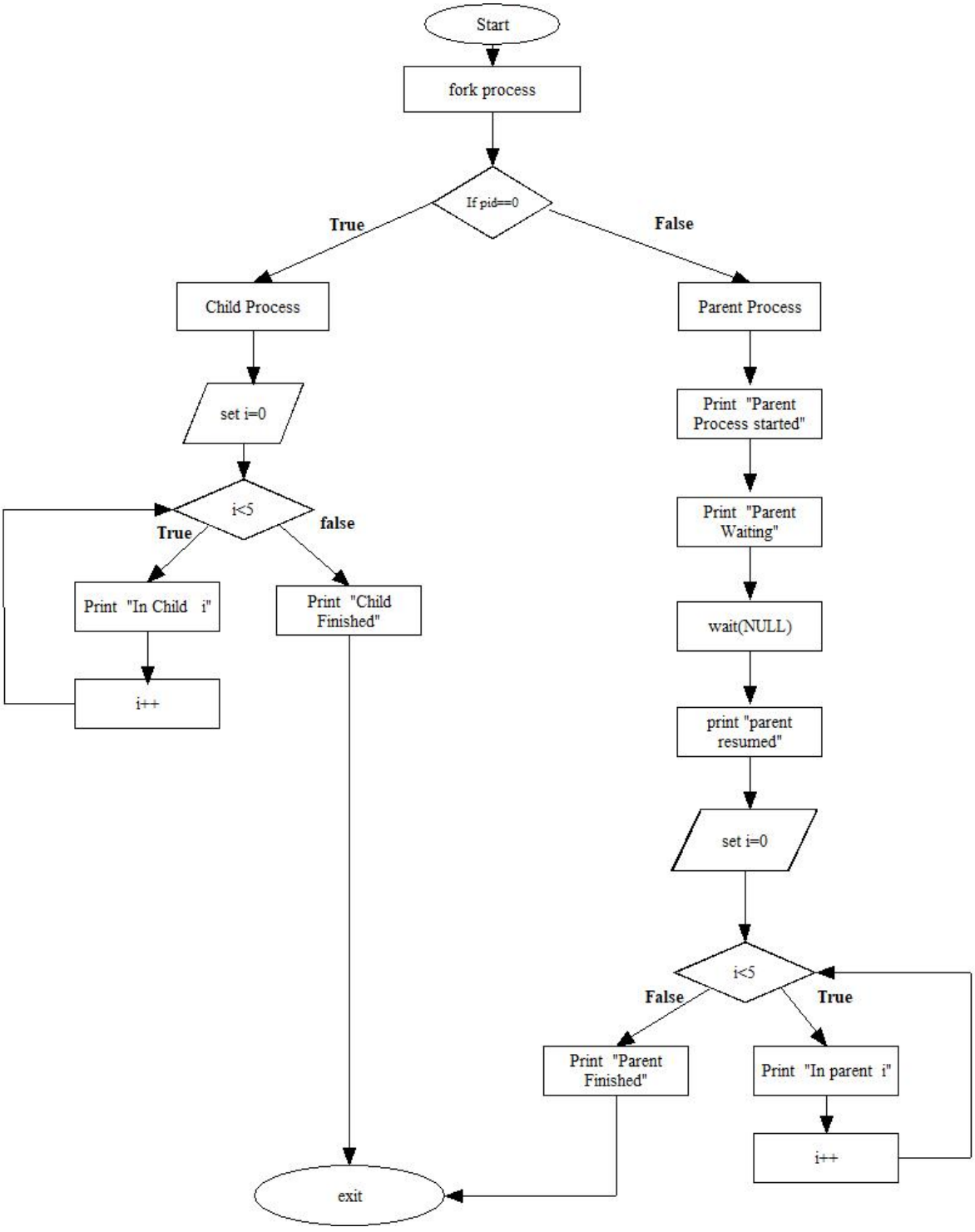
1. To learn about Processing Environment.
2. To know the difference between fork/vfork and various execs variations.
3. Use of system call to write effective programs.

Theory:

Aspect	**wait()**	**waitpid()**
Blocking	Blocks the caller until a child process terminates.	Can be either blocking or non-blocking depending on options.
Behavior	Waits until one child process terminates.	-
Handling	If more than one child process is running, wait() returns the first time one of the parent's offspring exits.	Offers more flexibility in handling multiple children.

Usage	Less flexible.	More flexible: - If pid == -1, waits for any child process. - If pid > 0, waits for child with specific PID. - If pid == 0, waits for any child in the same process group. - If pid < -1, waits for child in specific process group.

Flowchart:



Data Dictionary:

Variable/Identifier	Type	Description

id	pid_t	Stores the process ID returned by the fork() call.
i	int	Loop variable used for iteration.

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(In Text Format):

Parent Process Started.. ProcessID = 27312

```
In Parent
Parent waiting
Child Process Started.. ProcessID = 27313
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:

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:

http://www.yolinux.com/TUTORIALS/ForkExecProcesses.html

Assignment 1E

Title : 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.

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-

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.

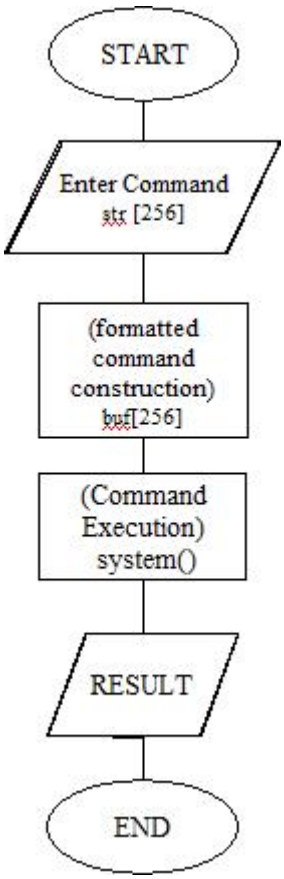
Return Value:

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:

Variables	Data Type	Use of Variable
str	char[256]	Stores user input command
buf	char[256]	Stores formatted command for system execution

Flowchart-



Program-

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
char str[256], buf[256];
printf("Enter command ");
scanf("%s",str);
sprintf(buf, "/bin/sh -c %s", str);
system(buf);
return 0;
}
```

Output-

```
Enter command hostnamectl
Static hostname: aadi
Icon name: computer-laptop
Chassis: laptop
 Machine ID: 82ddc2169081478eb21ea88b69852886
 Boot ID: 13b028f673bc41a5b6ccb42226fab4b4
Operating System: Ubuntu 22.04.3 LTS
 Kernel: Linux 6.5.0-26-generic
 Architecture: x86-64
 Hardware Vendor: Lenovo
```

Hardware Model: IdeaPad 3 15ITL6
aadi@aadi:~$

Conclusion:

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

References:

www.tutorialspoint.com/unix_system_calls/

Assignment 2A

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

Objectives:

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

Theory-

1.alarm()

Syntax-

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

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

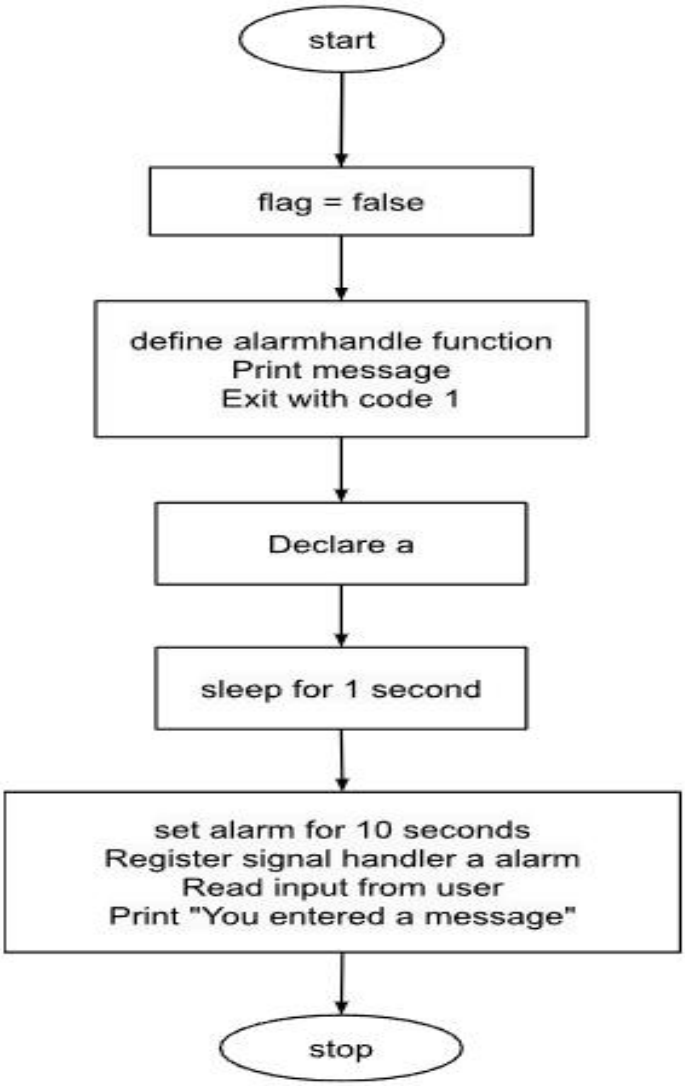
signal()

Syntax-

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

The signal() system call installs a new signal handler for the signal with number signum. The signal handler is set to sighandler which may be a user specified function, or either SIG_IGN or SIG_DFL.
Upon arrival of a signal with number signum the following happens. If the corresponding handler is set to SIG_IGN, then the signal is ignored. If the handler is set to SIG_DFL, then the default action associated with the signal (see signal(7)) occurs. Finally, if the handler is set to a function sighandler then first either the handler is reset to SIG_DFL or an implementation-dependent blocking of the signal is performed and next sighandler is called with argument signum.
Using a signal handler function for a signal is called "catching the signal". The signals SIGKILL and SIGSTOP cannot be caught or ignored.
The signal() function returns the previous value of the signal handler, or SIG_ERR on error. The original Unix signal() would reset the handler to SIG_DFL, and System V (and the Linux kernel and libc4,5) does the same. On the other hand, BSD does not reset the handler, but blocks new instances of this signal from occurring during a call of the handler. The glibc2 library follows the BSD behavior.

Flowchart:



Term	Description
SIGALRM	Signal generated by the alarm() system call when the timer expires
alarm()	System call used to set a timer that generates a SIGALRM signal
alarm_handler	Signal handler function called when the SIGALRM signal is received

input	Character array to store user input

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:

```
aadi@ubuntu: ./a.out
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:

www.tutorialspoint.com/unix_system_calls/

Assignment 2B

Title: Write an 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()

Syntax:

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

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.

signal()

Syntax-

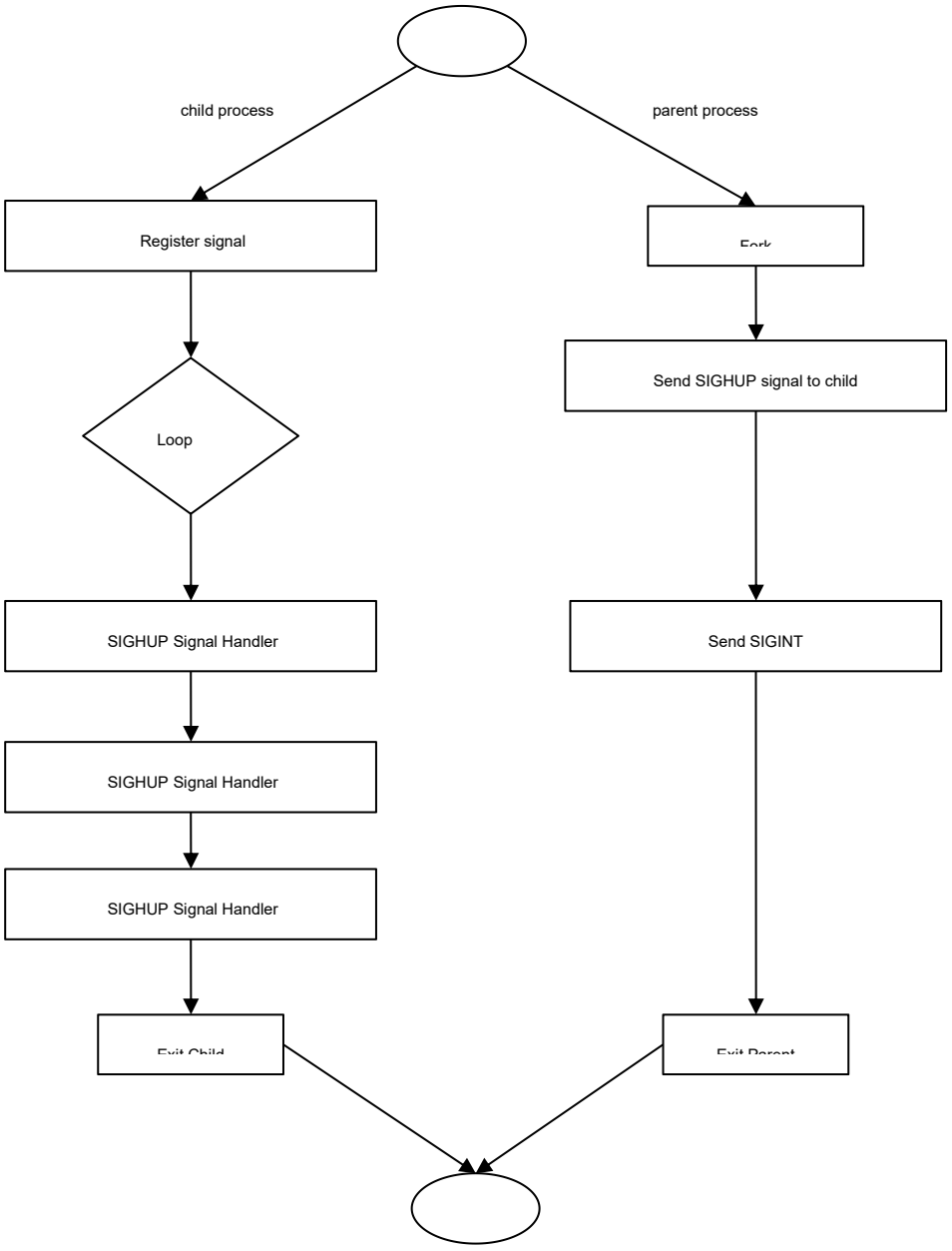
```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

The signal() system call installs a new signal handler for the signal with number signum. The signal handler is set to sighandler which may be a user specified function, or either SIG_IGN or SIG_DFL.
Upon arrival of a signal with number signum the following happens. If the corresponding handler is set to SIG_IGN, then the signal is ignored. If the handler is set to SIG_DFL, then the default action associated with the signal (see signal(7)) occurs. Finally, if the handler is set to a function sighandler then first either the handler is reset to SIG_DFL or an implementation-dependent blocking of the signal is performed and next sighandler is called with argument signum.
Using a signal handler function for a signal is called "catching the signal". The signals SIGKILL and SIGSTOP cannot be caught or ignored.
The signal() function returns the previous value of the signal handler, or SIG_ERR on error. The original Unix signal() would reset the handler to SIG_DFL, and System V (and the Linux kernel and libc4,5) does the same. On the other hand, BSD does not reset the handler, but blocks new instances of this signal from occurring during a call of the handler. The glibc2 library follows the BSD behavior.

Data Dictionary:

Sr. No	Variable/ Function	Datatype	Use
1	pid	int	Process ID
2	fork()		System call to create a new process (child process) by duplicating the calling
3	exit()		Function to terminate the calling process and return a status code to the parent process
4	signal()		Function to set a signal handler for a specific signal. When the signal with number signum occurs, the function handler will be called

Flowchart for parent, child communications



Program:

```
#include <signal.h> // library for signal handling
#include <stdio.h>  // library for input and output functions
#include <stdlib.h> // library for exit function
#include <sys/types.h> // library for process ID functions
```

```
#include <unistd.h> // library for sleep function

// Signal handler functions
void sighup();
void sigint();
void sigquit();

// Main function
void main()
{
   int pid;

   // Fork a child process
   if ((pid = fork()) < 0)
   {
      // Print error message if fork fails
      perror("fork");
      exit(1);
   }
   if (pid == 0)
   {Variable/Identifier
```

Type

Description

id

pid_t

Stores the process ID returned by the fork() call.

i

int

Loop variable used for iteration.Variable/Identifier

Type

Description

id

pid_t

Stores the process ID returned by the fork() call.

i

int

```
Loop variable used for iteration.
        // Child process
        // Register signal handlers for SIGHUP, SIGINT, and SIGQUIT
        signal(SIGHUP, sighup);
        signal(SIGINT, sigint);
        signal(SIGQUIT, sigquit);

        for (;;)
            ; // infinite loop
    }
    else
    {
        // Parent process
        printf("\nPARENT: sending SIGHUP\n\n");
        // Send SIGHUP signal to child process
        kill(pid, SIGHUP);
        sleep(3);
        printf("\nPARENT: sending SIGINT\n\n");
        // Send SIGINT signal to child process
        kill(pid, SIGINT);
        sleep(3);
        printf("\nPARENT: sending SIGQUIT\n\n");
        // Send SIGQUIT signal to child process
        kill(pid, SIGQUIT);
        sleep(3);
    }
}

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

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

// Signal handler function for SIGQUIT
void sigquit()
{
    printf("My DADDY has Killed me!!!\n");
    exit(0);
}
```

Output:

PARENT: sending SIGHUP

CHILD: I have received a SIGHUP

PARENT: sending SIGINT

CHILD: I have received a SIGINT

PARENT: sending SIGQUIT

My DADDY has killed me!!!

Conclusion:

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

References:

www.tutorialspoint.com/unix_system_calls/

Assignment 2C

Title- Write a application or program that communicates between to process opened in two terminal 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

kill()

Syntax-

#include<sys/types.h>

#include<signal.h>

int kill(pid_t pid, int sig);

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

signal()

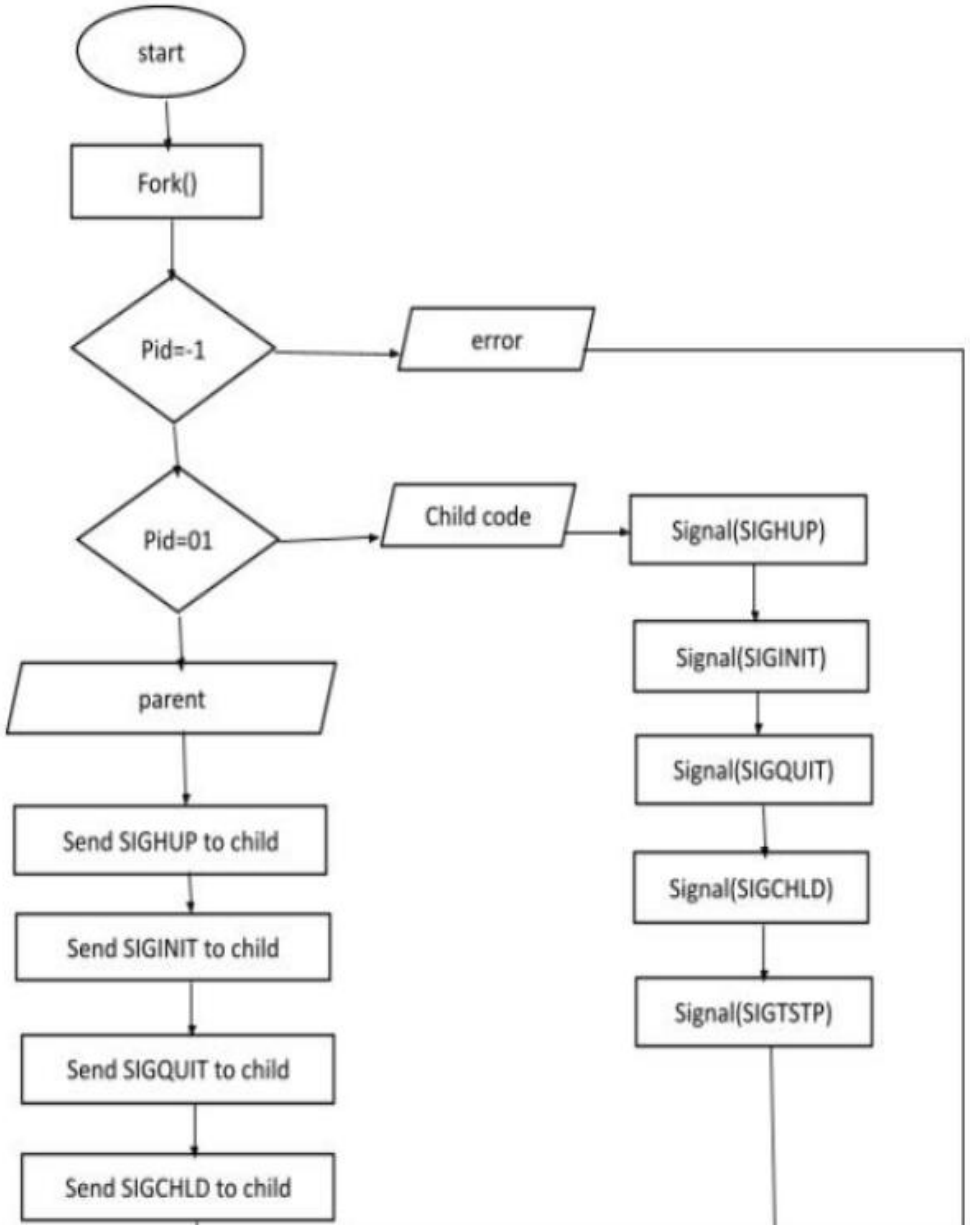
Syntax-

#include<signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler); The signal() system call installs a new signal handler for the signal with number signum. The signal handler is set to sighandler which may be a user specified function, or either SIG_IGN or SIG_DFL. Upon arrival of a signal with number signum the following happens. If the corresponding handler is set to SIG_IGN, then the signal is ignored. If the handler is set to SIG_DFL, then the default action associated with the signal (see signal(7)) occurs. Finally, if the handler is set to a function sighandler then first either the handler is reset to SIG_DFL or an implementation-dependent blocking of the signal is performed and next sighandler is called with argument signum. Using a signal handler function for a signal is called "catching the signal". The signals SIGKILL and SIGSTOP cannot be caught or ignored. The signal() function returns the previous value of the signal handler, or SIG_ERR on error. The original Unix signal() would reset the handler to SIG_DFL,and System V (and the Linux kernel and libc4,5) does the same. On the other hand, BSD does not reset the handler, but blocks new instances of this signal from occurring during a call of the handler. The glibc2 library follows the BSD behavior.

Flowchart-



Program-

```
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include<stdlib.h>
```

```
#include<unistd.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: http://www.csl.mtu.edu/cs4411.ck/www/NOTES/signa l/kill.htm

Assignment 2D: IPC

Title: Write an 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.Write a C program that doesn't terminate when Ctrl+C is pressed. It prints a message "Cannot be terminated using Ctrl+c" and continues execution. 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.
This program sets up a signal handler using the signal() function from the signal.h header. It specifies the signal_handler function as the handler for the SIGINT signal.

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.

Inside the signal_handler function, we can implement any custom behavior we want when the signal is received. In this example, it simply prints a message indicating that the Ctrl+C signal was trapped.

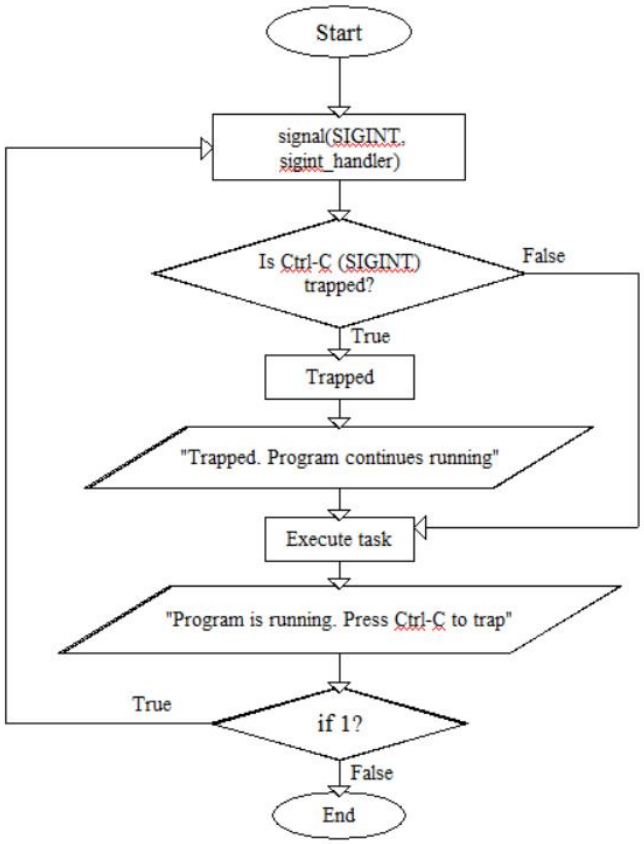
When we run this program and press Ctrl+C, instead of terminating immediately, it will execute the signal_handler function and then continue running indefinitely in the while(1) loop.

This demonstrates how to handle signals in C and how to trap the Ctrl+C signal without quitting the program

Data Dictionary:

Name	Type	Description
signum	int	Integer representing the signal number passed to the signal handler
sigint_handler	void	Signal handler function for the SIGINT signal (Ctrl+C)
main	int	Main function of the program
SIGINT	macro	Constant representing the SIGINT signal

Flowchart:



Program:

```
#include &lt;signal.h&gt;
#include &lt;stdio.h&gt;
#include &lt;stdlib.h&gt;
#include &lt;unistd.h&gt;
// Custom signal handler for SIGINT
void sigint_handler(int signum) {
printf(&quot;\nCtrl-C (SIGINT) trapped. Program continues running.\n&quot;);
}
int main() {

// Set up custom signal handler for SIGINT
signal(SIGINT, sigint_handler);
printf(&quot;Program is running. Press Ctrl-C to trap the signal.\n&quot;);
while (1) {
// Your program&#39;s main logic goes here
// ...
// Sleep or perform some non-intensive task to keep the program running
sleep(1);
}
return 0;
}
```

Output:

