WALCHAND COLLEGE OF ENGINEERING SANGLI

Department of Information Technology

UNIX OPERATING SYSTEM LAB (4IT 371)

Year of Studentship: 2020-21

Term: Semester-2(Even)

Class: T.Y. IT

Name	PRN. No
Aniket Sable	2019BTEIT00020
Sumit Koundanya	2019BTEIT00023

Certificate

This is certified that

Aniket Sable(2019BTEIT00020)

Sumit Koundanya(2019BTEIT00023)

of

T. Y. I. T. class has completed satisfactorily 13 experiments in Unix Operating System Lab(4IT371) during the

Year :2021-22

Submission Date:
13/05/2022

COURSE TEACHER & HOD

Dr. A.J.Umbarkar

UNIX Operating System Lab. 4IT 371

Course Objectives:

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

Percentage of objective achieved by students

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

Please tick appropriate box.

Course Learning Outcomes:

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

Percentage of Outcomes achieved by students

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

Please tick appropriate box

Name Of Student	Roll No
1 Aniket Sable	2019BTEIT00020
2 Sumit Koundanya	2019BTEIT00023

UNIX Operating System

Assignment list

Sr. no.	Assignments	
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 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

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

Objectives:

1. To learn about File system Internals.

a. Write the program to show file statistics using the stat system call. Take the filename / directory name from user including path. (B)
b. Write the program to show file statistics using the fstat system call. Take the file name / directory name from user including path. Print only inode no, UID, GID, FAP and File type only. (B)

c. Write a program to use link/unlink system call for creating logical link and identifying the difference using stat. (I)
d. Implement a program to print the various types of file in Linux. (Char, block etc.) (E)
Ref: *System call search*

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

Objectives:

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

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

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

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

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

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

a. Write a multithreaded program in java/c for chatting (multiuser and multi-terminal) using threads. (B)
b. Write a program which creates 3 threads, first thread printing even number, second thread printing odd number and third thread printing prime number in terminals. (B)
1. Write program to synchronize threads using construct – monitor/serialize/semaphore of Java (In java only) (I)
c. Write a program in Linux to use clone system call and show how it is different from fork system call. (I)
d. Write a program using p-thread library of Linux. Create three threads to take odd, even and prime respectively and print their average respectively. (In C) (I)
Ref:http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html#BASICS

e. Write a program using p thread library of Linux. Create three threads to take numbers and use join to print their average. (in C) (E)
Ref:http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html#BASICS

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

g. Write program using semaphore to ensure that function *f1* should executed after *f2*. (In java only) (E)
Ref: ftp://10.1013.3/pub/UOS..../
System call search

5. **Shell programming**: shell scripts

Objectives:

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

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

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

Objectives:

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

a. Write a program to illustrate the semaphore concept. Use fork so that 2 process running simultaneously and communicate via semaphore. (B)
Ref:http://www.cs.cf.ac.uk/Dave/C/node26.html#SECTION00260000000000000000
Ref: ftp://10.1013.3/pub/UOS..../ OR
Ref:www.cs.cf.ac.uk/Dave/C/CE.html
OR System call search

b. Write a program to implement producer consumer problem. (Use suitable synchronization system calls 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 semaphoredeclaredinthe1stprogram.
(E)Ref.:http://www.minek.com/files/unix_examples/semab.html

7.

IPC: Message Queues: msgget, msgsnd, msgrcv.
Objectives:

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

---

a. Write a program to perform IPC using message and send did u get this? and then reply. (B)
b. Write a 2 programs that will both send and messages and construct the following dialog between them

- (Process 1) Sends the message "Are you hearing me?"
- (Process 2) Receives the message and replies "Loud and Clear".
- (Process 1) Receives the reply and then says "I can hear you too". (I)

c. Write a *server* program and two *client* programs so that the *server* can communicate privately to *each client* individually via a *single* message queue. (E)
Ref: ftp://10.1013.3/pub/UOS..../ OR
Ref:www.cs.cf.ac.uk/Dave/C/CE.html
OR System call search

8.

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

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

a. Write a program to perform IPC using shared memory to illustrate the passing of a simple piece of memory (a string) between the processes if running simultaneously. (B)
b. Write 2 programs that will communicate via shared memory and semaphores. Data will be exchanged via memory and semaphores will be used to synchronize and notify each process when operations such as memory loaded and memory read have been performed. (I)

c. Write 2 programs. 1st program will take small file from the user and write inside the shared memory. 2nd program will read from the shared memory and write into the file. (E)
Ref: ftp://10.1013.3/pub/UOS..../ OR
Ref:www.cs.cf.ac.uk/Dave/C/CE.html
OR System call search

9.