```
aadi@ubuntu:~$ nano 2d.c
aadi@ubuntu:~$ gcc 2d.c -o 2d
aadi@ubuntu:~$./2d
Program is running. Press Ctrl-C to trap the signal.
^C
Ctrl-C (SIGINT) trapped. Program continues running.
^C
Ctrl-C (SIGINT) trapped. Program continues running.
^Z
[1]+ Stopped ./2d
aadi@ubuntu:~$
```

Conclusion:

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

2. we have done successful program to trap ctrl +c signal but not stopped the quit options.

3.Instead we provided the another options for it such as ctrl + z and ctrl +Backslash

References:

1. https://www.geeksforgeeks.org/write-a-c-program-that-doesnt-terminate-when-ctrlc-is-pressed/
2. https://chat.openai.com/

Assignment 2E

Title-

IPC: Interrupts and Signals: signal(any fives type of signal), alarm, kill, raise, killpg, signal , sigaction.

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.**

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

Theory –

- kill() - Sends a signal to a specific process or group of processes.
- raise() - Sends a signal to the current process.

- killpg() - Sends a signal to all processes in a process group.
- alarm() - Sets an alarm to send a SIGALRM signal to the process after a specified time.
- abort() - Sends a SIGABRT signal to the calling process to terminate it abnormally.

1. **kill():**

   Syntax: int kill(pid_t pid, int sig);
   - Sends the signal specified by sig to the process with the process ID pid.
   - If pid is positive, the signal is sent to the process with the ID pid.
   - If pid is 0, the signal is sent to all processes in the current process group.
   - If pid is -1, the signal is sent to all processes for which the calling process has permission to send signals, except for process 1 (init).
   - Returns 0 on success, -1 on error.

2. **raise():**

   Syntax: int raise(int sig);
   - Sends the signal specified by sig to the calling process.
   - Returns 0 on success, a non-zero value on error.

3. **killpg():**

   Syntax: int killpg(int pgrp, int sig);
   - Sends the signal specified by sig to all processes in the process group identified by pgrp.
   - Returns 0 on success, -1 on error.

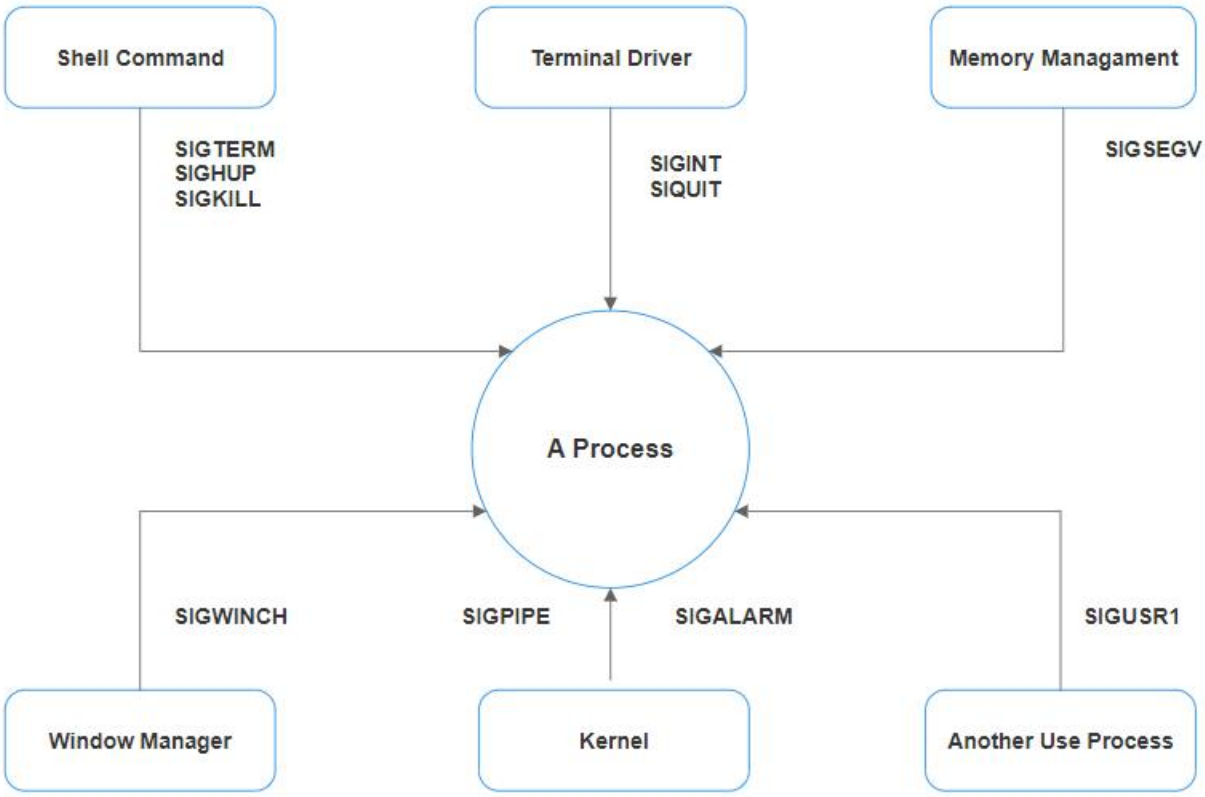
4. **alarm():**

   Syntax: unsigned int alarm(unsigned int seconds);
   - Sets an alarm to deliver a SIGALRM signal to the calling process after seconds seconds.
   - If seconds is 0, any previously set alarm is canceled.
   - Returns the number of seconds remaining on any previous alarm, or 0 if there was no previous alarm.

5. **abort():**

   Syntax: void abort(void);
   - Generates a SIGABRT signal to terminate the calling process abnormally.
   - This signal is caught by the default signal handler, which typically terminates the process and creates a core dump file.

Flowchart-



Code-

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

    void signal_handler(int signo)
    {
       printf("Received signal %d\n", signo);
    }
```

```
int main()
{
    // Set a signal handler for SIGALRM
    signal(SIGALRM, signal_handler);

    // 1. Using kill() to send a signal to the current process
    kill(getpid(), SIGALRM);

    // 2. Using raise() to send a signal to the current process
    raise(SIGALRM);

    // 3. Using killpg() to send a signal to the process group
    killpg(getpgrp(), SIGALRM);

    // 4. Using alarm() to set an alarm
    alarm(2); // Wait for 2 seconds before sending SIGALRM

    // 5. Using abort() to generate a SIGABRT signal
    abort();

    return 0;
}
```

Output-

```
Received signal 14
```

Received signal 14

Received signal 14

Conclusion:

Signals are a powerful mechanism for inter-process communication (IPC) in Unix and Unix-like operating systems. They allow processes to notify each other about specific events, such as interrupts, alarms, or requests for termination. Signals can be used to implement features like graceful process shutdown, timer-based tasks, and custom event handling.

References:

https://www.tutorialspoint.com/unix/unix-signals-traps.html

Assignment 2F

Title- Write application of signal handling in Linux OS and program any one.

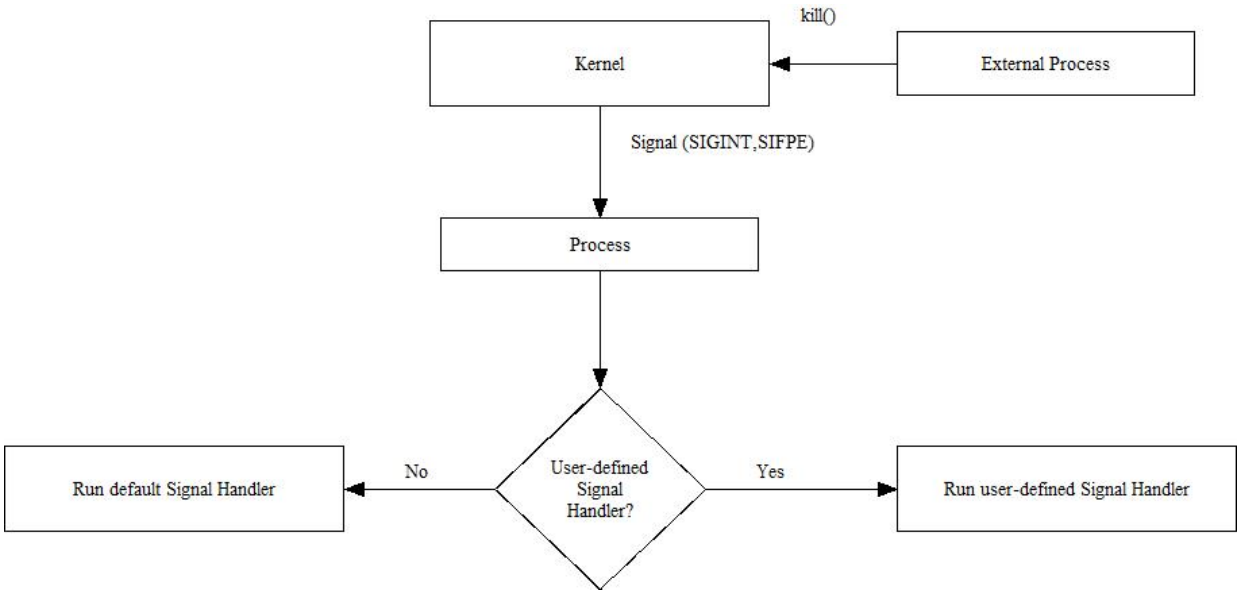
Objectives –

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 – Signals are a form of inter-process communication (IPC) used in Unix and Unix-like systems to notify a process that a specific event has occurred. Here are five types of signals and their explanations:

- **SIGINT (2) -** This signal is sent to a process by the terminal interrupt character (Ctrl+C by default) to request the process to terminate**.**
- **SIGALRM (14) -** This signal is used to set a timer. When the timer expires, a SIGALRM signal is sent to the process.
- **SIGKILL (9) -** This signal is used to forcefully terminate a process. It cannot be caught or ignored by the process**.**
- **SIGUSR1 (10) -** This signal is user-defined and can be used by the application to trigger specific actions.
- **SIGTERM (15) -** This signal is used to request the termination of a process. Unlike SIGKILL, the process can catch and handle this signal.

Flowchart –



Program –

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

// Signal handler function
void sigint_handler(int sig) {
    printf("Caught signal %d (SIGINT)\n", sig);
    printf("Exiting...\n");
    _exit(0); // Ensure immediate exit to avoid further signal interruption
}
```

```
int main() {
    // Register the signal handler
    if (signal(SIGINT, sigint_handler) == SIG_ERR) {
        fprintf(stderr, "Failed to register signal handler for SIGINT\n");
        return 1;
    }

    printf("Press Ctrl+C to send a SIGINT signal...\n");

    // Infinite loop to keep the program running
    while (1) {
        sleep(1); // Sleep for 1 second
    }

    return 0;
}
```

Output –

```
Press Ctrl+C to send a SIGINT signal...
^CCaught signal 2 (SIGINT)
Exiting...
```

Conclusion –

Signals are a powerful mechanism for inter-process communication (IPC) in Unix and Unix-like operating systems. They allow processes to notify each other about specific events, such as interrupts, alarms, or requests for termination. Signals can be used to implement features like graceful process shutdown, timer-based tasks, and custom event handling.

References –

Unix / Linux - Signals and Traps (tutorialspoint.com)

Assignment 3A_a

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

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:

The file status functions stat, fstat, and lstat are used to obtain information about files. They do not require permissions on the file itself, but in the case of stat() and lstat(), execute (search) permission is required on all directories in the path leading to the file.

- stat(): Stats the file pointed to by path and fills in the buf.
- lstat(): Similar to stat(), but if path is a symbolic link, it stats the link itself, not the file it refers to.

- fstat(): Similar to stat(), but the file to be stated is specified by the file descriptor fd.

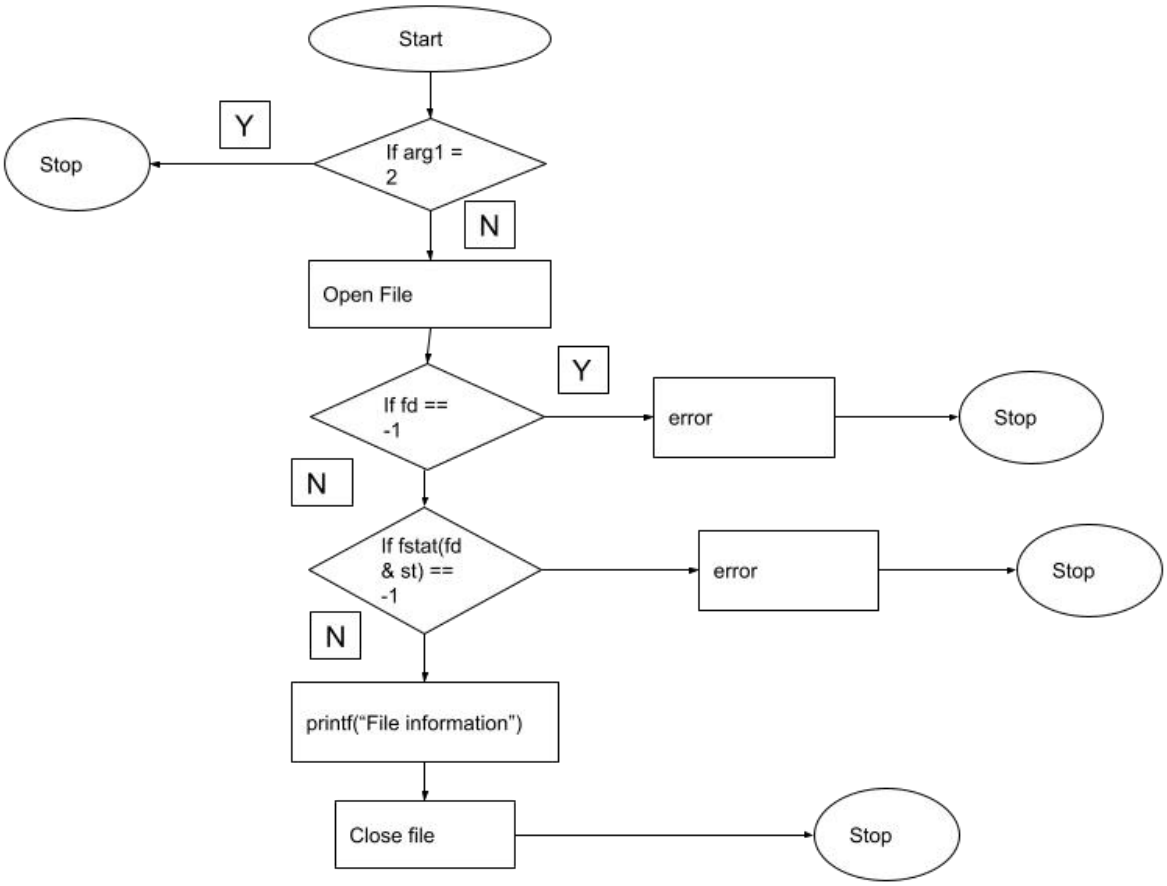
All these functions return a struct stat which contains the following fields:

```
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 filesystem 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	Data Type	Use
1	s	char[]	Get file name.
2	fp	FILE*	Pointer to file.
3	fn	int	File descriptor number.
4	sta		t Store information about files.

Flowchart :



Program:

```
#include<stdio.h>

#include<stdlib.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>


int main() {
```

```
char s[100];
printf("Enter file name: ");
fgets(s, sizeof(s), stdin); // Using fgets instead of gets for safer input handling
s[strcspn(s, "\n")] = '\0'; // Remove newline character from fgets input
// printf("%s",s);
FILE *fp;

if ((fp = fopen(s, "r")) == NULL) {
   perror("Error");
   return 1;
}
int fn = 0;
fn = fileno(fp);
struct stat sta;
if (fstat(fn, &sta) < 0) {
   perror("Error");
   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");
        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:

```
Enter file name: sample.txt

File size : 4252

File INode Number : 254232
```

File UID : 1234

File GID : 1224

File Permissions: -rw-r--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:

https://www.lix.polytechnique.fr/~liberti/public/computing/prog/c/C/FUNCTIONS/stat.html

Assignment 3A_b

Theory:

File system internals refer to the inner workings of how data is organized, stored, and managed on storage devices by an operating system. This includes structures like inodes, allocation tables, and metadata, as well as methods for allocating disk space and managing file operations. Understanding file system internals is crucial for optimizing performance, ensuring data integrity, and troubleshooting storage-related issues.

stat:

- stat is a system call in Unix-like operating systems (including Linux) that retrieves information about a file identified by its pathname.

- It returns a struct stat containing various attributes of the file, such as its size, type, permissions, ownership, timestamps (last access, modification, and status change), and more.

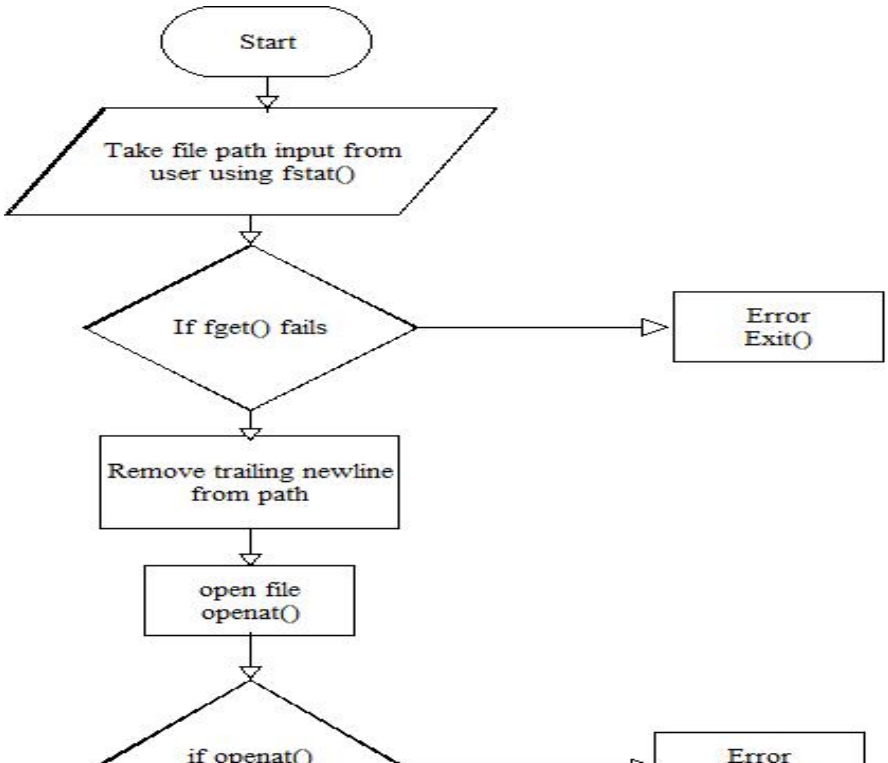
fstat:

- fstat is similar to stat but instead of taking a filename as an argument, it operates on an open file descriptor.
- This means it retrieves information about the file associated with an already opened file descriptor rather than needing the filename.

ustat:

- ustat is a system call that provides information about a mounted file system.
- It returns a struct ustat containing information such as the total number of inodes, the number of free inodes, the total number of blocks, and the number of free blocks.

Flowchart:



CODE:

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

#define MAX_PATH 256

int main() {
    char path[MAX_PATH];
    struct stat statbuf;
    int fd;

    printf("Enter the file/directory path: ");
    if (fgets(path, MAX_PATH, stdin) == NULL) {
        perror("fgets");
        return 1;
    }
```

```
// Remove trailing newline (if any)
path[strcspn(path, "\n")] = '\0';
// Open the file/directory (handle symbolic links)
fd = openat(AT_FDCWD, path, O_RDONLY | AT_SYMLINK_NOFOLLOW);
if (fd == -1) {
   perror("openat");
   return 1;
}

// Get file statistics using fstat
if (fstat(fd, &statbuf) == -1) {
   perror("fstat");
   close(fd);
   return 1;
}

close(fd);  // Close the file descriptor

// Print requested file statistics
printf("Inode number: %lu\n", statbuf.st_ino);
printf("UID: %u\n", statbuf.st_uid);
printf("GID: %u\n", statbuf.st_gid);

// Determine and print file type (regular, directory, etc.)
switch (S_ISREG(statbuf.st_mode)) {
   case 1:
      printf("File type: Regular file\n");
      break;
   case 0:
      if (S_ISDIR(statbuf.st_mode)) {
         printf("File type: Directory\n");
      } else if (S_ISCHR(statbuf.st_mode)) {
         printf("File type: Character device\n");
      } else if (S_ISBLK(statbuf.st_mode)) {
         printf("File type: Block device\n");
      } else if (S_ISFIFO(statbuf.st_mode)) {
         printf("File type: FIFO (named pipe)\n");
      } else if (S_ISLNK(statbuf.st_mode)) {
         printf("File type: Symbolic link\n");
      } else {
         printf("File type: Unknown\n");
      }
      break;
```

```
    }

    return 0;
}
```

Output:

```
Inode number: 5020423
UID: 1000
GID: 1000
File type: Directory
```

Explanation and Improvements:

Header Inclusion: Includes necessary headers for system calls (sys/types.h, sys/stat.h, unistd.h), standard input/output (stdio.h), and string manipulation (string.h).

Symbolic Link Handling: Uses openat with AT_SYMLINK_NOFOLLOW to get file statistics of the file the symbolic link points to, not the link itself. This aligns with the prompt's behavior.

Error Handling: Includes error handling for fgets, openat, and fstat using perror to provide informative error messages.
File Descriptor Closing: Ensures the file descriptor is closed using close(fd) after fstat to release resources.

Concise File Type Determination: Uses a switch statement for a more compact way to determine and print the file type based on the file mode.

Clarity and Consistency: Maintains consistent formatting and indentation for readability.

User Input Validation: While not explicitly included in the prompt, consider adding input validation mechanisms (e.g., checking for path length) to enhance robustness.

Assignment 3A_c

Title:

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

Objective:

1. To learn about File system Internals.

Theory:

The stat function in Unix-like operating systems is used to retrieve information about a file. It takes a filename as input and fills a structure (struct stat) with details such as the file's size, permissions, ownership, timestamps, and more. The stat function is declared in the <sys/types.h>, <sys/stat.h>, and <unistd.h> header files. Its parameters include the pathname of the file and a pointer to a struct stat where the file information will be stored. Upon successful execution, stat returns 0, while failure results in a return value of -1, accompanied by setting errno to indicate the specific error encountered. For example, using stat allows developers to access information like file size (st_size), permissions (st_mode), user ID of the owner (st_uid), group ID of the owner (st_gid), and last modification time (st_mtime).

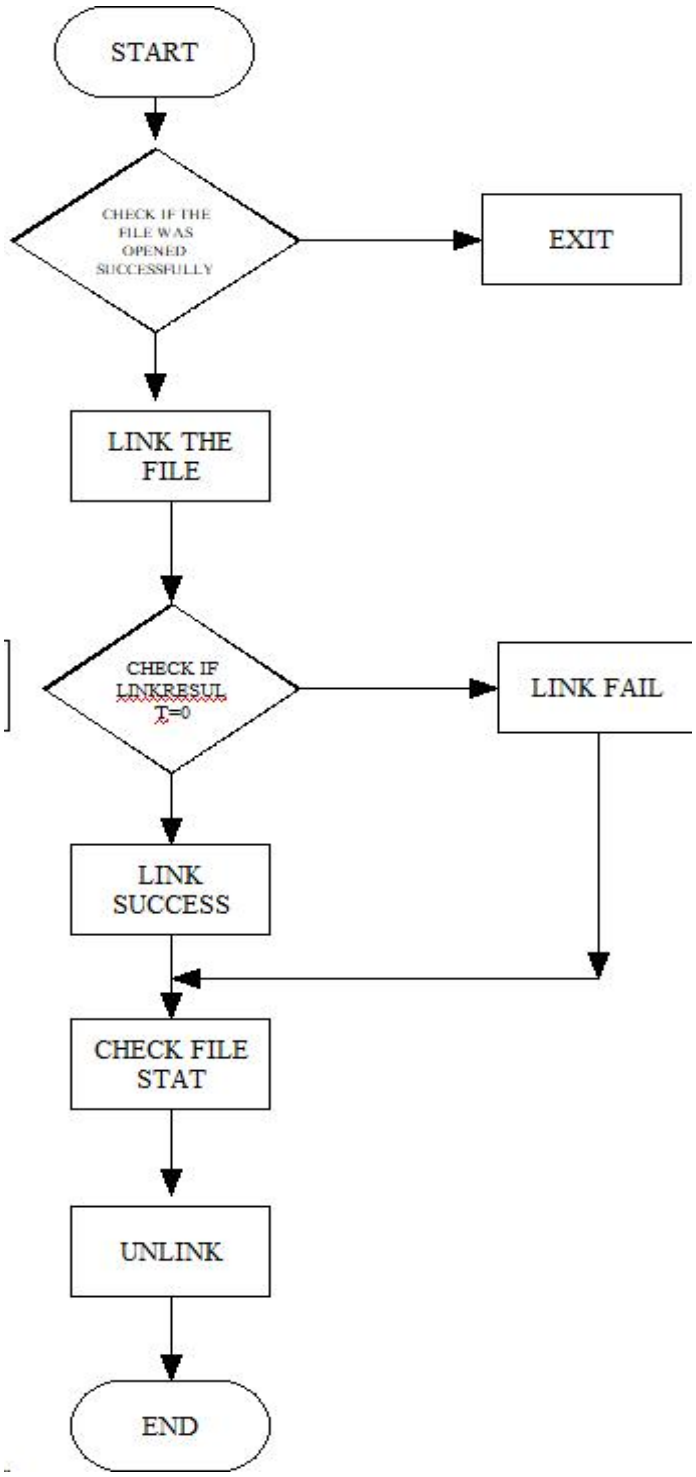
On the other hand, the link function creates a new link, either a hard link or a symbolic link, to an existing file. Its declaration resides in the <unistd.h> header file. link takes two parameters: oldpath, the path to the existing file, and newpath, the location where the new link will be created. Upon successful creation of the link, link returns 0; otherwise, it returns -1, along with setting errno to indicate the encountered error. Hard links created using link establish multiple directory entries (links) to the same physical file, whereas symbolic links, created with symlink, act as pointers to another file. Symbolic links can

span different file systems and are often used to link directories or files located in different locations within a system. These functions, stat and link, are fundamental tools in file management within Unix environments, facilitating the retrieval of file attributes and the creation of different types of file links.

Data Dictionary:

Serial No.	Variable/Function	Data Types	Use
1	argc	int	Number of command-line arguments passed to the program.
2	argv[]	char*[]	Array of strings containing the command-line arguments.
3	file	file*	Pointer to original file
4	linkresult	int	storing the result of the link system call for creating a logical link.
5	originalStat	Struct stat	Original file info.
6	linkStat	Struct stat	Info about link file

Flowchart:



CODE:

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

```
int main() {
    const char* originalFile = "original.txt";
    const char* linkFile = "logical_link.txt";

    // Create the original file
    FILE* file = fopen(originalFile, "w");
    if (file != nullptr) {
        fputs("Hello, this is the original file.", file);
        fclose(file);
    }

    // Create a logical link to the original file
    int linkResult = link(originalFile, linkFile);
    if (linkResult == 0) {
        std::cout << "Logical link created successfully." << std::endl;
    } else {
        std::cerr << "Error creating logical link." << std::endl;
        return 1;
    }

    // Use stat to compare the files
    struct stat originalStat, linkStat;
    stat(originalFile, &originalStat);
    stat(linkFile, &linkStat);

    std::cout << "Original File Size: " << originalStat.st_size << " bytes" << std::endl;
    std::cout << "Link File Size: " << linkStat.st_size << " bytes" << std::endl;

    // Clean up: remove the files
    unlink(originalFile);
    unlink(linkFile);

    return 0;
}
```

Conclusion:

Using link() two files can be linked

References:

https://www.lix.polytechnique.fr/~liberti/public/computing/prog/c/C/FUNCTIONS/stat.html

Assignment 3A_d

Title:

Implement a program to print the various types of file in Linux.
(Char, block etc.)

Objective:

1. To learn about File system Internals.

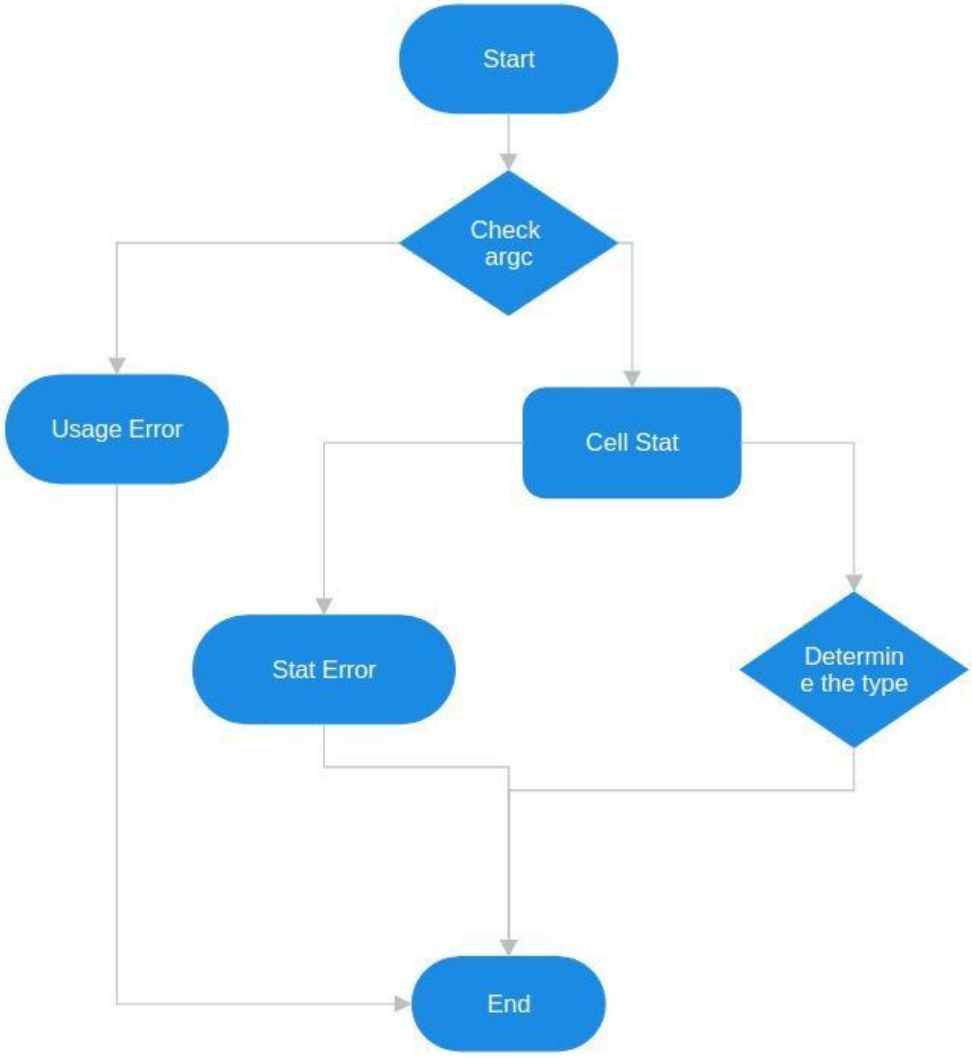
Theory:

The "stat" system call is a function in Unix-like operating systems that is used to retrieve information about a file or directory. The call takes a pathname as its argument and returns information about the file, such as its type (file, directory, symbolic link, etc.), permissions, timestamps (last modification, last access, and creation times), and size.
The information about the file is returned in a "struct stat" data structure, which is defined in the header file "sys/stat.h". The struct stat includes information about the file's inode number, device ID, number of hard links, user and group IDs, and other details.
The "stat" system call is often used in conjunction with the "open" system call to retrieve information about a file before it is opened, or to check whether a file exists and is accessible. It can also be used to retrieve information about symbolic links, which are often used to create shortcuts or aliases to files or directories.

Data Dictionary:

Serial No.	Variable/Function	Data Types	Use
1	argc	int	Number of command-line arguments passed to the program.
2	argv[]	char*[]	Array of strings containing the command-line arguments.
3	st	struct stat	Data structure used to store information about a file(eg. type,permissions,timestamps).
4	argv[1]	char*	String representing the file path provided as a command-line argument
5	st_mode	mode_t	Data type representing file mode and file type information.
6	S_IFMT	macro	Macro used to mask the filetype portion of the st_mode field.

Flowchart:



Code:

```
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[])
{

        struct stat st;
        if (argc != 2) {
                fprintf(stderr, "Usage: %s <file>\n", argv[0]);
                return 1;
        }
        // Retrieve information about the file using stat
        if (stat(argv[1], &st) == -1) {
                perror("stat");
                return 1;
        }
        // Print information about the file
        printf("File type: ");
        switch (st.st_mode & S_IFMT) {
                case S_IFREG: printf("regular file\n"); break;
                case S_IFDIR: printf("directory\n"); break;
                case S_IFCHR: printf("character device\n"); break;
                case S_IFBLK: printf("block device\n"); break;
                case S_IFLNK: printf("symbolic link\n"); break;
                case S_IFIFO: printf("FIFO (named pipe)\n"); break;
                case S_IFSOCK: printf("socket\n"); break;
                default: printf("unknown\n"); break;
        }
        return 0;
}
```

Output:

File type: regular filr

References:

https://www.lix.polytechnique.fr/~liberti/public/computing/prog/c/C/FUNCTIONS/stat.html

Assignment 3B_b

Title -

Write a program to lock the file using flock system call File locking system call : fnctl.h: flock/lockf

Objective -

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

Theory -

File Locking with flock

Problem: Multiple processes accessing the same file concurrently can lead to data corruption.

Solution: File locking using flock (advisory locking) helps prevent this.

Steps:

Lock Acquisition:

Process opens the file.
flock is used to acquire an exclusive lock (prevents other processes from modifying the file).
If the lock is already held, the process might block or fail depending on configuration.

Critical Section:

The process has exclusive access to the file and can safely read or write data.

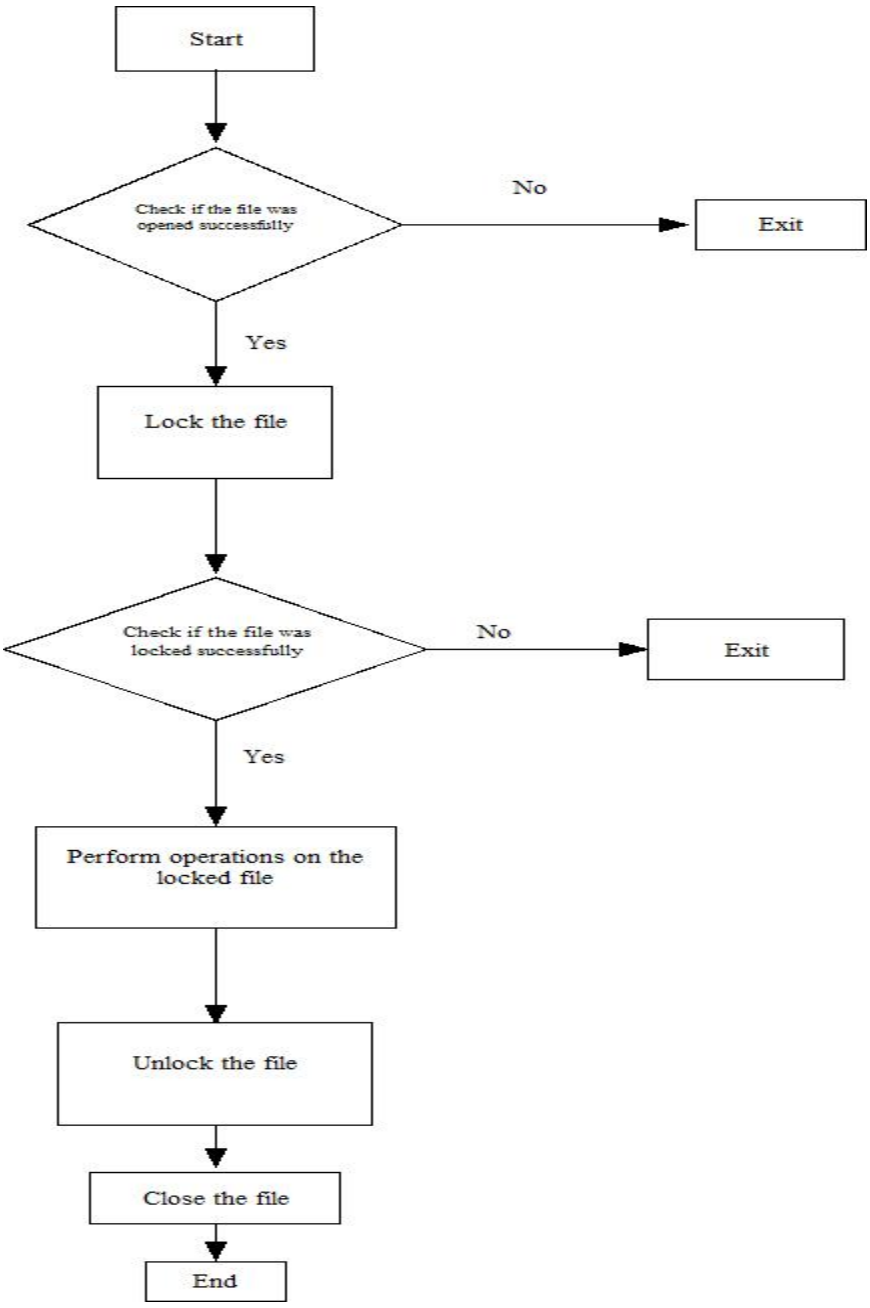
Unlocking:

flock is used again to release the lock, allowing other processes to access the file.

Benefits:

Ensures data integrity when multiple processes access the file.
Improves coordination and avoids race conditions.

Flowchart -



Code -

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
```

```
void lock_file(int fd) {
    struct flock lock;
    lock.l_type = F_WRLCK;    // Exclusive write lock
    lock.l_whence = SEEK_SET; // Start of file
    lock.l_start = 0;         // Starting offset
    lock.l_len = 0;           // Whole file
    lock.l_pid = getpid();    // PID of process

    if (fcntl(fd, F_SETLKW, &lock) == -1) {
        perror("fcntl");
        exit(EXIT_FAILURE);
    }

    printf("File locked successfully.\n");
}

void unlock_file(int fd) {
    struct flock unlock;
    unlock.l_type = F_UNLCK;  // Unlock
    unlock.l_whence = SEEK_SET;
    unlock.l_start = 0;
    unlock.l_len = 0;
    unlock.l_pid = getpid();

    if (fcntl(fd, F_SETLK, &unlock) == -1) {
        perror("fcntl");
        exit(EXIT_FAILURE);
    }

    printf("File unlocked.\n");
}

int main() {
    const char *filename = "test.txt";
    int fd;

    // Open the file
    if ((fd = open(filename, O_RDWR | O_CREAT, 0666)) == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    // Lock the file
```

```
    lock_file(fd);

    // Perform operations on the locked file
    printf("Performing operations on the locked file...\n");

    // Unlock the file
    unlock_file(fd);

    // Close the file
    close(fd);

    return 0;
}
```

Output -

File locked successfully.
Performing operations on the locked file…
File unlocked.

Conclusion -

Lock: The program acquires an exclusive lock on the file, ensuring only one process can modify it at a time.
Work: The program safely reads from or writes to the file within this locked section.

Unlock: When finished, the program releases the lock, allowing other processes to access the file.

Assignment: 3B.C

Title:

Write a program to lock file using fnctl system call (E)

Objective:

To create a program that demonstrates file locking using the fcntl system call in Unix-like operating systems.

Theory:

.File locking is a mechanism used to restrict access to a file by multiple processes or threads.
•It helps in preventing data corruption or inconsistencies when multiple processes attempt to read from or write to the same file simultaneously.

•The fcntl system call is commonly used to implement file locking in Unix-like operating systems.
.It allows processes to lock regions of a file or the entire file itself, thereby controlling access to it.

Source Code:

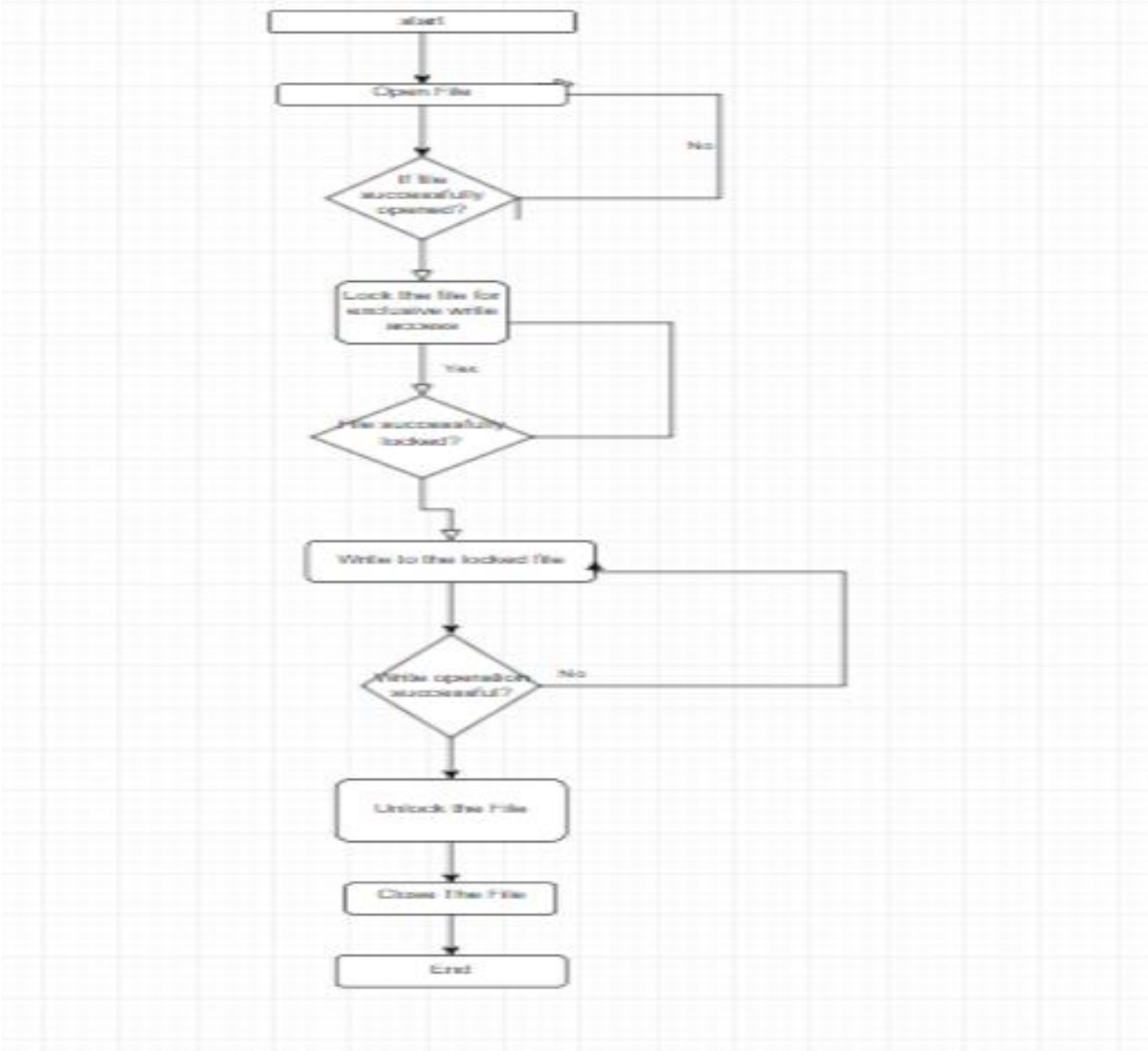
```
#include <iostream>
#include <unistd.h>
#include <fcntl.h>
#include <cstring>
#include <cstdlib>
using namespace std;
int main() {
   const char* filename = "example.txt";
   int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
   if (fd == -1) {
      cerr << "Error opening file\n";
      exit(EXIT_FAILURE);
   }
   // Locking the entire file for exclusive write access
   struct flock fl;
   fl.l_type = F_WRLCK;    // Exclusive write lock
   fl.l_whence = SEEK_SET; // Start from beginning of file
   fl.l_start = 0;         // Offset from whence
   fl.l_len = 0;           // Lock whole file
   fl.l_pid = getpid();    // Process ID
   if (fcntl(fd, F_SETLKW, &fl) == -1) {
      cerr << "Error locking file\n";
      exit(EXIT_FAILURE);
   }
   // Write to the locked file
   const char* message = "This is a locked file.";
   ssize_t bytes_written = write(fd, message, strlen(message));
   if (bytes_written == -1) {
      cerr << "Error writing to file\n";
      exit(EXIT_FAILURE);
   }
   cout << "File locked and written to successfully.\n";
   // Unlocking the file
   fl.l_type = F_UNLCK;
   if (fcntl(fd, F_SETLK, &fl) == -1) {
      cerr << "Error unlocking file\n";
      exit(EXIT_FAILURE);
   }
```

```
    close(fd);
    return 0;
}
```

Output :

File locked and written successfully.

FlowChart:



Conclusion:

This program demonstrates file locking using the fcntl system call in Unix-like operating systems. It locks the entire file for exclusive write access, preventing other processes from modifying it concurrently. After performing some work, the program unlocks the file, allowing other processes to access it.

References:

- Linux man pages
- Advanced Programming in the UNIX Environment" by W. Richard Stevens and Stephen A. Rago

Assignment 4A

Title : Write a multithreaded program in java/c for chatting (multi-user and multi-terminal) using threads.

Theory :

1. **Purpose of Threading:**

- Threading is used in network programming to handle multiple client requests simultaneously.
- Without threading, only one client can connect to the server at a time, and subsequent requests are rejected.

2. **Example Scenario:**
   - Consider a Date-Time server located at a place X, serving multiple generic clients.
   - At a particular time, multiple requests arrive at the server.
   - Without threading, the first request gets served, and subsequent requests are rejected.

3. **Server Side Programming (Server.java):**
   - Server class: Handles the server-side operations.
   - Establishing Connection: Initializes a ServerSocket object and continuously accepts incoming connections in a loop.
   - Obtaining Streams: Extracts InputStream and OutputStream from the current request's Socket object.
   - Creating Handler Object: Instantiates a new ClientHandler object with the extracted parameters.
   - Invoking start(): Invokes the start() method on the newly created thread object.

4. **ClientHandler class:**
   - Extends Thread: Inherits properties of Thread class for concurrent execution.
   - Constructor Parameters: Takes a Socket, a DataInputStream, and a DataOutputStream to uniquely identify each request.
   - Thread Creation: Instantiates an object of this class for each incoming request and invokes start() method.
   - Run Method: Performs three operations - prompts user for time or date, reads answer from input stream, and writes output to output stream.

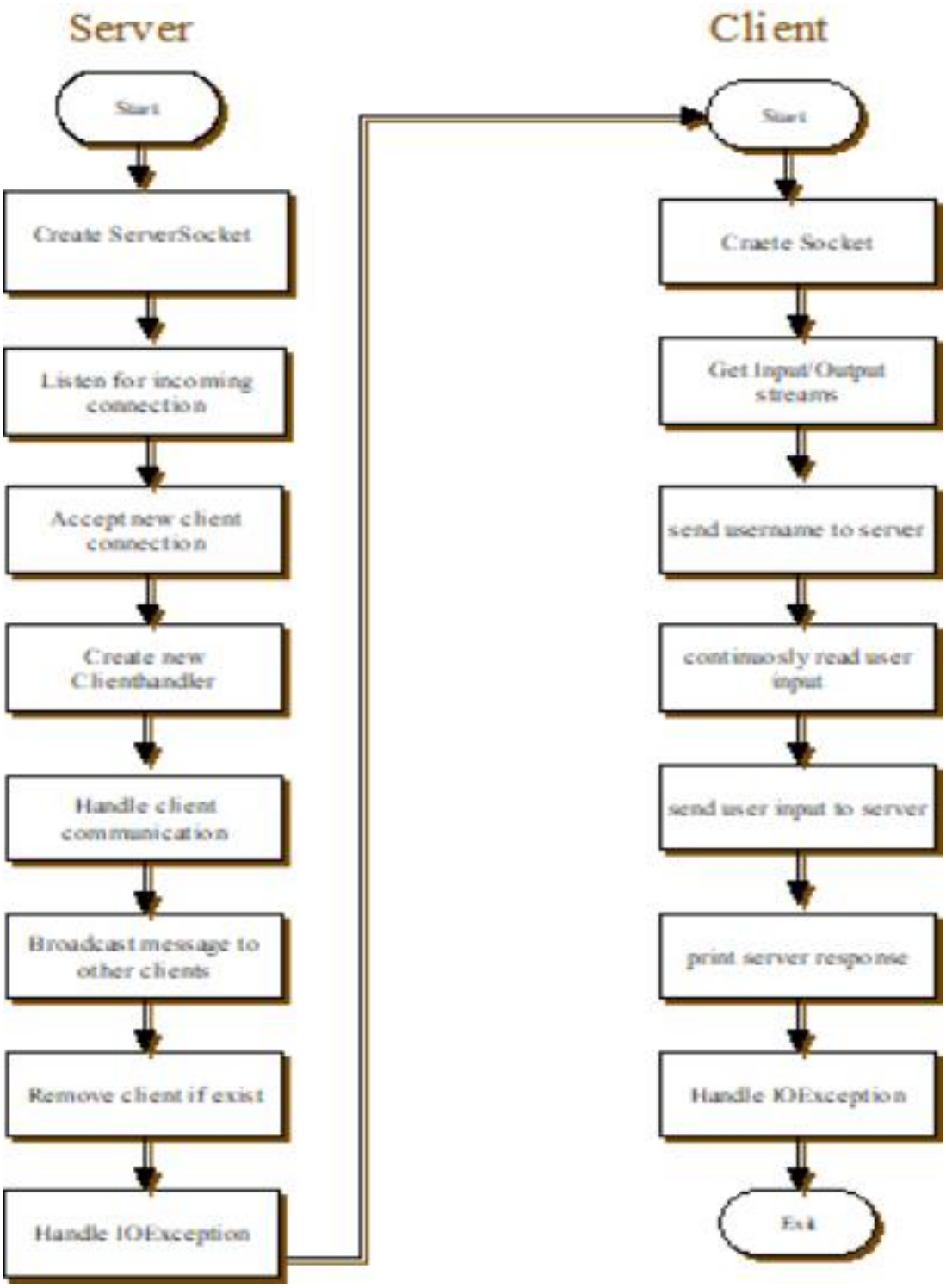
5. **Threading Implementation:**
   - Each client request is handled by a separate thread, allowing multiple clients to be served simultaneously.
   - Threaded architecture ensures that each request is processed independently without blocking other requests.

Data Dictionary :

Sr Number	Variable	Datatype	Use
ClassServer			
1	Ar	Vector	Store which clients are there

2	i	int	Count of clients
3	ss	serversocket	Create socket for server side
4	s	socket	Socket is craeted
5	dis	Datainputstream	Input data
6	t	Thread	Used to create new thread
7	mtch	ClientHandler	Object of clienthandler
Class ClientHandler			
1	scn	Scanner	Used for any input
2	name	string	Name of client
3	dis	DataInputStream	Input message
4	recieved	string	Store message

Flowchart :



Code :
Server code

```
import java.io.*;
import java.net.*;
```

```
import java.util.*;
public class MultiThreadedChatServer {
private static final int PORT = 12345;
private static List<ClientHandler> clients = new ArrayList<>();
public static void main(String[] args) {
try (ServerSocket serverSocket = new ServerSocket(PORT)) {
System.out.println("Server started. Listening on port " + PORT);
while (true) {
Socket clientSocket = serverSocket.accept();
System.out.println("New client connected: " + clientSocket);
ClientHandler clientHandler = new ClientHandler(clientSocket);
clients.add(clientHandler);
clientHandler.start();
}
} catch (IOException e) {
e.printStackTrace();
}
}
static class ClientHandler extends Thread {
private final Socket clientSocket;
private PrintWriter out;
private BufferedReader in;
public ClientHandler(Socket socket) {
this.clientSocket = socket;
}
public void run() {
try {
out = new PrintWriter(clientSocket.getOutputStream(), true);
in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
String username = in.readLine();
System.out.println(username + " joined the chat.");
String inputLine;
while ((inputLine = in.readLine()) != null) {
if ("exit".equalsIgnoreCase(inputLine)) {
break;
}
System.out.println(username + ": " + inputLine);
for (ClientHandler client : clients) {
if (client != this) {
client.sendMessage(username + ": " + inputLine);
}
}
}
clients.remove(this);
```

```
System.out.println(username + " left the chat.");
clientSocket.close();
} catch (IOException e) {
e.printStackTrace();
}
}
public void sendMessage(String message) {
out.println(message);
}
}
}
```

Client code :

```
import java.io.*;
import java.net.*;
public class ChatClient {
private static final String SERVER_ADDRESS = "localhost";
private static final int SERVER_PORT = 12345;
public static void main(String[] args) {
try (Socket socket = new Socket(SERVER_ADDRESS, SERVER_PORT);
PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in))) {
System.out.print("Enter your username: ");
String username = stdIn.readLine();
out.println(username);
String userInput;
while ((userInput = stdIn.readLine()) != null) {
out.println(userInput);
if ("exit".equalsIgnoreCase(userInput)) {
break;
}
System.out.println("Server: " + in.readLine());
}
} catch (IOException e) {
e.printStackTrace();
}
}
}
```

Output :

Enter you username:aadi

Hello i am fine
Server:chin

Conclusions:

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

References:

https://www.geeksforgeeks.org/multithreading-in-java/

Assignment 4C

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

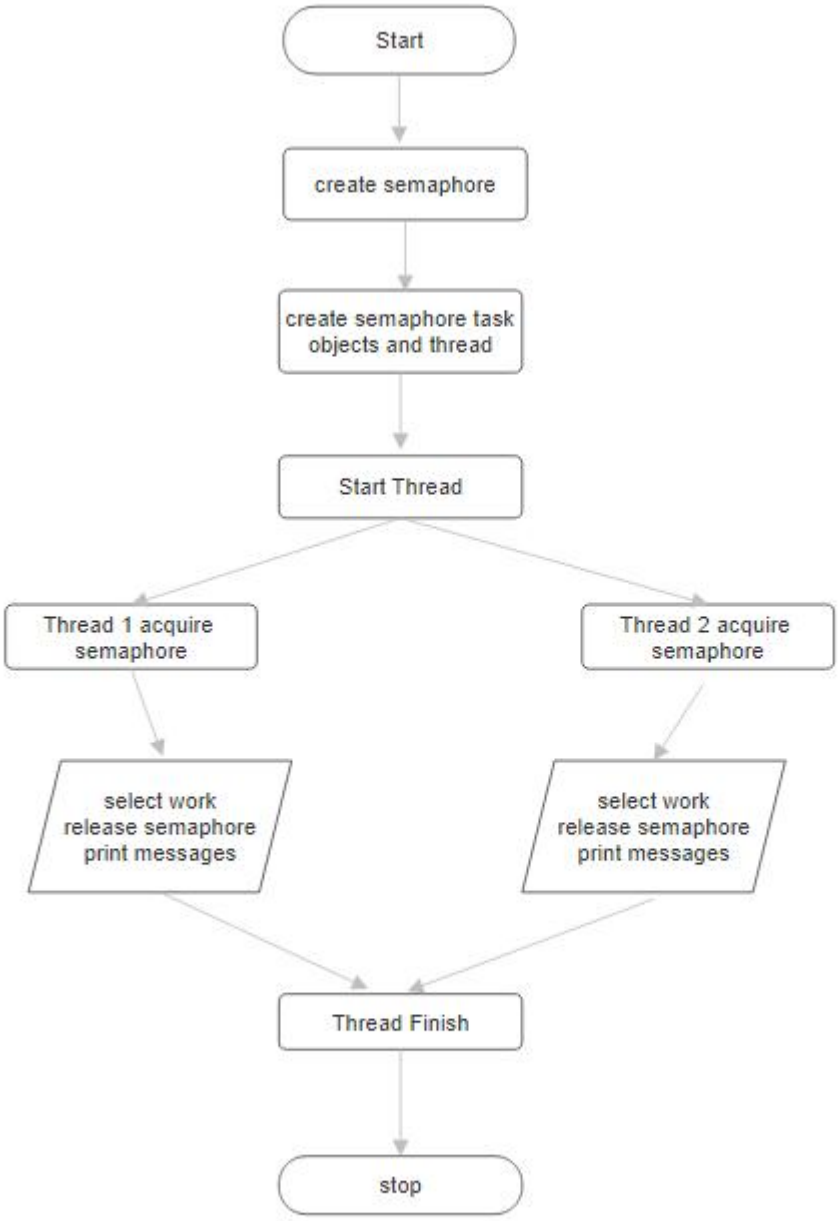
• If the semaphore's count is greater than zero, then the thread acquires a permit, which causes the semaphore's count to be decremented.

• Otherwise, the thread will be blocked until a permit can be acquired.

• When the thread no longer needs 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 provides a Semaphore class in java.util.concurrent package that implements this mechanism, so you don't have to implement your own semaphores.

Flowchart:



Code:

```
import java.util.concurrent.Semaphore;
public class SemaphoreThreadExample {
public static void main(String[] args) {
Semaphore semaphore = new Semaphore(1); // create a semaphore with a permit count
```

```
of 1
//Create two threads that use the same semaphore
Thread thread1 = new Thread(new SemaphoreTask(semaphore));
Thread thread2 = new Thread(new SemaphoreTask(semaphore));
// start both threads
thread1.start();
thread2.start();
}
static class SemaphoreTask implements Runnable
{ private Semaphore semaphore;
public SemaphoreTask(Semaphore semaphore)
{ this.semaphore = semaphore;
}
@Override
public void run() {
try {
semaphore.acquire(); // acquire a permit from the semaphore
System.out.println(&quot;Thread &quot; + Thread.currentThread().getName() + &quot;
acquired the

semaphore.&quot;);
Thread.sleep(1000); // simulate some work
semaphore.release(); // release the permit
System.out.println(&quot;Thread &quot; + Thread.currentThread().getName() + &quot; released

the semaphore.&quot;);
} catch (InterruptedException e)
{ e.printStackTrace();
}
}
}
}
```

Output:

```
Thread Thread-1 acquire semaphore
Thread Thread-0 acquire semaphore
Thread Thread-1 released semaphore
Thread Thread-0  released semaphore
```

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

Assignment 4D

4d)Thread concept: clone, threads of java

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

Objectives:

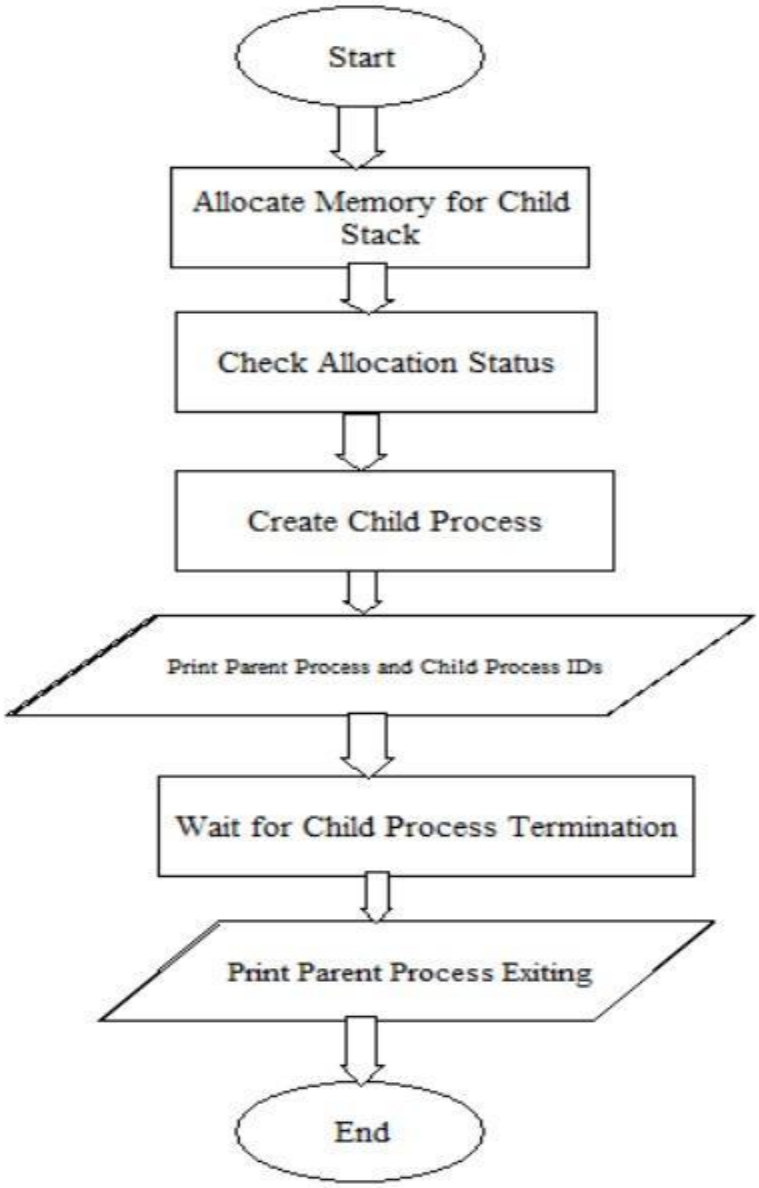
1. To learn about threading in Linux/Unix and Java and difference between them..
2. Use of system call/library to write effective programs

Theory:

1. **Functionality**:
   - ① Both clone() and fork() are used to create new processes in Linux.
   - ① fork() creates a new process that is a copy of the parent process. The child process has its own memory space but inherits copies of the parent's file descriptors, among other things.
   - ① clone() is more flexible than fork(). It allows for the creation of processes with different memory layouts, sharing specific resources between parent and child, etc.
2. **Signature**:
   - ① fork() has the signature pid_t fork(void);. It doesn't take any arguments.
   - ① clone() has the signature pid_t clone(int (*fn)(void *), void *child_stack, int flags, void *arg, ...);. It takes multiple arguments including a function pointer to the function to be executed by the child, a pointer to the child's stack, flags to specify cloning options, and an argument passed to the function.
3. **Options**:
   - ① fork() creates a new process with a copy of the parent's memory space, file descriptors, and other resources.
   - ① clone() allows more fine-grained control over what resources are shared between the parent and child processes. For example, you can specify whether memory, file descriptors, signal handlers, etc., should be shared or duplicated between parent and child.
4. **Usage**:
   - ① fork() is simpler and more commonly used for general-purpose process creation.
   - ① clone() is used in more specialized scenarios where finer control over resource sharing is needed, such as in implementing threads or in certain system-level programming tasks.
5. **Portability**:clone() is Linux-specific and provides more advanced features compared to fork().

In summary, while both fork() and clone() are used to create new processes in Linux, clone() offers greater flexibility and control over resource sharing between parent and child processes, making it suitable for specialized use cases.

Flowchart:



Code:

```
 #define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sched.h>
#include <sys/types.h>
#include <sys/wait.h>

#define STACK_SIZE 65536 // 64KB stack size for the cloned process
```

```
int child_func(void *arg) {
        printf("Child Process: PID=%d, Parent PID=%d\n", getpid(), getppid());
        return 0;
}

int main() {
        // Memory allocation for the child's stack
        void *child_stack = malloc(STACK_SIZE);
        if (child_stack == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
        }

        // Create a child process using clone() system call
        pid_t pid = clone(child_func, child_stack + STACK_SIZE, SIGCHLD, NULL);
        if (pid == -1) {
        perror("clone");
        exit(EXIT_FAILURE);
        }

        printf("Parent Process: PID=%d, Child PID=%d\n", getpid(), pid);

        // Wait for the child process to terminate
        if (waitpid(pid, NULL, 0) == -1) {
        perror("waitpid");
        exit(EXIT_FAILURE);
        }

        printf("Parent Process exiting.\n");

        return 0;
}
```

Output:

```
Parent Process: PID=7713, Child PID=7714
Child Process: PID=7714, Parent PID=7713
Parent Process exiting.
```

Conclusion:

This program demonstrates the use of the clone() system call to create a child process in Linux. The child process executes a specified function, and the parent process waits for the child to

terminate before exiting. This mechanism allows for more control over the creation and execution of processes in a Linux environment.

References:

1. http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html#BASICS
2. http://www.csc.villanova.edu/~mdamian/threads/posixthreads.html

Assignment 4E

Title: 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:

Threading in Linux/Unix and Java shares many fundamental concepts, but there are notable differences in their implementation and usage due to the nature of the underlying systems and programming languages. Here's an overview of threading in both environments along with their key distinctions:

Threading in Linux/Unix (pthreads):

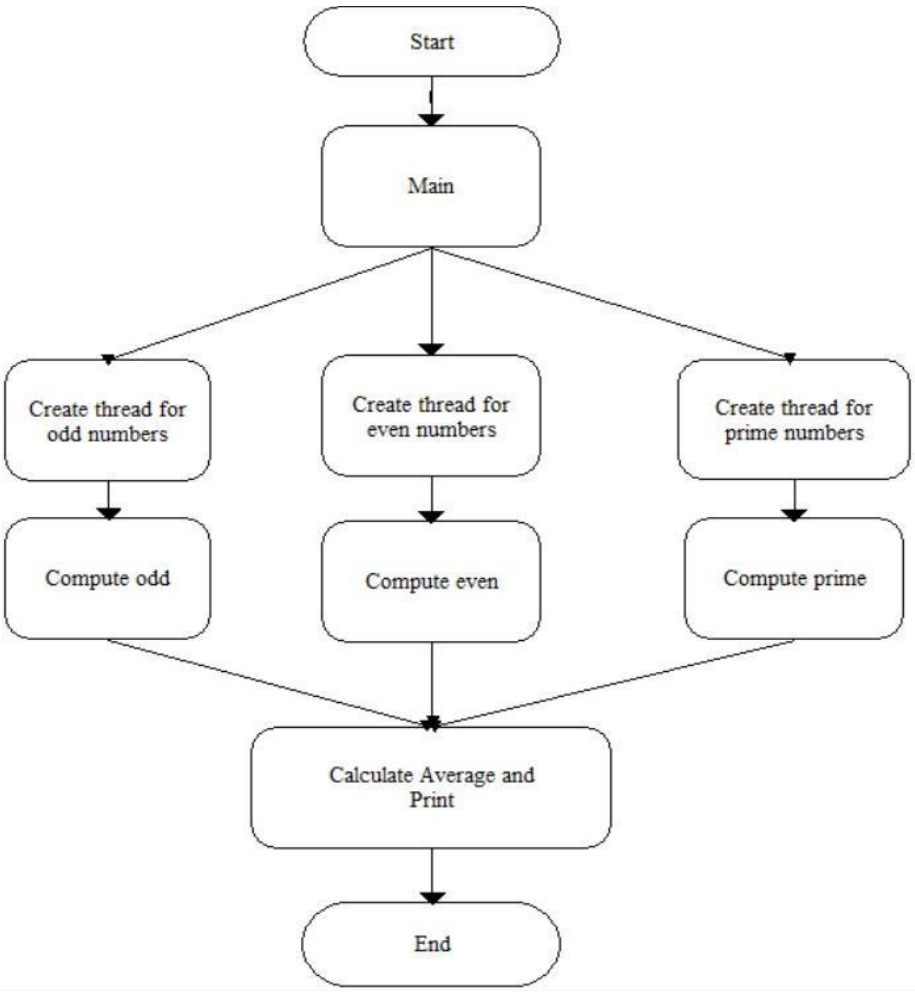
Threading in Linux/Unix is typically accomplished using the POSIX Threads (pthreads) library. Threads in pthreads are lightweight processes that share the same memory space and resources within a process.
Pthreads provide low-level control over thread creation, management, and synchronization through system calls and C library functions. Thread management involves explicit manipulation of thread attributes, synchronization primitives (e.g., mutexes, condition variables), and scheduling parameters. Pthreads programming often requires a good understanding of system-level concepts such as process management, memory allocation, and synchronization mechanisms.

Threading in Java:
Java provides built-in support for threading through its java.lang.Thread class and java.lang.Runnable interface.
Threads in Java are managed by the Java Virtual Machine (JVM) and abstracted from the underlying operating system, providing platform independence.
Java's threading model is higher-level compared to pthreads, offering features like thread lifecycle management, thread pooling, and synchronization utilities (e.g., synchronized keyword, wait-notify mechanism).
Java's thread management is simpler and more user-friendly than pthreads, with constructs like thread groups, daemon threads, and thread-safe collections.

Data Dictionary:

Variable	**Description**
oddSum, evenSum, primeSum	Sum of odd, even, and prime numbers respectively
oddCount, evenCount, primeCout	Count of odd, even, and prime numbers respectively
oddThread, evenThread, primeThread	Threads for computing count and Sum of odd, even, and prime numbers respectively
oddAverage, evenAverage, primeAverage	Average of odd, even, and prime numbers respectively

Flowchart:



Code:

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

#define MAX_NUMBERS 30

// Global variables to store odd, even, and prime numbers
int oddSum = 0, evenSum = 0, primeSum = 0;
int oddCount = 0, evenCount = 0, primeCount = 0;

// Function to check if a number is prime
int isPrime(int n) {
    if (n <= 1) return 0;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) return 0;
    }
    return 1;
}
```

```
// Function to compute the sum of odd numbers
void *computeOdd(void *arg) {
    for (int i = 1; i <= MAX_NUMBERS; i += 2) {
        oddSum += i;
        oddCount++;
    }
    pthread_exit(NULL);
}

// Function to compute the sum of even numbers
void *computeEven(void *arg) {
    for (int i = 2; i <= MAX_NUMBERS; i += 2) {
        evenSum += i;
        evenCount++;
    }
    pthread_exit(NULL);
}

// Function to compute the sum of prime numbers
void *computePrime(void *arg) {
    for (int i = 2; i <= MAX_NUMBERS; i++) {
        if (isPrime(i)) {
            primeSum += i;
            primeCount++;
        }
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t oddThread, evenThread, primeThread;

    // Creating threads
    pthread_create(&oddThread, NULL, computeOdd, NULL);
    pthread_create(&evenThread, NULL, computeEven, NULL);
    pthread_create(&primeThread, NULL, computePrime, NULL);

    // Waiting for threads to finish
    pthread_join(oddThread, NULL);
    pthread_join(evenThread, NULL);
    pthread_join(primeThread, NULL);

    // Calculating averages
```

```
    float oddAverage = (float)oddSum / oddCount;
    float evenAverage = (float)evenSum / evenCount;
    float primeAverage = (float)primeSum / primeCount;

    // Printing results
    printf("Average of odd numbers: %.2f\n", oddAverage);
    printf("Average of even numbers: %.2f\n", evenAverage);
    printf("Average of prime numbers: %.2f\n", primeAverage);

    return 0;
}
```

Output:
Average of odd numbers: 15.00
Average of even numbers: 16.00
Average of prime numbers: 12.90

Conclusion:

1. Threading in C using the pthread library allows for concurrent execution of tasks, which can significantly improve program performance by leveraging multiple CPU cores.
2. By distributing the computation of sum and count of odd, even, and prime numbers across separate threads, the program demonstrates parallelism, showcasing the ability to efficiently handle independent tasks concurrently.

References:

3. http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html#BASICS
4. http://www.csc.villanova.edu/~mdamian/threads/posixthreads.html

Assignment 4F

Title: Write a program using the p thread library of Linux. Create three threads to take numbers and use join to print their average.

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:

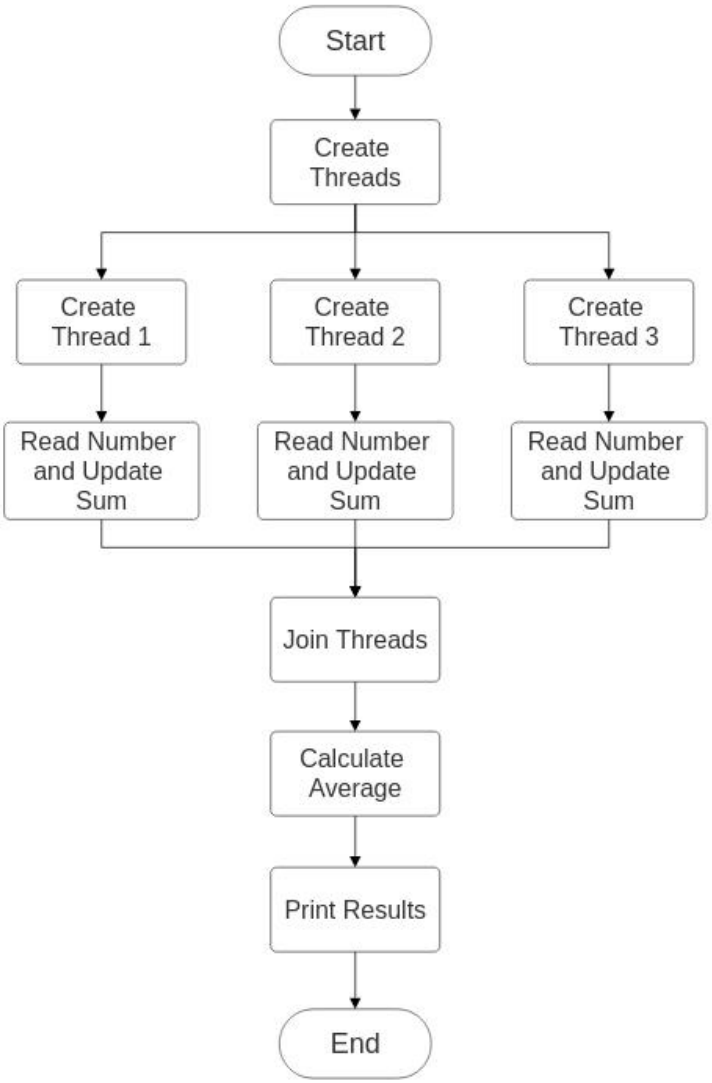
The pthread library in Linux provides a set of functions for creating and managing threads in a multi-threaded program. Threads are individual units of execution within a process that can run concurrently, allowing for parallelism and improved performance. In the provided program, three threads are created using pthread_create, each executing the thread_function. This function is designed to read a number from the user, update the global sum variable with the input, and increment the count variable. By using threads, the program can prompt the user for input concurrently, potentially reducing the time it takes to gather multiple inputs.

The pthread_join function is crucial for synchronization in this program. After creating the threads, the main function waits for each thread to complete its execution using pthread_join. This ensures that the program does not proceed to calculate the average until all threads have finished their tasks. Without this synchronization, the main thread might try to calculate the average before all numbers are entered, leading to incorrect results. Once all threads have completed, the program calculates the average by dividing the sum by the count and prints the result. This demonstrates the use of pthread library functions to create a simple parallel program that efficiently calculates the average of numbers entered by the user.

Data Dictionary:

Variable	Description
sum, count	Sum of numbers and number of elements respectively
num, rc	Input numbers and status of thread creation(0 if successful else non zero)
average	Stores average of input numbers

Flowchart:



Code:

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

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

#define NUM_THREADS 3

// Global variables for sum and number of elements
double sum = 0;
int count = 0;

// Thread function to take numbers from the user
void *thread_function(void *arg) {
        double num;
        printf("Enter a number: ");
        scanf("%lf", &num);

        // Add the number to the sum
        sum += num;
        count++;

        pthread_exit(NULL);
}

int main() {
        pthread_t threads[NUM_THREADS];
        int i, rc;

        // Create three threads
        for (i = 0; i < NUM_THREADS; i++) {
        rc = pthread_create(&threads[i], NULL, thread_function, NULL);
        if (rc) {
        fprintf(stderr, "Error: pthread_create() failed\n");
        exit(-1);
        }
        }

        // Wait for all threads to finish
        for (i = 0; i < NUM_THREADS; i++) {
        rc = pthread_join(threads[i], NULL);
        if (rc) {
        fprintf(stderr, "Error: pthread_join() failed\n");
        exit(-1);
        }
        }
```

```
        // Calculate and print the average
        if (count > 0) {
        double average = sum / count;
        printf("Average of the numbers: %.2f\n", average);
        } else {
        printf("No numbers were entered.\n");
        }

        pthread_exit(NULL);
}
```

Output:
Average of the numbers: 11.67

Conclusion:
The pthread library in Linux enables the creation of concurrent threads for parallel execution. The provided program creates three threads to read numbers from the user, updating a shared sum and count. By using pthreads, the program efficiently calculates the average of these numbers, demonstrating the power of multi-threaded programming and synchronization with pthread_join. This approach is fundamental for developing responsive and efficient applications in Linux, leveraging the benefits of parallelism.

References:

1. http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html#BASICS
2. ftp://10.1013.3/pub/UOS..../

Assignment 4G

Title - Write program to synchronize threads using construct –monitor/serialize/semaphore of Java (In java only) (E)

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:

SemaphoreExample class This is the main class of the program It contains the main() method, which is the entry point of the program. It creates instances of the SharedResource class and multiple threads of the MyThread class to demonstrate synchronization using semaphores.

SharedResource class:This class represents a shared resource that needs to be accessed by multiple threads.It contains a static Semaphore object named semaphore, initialized with a permit count of 1.The useResource() method is defined to simulate the usage of the shared resource.Inside useResource(), the semaphore is acquired using semaphore.acquire(), ensuring that only one thread can access the shared resource at a time.

Some work is simulated using Thread.sleep(100) to represent work being done.

After completing the work, the semaphore is released using semaphore.release().

MyThread class: This class extends the Thread class and represents the threads that will access the shared resource. It has a reference to an instance of the SharedResource class. The run() method is overridden to call the useResource() method of the SharedResource object associated with the thread.main() method:

In the main() method, an instance of the SharedResource class is created.

Multiple instances of the MyThread class are created and associated with the same SharedResource object.

The threads are started using the start() method.

The join() method is called on each thread to wait for them to finish executing before proceeding with further instructions.

Thread Synchronization:

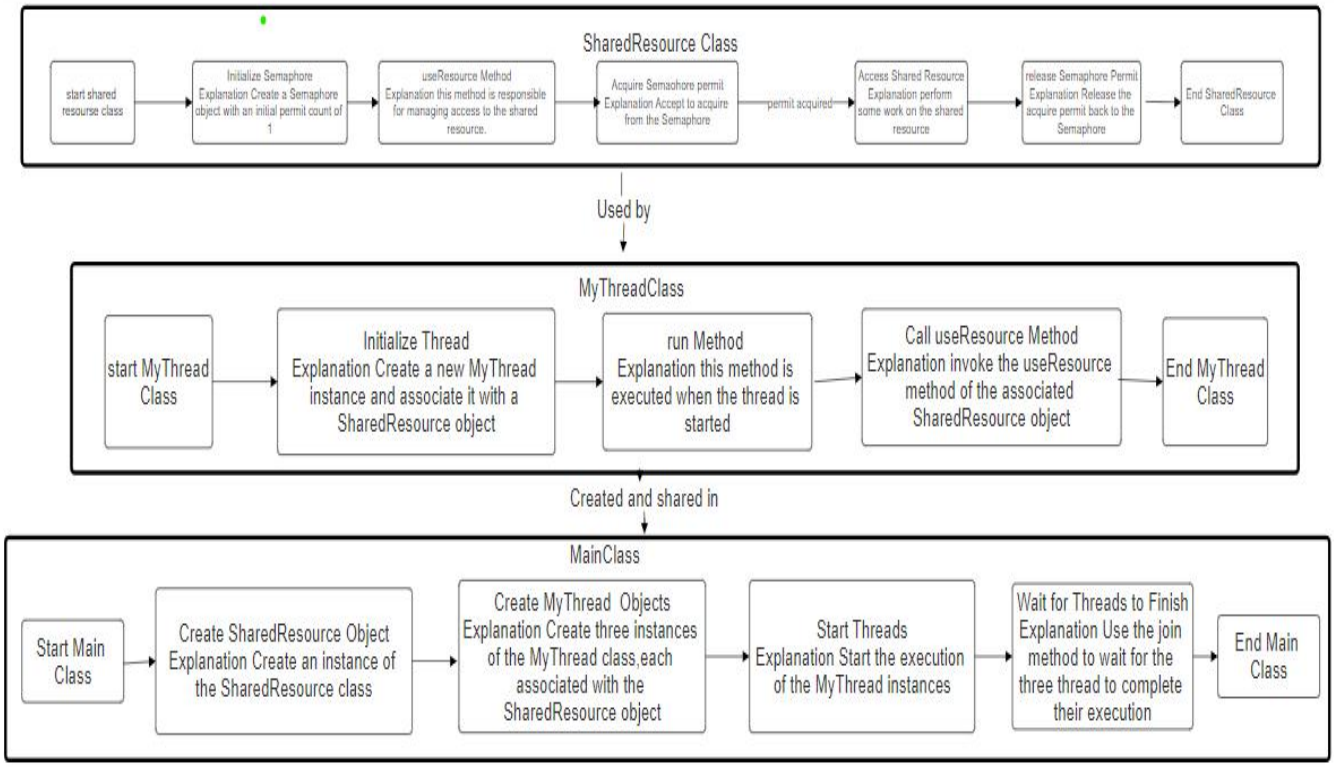
The Semaphore is used to synchronize access to the sharedr esource.

With a permit count of 1, only one thread can acquire the semaphore at a time, ensuring mutual exclusion.

Once a thread has finished using the shared resource, it releases the semaphore, allowing another waiting thread to acquire it and access the resource.

Overall, this program demonstrates how to use semaphores in Java to synchronize access to shared resources among multiple threads, ensuring thread safety and preventing race conditions.

Flowchart:



Program:

```
import java.util.concurrent.Semaphore;

class SharedResource {
   static Semaphore semaphore = new Semaphore(1);

   static void useResource() {
      try {
         semaphore.acquire();
         System.out.println(Thread.currentThread().getName() + " is accessing the shared
resource.");
         Thread.sleep(100); // Simulating some work being done
      } catch (InterruptedException e) {
         e.printStackTrace();
      } finally {
         System.out.println(Thread.currentThread().getName() + " has released the shared
resource.");
         semaphore.release();
      }
   }
```

```
}

class MyThread extends Thread {
   SharedResource resource;

   public MyThread(SharedResource resource) {
      this.resource = resource;
   }

   public void run() {
      resource.useResource();
   }
}

public class SemaphoreExample {
   public static void main(String[] args) {
      SharedResource resource = new SharedResource();

      // Creating multiple threads
      MyThread thread1 = new MyThread(resource);
      MyThread thread2 = new MyThread(resource);
      MyThread thread3 = new MyThread(resource);

      // Start the threads
      thread1.start();
      thread2.start();
      thread3.start();

      // Wait for threads to finish using join()
      try {
         thread1.join();
         thread2.join();
         thread3.join();
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }
}
```

Output:

```
$ gedit  SemaphoreExample.java
$ Javac SemaphoreExample.java
$ Java SemaphoreExample
Thread-1 is accessing the shared resource.
```

Thread-1 has released the shared resource.
Thread-2 is accessing the shared resource.
Thread-2 has released the shared resource.
Thread-0 is accessing the shared resource.
Thread-0 has released the shared resource.

Conclusion:

The provided Java code demonstrates the usage of semaphores for thread synchronization. It consists of a SharedResource class representing a shared resource, a MyThread class representing threads accessing the resource, and a SemaphoreExample class orchestrating the threads. Semaphores are used to ensure that only one thread accesses the shared resource at a time, preventing race conditions and ensuring thread safety. Through the use of semaphores, the program demonstrates controlled access to the shared resource, allowing threads to execute concurrently while avoiding conflicts.

References:

1] https://chat.openai.com/c/4e69cd0f-e135-4643-9420-94bfe9d11ea2
2] https://youtu.be/Kk2UjpOPT-Y?si=iLNRREHZElHdF7gW

Assignment 5A

Title- Write a program to implement a shell script for calculator

Objectives –

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

Theory-

A shell in a Linux operating system takes input from you in the form of commands, processes it, and then gives an output. It is the interface through which a user works on the programs, commands, and scripts. A shell is accessed by a terminal which runs it.

When you run the terminal, the Shell issues a command prompt (usually $), where you can type your input, which is then executed when you hit the Enter key. The output or the result is thereafter displayed on the terminal.

The Shell wraps around the delicate interior of an Operating system protecting it from accidental damage. Hence the name Shell.

Flowchart-

Assignment 3B_b

Title - Write a program to lock the file using flock system call File locking system call : fnctl.h: flock/lockf

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

Theory -

File Locking with flock

Problem: Multiple processes accessing the same file concurrently can lead to data corruption.

Solution: File locking using flock (advisory locking) helps prevent this.

Steps:

Lock Acquisition:

Process opens the file.
flock is used to acquire an exclusive lock (prevents other processes from modifying the file).
If the lock is already held, the process might block or fail depending on configuration.

Critical Section:

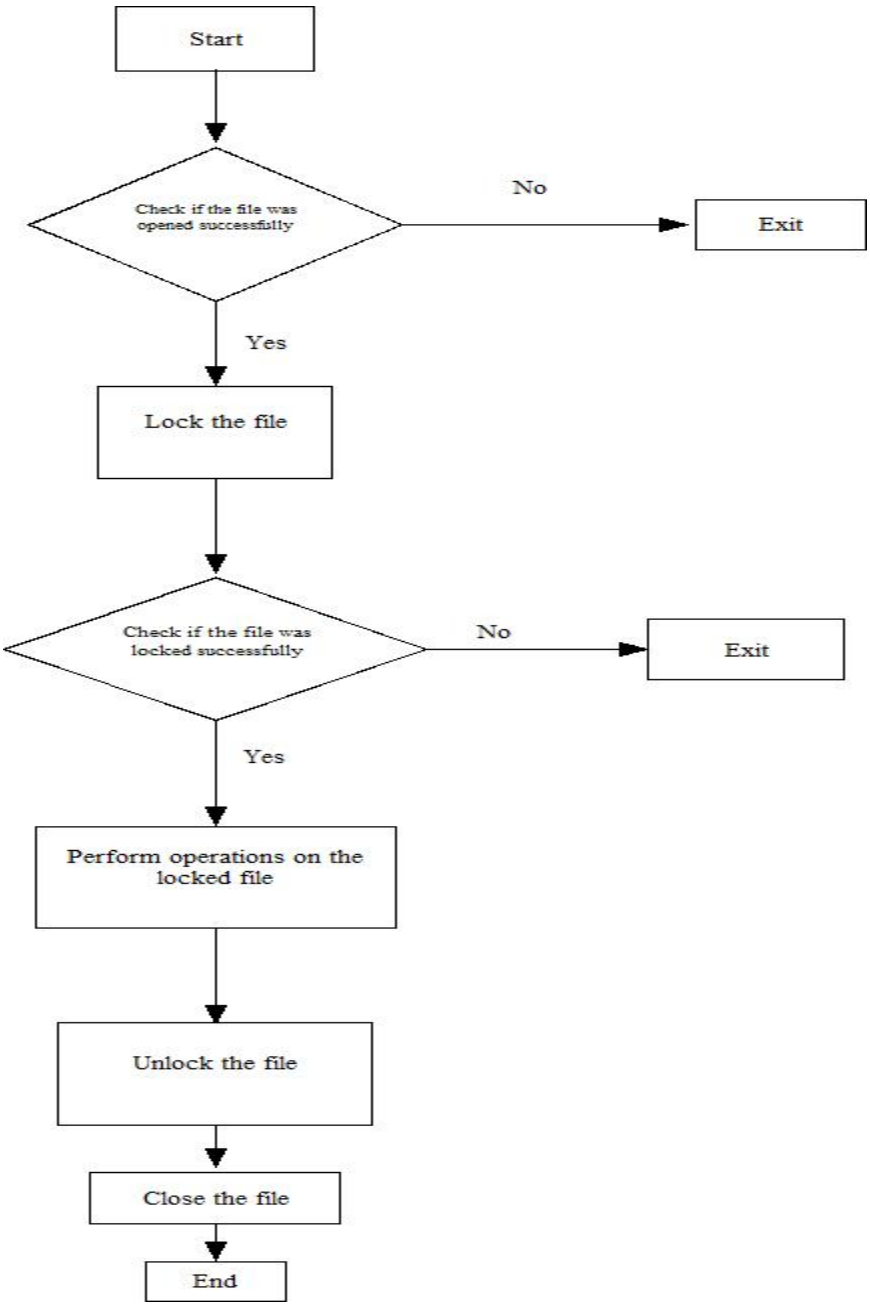
The process has exclusive access to the file and can safely read or write data.

Unlocking:

flock is used again to release the lock, allowing other processes to access the file.
Benefits:

Ensures data integrity when multiple processes access the file.
Improves coordination and avoids race conditions.

Flowchart -



Code -

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

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

void lock_file(int fd) {
    struct flock lock;
    lock.l_type = F_WRLCK;    // Exclusive write lock
    lock.l_whence = SEEK_SET; // Start of file
    lock.l_start = 0;         // Starting offset
    lock.l_len = 0;           // Whole file
    lock.l_pid = getpid();    // PID of process

    if (fcntl(fd, F_SETLKW, &lock) == -1) {
        perror("fcntl");
        exit(EXIT_FAILURE);
    }

    printf("File locked successfully.\n");
}

void unlock_file(int fd) {
    struct flock unlock;
    unlock.l_type = F_UNLCK;  // Unlock
    unlock.l_whence = SEEK_SET;
    unlock.l_start = 0;
    unlock.l_len = 0;
    unlock.l_pid = getpid();

    if (fcntl(fd, F_SETLK, &unlock) == -1) {
        perror("fcntl");
        exit(EXIT_FAILURE);
    }

    printf("File unlocked.\n");
}

int main() {
    const char *filename = "test.txt";
    int fd;

    // Open the file
    if ((fd = open(filename, O_RDWR | O_CREAT, 0666)) == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }
```

```
    // Lock the file
    lock_file(fd);

    // Perform operations on the locked file
    printf("Performing operations on the locked file...\n");

    // Unlock the file
    unlock_file(fd);

    // Close the file
    close(fd);

    return 0;
}
```

Output -

File locked successfully.
Performing operations on the locked file…
File unlocked.