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

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

---

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

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

b. Write programs (server and client) to implement concurrent/iteraitve server to conntect multiple clients requests handled through concurrent/iterative logic using UDP/TCP socket connection.(C or Java only) (use process concept for c server and thread for java server) (B)

c. Write two programs (server and client) to show how you can establish a TCP socket connection using the above functions.(in C only) (in exam allowed to do in java and python) (I)

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

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

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

Ref: pdfbooks given

e. Implement echo server using TCP/UDP in iterative/concurrent logic. (E)

f. Implement 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

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)

(Hint: Basics n loops)

b. Write a python function for prime number input limit in as parameter to it. (B)
Ref: ftp://10.1013.3/pub/UOS..../python by AJU

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

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

1. To learn about IPC through MPI.
2. Use of IPC mechanism to write effective application programs.

a. Implement the program IPC/IPS using MPI library. Communication in processes of users. (B)
b. Implement the program IPC/IPS using MPI library. Communication in between processes OS's: Unix. (I)
c. configure cluster and experiment MPI program on it. (E)

Ref:
ftp://10.10.13.16/pub/Academic_Material/TYIT/Semister_2/Unix_Operating_System/OpenMP..... OR
Search on internet

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

Objectives:

1. To learn about openMP for better use multicore system.

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

	Ref: ftp://10.10.13.16/pub/Academic_Material/TYIT/Semister_2/Unix_Operating_System/OpenMP..... OR Search on internet	
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)	

1a: Processing Environment

Name:- Sumit Sunil Koundanya

PRN:-2019BTEIT00023 **Class:- TYIT**

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 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:

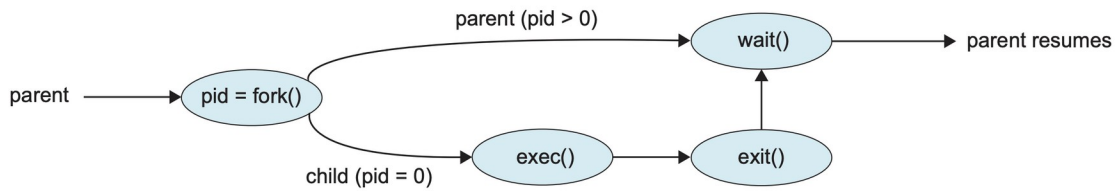


Fig: 1.1 Flowchart of fork

Data Dictionary:

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

Fig:1.1 Data Dictionary

Program:

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

int main(){

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

Output:

```
sumit@sumit-15:~/Documents/UOS$ gedit fork.c

sumit@sumit-15:~/Documents/UOS$ gcc fork.c

sumit@sumit-15:~/Documents/UOS$ ./a.out

Beginning

Parent process = 1

Parent process = 2

Parent process = 3

Parent process = 4

Parent process = 5

Parent Ended

Child process = 1

Child process = 2

Child process = 3

Child process = 4

Child process = 5

Child Ended
```

Conclusion:

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

References:

[1] www.tutorialspoint.com/unix_system_calls/

[2] https://users.cs.cf.ac.uk/Dave.Marshall/C/node22.html

Processing Environment

Subject:- Unix OperatingSystem

System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

Assignment No - 1b

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

Objectives –

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

Theory-

fork vs exec:

The use of fork and exec exemplifies the spirit of UNIX in that it provides a very simple way to start new processes. The fork call basically makes a duplicate of the current process, identical in almost every way (not everything is copied over, for example, resource limits in some implementations but the idea is to create as close a copy as possible).

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

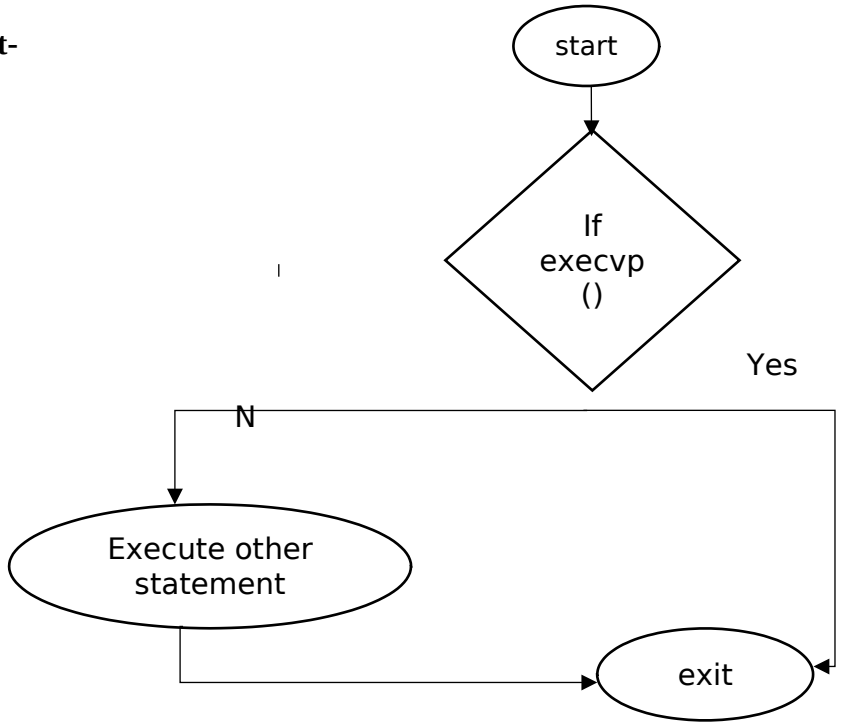
The exec call is a way to basically replace the entire current process with a new program. It loads the program into the current process space and runs it from the entry point. So, fork and exec are often used in sequence to get a new program running as a child of a current process. Shells typically do this whenever you try to run a program like find - the shell forks, then the child loads the find program into memory, setting up all command line arguments, standard I/O and so forth.

Processing Environment:

Process creation Unless the system is being bootstrapped a process can only come into existance as the child of another process. This done by the fork system call. The first process created is "hand tooled" by the boot process. This is the swapper process.

The swapper process creates the init process, which is the ancestor of all further processes. In particular, init forks off a process getty, which monitors terminal lines and allows users to log in. Upon login, the command shell is run as the first process. The command shell for a given user is specified in the /etc/passwd file. From thereon, any process may fork to produce new processes, considered to be children of the forking process. The process table and uarea Information about processes is described in two data structures, the kernel process table and a "uarea" associated with each process. The process table holds information required by the kernel The uarea holds information required by the process itself.

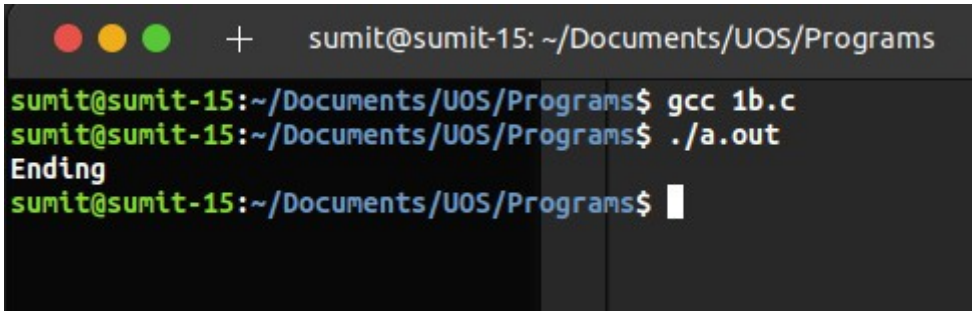
Flowchart-

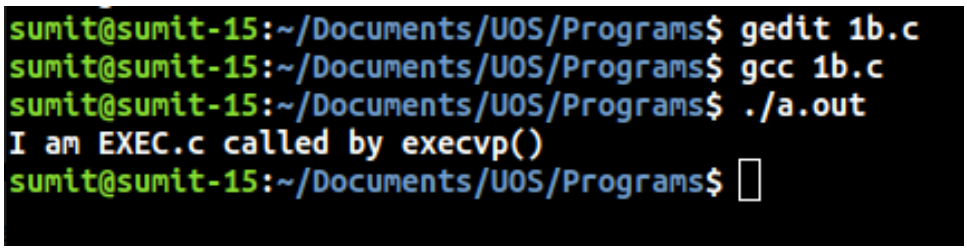


Program-

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

#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main()
{
printf("I am EXEC.c called by execvp()\n");
return 0;
}
```

Output-





Conclusion:

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

References:

www.tutorialspoint.com/unix_system_calls/

Processing Environment

Subject:- Unix OperatingSystem

System Lab Class :- TYIT

Name	**PRN**
Sumit Sunil Koundanya	**2019BTEIT00023**
Aniket Shivnath Sable	**2019BTEIT00020**

Assignment No - 1c

Title-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.

Basis for comparision	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	If child process alters any

	page in the address space, it is invisible to the parent process as the address space are separate.	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

Program-

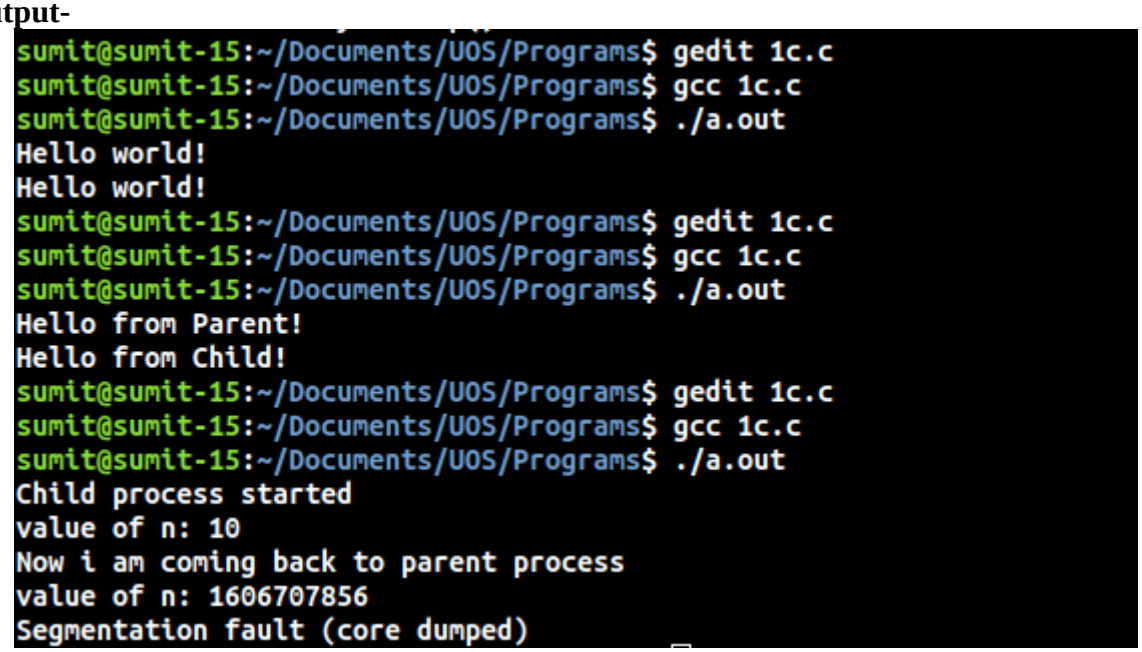
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
   fork();
   printf("Hello world!\n");
   return 0;
}
```