Conclusion -

Lock: The program acquires an exclusive lock on the file, ensuring only one process can modify it at a time.
Work: The program safely reads from or writes to the file within this locked section.

Unlock: When finished, the program releases the lock, allowing other processes to access the file.

Assignment 5C

Title: 1.To learn shell programming and use it for write effective programs.Using shell sort the given 10 number in ascending order (use of array). (B)

Theory: In a lab manual, you'd typically want to provide clear instructions along with some theoretical background to help students understand the purpose and principles behind the algorithm being implemented. Below is a suggested outline for the theory section of your lab manual for the Shell Sort algorithm

Introduction

Shell Sort, also known as Shell's method, is an in-place comparison-based sorting algorithm. It is an extension of insertion sort with the aim of improving its performance, particularly for larger datasets.

Algorithm Overview
Shell Sort starts by sorting pairs of elements far apart from each other, then progressively reducing the gap between elements to be compared. The basic idea is to rearrange the array so that, starting anywhere, considering every \(h^\text{th}\) element gives a sorted subsequence. The gaps between elements are gradually reduced until the final pass when the gap becomes 1, effectively performing an insertion sort.

How Shell Sort Works
1. **Gap Sequence Selection**: Shell Sort begins by selecting a sequence of gap values. Common sequences include the original sequence proposed by Donald Shell (incremented halving method) and other variants.

2. **Sorting Passes**: Shell Sort performs multiple sorting passes, each time using a different gap value. In each pass, elements that are 'gap' distance apart are compared and swapped if necessary to partially sort the array.

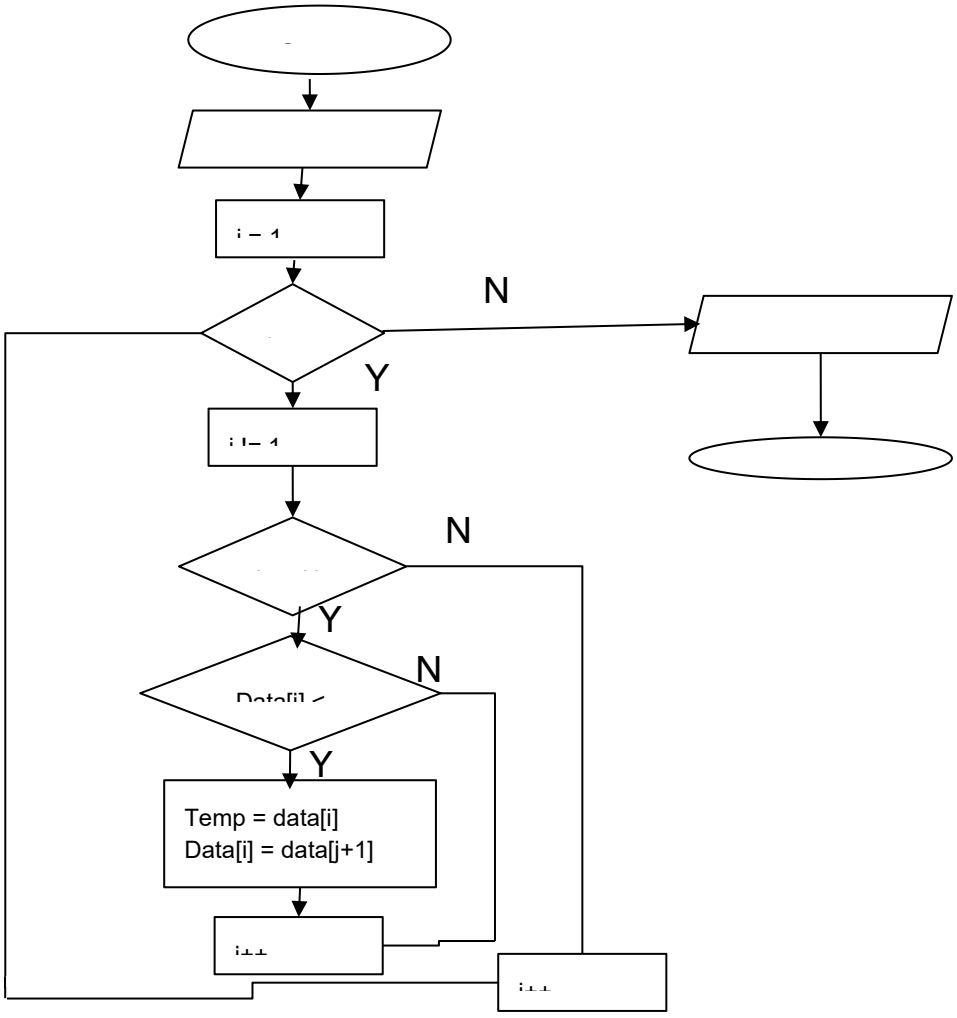
3. **Final Pass**: The final pass of the algorithm is essentially an insertion sort with a gap of 1. At this point, the array is nearly sorted, and a single insertion sort pass efficiently finishes the sorting process.

Time Complexity
The time complexity of Shell Sort depends on the choice of gap sequence. Although its time complexity is not as good as some other efficient sorting algorithms like Merge Sort or Quick Sort, it can still perform better than the simpler algorithms like Bubble Sort or Insertion Sort for medium-sized datasets. The worst-case time complexity of Shell Sort is (O(n^2)), but with optimal gap sequences, it can approach (O(n log n)).

Space Complexity
Shell Sort is an in-place sorting algorithm, meaning it does not require additional space proportional to the size of the input array.
Data Dictionary:
numbers: Array of integers to be sorted.
length: Length of the numbers array.
gap: Gap between elements to be compared during each sorting pass.
temp: Temporary storage for a value during sorting.
i: Loop counter for iterating over elements.

j: Loop counter for comparisons within a pass.

Flowchart(Photo+URL):
https://drive.google.com/drive/folders/1UHzim_45uAaB3gkrGlWX2NXR8qnJat0C



Code:

File name :: **shell_sort.sh**

#!/bin/bash

```
# Define the array of numbers
numbers=(9 5 7 2 1 8 4 6 3 10)


# Define the length of the array
length=${#numbers[@]}


# Perform shell sort
for (( gap = $length / 2; gap > 0; gap /= 2 )); do
  for (( i = gap; i < length; i++ )); do
    temp=${numbers[$i]}
    j=$i
    while (( j >= gap && numbers[j - gap] > temp )); do
      numbers[j]=${numbers[j - gap]}
      (( j -= gap ))
    done
    numbers[j]=$temp
  done
done


# Print the sorted array
echo "Sorted numbers in ascending order:"
for (( i = 0; i < length; i++ )); do
  echo "${numbers[i]}"
done
```

Output:

Use commands

1. chmod +x shell_sort.sh
2. ./shell_sort.sh

```
chmod +x shell_sort.sh
./shell_sort.sh Sorted numbers in ascending order:
1
2
3
4
5
6
7
```

8
9
10

Assignment 5D

Title: Shell script to print "Hello World" message, in Bold, Blink effect, and in different colors like red, brown etc. (B)

Objectives :

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

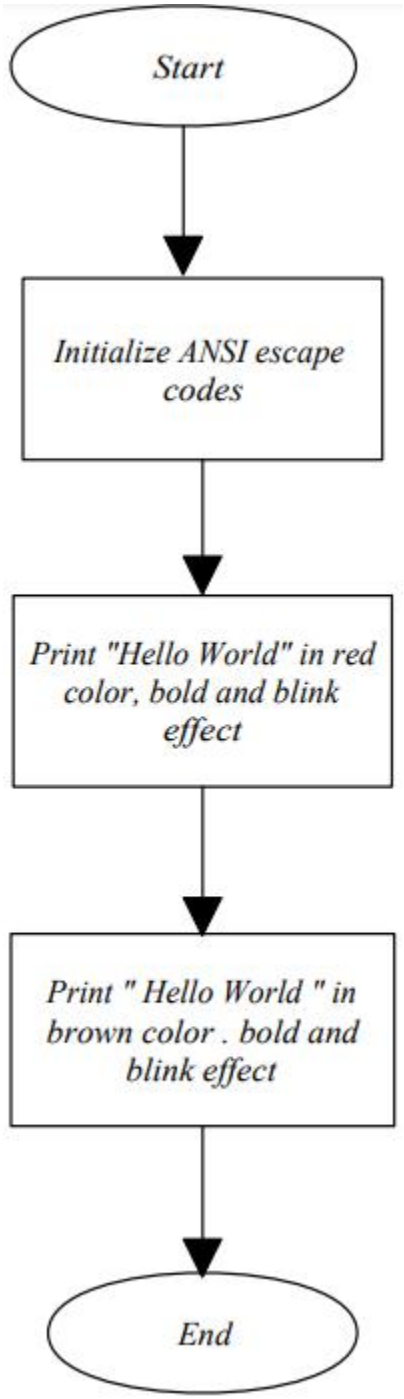
Theory: Shell scripting is a powerful way to automate tasks and write effective programs by leveraging the capabilities of the Unix shell. The Unix shell provides a command-line interface to interact with the operating system. Shell scripts are text files that contain a sequence of shell commands and can be executed to perform various tasks.

To print messages with special formatting like bold, blink effect, and different colors, we can use ANSI escape codes. ANSI escape codes are sequences of characters that are embedded in the text to control formatting, color, and other text attributes when printed to the terminal.

Data Dictionary :

Variables	Description
Bold	Escape code for bold text formatting.
Blink	Escape code for text with blinking effect.
Reset	Escape code for resetting text attributes to default.

Flow Chart:



Code:

```
#!/bin/bash
# Bold text escape code
bold=$(tput bold)
# Blink text escape code
```

```
blink=$(tput blink)
# Reset all text attributes escape code
reset=$(tput sgr0)
# Print "Hello World" in red color, bold, and blink effect
echo -e "${blink}${bold}\033[31mHello World${reset}"
# Print "Hello World" in brown color, bold, and blink effect
echo -e "${blink}${bold}\033[33mHello World${reset}"
```

Output :

```
$ bash ./5b.sh

Hello World
Hello World
```

Conclusion : Shell scripting allows for automation and customization of various tasks in a Unix environment. By utilizing ANSI escape codes, we can enhance the visual appearance of text in shell scripts, making them more interactive and user-friendly. This simple "Hello World" script demonstrates the use of ANSI escape codes to print text with
different attributes such as bold, blink effect, and colors. With further exploration and practice, one can create more complex and useful shell scripts for various purposes.

References :

https://www.tutorialspoint.com/unix/unix-what-is-shell.htm/

Assignment 6A

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

Objective:

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:

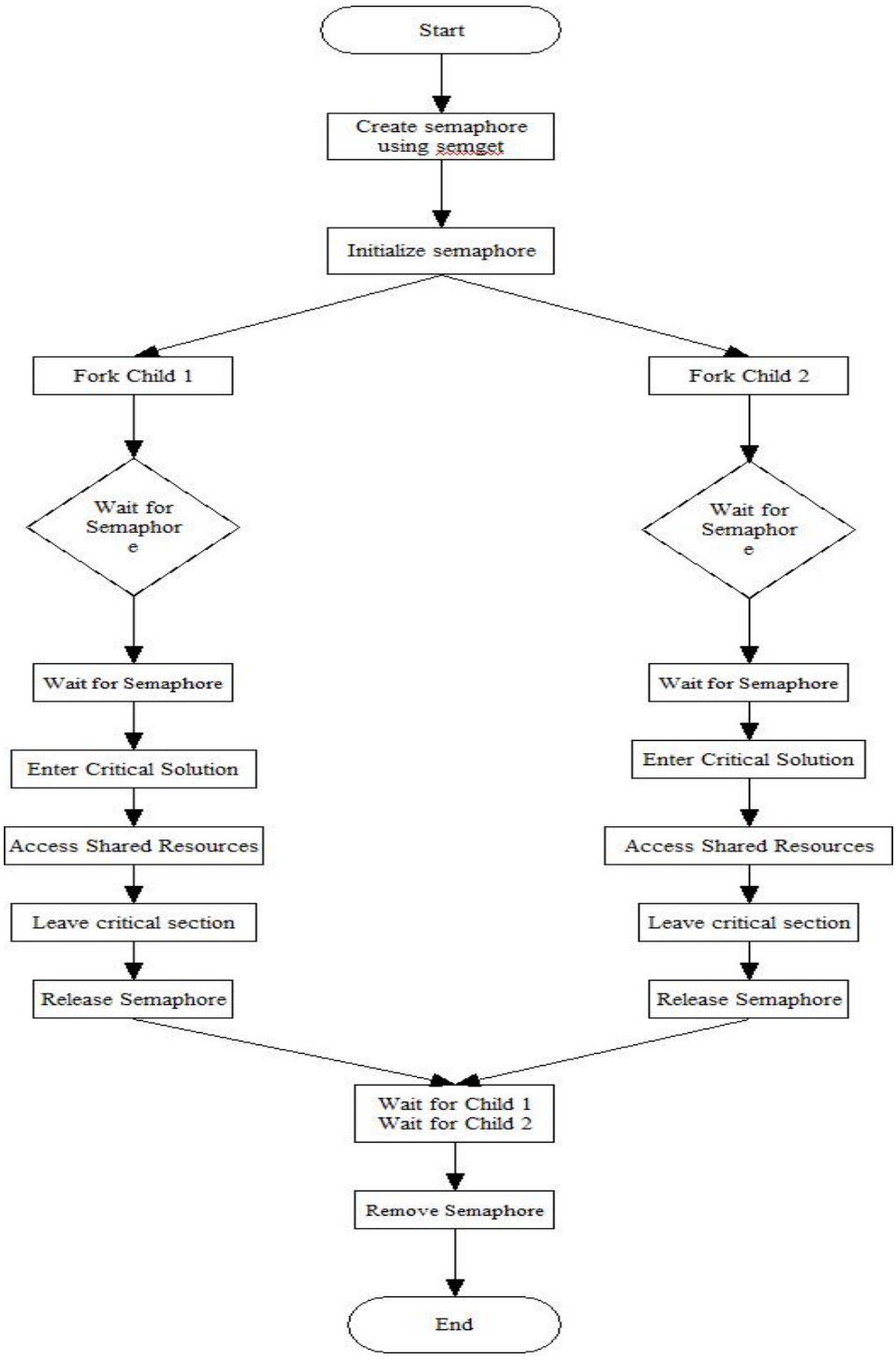
- sem_open() -- Connects to, and optionally creates, a named semaphore
- sem_init() -- Initializes a semaphore structure (internal to the calling program, so not a named semaphore).
- sem_close() -- Ends the connection to an open semaphore.
- sem_unlink() -- Ends the connection to an open semaphore and 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	DataType	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

Flowchart:



Program-

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

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

#define SEM_NAME "/my_semaphore"

int main() {
    sem_t *sem;
    pid_t pid;

    // Create the semaphore
    sem = sem_open(SEM_NAME, O_CREAT, 0644, 1);
    if (sem == SEM_FAILED) {
        perror("Failed to create semaphore");
        exit(EXIT_FAILURE);
    }

    // Fork two processes
    pid = fork();
    if (pid == -1) {
        perror("Failed to fork process");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        // Child process
        printf("Child process waiting for semaphore...\n");
        sem_wait(sem); // Acquire the semaphore
        printf("Child process acquired semaphore\n");
        sleep(2);
        sem_post(sem); // Release the semaphore
        printf("Child process released semaphore\n");
        exit(EXIT_SUCCESS);
    } else {
        // Parent process
        printf("Parent process waiting for semaphore...\n");
        sem_wait(sem); // Acquire the semaphore
        printf("Parent process acquired semaphore\n");
        sleep(2);
        sem_post(sem); // Release the semaphore
        printf("Parent process released semaphore\n");
        wait(NULL);
        sem_unlink(SEM_NAME); // Destroy the semaphore
        exit(EXIT_SUCCESS);}
    }
```

Output-

./semaphore
Parent process waiting for semaphore…
Parent process acquired semaphore
Child process waiting for semaphore…
Parent process released semaphore

Child process acquired semaphore

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

Reference-

Dave's Programming in C Tutorials

Assignment 6B

Title:

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

Assignment 6C

Title: Write two programs that will communicate both ways (i.e each process can read and write) when run concurrently via semaphores.

Theory:

A semaphore is like a counter that helps processes coordinate their actions. It contains a number, typically an integer, and allows processes to work together without stepping on each other's toes.

Imagine you're in a line waiting for something. The number of people ahead of you represents the semaphore value. When it's your turn, you decrease the count by one as you move forward. If the count is zero, you have to wait until someone else finishes and increases the count again.

There are two main actions you can perform with a semaphore: wait and signal.

1.Wait: You check if the count is greater than zero. If it is, you decrease the count and proceed. If not, you wait until it's your turn.

2.Signal: You increase the count, indicating that you've finished your task and allowing others to proceed if they were waiting.

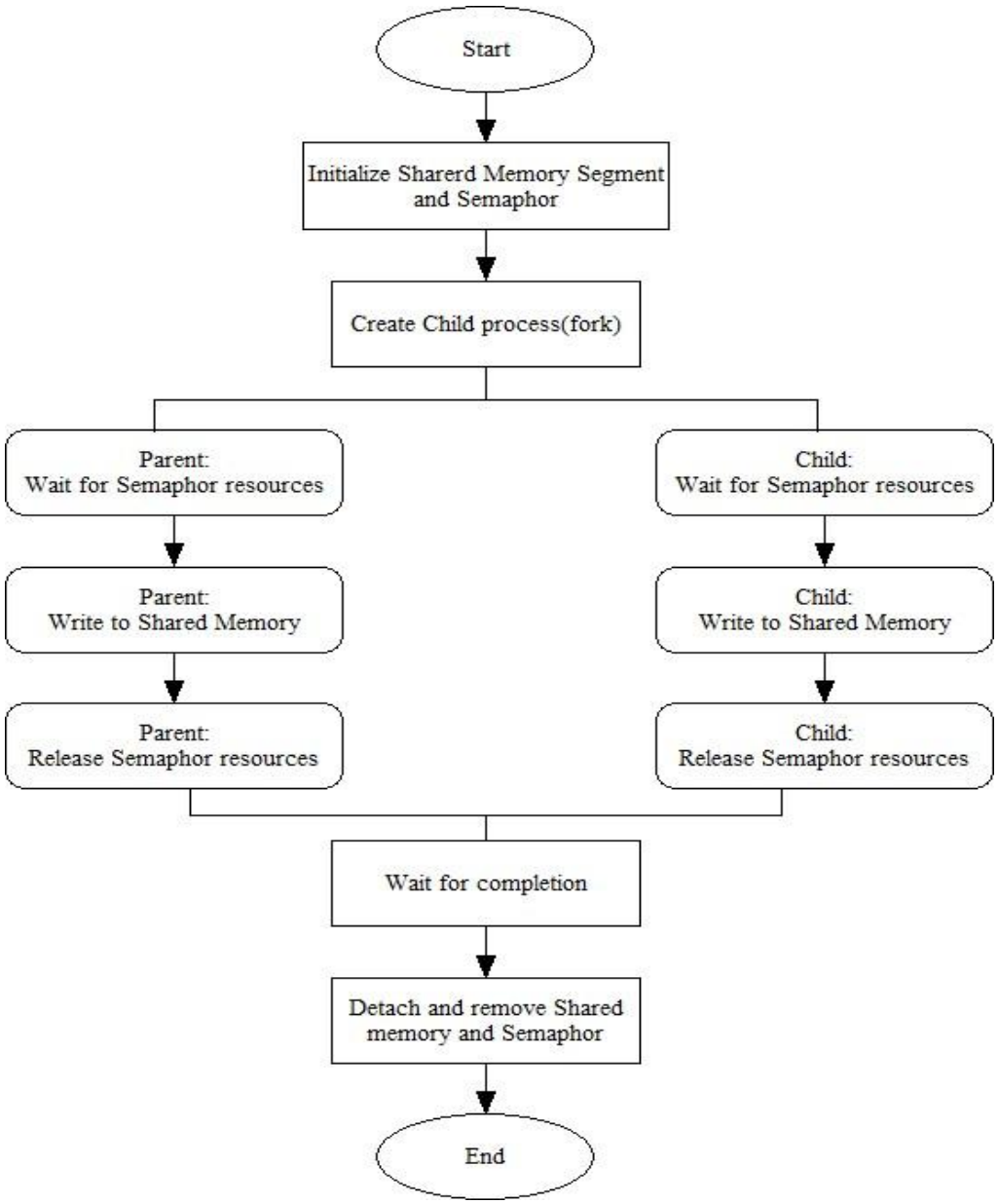
This mechanism ensures that only one process can modify the semaphore at a time, avoiding conflicts.

Semaphores can also be grouped together in a semaphore set, which allows processes to perform coordinated actions on multiple semaphores at once. This ensures that either all actions in the group succeed or none do, providing consistency and reliability. However, it's important to note that semaphore sets are specific to certain systems and are not a universal concept in parallel programming.

Data Dictionary:

1. SHM_KEY (Shared Memory Key): Identifier for accessing shared memory.
2. SEM_KEY (Semaphore Key): Identifier for semaphore synchronization.
3. shmseg Structure: Defines shared memory layout.
4. retval: A variable storing the return value of the semop function.

Flowchart(Photo+URL):



1. IPC_sem_write.c

```
#include<stdio.h>
#include<sys/shm.h>
#include<sys/types.h>
#include<string.h>
#include<errno.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#define SHM_KEY 0x12345
struct shmseg {
```

```
  int cntr;
  int write_complete;
  int read_complete;
};
void shared_memory_cntr_increment(int pid, struct shmseg *shmp, int total_count);
int main(int argc, char *argv[]) {
  int shmid;
  struct shmseg *shmp;
  char *bufptr;
  int total_count;
  int sleep_time;
  pid_t pid;
  if (argc != 2)
  total_count = 10000;
  else {
        total_count = atoi(argv[1]);
        if (total_count < 10000)
        total_count = 10000;
  }
  printf("Total Count is %d\n", total_count);
  shmid = shmget(SHM_KEY, sizeof(struct shmseg), 0644|IPC_CREAT);
  if (shmid == -1) {
        perror("Shared memory");
        return 1;
  }
  // Attach to the segment to get a pointer to it.
  shmp = shmat(shmid, NULL, 0);
  if (shmp == (void *) -1) {
        perror("Shared memory attach");
        return 1;
  }
  shmp->cntr = 0;
  pid = fork();

  /* Parent Process - Writing Once */
  if (pid > 0) {
        shared_memory_cntr_increment(pid, shmp, total_count);
  } else if (pid == 0) {
        shared_memory_cntr_increment(pid, shmp, total_count);
        return 0;
  } else {
        perror("Fork Failure\n");
        return 1;
  }
  while (shmp->read_complete != 1)
  sleep(1);

  if (shmdt(shmp) == -1) {
        perror("shmdt");
        return 1;
  }
```

```
  if (shmctl(shmid, IPC_RMID, 0) == -1) {
        perror("shmctl");
        return 1;
  }
  printf("Writing Process: Complete\n");
  return 0;
}

/* Increment the counter of shared memory by total_count in steps of 1 */
void shared_memory_cntr_increment(int pid, struct shmseg *shmp, int total_count) {
  int cntr;
  int numtimes;
  int sleep_time;
  cntr = shmp->cntr;
  shmp->write_complete = 0;
  if (pid == 0)
  printf("SHM_WRITE: CHILD: Now writing\n");
  else if (pid > 0)
  printf("SHM_WRITE: PARENT: Now writing\n");
  //printf("SHM_CNTR is %d\n", shmp->cntr);

  /* Increment the counter in shared memory by total_count in steps of 1 */
  for (numtimes = 0; numtimes < total_count; numtimes++) {
        cntr += 1;
        shmp->cntr = cntr;

        /* Sleeping for a second for every thousand */
        sleep_time = cntr % 1000;
        if (sleep_time == 0)
        sleep(1);
  }
  shmp->write_complete = 1;
  if (pid == 0)
  printf("SHM_WRITE: CHILD: Writing Done\n");
  else if (pid > 0)
  printf("SHM_WRITE: PARENT: Writing Done\n");
  return;
}
```

Output:

```
$ gedit IPC_sem_write.c

$ gcc IPC_sem_write.c -o IPC_sem_write -lpthread

$ ./IPC_sem_write

Total Count is 10000

SHM_WRITE: PARENT: Now writing
```

SHM_WRITE: CHILD: Now writing

SHM_WRITE: PARENT: Writing Done

SHM_WRITE: CHILD: Writing Done

2. IPC_sem_read.c

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/sem.h>
#include<string.h>
#include<errno.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>

#define SHM_KEY 0x12345
#define SEM_KEY 0x54321
#define MAX_TRIES 20

struct shmseg {
  int cntr;
  int write_complete;
  int read_complete;
};
void shared_memory_cntr_increment(int, struct shmseg*, int);
void remove_semaphore();

int main(int argc, char *argv[]) {
  int shmid;
  struct shmseg *shmp;
  char *bufptr;
  int total_count;
  int sleep_time;
  pid_t pid;
  if (argc != 2)
  total_count = 10000;
  else {
        total_count = atoi(argv[1]);
        if (total_count < 10000)
        total_count = 10000;
  }
  printf("Total Count is %d\n", total_count);
  shmid = shmget(SHM_KEY, sizeof(struct shmseg), 0644|IPC_CREAT);
  if (shmid == -1) {
        perror("Shared memory");
```

```
        return 1;
    }
    // Attach to the segment to get a pointer to it.
    shmp = shmat(shmid, NULL, 0);

    if (shmp == (void *) -1) {
        perror("Shared memory attach: ");
        return 1;
    }
    shmp->cntr = 0;
    pid = fork();

    /* Parent Process - Writing Once */
    if (pid > 0) {
        shared_memory_cntr_increment(pid, shmp, total_count);
    } else if (pid == 0) {
        shared_memory_cntr_increment(pid, shmp, total_count);
        return 0;
    } else {
        perror("Fork Failure\n");
        return 1;
    }
    while (shmp->read_complete != 1)
    sleep(1);
    if (shmdt(shmp) == -1) {
        perror("shmdt");
        return 1;
    }
    if (shmctl(shmid, IPC_RMID, 0) == -1) {
        perror("shmctl");
        return 1;
    }
    printf("Writing Process: Complete\n");
    remove_semaphore();
    return 0;
}
/* Increment the counter of shared memory by total_count in steps of 1 */
void shared_memory_cntr_increment(int pid, struct shmseg *shmp, int total_count) {
    int cntr;
    int numtimes;
    int sleep_time;
    int semid;
    struct sembuf sem_buf;
    struct semid_ds buf;
    int tries;
    int retval;
    semid = semget(SEM_KEY, 1, IPC_CREAT | IPC_EXCL | 0666);
    //printf("errno is %d and semid is %d\n", errno, semid);
    /* Got the semaphore */
    if (semid >= 0) {
        printf("First Process\n");
```

```
        sem_buf.sem_op = 1;
        sem_buf.sem_flg = 0;
        sem_buf.sem_num = 0;
        retval = semop(semid, &sem_buf, 1);
        if (retval == -1) {
        perror("Semaphore Operation: ");
        return;
        }
} else if (errno == EEXIST) { // Already other process got it
        int ready = 0;
        printf("Second Process\n");
        semid = semget(SEM_KEY, 1, 0);
        if (semid < 0) {
        perror("Semaphore GET: ");
        return;
        }
        /* Waiting for the resource */
        sem_buf.sem_num = 0;
        sem_buf.sem_op = 0;
        sem_buf.sem_flg = SEM_UNDO;
        retval = semop(semid, &sem_buf, 1);
        if (retval == -1) {
        perror("Semaphore Locked: ");
        return;
        }
}
sem_buf.sem_num = 0;
sem_buf.sem_op = -1; /* Allocating the resources */
sem_buf.sem_flg = SEM_UNDO;
retval = semop(semid, &sem_buf, 1);

if (retval == -1) {
        perror("Semaphore Locked: ");
        return;
}
cntr = shmp->cntr;
shmp->write_complete = 0;
if (pid == 0)
printf("SHM_WRITE: CHILD: Now writing\n");
else if (pid > 0)
printf("SHM_WRITE: PARENT: Now writing\n");
//printf("SHM_CNTR is %d\n", shmp->cntr);

/* Increment the counter in shared memory by total_count in steps of 1 */
for (numtimes = 0; numtimes < total_count; numtimes++) {
        cntr += 1;
        shmp->cntr = cntr;
        /* Sleeping for a second for every thousand */
        sleep_time = cntr % 1000;
        if (sleep_time == 0)
        sleep(1);
```

```
  }
  shmp->write_complete = 1;
  sem_buf.sem_op = 1; /* Releasing the resource */
  retval = semop(semid, &sem_buf, 1);

  if (retval == -1) {
        perror("Semaphore Locked\n");
        return;
  }
  if (pid == 0)
        printf("SHM_WRITE: CHILD: Writing Done\n");
        else if (pid > 0)
        printf("SHM_WRITE: PARENT: Writing Done\n");
        return;
}
void remove_semaphore() {
  int semid;
  int retval;
  semid = semget(SEM_KEY, 1, 0);
        if (semid < 0) {
        perror("Remove Semaphore: Semaphore GET: ");
        return;
        }
  retval = semctl(semid, 0, IPC_RMID);
  if (retval == -1) {
        perror("Remove Semaphore: Semaphore CTL: ");
        return;
  }
  return;
}
```

Output:

aadi@ubuntu:~$ gedit IPC_sem_read.c

aadi@ubuntu:~$ gcc IPC_sem_read.c -o IPC_sem_read -lpthread

aadi@ubuntu:~$ ./IPC_sem_read

Total Count is 10000

First Process

SHM_WRITE: PARENT: Now writing

Second Process

SHM_WRITE: PARENT: Writing Done

SHM_WRITE: CHILD: Now writing

SHM_WRITE: CHILD: Writing Done

Now, we will check the counter value by the reading process.