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void forkexample()
{

        if (fork() == 0)
                printf("Hello from Child!\n");
        else
                printf("Hello from Parent!\n");
}
int main()
{
        forkexample();
        return 0;
}
```

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
   int n =10;
   pid_t pid = vfork();
   if (pid == 0)
   {
      printf("Child process started\n");
```

```
    }
    else
    {
       printf("Now i am coming back to parent process\n");
    }
    printf("value of n: %d \n",n);
    return 0;
}
```

Output-



Conclusion:

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

References:

www.tutorialspoint.com/unix_system_calls/

Processing Environment

Subject:- Unix Operating System

System Lab Class :- TYIT

Name	**PRN**
Sumit Sunil Koundanya	**2019BTEIT00023**
Aniket Shivnath Sable	**2019BTEIT00020**

Assignment No - 1d

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

Objectives –

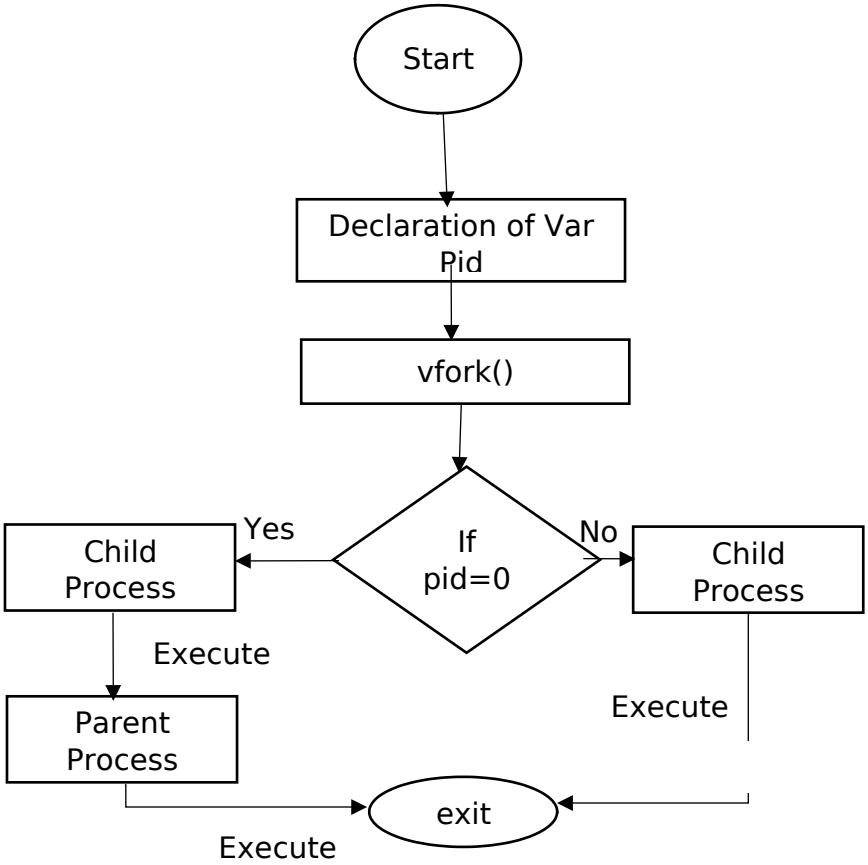
1. To learn about Processing Environment.
2. To know the difference between fork/vfork and various execs variations.
3. Use of system call to write effective programs.

Theory-

Difference between wait() and waitpid()

Wait()	Waitpid()
wait blocks the caller until a child process terminates	waitpid can be either blocking or nonblocking: • If options is 0, then it is blocking • If options is WNOHANG, then is it non-blocking
if more than one child is running then wait() returns the first time one of the parent's offspring exits	waitpid is more flexible: • If pid == -1, it waits for any child process. In this respect, waitpid is equivalent to wait • If pid > 0, it waits for the child whose process ID equals pid • If pid == 0, it waits for any child whose process group ID equals that of the calling process • If pid < -1, it waits for any child whose process group ID equals that absolute value of pid

Flowchart-

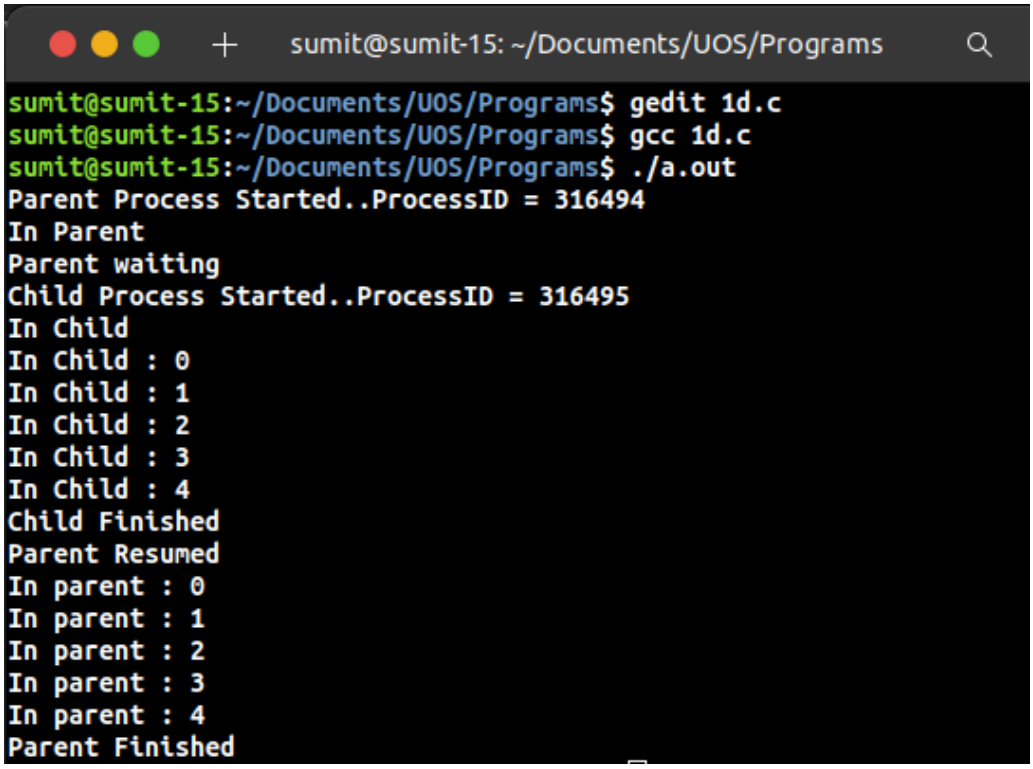


Program-

```
#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-



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:

www.tutorialspoint.com/unix_system_calls/

Processing Environment

Subject:- Unix Operating System

System Lab Class :- TYIT

Name	**PRN**
Sumit Sunil Koundanya	**2019BTEIT00023**
Aniket Shivnath Sable	**2019BTEIT00020**

Assignment No - 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

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-

```
sumit@sumit-15:~/Documents/UOS/Programs$ gedit 1e.c
sumit@sumit-15:~/Documents/UOS/Programs$ gcc 1e.c
      In function ‘    ’:
          warning: ‘  ’ directive writing up to 255 bytes into a regio
n of size 245 [-Wformat-overflow=]
    8 | sprintf(buf, "/bin/sh -c %s", str);
      |                          ^~   ~~~
         note: ‘       ’ output between 12 and 267 bytes into a destin
ation of size 256
    8 | sprintf(buf, "/bin/sh -c %s", str);
      | ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
sumit@sumit-15:~/Documents/UOS/Programs$ ./a.out
Enter command hostnamectl
   Static hostname: sumit-15
         Icon name: computer-laptop
           Chassis: laptop
        Machine ID: 9994117559e84f92b96cd4e8bbc2643f
           Boot ID: 8c86cbaffd1a4dedbc6f83edeb26ae3f
  Operating System: Ubuntu 20.04.4 LTS
            Kernel: Linux 5.4.0-52-generic
      Architecture: x86-64
```

Conclusion:

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

References:

www.tutorialspoint.com/unix_system_calls/

IPC: Interrupts and Signals

Subject:- Unix Operating System

System Lab Class :- TYIT

Name	**PRN**
Sumit Sunil Koundanya	**2019BTEIT00023**
Aniket Shivnath Sable	**2019BTEIT00020**

Assignment No - 2a

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

Objectives –

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

Theory-

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 cancelled.
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.

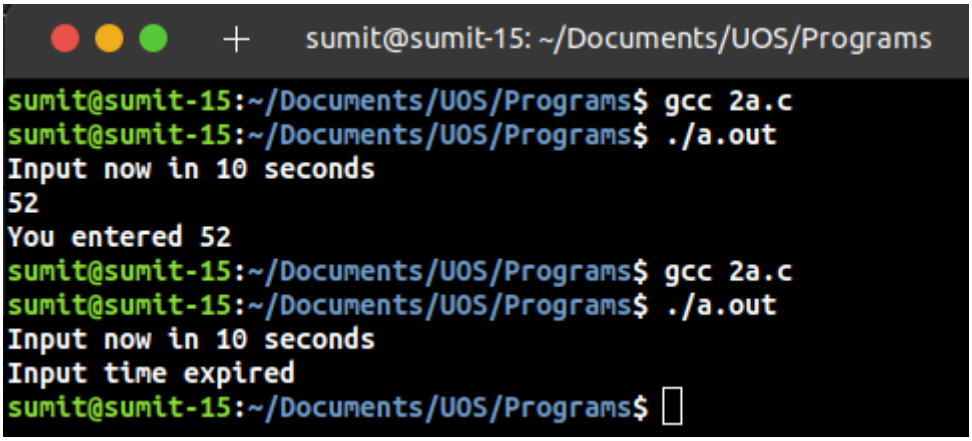
Using a signal handler function for a signal is called "catching the signal". The signals SIGKILL and SIGSTOP cannot be caught or ignored.

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 behaviour.

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-



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/

IPC: Interrupts and Signals

Assignment No - 2b

Title- Write a application or program that communicates between child and parent processes using kill() and signal().

Objectives –

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

Theory-

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);
```