Execution Steps

Reading Process: Shared Memory: Counter is 20000

Reading Process: Reading Done, Detaching Shared Memory

Reading Process: Complete

Conclusion:

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

References:

https://users.cs.cf.ac.uk/dave/C/node26.html#SECTION00260000000000000000

Assignment 6D

Title: Semaphore IPC Programming in C

Theory:
Semaphore is a synchronization mechanism used in inter-process communication (IPC) to control access to shared resources. It maintains a count of available resources and provides two fundamental operations: P (wait) and V (signal). P decrements the semaphore count, blocking if the count is zero, while V increments the count. Semaphore operations are atomic and thus suitable for coordinating access to shared resources among multiple processes.

Data Dictionary:

1. semget(): System call to create or access a semaphore set.
2. semctl(): System call to control semaphore operations.
3. semop(): System call to perform semaphore operations (P and V).

Flowchart:

Code:

1. **Program to initialize the semaphore and display the semaphore ID (sem_init.c):**

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#define KEY 1234 // Unique key for semaphore

int main() {
  int semid;

  // Create or access semaphore set with one semaphore
  semid = semget(KEY, 1, IPC_CREAT | 0666);
  if (semid == -1) {
     perror("semget");
     exit(1);
  }


  printf("Semaphore ID: %d\n", semid);
```

```
  return 0;
}
```

Output:

```
$ gcc semi_init.c
$ ./a.out
```

2. Program to perform the P operation (semaphore_wait.c):

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#define KEY 1234 // Same key used in sem_init.c

int main() {
  int semid;
  struct sembuf sem_op;

  // Access semaphore set created by first program
  semid = semget(KEY, 1, 0666);
  if (semid == -1) {
     perror("semget");
     exit(1);
  }
  // Perform P operation
  sem_op.sem_num = 0;
  sem_op.sem_op = -1;
  sem_op.sem_flg = 0;
  if (semop(semid, &sem_op, 1) == -1) {
     perror("semop");
     exit(1);
  }
  printf("Semaphore acquired.\n");

   return 0;
}
```

3. Program to perform the V operation (semaphore_signal.c):

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

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


#define KEY 1234 // Same key used in sem_init.c


int main() {
  int semid;
  struct sembuf sem_op;


  // Access semaphore set created by first program
  semid = semget(KEY, 1, 0666);
  if (semid == -1) {
     perror("semget");
     exit(1);
  }


  // Perform V operation
  sem_op.sem_num = 0;
  sem_op.sem_op = 1;
  sem_op.sem_flg = 0;
  if (semop(semid, &sem_op, 1) == -1) {
     perror("semop");
     exit(1);
  }


  printf("Semaphore released.\n");


  return 0;
}
```

OUTPUT for 2 and 3:
(Run 2nd program and open another terminal and run the 3rd program)
$ gcc -o semaphore_wait semaphore_wait.c

$ ./semaphore_wait
Semaphore acquired

$ gcc -o semaphore_signal semaphore_signal.c
$ ./semaphore_signal
Semaphore released

Conclusion:

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

References:

https://users.cs.cf.ac.uk/dave/C/node26.html#SECTION00260000000000000000

Assignment 7A

Title: Write a program to perform IPC using message and send did u get this

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

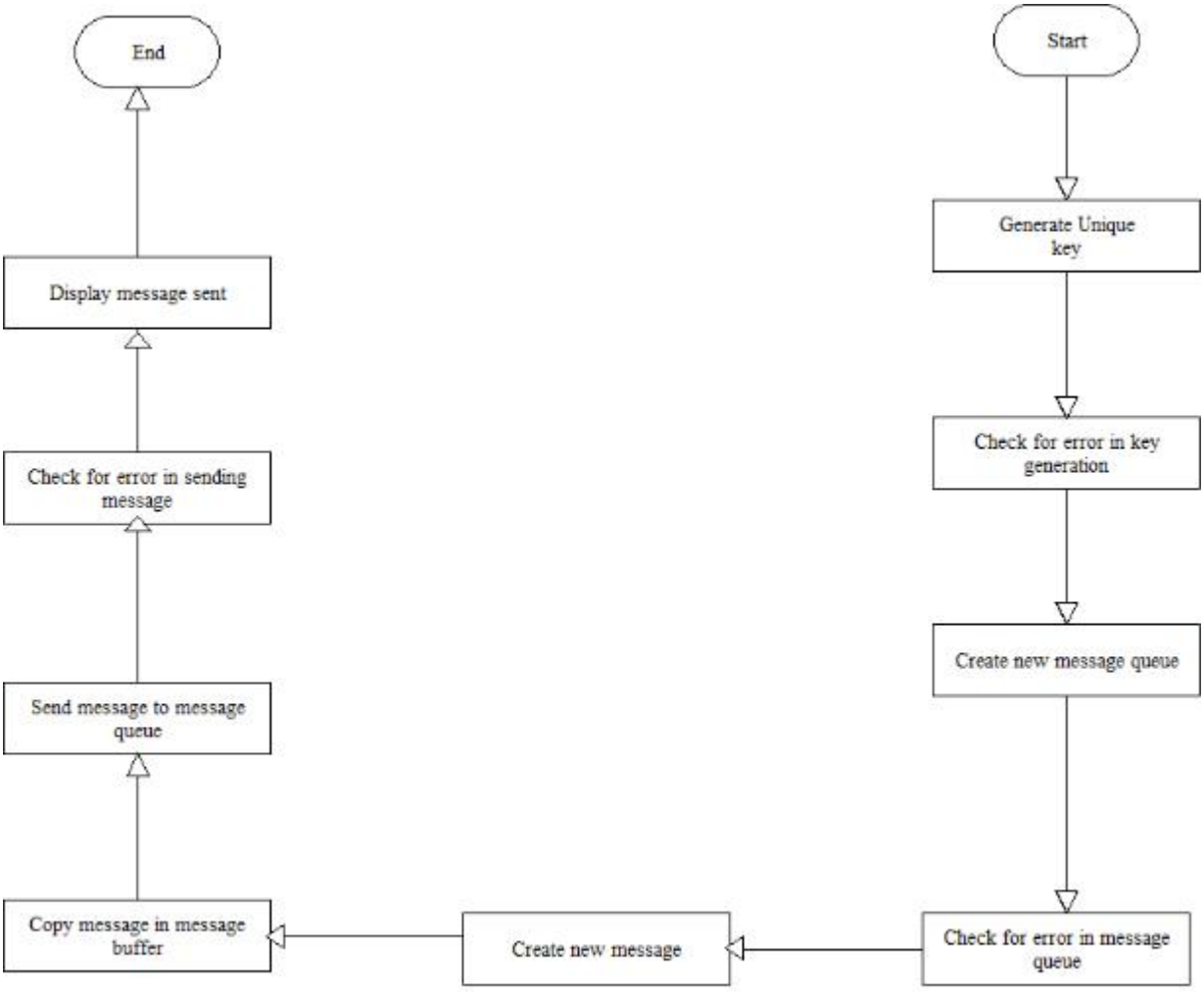
Each message is given an identification or type so that processes can select the appropriate message. Process must share a common key in order to gain access to the queue in the first place. Basic Message Passing IPC messaging lets processes send and receive messages, and queue messages for processing 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. 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.
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.

Data Dictionary:

Sr. No	Variable/Function	Data type	Use
1	msqid	int	For Sucket tuple
2	msgflg	int	For semaphore
3	key	key_t	Semaphore

Flowchart:



Program-
Messagesend.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <string.h>
#include<stdlib.h>
#define MSGSZ
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;
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);
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);
}
```

Messagreceive.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#define MSGSZ 128
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:

```
$  ./a.out

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
```

Assignment 7B

Title:

Write 2 programs that will both send and receive messages and construct the following dialog between them.

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

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.

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.

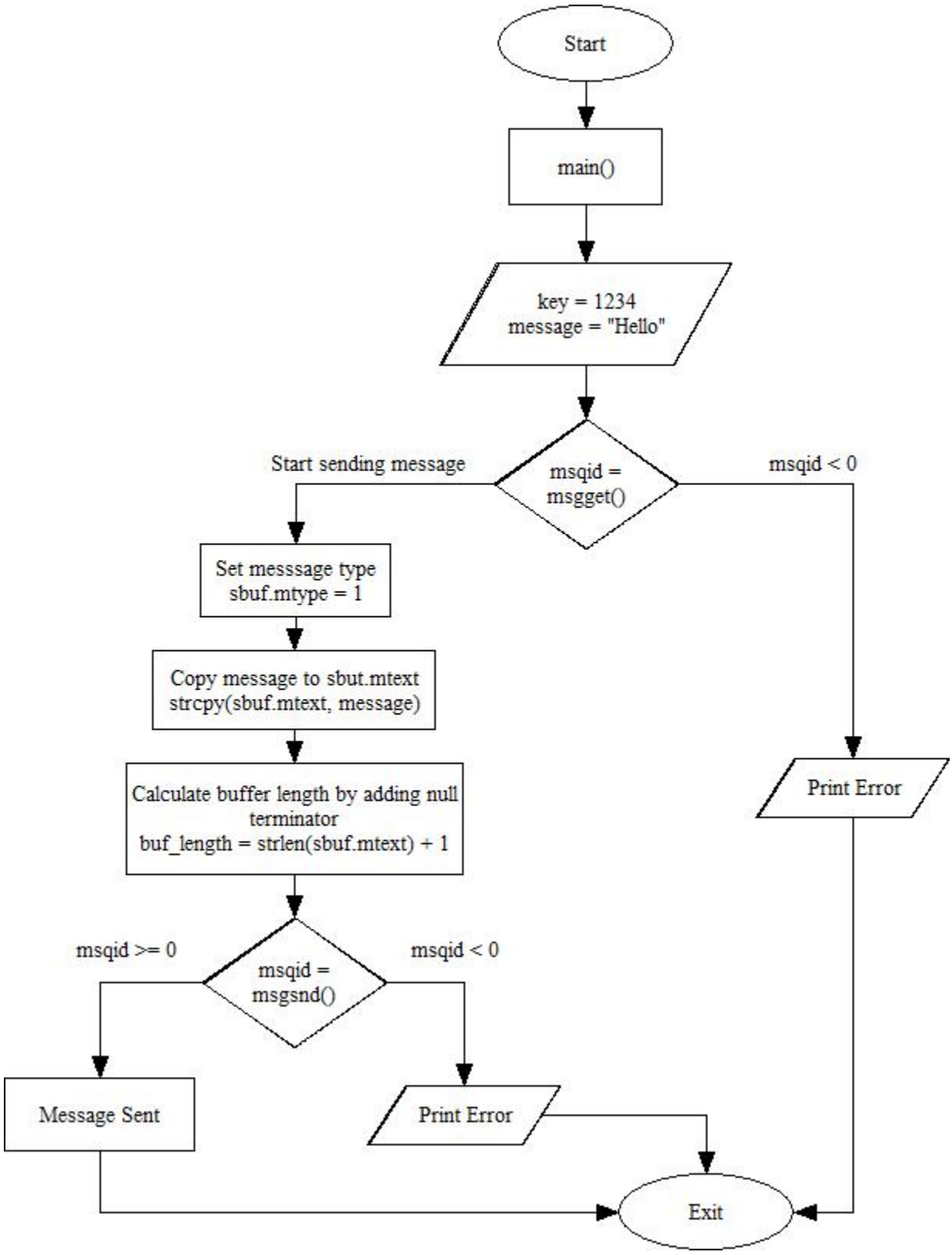
The owner or creator of a queue can change its ownership or permissions using msgctl() Also, any process with permission to do so can use msgctl() for control operations.

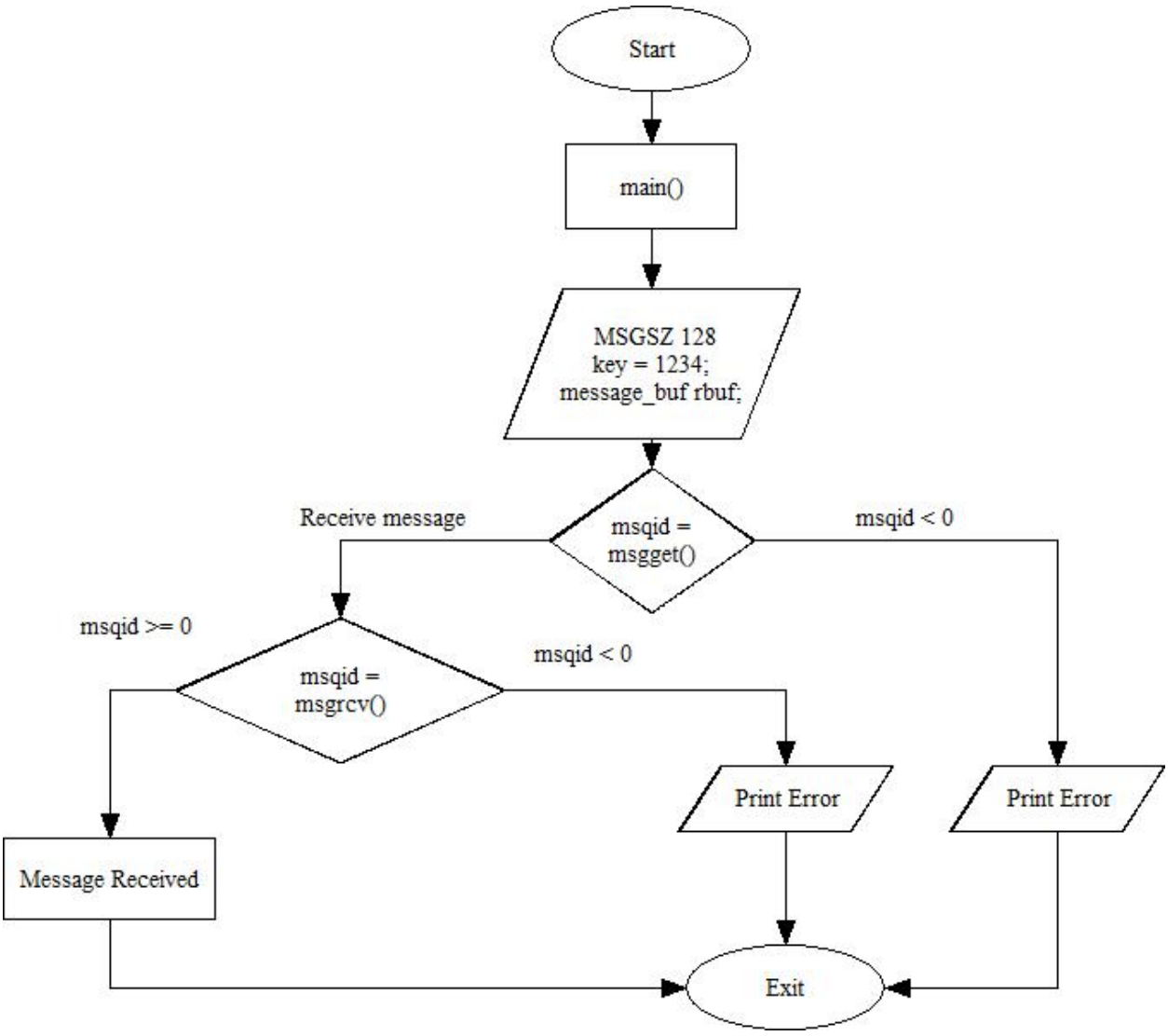
- int msgctl(int msqid, int cmd, struct msqid_ds *buf)

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.

Flowchart:


Start
main()
key = 1234
message = "Hello"
msqid = msgget()
Start sending message
msqid < 0
Set messsage type
sbuf.mtype = 1
Copy message to sbut.mtext
strcpy(sbuf.mtext, message)
Calculate buffer length by adding null terminator
buf_length = strlen(sbuf.mtext) + 1
Print Error
msqid >= 0
msqid = msgsnd()
msqid < 0
Message Sent
Print Error
Exit



Data Dictionary:

Variable / Function	Datatype / Return Type	Description
msqid	int	Message queue identifier
msgflg	int	Flags for message queue creation and permissions
key	key_t	Key used for accessing the message queue
sbuf	struct message_buf	Structure for sending messages
rbuf	struct message_buf	Structure for receiving messages
buf_length	size_t	Length of the message text
msgget(key, msgflg)	int	Initializes a new message queue or retrieves an existing one

msgsnd(msqid, &sbuf, buf_length, IPC_NOWAIT)	int	Sends a message to the message queue
msgrcv(msqid, &rbuf, MSGSZ, 1, 0)	int	Receives a message from the message queue
message_buf	struct	Structure representing a message with type and text content

Constants	Value/Datatype	Description
MSGSZ	int	Maximum size of the message text
IPC_NOWAIT	int	Flag indicating non-blocking message sending
0666	int	Permissions for message queue creation
1234	key_t	Key used for accessing the message queue

Code:

Program 1 (Sender):

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <string.h>
#define MSGSZ     128
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;
        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);
        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);
}
```

Program 2 (Receiver):

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#define MSGSZ     128
typedef struct msgbuf {
        long    mtype;
        char    mtext[MSGSZ];
} message_buf;

main()
{
        int msqid;
        key_t key;
```

```
        message_buf  rbuf;
        key = 1234;

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

        if (msgrcv(msqid, &rbuf, MSGSZ, 1, 0) < 0) {
        perror("msgrcv");
        exit(1);
        }

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

Output:
Program 1 (Sender):

```
$ nano sender.c && gcc sender.c && ./a.out
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
```

Program 2 (Receiver):

```
$ nano receiver.c && gcc receiver.c && ./a.out
Did you get this?
```

Conclusion:
The implementation of message queues for interprocess communication (IPC) involved the utilization of functions such as msgget, msgsnd, and msgrcv to establish a reliable and efficient messaging mechanism between two programs. Program 1 served as the message sender, while Program 2 acted as the receiver, demonstrating the basic usage of message queues for IPC. This process highlighted the capability of message queues to facilitate communication between processes, showcasing their fundamental role in building various types of applications, including but not limited to chat applications, where seamless exchange of information is crucial.

Assignment 7C

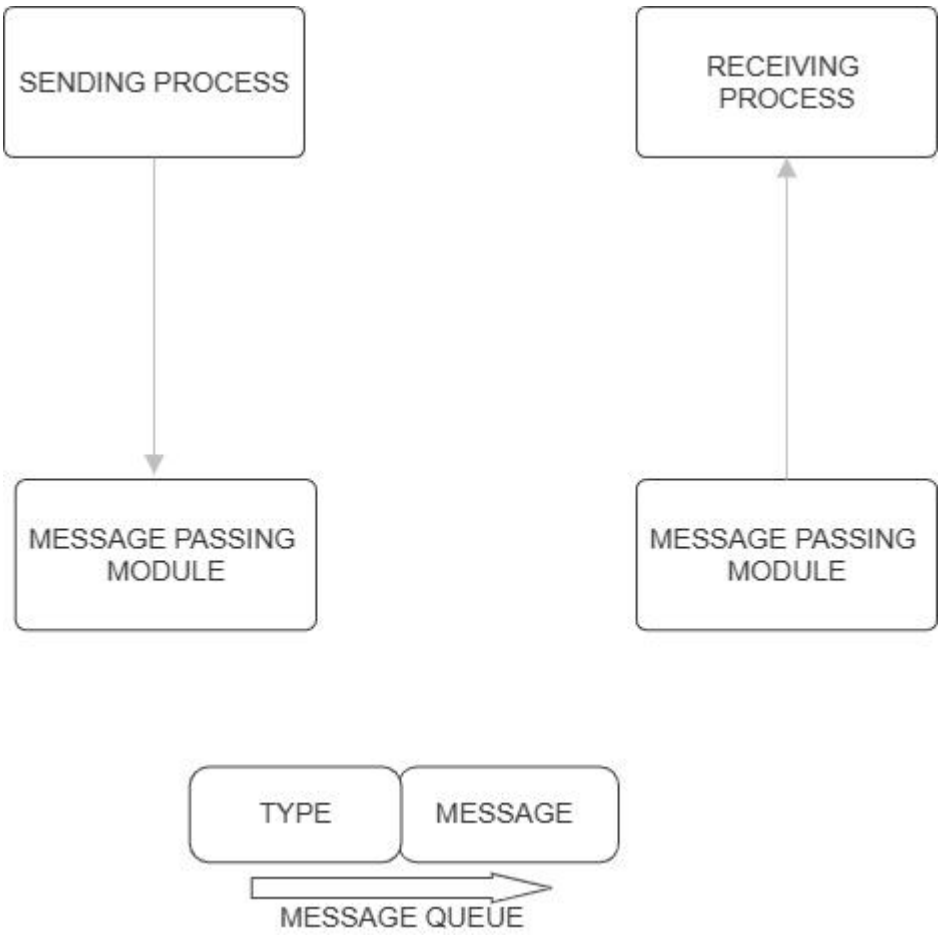
Title:
Interprocess Communication (IPC) using Message Queues for Private Communication

Theory:
Message queues facilitate interprocess communication (IPC) by allowing processes to exchange messages through a shared queue. In this scenario, two processes will communicate privately via a message queue. Process 1 sends a message, and Process 2 receives and replies, creating a private conversation.

Data Dictionary:

Queue ID: Identifier for the message queue
Message Type: Type of the message being sent
Message Content: Data contained within the message
Code:

Flowchart:



Program 1 (Process 1 - Sender):

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>

#define MSG_SIZE 128

// Structure for message
struct message_buffer {
   long message_type;
   char message_text[MSG_SIZE];
};

int main() {
   key_t key;
   int msg_id;
   struct message_buffer message;

   // Generate unique key
   key = ftok("progfile", 65);

   // Create message queue
   msg_id = msgget(key, 0666 | IPC_CREAT);

   // Send message
   strcpy(message.message_text, "Are you hearing me?");
   message.message_type = 1;
   msgsnd(msg_id, &message, sizeof(message), 0);

   printf("Process 1: Sent message: %s\n", message.message_text);

   // Receive reply
   msgrcv(msg_id, &message, sizeof(message), 2, 0);
   printf("Process 1: Received reply: %s\n", message.message_text);

   // Close message queue
   msgctl(msg_id, IPC_RMID, NULL);
```

```
    return 0;
}
```

Program 2 (Process 2 - Receiver):

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>

#define MSG_SIZE 128

// Structure for message
struct message_buffer {
    long message_type;
    char message_text[MSG_SIZE];
};

int main() {
    key_t key;
    int msg_id;
    struct message_buffer message;

    // Generate unique key
    key = ftok("progfile", 65);

    // Get message queue ID
    msg_id = msgget(key, 0666 | IPC_CREAT);

    // Receive message
    msgrcv(msg_id, &message, sizeof(message), 1, 0);
    printf("Process 2: Received message: %s\n", message.message_text);