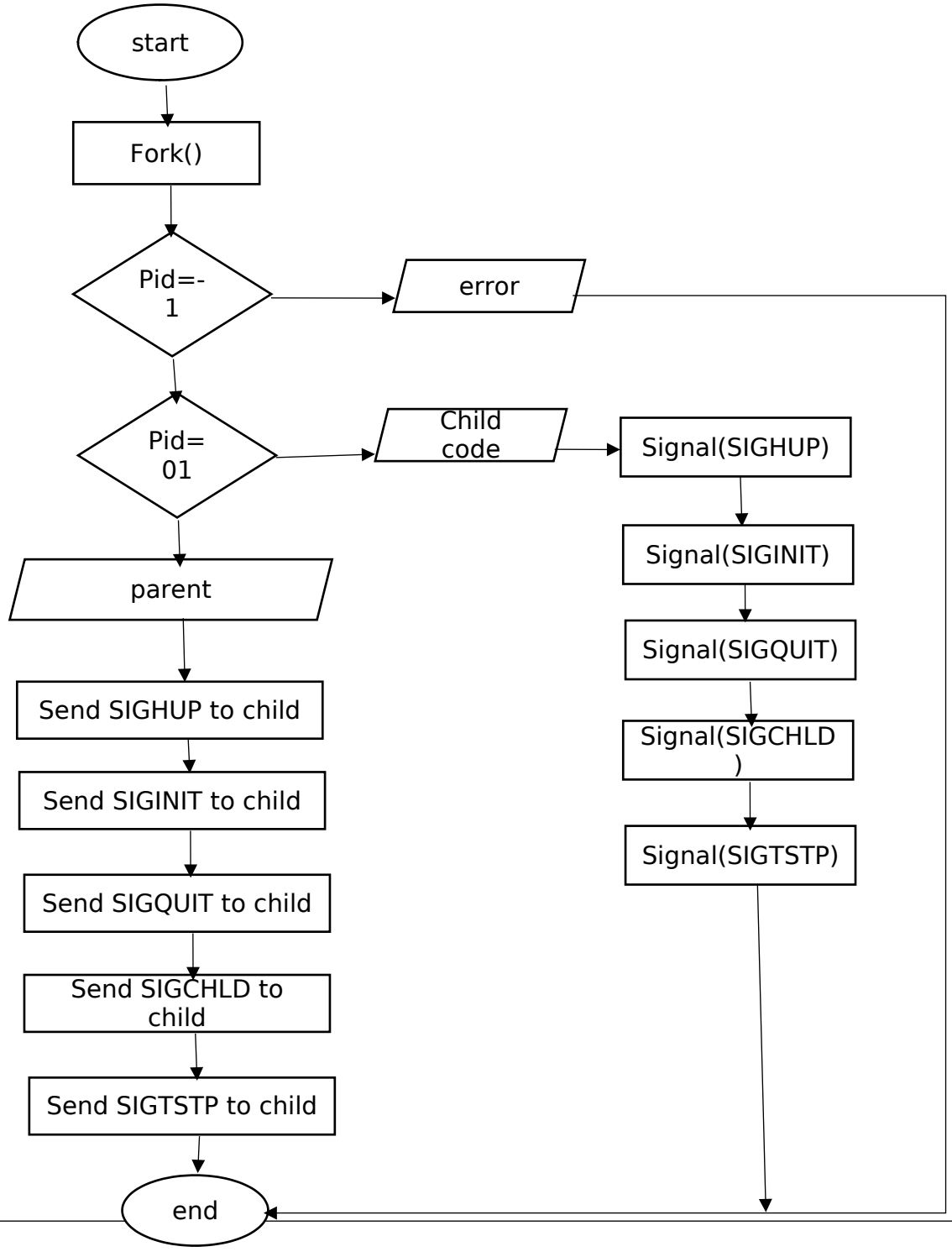
The signal() system call installs a new signal handler for the signal with number signum. The signal handler is set to sighandler which may be a user specified function, or either SIG_IGN or SIG_DFL.
Upon arrival of a signal with number signum the following happens. If the corresponding handler is set 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 behaviour.