    // Reply to message
    strcpy(message.message_text, "Loud and Clear");
    message.message_type = 2;
    msgsnd(msg_id, &message, sizeof(message), 0);

    return 0;
}
```

Output:
Process 1:
Process 1: Sent message: Are you hearing me?
Process 1: Received reply: Loud and Clear
Process 2:

arduino
Process 2: Received message: Are you hearing me?

Conclusion:
The two processes communicate privately using a message queue. Process 1 sends a message, Process 2 receives and replies, and then Process 1 receives the reply, completing the private conversation. This demonstrates the use of message queues for interprocess communication in a private setting.

Assignment 8A

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.

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 the 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 another process, process1 generates information about certain computations or resources being used and keeps it as a record in shared memory. When process2 needs 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 processes as well as for delivering any specific information to other processes.

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	Use

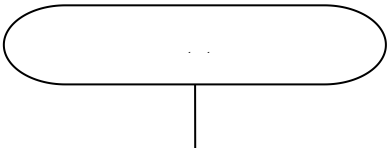
		Type	
1	shmid	int	Store value of identifier of System V shared memory
2	key	key_t	Used to pass the key to shmget

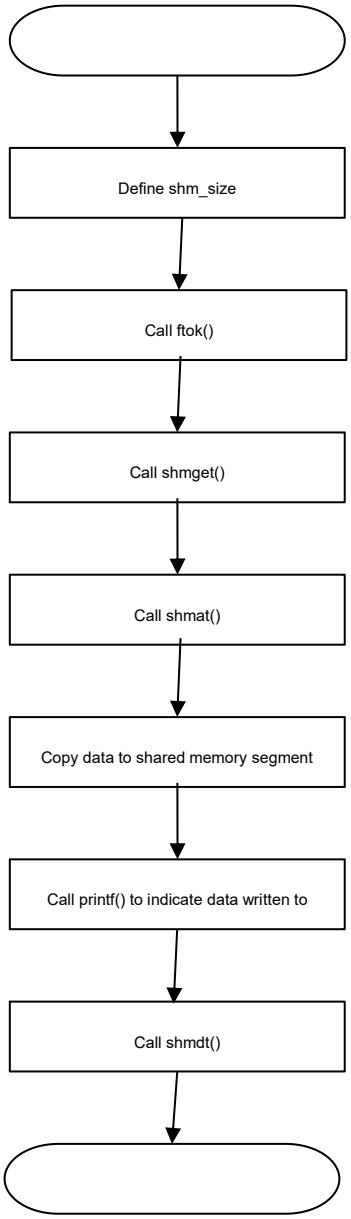
Server:

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

Flowchart:





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:

abcdefghijklmnopqrstuvwxyz

Conclusion:

1.Memory shared between client and server using IPC-SHM functions.
2.The data placed can be accessed by both.

References:

[1] Dave's Programming in C Tutorials

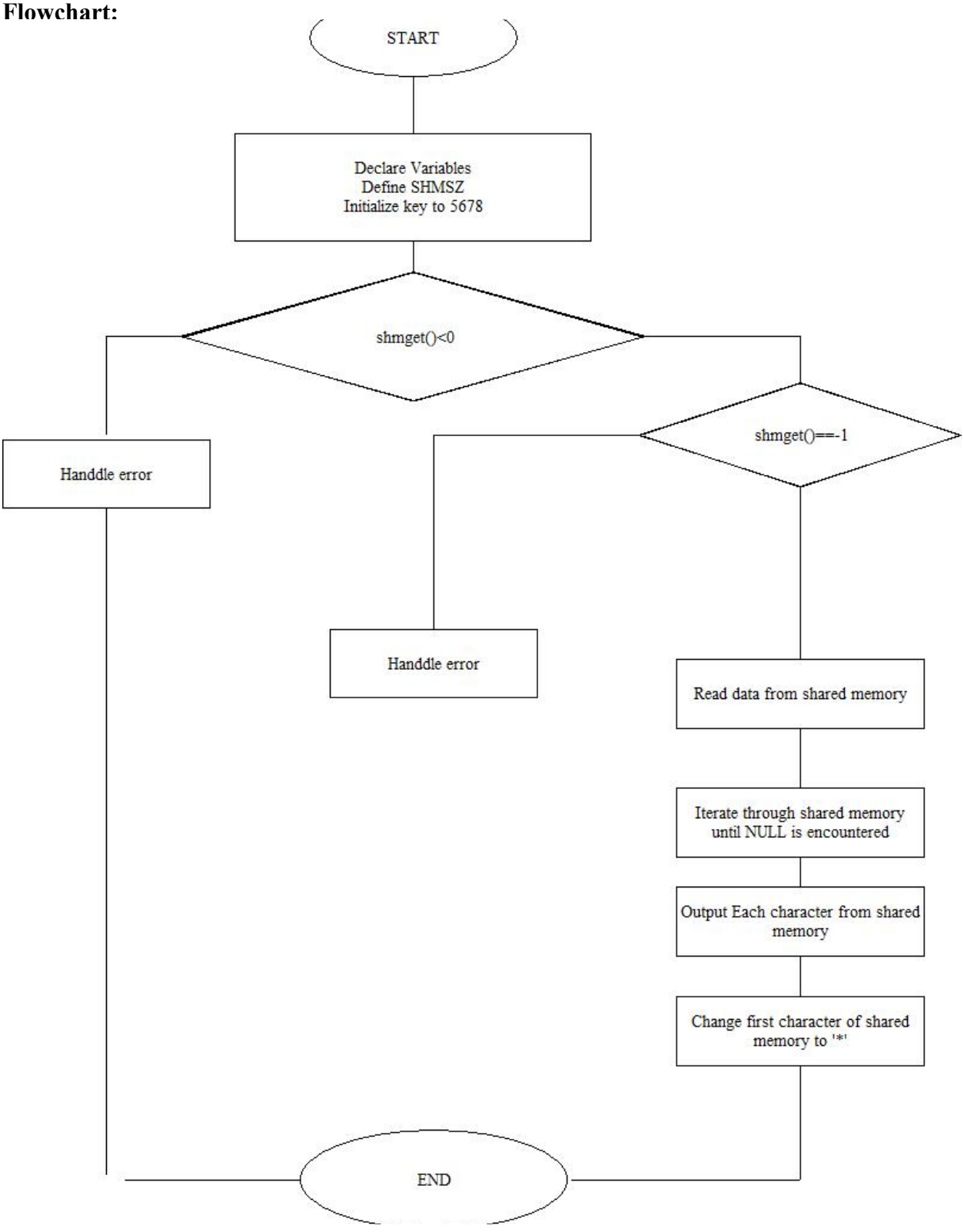
Assignment 8B

Title: Interprocess Communication (IPC) Using Shared Memory and Seamaphores.

Theory:
Shared memory is a mechanism for interprocess communication (IPC), where multiple processes can access the same region of memory to exchange data. Semaphores are used to synchronize access to shared resources among multiple processes.

Data Dictionary:
- Shared Memory: A region of memory shared among multiple processes.
- Semaphore: A synchronization primitive used to control access to shared resources.
- Memory Loaded: A semaphore indicating that data has been loaded into the shared memory.
- Memory Read: A semaphore indicating that data has been read from the shared memory.

Flowchart:



Code:
Client.c :

/*shm_client.c*/

```
/*
* shm-client - client program to demonstrate shared memory.
*/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSZ 27
main()
{
int shmid;
key_t key;
char *shm, *s;
/*
* We need to get the segment named
* "5678", created by the server.
*/
key = 5678;
/*
* Locate the segment.
*/
if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
perror("shmget");
exit(1);
}
/*
* Now we attach the segment to our data space.
*/
if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
perror("shmat");
exit(1);
}
/*
* Now read what the server put in the memory.
*/
for (s = shm; *s != NULL; s++)
putchar(*s);
putchar('\n');
/*
* Finally, change the first character of the
* segment to '*', indicating we have read
* the segment.
*/
*shm = '*';
exit(0);
}
```

Server.c :

```
/*shm_server.c*/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSZ 27
main()
{
```

```
char c;
int shmid;
key_t key;
char *shm, *s;
/*
* We'll name our shared memory segment
* "5678".
*/
key = 5678;
/*
* Create the segment.
*/
if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
perror("shmget");
exit(1);
}
/*
* Now we attach the segment to our data space.
*/
if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
perror("shmat");
exit(1);
}
/*
* Now put some things into the memory for the
* other process to read.
*/
s = shm;
for (c = 'a'; c <= 'z'; c++)
*s++ = c;
*s = NULL;
/*
* Finally, we wait until the other process
* changes the first character of our memory
* to '*', indicating that it has read what
* we put there.
*/
while (*shm != '*')
sleep(1);
exit(0);
}
```

Output:

```
Client:
$ ./cl
abcdefghijklmnopqrstuvwxyz
Server:
$ ./server
```

Assignment No 8c

Title-

Write 2 programs. 1st program will take small file from the user and write inside the shared memory. 2 nd program will read from the shared memory and write into the file.

Objective:

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 efficeint 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.

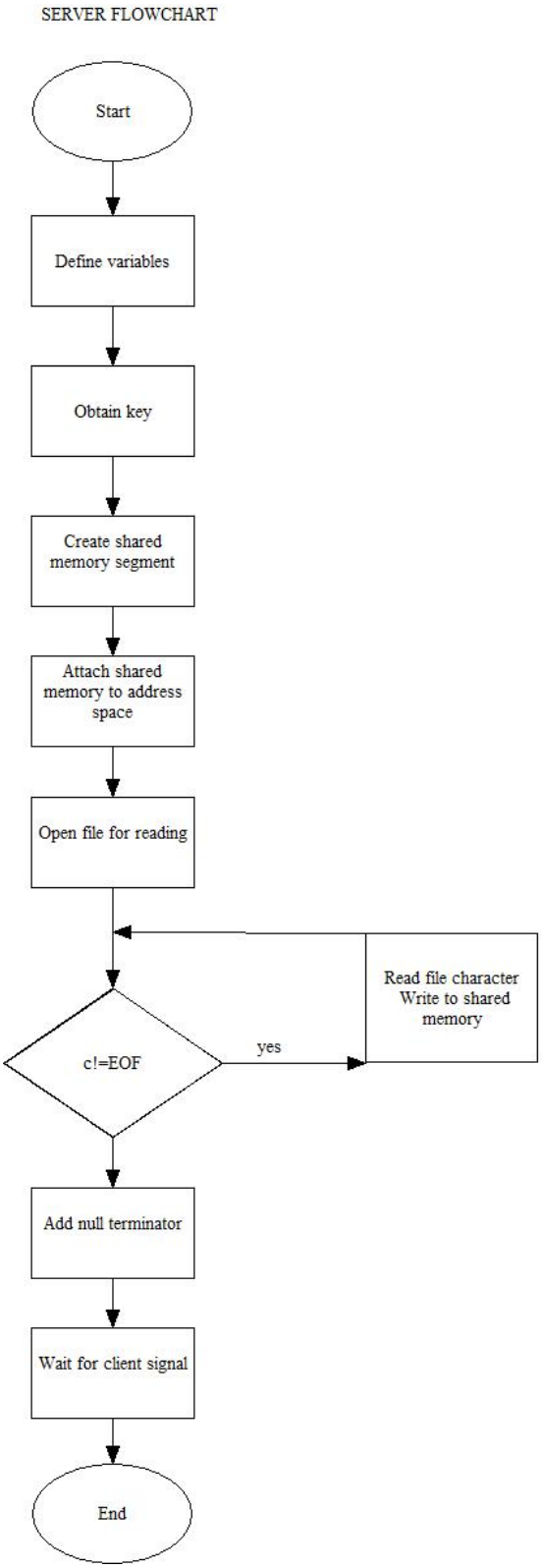
When process2 need to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from other process as well as for delivering any specific 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.

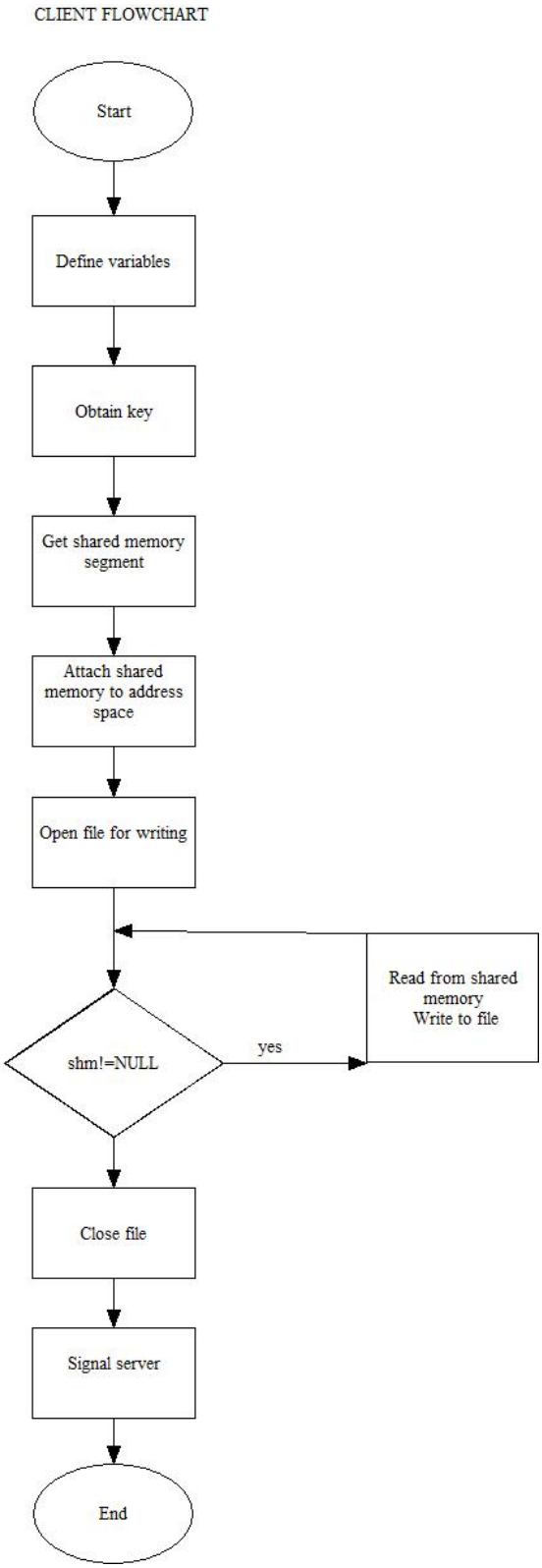
To use shared memory, we have to perform 2 basic steps:
• Request to the operating system a memory segment that can be shared between processes. The user can create/destroy/open this memory using a shared memory object: An object that represents memory that can be mapped concurrently into the address 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

Here is the data dictionary converted into a table format:

SR.NO	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

Flowchart -





Program-

Server-

```
#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);
}
```

Client-

```
#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;
   FILE *file;
   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);
   }
   file = fopen("output.txt", "w");
   if (!file) {
      perror("fopen");
      exit(1);
   }
   for (s = shm; *s != NULL; s++)
      fputc(*s, file);
   fclose(file);
   *shm = '#';
   exit(0);
}
```

Output :

Text from test.txt is copied into output.txt using shared memory.

Conclusion :

Sharing of files between client and server by using shared memory IPC was implemented.

Reference :

Dave's Programming in C Tutorials

Assignment 9A IPC: Sockets

Title- Write two programs (server/client) and establish a socket to communicate.

Objective:

1. To learn about the 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:

A very basic one-way Client and Server setup where a client connects, sends messages to the 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 machines, 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 the 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 the 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 ())

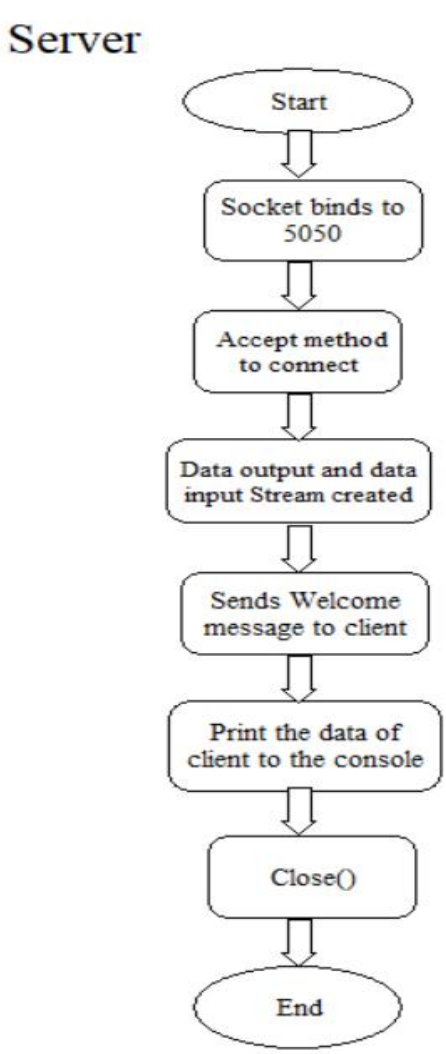
▪ A plain old Socket socket to use for communication with the client. getOutputStream() method is used to send the output through the socket. Close the Connection After finishing, it is important to close the connection by closing the socket as well as the input/output streams

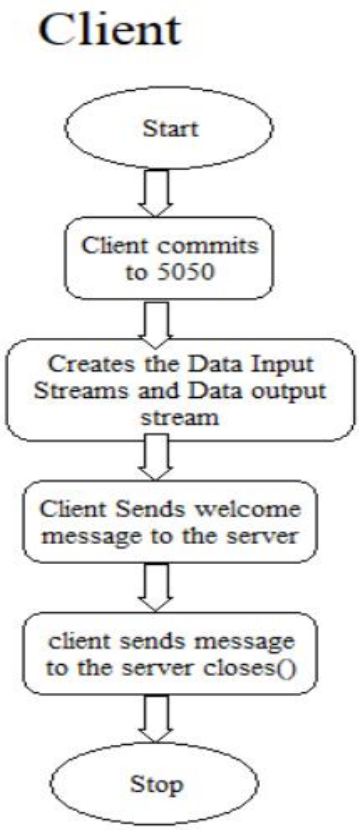
Data Dictionary:

SR.N	Variable/Function	Data Type	Use
1	ss	ServerSocket	Create a socket for server-side communication.
2	s	Socket	Socket is created
3	dos	DatOutputStream	Output Stream

4	dis	DataInputStrea m	Input Stream.
5	str	String	String to display messages from clients.

Flowchart





Program-

Server-

```
import java.net.*;
import java.io.*;
class uos91server
{
public static void main(String []args)throws Exception
{
ServerSocket ss=new ServerSocket(5050);
System.out.println("Server is Waiting.....");
Socket s=ss.accept();
DataOutputStream dos=new DataOutputStream(s.getOutputStream());
DataInputStream dis=new DataInputStream(s.getInputStream());
String str="Welcomes you are connected \n";
dos.writeUTF(str);
str=dis.readUTF();
System.out.println("From client"+" "+str);
```

```
ss.close();
s.close();
dos.close();
dis.close();
}
}
```

Client-

```
import java.net.*;
import java.io.*;
class uos91client
{
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();
System.out.println("From server"+" "+str);
str="Thank u for connecting";
dos.writeUTF(str);
s.close();
dos.close();
dis.close();
}
}
```

Output-

aadi@ubuntu:~$ javac uos91server.java

aadi@ubuntu:~$ java uos91server

server is waiting….

From client Thank u for connecting

aadi@ubuntu:~$ gedit 9a1.c

aadi@ubuntu:~$ javac uos91server.java

aadi@ubuntu:~$ java uos91server

From server Welcomes you are connected

Conclusion-Java can be used to establish communication between two programs on remote or same machine using sockets and system calls.

Reference-

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

(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).

9.B Write programs (server and client) to implement concurrent/iteraitve server to conntect 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 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.

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 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.

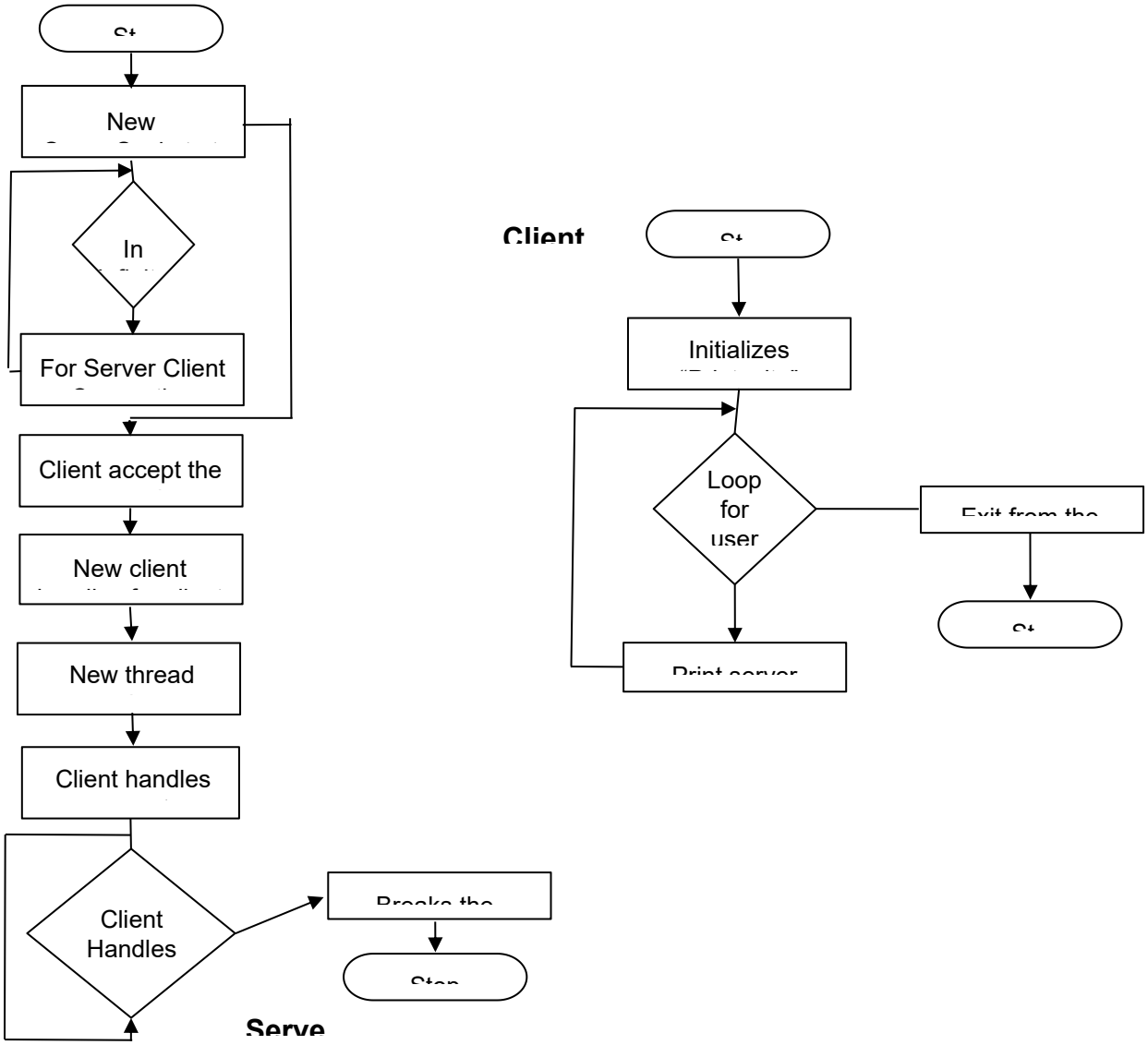
Data Dictionary:

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	cnm	String	String to display message from clients.
6	br	BufferedReader	Input data.

Table: 9.2 Data Dictionary

Flowchart:



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 aurgument 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:

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

Assignment 9C

Title-Write two programs (server and client) to show how you can establish a TCP socket connection using the above functions

Objective:

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

Theory:

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
#define MAX_SIZE 80
int main(int argc, char *argv[])
{
int sockfd, newsockfd, clilen;
struct sockaddr_in cli_addr, serv_addr;
int port;
char string[MAX_SIZE];
int len;
if(argc == 2)
sscanf(argv[1], "%d", &port);
else
port = SERV_TCP_PORT;
if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
```

```
perror("can't open stream socket");
exit(1);
}
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(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");
}
len = read(newsockfd, string, MAX_SIZE);
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
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;
if(argc >= 2)
serv_host = argv[1];
if(argc == 3)
```

```
sscanf(argv[2], "%d", &port);
else
port = SERV_TCP_PORT;
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);
if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
perror("can't open stream socket");
exit(1);
if(connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
perror("can't connect to server");
exit(1);
}
write(sockfd, "hello world connetion established", sizeof("hello world connetion
established"));
close(sockfd);
}
```

Output-

Conclusion-

TCP socket connetion using system calls in C studied and client server connection established.

Reference-

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

Assignment 9D

Title :- 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 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 know the iterative and concurrent server concept.

Theory :-

Socket Creation:

- Both the server and the client use the socket() system call to create a socket.
- In this case, they specify the address family AF_INET (IPv4), type SOCK_DGRAM for UDP communication, and protocol 0.

Address Initialization:

- The server initializes a sockaddr_in structure (servaddr) with the server's address details.
- The client also initializes a similar structure (servaddr) with the server's address details.

Binding (Server Only):

- The server binds the socket to a specific address and port using the bind() system call.
- This step is not required for the client in UDP, as UDP is connectionless.

Message Exchange:

- Both the client and the server prompt the user to enter a message using fgets() and read the message from standard input.
- The client then sends the message to the server using the sendto() system call, specifying the server's address.
- The server receives the message from the client using the recvfrom() system call, which also provides the client's address.
- Similarly, the server sends a response back to the client using sendto(), specifying the client's address.
- The client receives the response from the server using recvfrom().

Closing the Socket:

- Both the client and the server close the socket using the close() system call when communication is complete.

In summary, the server and client establish communication through the following steps:

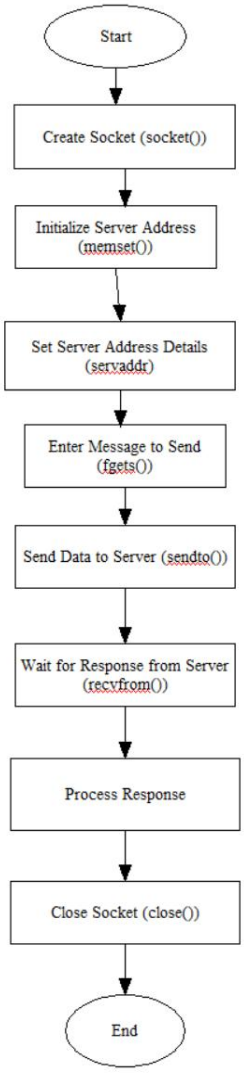
- The server creates a UDP socket, binds it to a specific address (optional), and waits for messages from clients.
- The client creates a UDP socket, sends a message to the server, and waits for a response.
- Both sides exchange messages using sendto() and recvfrom() system calls, specifying the destination address.
- After communication is complete, both sides close the socket using the close() system call.

Data Dictionary : -

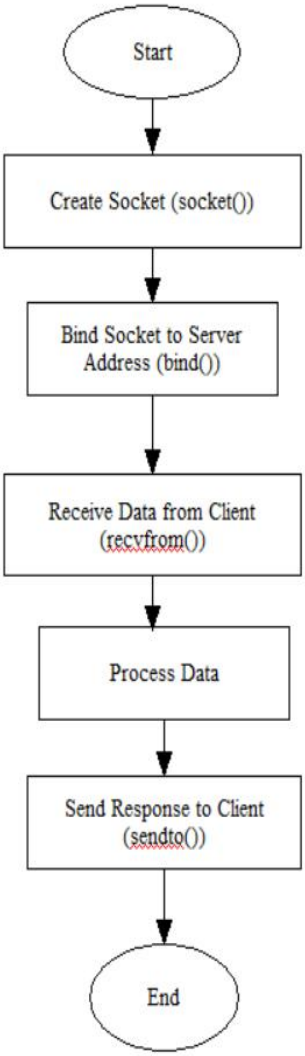
Number	Variable/Function	Data Type	Use
1	server_socket	int	Server socket file descriptor
2	servaddr	struct sockaddr_in	Server address structure
3	buffer	char[MAXLINE]	Buffer to store messages
4	PORT	int	Port number

Flowchart : -

Client



server



Program :-

socket_server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define PORT 12345
#define MAXLINE 1024

int main() {
   int sockfd;
   struct sockaddr_in servaddr, cliaddr;

   // Creating socket file descriptor
   if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
      perror("socket creation failed");
      exit(EXIT_FAILURE);
   }

   memset(&servaddr, 0, sizeof(servaddr));
   memset(&cliaddr, 0, sizeof(cliaddr));

   // Filling server information
   servaddr.sin_family = AF_INET;
   servaddr.sin_addr.s_addr = INADDR_ANY;
   servaddr.sin_port = htons(PORT);

   // Bind the socket with the server address
   if (bind(sockfd, (const struct sockaddr *)&servaddr, sizeof(servaddr)) < 0) {
      perror("bind failed");
      exit(EXIT_FAILURE);
   }

   int len, n;
   char buffer[MAXLINE];

   len = sizeof(cliaddr);  // len is value/resuslt

   n = recvfrom(sockfd, (char *)buffer, MAXLINE, MSG_WAITALL, (struct sockaddr
*)&cliaddr, &len);
   buffer[n] = '\0';
   printf("Client : %s\n", buffer);
```

```
    sendto(sockfd, (const char *)"Message from server.", strlen("Message from server."),
MSG_CONFIRM, (const struct sockaddr *)&cliaddr, len);
    printf("Message sent to client.\n");

    return 0;
}
```

socket_client.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h> // Include this header for close()

#define PORT 12345
#define MAXLINE 1024

int main() {
    int sockfd;
    struct sockaddr_in servaddr;

    // Creating socket file descriptor
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    memset(&servaddr, 0, sizeof(servaddr));

    // Filling server information
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(PORT);
    servaddr.sin_addr.s_addr = INADDR_ANY;

    int n, len;
    char buffer[MAXLINE];

    printf("Enter message to send: ");
    fgets(buffer, MAXLINE, stdin);

    sendto(sockfd, (const char *)buffer, strlen(buffer), MSG_CONFIRM, (const struct sockaddr
*)&servaddr, sizeof(servaddr));
```

```
    printf("Message sent to server.\n");

    n = recvfrom(sockfd, (char *)buffer, MAXLINE, MSG_WAITALL, (struct sockaddr
*)&servaddr, &len);
    buffer[n] = '\0';
    printf("Server : %s\n", buffer);

    close(sockfd); // Close the socket

    return 0;
}
```

Output-

```
~/Desktop/9d$ ./server
Client : hello from client!!!!
Message sent to client.

~/Desktop/9d$ ./client
Enter message to send: hello from client!!!!
Message sent to server.
Server : Message from server.
```

Conclusion-

UDP socket connection using system calls in C studied and client server connection established.