Flowchart-

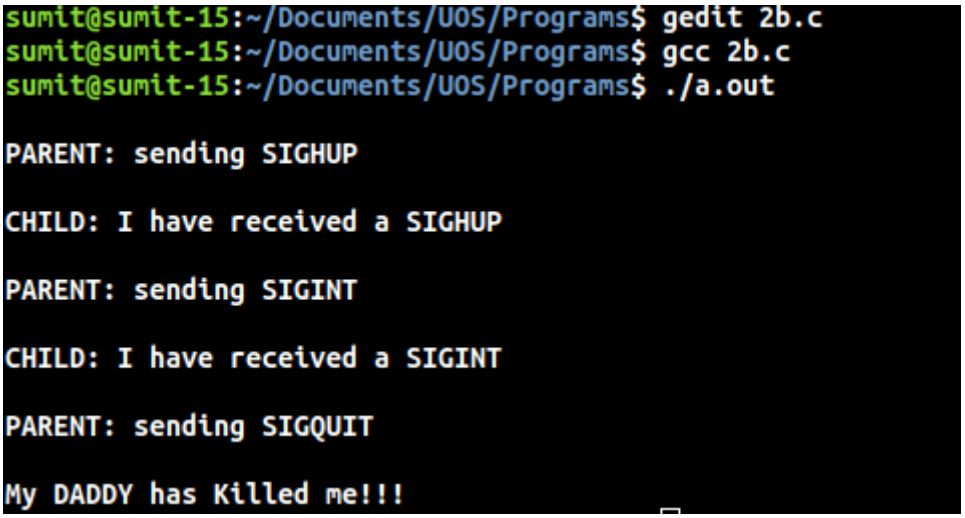


Program-

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
void sighup();
void sigint();
void sigquit();
void main()
{
        int pid;

        if ((pid = fork()) < 0) {
                perror("fork");
                exit(1);
        }
        if (pid == 0) {
                signal(SIGHUP, sighup);
                signal(SIGINT, sigint);
                signal(SIGQUIT, sigquit);
                for (;;)        ;
        }
        else
        {
                printf("\nPARENT: sending SIGHUP\n\n");
                kill(pid, SIGHUP);
                sleep(3);
                printf("\nPARENT: sending SIGINT\n\n");
                kill(pid, SIGINT);
                sleep(3);
                printf("\nPARENT: sending SIGQUIT\n\n");
                kill(pid, SIGQUIT);
                sleep(3);
        }
}


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

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

Output-



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/

IPC: Interrupts and Signals

Subject:- Unix Operating System

System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

Assignment No - 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*);

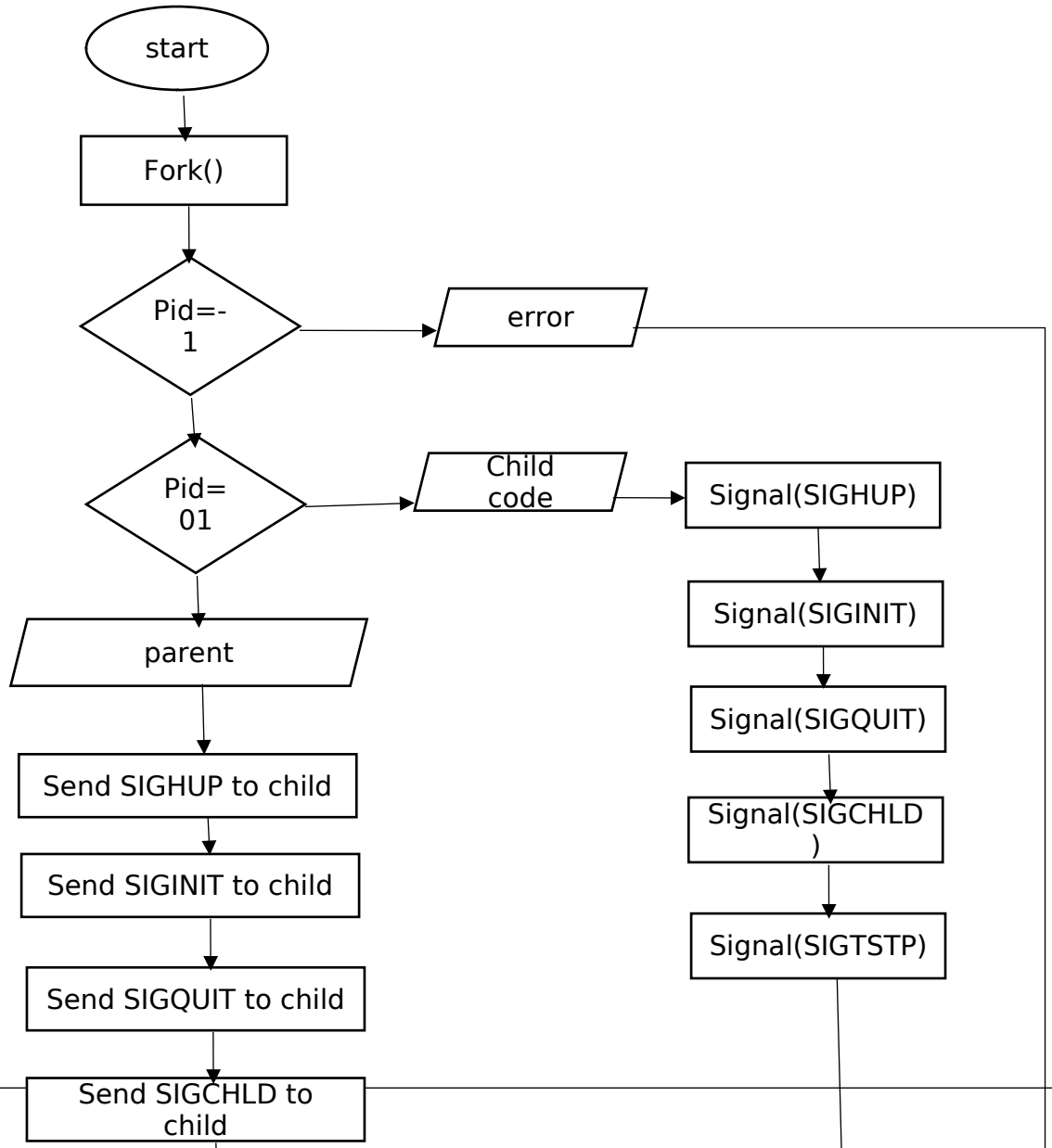
The signal() system call installs a new signal handler for the signal with number signum. The signal handler is set to sighandler which may be a user specified function, or either SIG_IGN or SIG_DFL.