Reference : -

https://users.cs.cf.ac.uk/Dave.Marshall/C/CE.html

Assignment 9F

Title : Implement chating using TCP/UDP socket (between two or more users.)

Theory :

This program implements a basic chat application using TCP sockets in both C and Java.

- TCP (Transmission Control Protocol) is used for reliable and connection-oriented communication.
- The server listens for incoming connections from clients on a specified port.

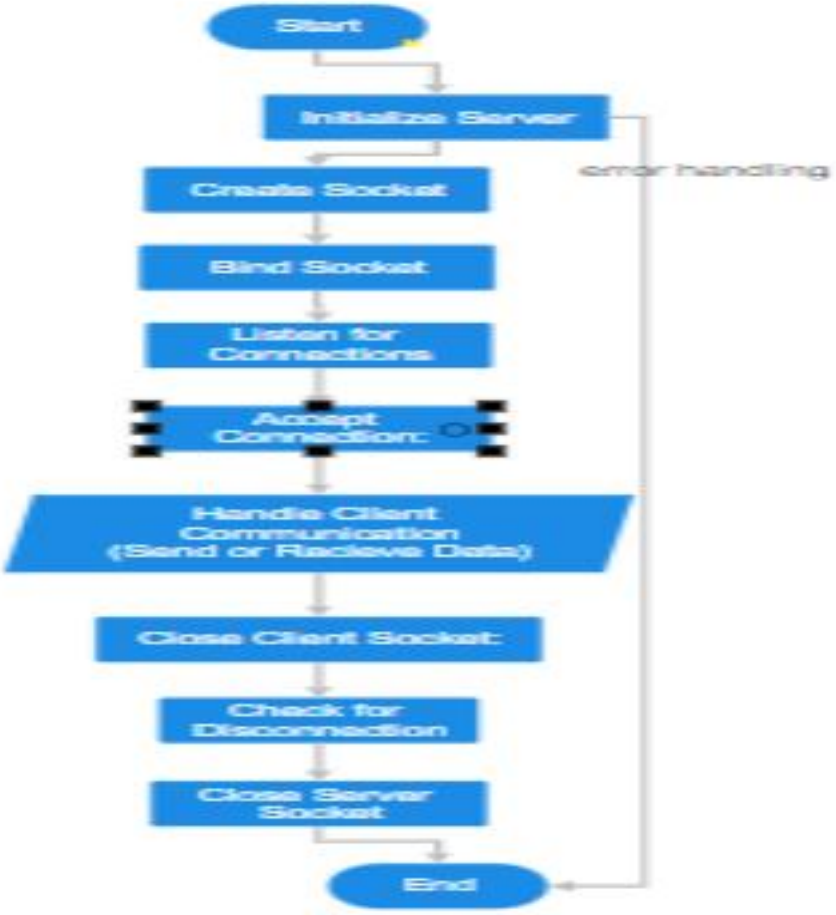
- Upon connection, the server creates a new thread or process to handle communication with each client concurrently.
- Clients connect to the server and can send messages, which are then echoed back to them by the server.
- The communication between the server and clients is bidirectional, allowing for real-time chat functionality.
- The server and clients use sockets for establishing connections and transmitting data over the network.
-
- These components together form a basic chat application that allows multiple clients to connect to a server and exchange messages.

Data Dictionary :

Variable	Type	Description
server_socket	int	File descriptor for the server socket.
server_addr	struct sockaddr_in	Structure containing server address information
client_addr	Struct sockaddr_in	Structure containing client address information
client_socket	int	File descriptor for the client socket
buffer	char[]	Array to hold data received from or sent to clients
message	char[]	Array to hold messages entered by the user
PORT	int	Port number for socket communication
BUFFER_SIZE	int	Size of the buffer used for data transmission

Flow Chart :

Server Side Flowchart



Client side Flowchart:

Code :

Client Side code :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define SERVER_IP "127.0.0.1"
#define PORT 5000
#define BUFFER_SIZE 1024

int main() {
    int client_socket;
    struct sockaddr_in server_addr;
    char buffer[BUFFER_SIZE], message[BUFFER_SIZE];

    // Create socket
    client_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (client_socket < 0) {
```

```
    perror("Socket creation failed");
    exit(EXIT_FAILURE);
}

// Initialize server address
memset(&server_addr, 0, sizeof(server_addr));
server_addr.sin_family = AF_INET;
server_addr.sin_addr.s_addr = inet_addr(SERVER_IP);
server_addr.sin_port = htons(PORT);

 // Connect to server
if (connect(client_socket, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
    perror("Connection failed");
    exit(EXIT_FAILURE);
}

printf("Connected to server.\n");

// Communication with the server
while (1) {
    // Read message from user
      printf("Enter message to send (type 'exit' to quit): ");
    fgets(message, BUFFER_SIZE, stdin);

    // Send message to server
    send(client_socket, message, strlen(message), 0);

    // Receive response from server
    int bytes_received = recv(client_socket, buffer, BUFFER_SIZE, 0);
    if (bytes_received < 0) {
        perror("Receiving data failed");
        exit(EXIT_FAILURE);
    } else if (bytes_received == 0) {
        printf("Server disconnected\n");
        break;
    }

    buffer[bytes_received] = '\0';
    printf("Received response from server: %s\n", buffer);

    // Check if user wants to quit
    if (strncmp(message, "exit", 4) == 0)
        break;
}
```

```
    close(client_socket);
    return 0;
}
```

Server Side :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 5000
#define BUFFER_SIZE 1024

int main() {
    int server_socket;
    struct sockaddr_in server_addr, client_addr;
    char buffer[BUFFER_SIZE];

    // Create socket
    server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket < 0) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    // Initialize server address
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    server_addr.sin_port = htons(PORT);

    // Bind the socket
    if (bind(server_socket, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
        perror("Binding failed");
        exit(EXIT_FAILURE);
    }

    // Listen for connections
    if (listen(server_socket, 5) < 0) {
        perror("Listening failed");
```

```
        exit(EXIT_FAILURE);
    }

    printf("Server started, waiting for clients...\n");

    while (1) {
        socklen_t client_addr_len = sizeof(client_addr);
        int client_socket = accept(server_socket, (struct sockaddr *)&client_addr,
&client_addr_len);
        if (client_socket < 0) {
            perror("Accepting connection failed");
            exit(EXIT_FAILURE);
        }

        printf("Client connected from %s:%d\n", inet_ntoa(client_addr.sin_addr),
ntohs(client_addr.sin_port));

        // Create a child process to handle each client
        if (fork() == 0) {
            close(server_socket);

            // Communication with the client
            while (1) {
                int bytes_received = recv(client_socket, buffer, BUFFER_SIZE, 0);
                if (bytes_received < 0) {
                    perror("Receiving data failed");
                    exit(EXIT_FAILURE);
                } else if (bytes_received == 0) {
                    printf("Client disconnected\n");
                    break;
                }

                buffer[bytes_received] = '\0';
                printf("Received message from client: %s\n", buffer);

                // Echo message back to the client
                if (send(client_socket, buffer, strlen(buffer), 0) != strlen(buffer)) {
                    perror("Sending data failed");
                    exit(EXIT_FAILURE);
                }
            }

            close(client_socket);
```

```
            exit(EXIT_SUCCESS);
        }

        close(client_socket);
    }

    close(server_socket);
    return 0;
}
```

Output :
Server started, waiting for clients…
Connected to server.
Enter message to send (type 'exit' to quit): exit

Conclusion: In TCP socket communication, the server listens for incoming connections, accepts them, and then communicates bidirectionally with clients over established connections, while in UDP socket communication, the server and client exchange datagrams without establishing a persistent connection, enabling faster but unreliable communication.

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

Assignment 10A

Title- 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.

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.*

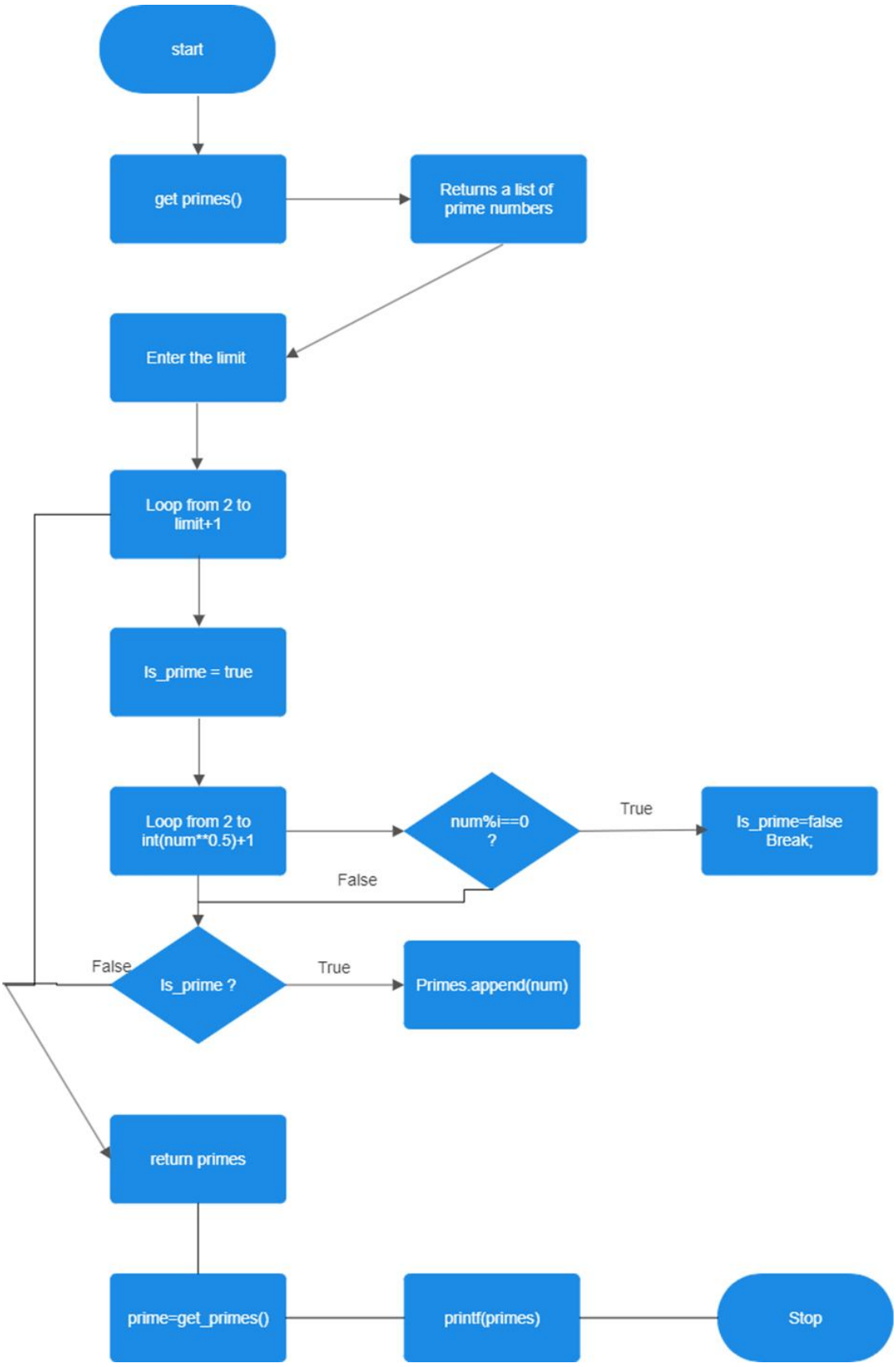
Using the subprocess module:
The recommended way to launch subprocesses is to use the following convenience functions. For more advanced use cases when these do not meet your needs, use the underlying Popen interface.
subprocess.call(args, *, stdin=None, stdout=None, stderr=None, shell=False)
Run the 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.

Data Dictionary

Variable/Function	Datatype	Use
r	int	Store the number of rows (terminal height)
c	int	Store the number of columns (terminal width)
n	int	Store the number of rows in the pyramid input by the user
rows, columns	str	Store the rows and columns obtained from the 'stty size' command

Flow Chart


start
get primes()
Returns a list of prime numbers
Enter the limit
Loop from 2 to limit+1
Is_prime = true
Loop from 2 to int(num**0.5)+1
num%i==0 ?
True
Is_prime=false Break;
False
False
Is_prime ?
True
Primes.append(num)
return primes
prime=get_primes()
printf(primes)
Stop

Code

```
import os
try:
    rows, columns = os.popen('stty size', 'r').read().split()
    r = int(rows)
    c = int(columns)
except ValueError:
    # Handle the case where 'stty size' fails to get terminal size
    # Set default terminal size values
    r = 24
    c = 80

n = int(input("Enter number of rows: "))

for i in range(int(r/2) - int(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) - int(n/2)):
    print()
```

Output

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

Conclusion-

1. .Basics of python like the concept of loops learnt
2. .Conditional statements learn

Reference-

https://docs.python.org/3/

Assignment 10B

Title: Write a python function for prime number input limit in as parameter to it.

Objectives:
1. To learn about python as scripting option.

Theory:

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance.

Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

Python code defines two functions:
is_prime(n) and print_prime_numbers(limit).

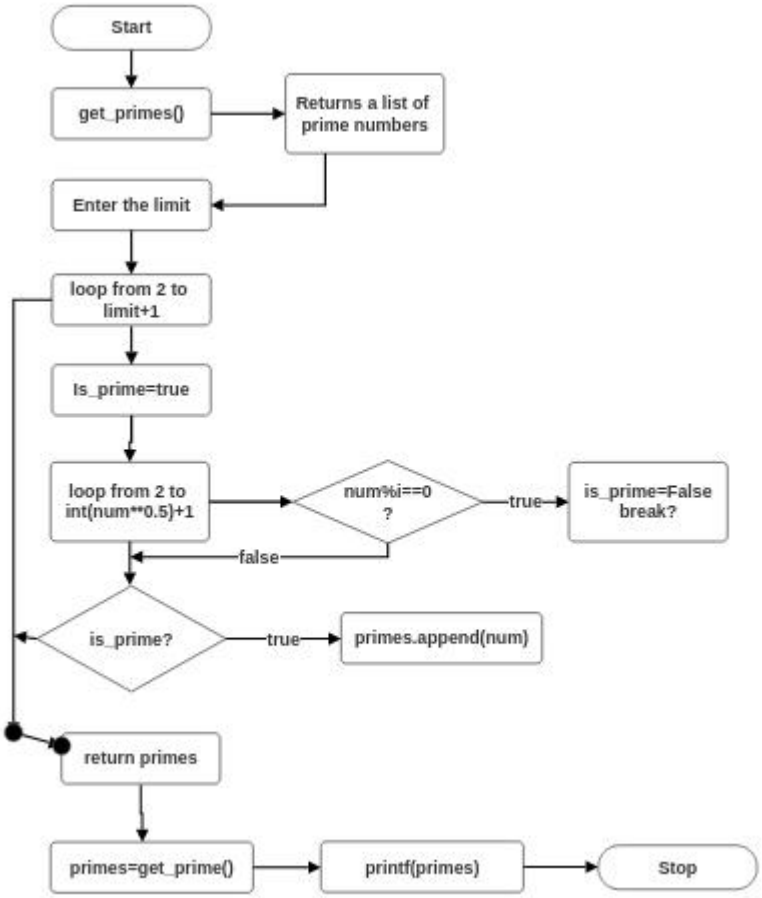
1. is_prime(n): This function checks whether a given number n is prime or not. It follows a basic prime number checking algorithm, where it returns False if n is less than or equal to 1, returns True if n is 2 or 3, returns False if n is divisible by 2 or 3, and then iterates through the odd numbers starting from 5 up to the square root of n to check for divisibility by those numbers. If n is divisible by any of those odd numbers, it returns False, otherwise, it returns True.

2. print_prime_numbers(limit): This function prints all prime numbers up to the given limit. It checks for each number from 2 to the limit using a loop and calls the is_prime function to determine if each number is prime. If a number is prime, it prints it.

Data Dictionary

Variable	**Description**
n	The number to be checked for primality in the is_prime function.
limit	The upper limit up to which prime numbers will be printed in the print_prime_numbers function.
i	A counter variable used in the is _prime function to iterate through potential divisors.
num	The current number being checked for primality in the print_prime_numbers function

Flowchart



Code:

```
def is_prime(n):
if n <= 1:
return False
if n <= 3:
return True
if n % 2 == 0 or n % 3 == 0:
return False
i = 5
while i * i <= n:
if n % i == 0 or n % (i + 2) == 0:
return False
i += 6
return True
```

```
def print_prime_numbers(limit):
if limit < 2:
print("There are no prime numbers up to", limit)
else:
print("Prime numbers up to", limit, "are:")
for num in range(2, limit + 1):
if is_prime(num):
print(num, end=" ")
limit = int(input("Enter the limit for prime numbers: "))
print_prime_numbers(limit)
```

Output:

```
aadi@ubuntu:~$ python3 ass10b.py
Enter the limit for prime numbers: 50
Prime numbers up to 50 are:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
```

Conclusion:

Python code effectively checks for prime numbers and prints them up to a specified limit. It employs a standard prime checking algorithm that efficiently determines the primality of numbers. The print_prime_numbers function nicely encapsulates the logic for printing prime numbers within the given limit.

References:

https://docs.python.org/3/

Assignment 10C

Title: Python: As a scripting language - Take any txt file and count word frequencies in a file.

Objectives: To learn about python as a 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:

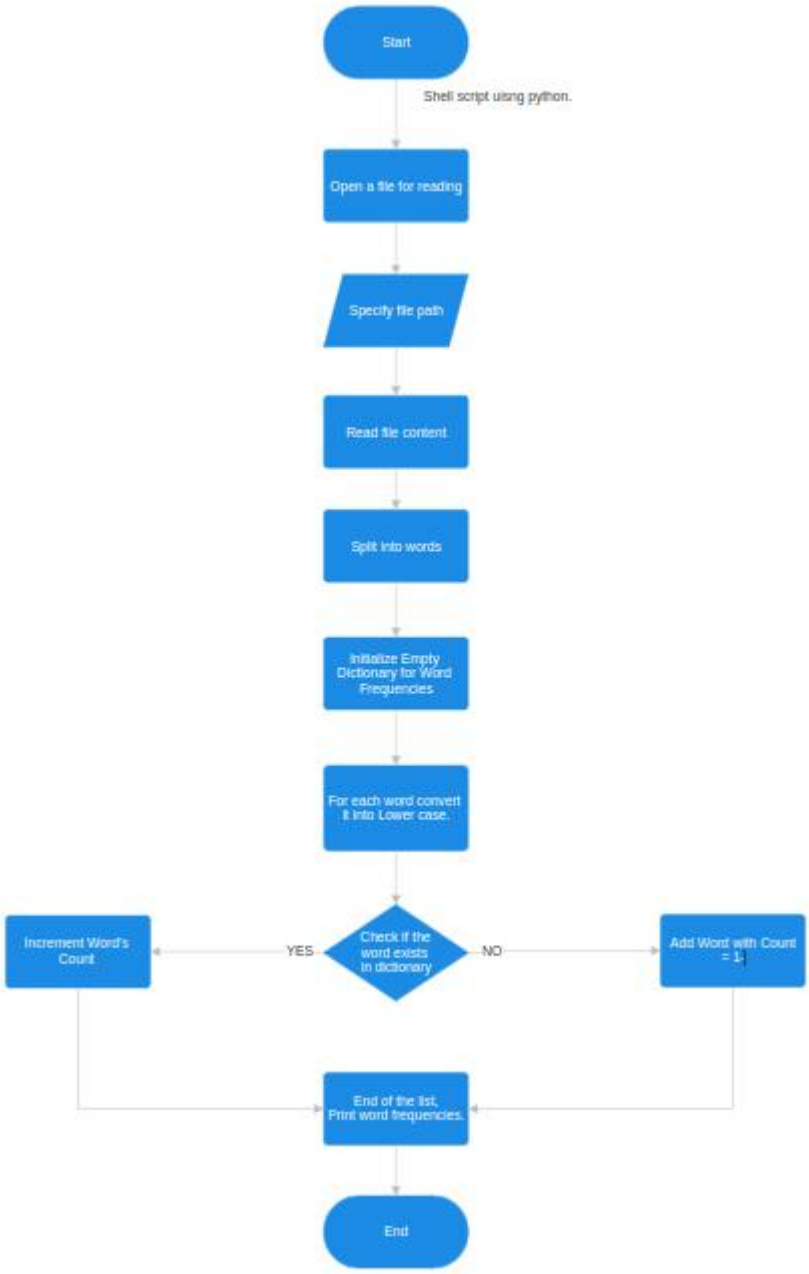
To read the whole file at once use the read() method.

> fobj = open("sample.txt")

> fobj.read()

'I love Python\n P1 loves KDE\nP2 loves Openoffice\n'

Flowchart:



Code:

```
# Open the file
with open("10cuos.txt") as f:
  d = {}

  # Process each line in the file
  for line in f:
    # Split the line into words
    l = line.split()

    # Process each word
    for word in l:
      # Update the word count
```

```
        if word in d:
            d[word] = d[word] + 1
        else:
            d[word] = 1

# Print the word counts
for key in d:
   print(key, " : ", d[key])
```

Output:

```
This  :  1
is  :  1
UOS  :  1
assignment  :  1
10c  :  1
.  :  1
TY  :  2
IT  :  2
WCE  :  2
sangli.  :  1
Unix  :  2
```

Conclusion:

1.File handling and manipulation of data using list and dictionary learnt.

References:

[1] https://docs.python.org/3/

Assignment 13C

Title: Simple filter to add two numbers.

Objectives:

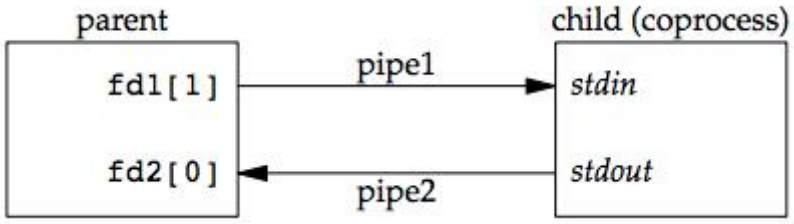
Simple filter to add two numbers. (B)

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.



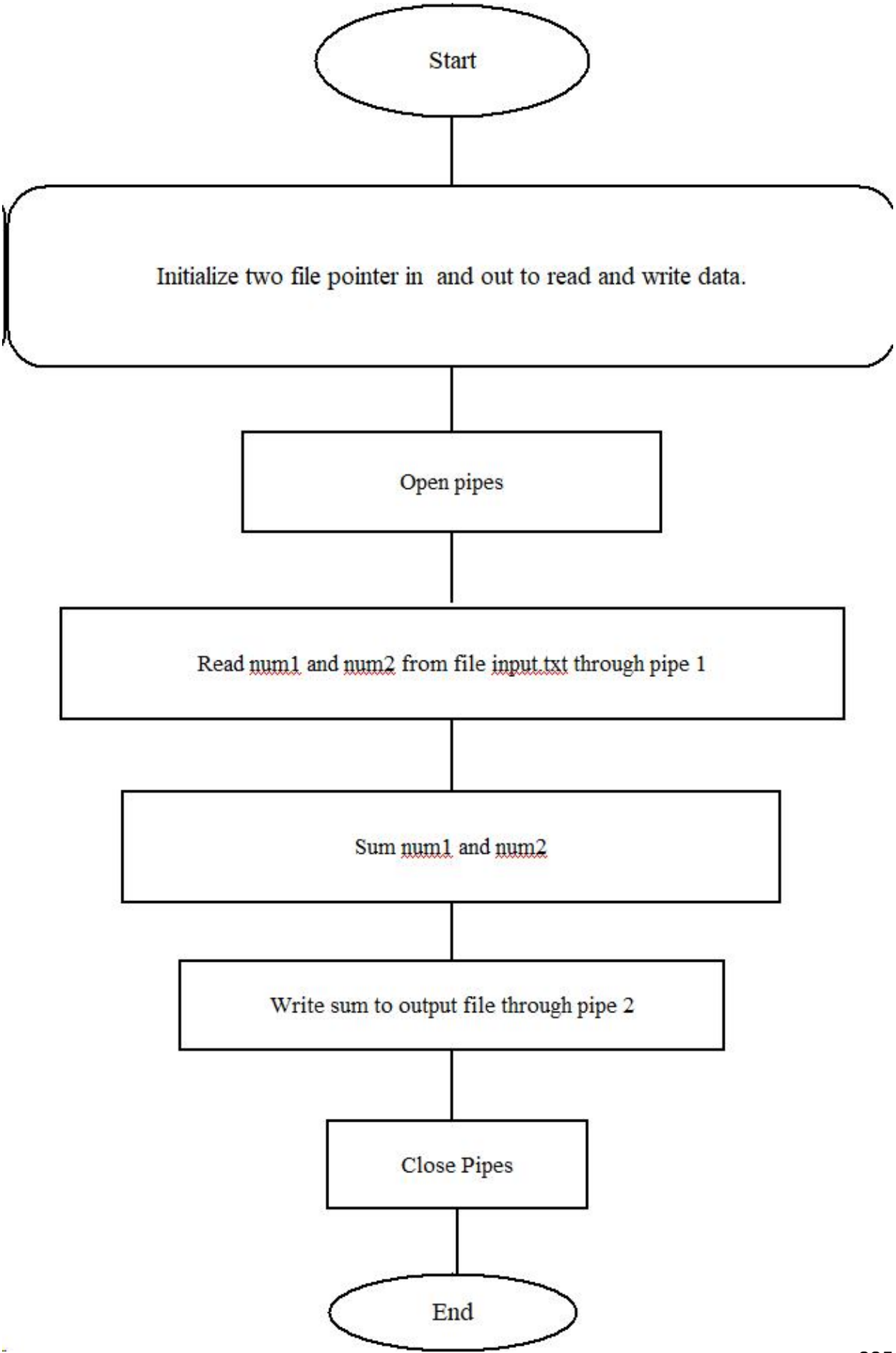
Example: invoking add2 as a coprocess

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.

Flowchart:


Start
Initialize two file pointer in and out to read and write data.
Open pipes
Read num1 and num2 from file input.txt through pipe 1
Sum num1 and num2
Write sum to output file through pipe 2
Close Pipes
End

Code:

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

int main() {
        int pipefd[2]; // declare an array to hold read and write file descriptors for the pipe
        pid_t pid; // declare a process ID variable

        if (pipe(pipefd) == -1) { // create the pipe and check for errors
        perror("pipe");
        exit(EXIT_FAILURE);
        }

        pid = fork(); // create a child process using fork

        if (pid == -1) { // check for fork errors
        perror("fork");
        exit(EXIT_FAILURE);
        }

        if (pid == 0) { // child process (addition filter)
 close(pipefd[1]); // close write end of pipe
 int num1, num2, sum;
 read(pipefd[0], &num1, sizeof(num1)); // read first number from read end of pipe
 read(pipefd[0], &num2, sizeof(num2)); // read second number from read end of pipe


 sum = num1 + num2; // compute sum
 printf("Sum of %d and %d is %d\n", num1, num2, sum); // print sum to stdout
 close(pipefd[0]); // close read end of pipe
 exit(EXIT_SUCCESS);
  }else { // parent process
        close(pipefd[0]); // close the read end of the pipe, since the parent only writes to it
        int num1 = 42, num2 = 24;
        write(pipefd[1], &num1, sizeof(num1)); // write the first number to the write end of the
pipe
        write(pipefd[1], &num2, sizeof(num2)); // write the second number to the write end of
the pipe
        close(pipefd[1]); // close the write end of the pipe
        // wait for the child process to finish
        // uncomment the next line if you want to wait for the child process explicitly
        // wait(NULL);
        exit(EXIT_SUCCESS); // exit the parent process
```

```
        }

        return 0;
}
```

Output:

```
$ cd 13c
$ ./a.out
Sum of 42 and 24 is 66
```