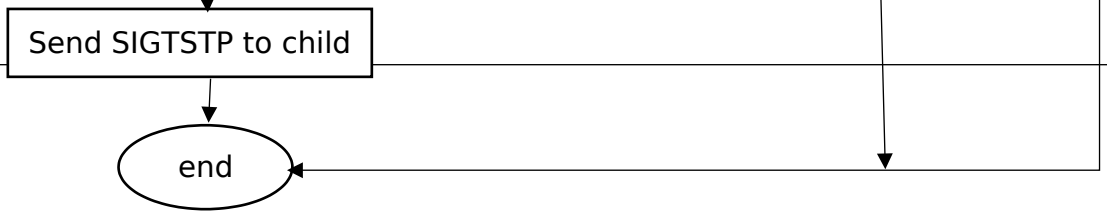
Upon arrival of a signal with number signum the following happens. If the corresponding handler is set to SIG_IGN, then the signal is ignored. If the handler is set to SIG_DFL, then the default action associated with the signal (see signal(7)) occurs. Finally, if the handler is set to a function sighandler 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 behaviour.

Flowchart-



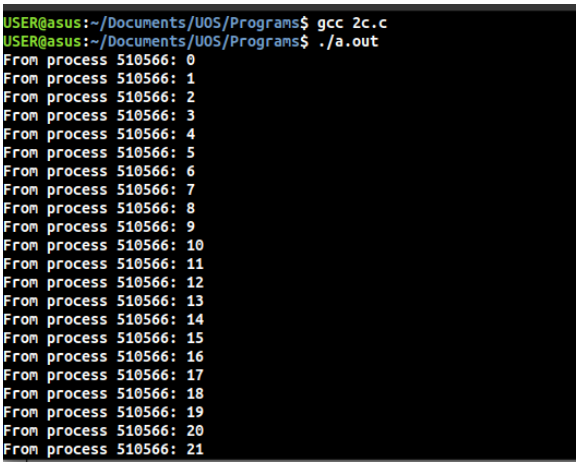


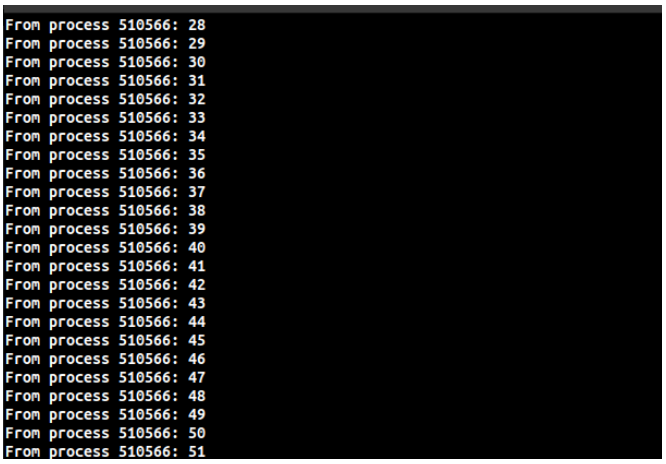
Program-

```
#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);
```

}

Output-





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/signal/kill.html

WALCHAND COLLEGE OF ENGINEERING SANGLI

Department of Information Technology

UNIX OPERATING SYSTEM LAB (3IT 371)

Year of Studentship: 2021-22

Term: Semester-2

Class: T.Y. IT

Name	PRN. No
Aniket Sable	2019BTEIT00020
Sumit Koundanya	2019BTEIT00023

Chapter 3 : File system Internals

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

Objectives :

1. To learn about File system Internals.

Theory :

Name:

stat, fstat, lstat - get file status

Syntax:

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

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

Description:

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

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

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

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

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

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

Data Dictionary :

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

Flowchart :

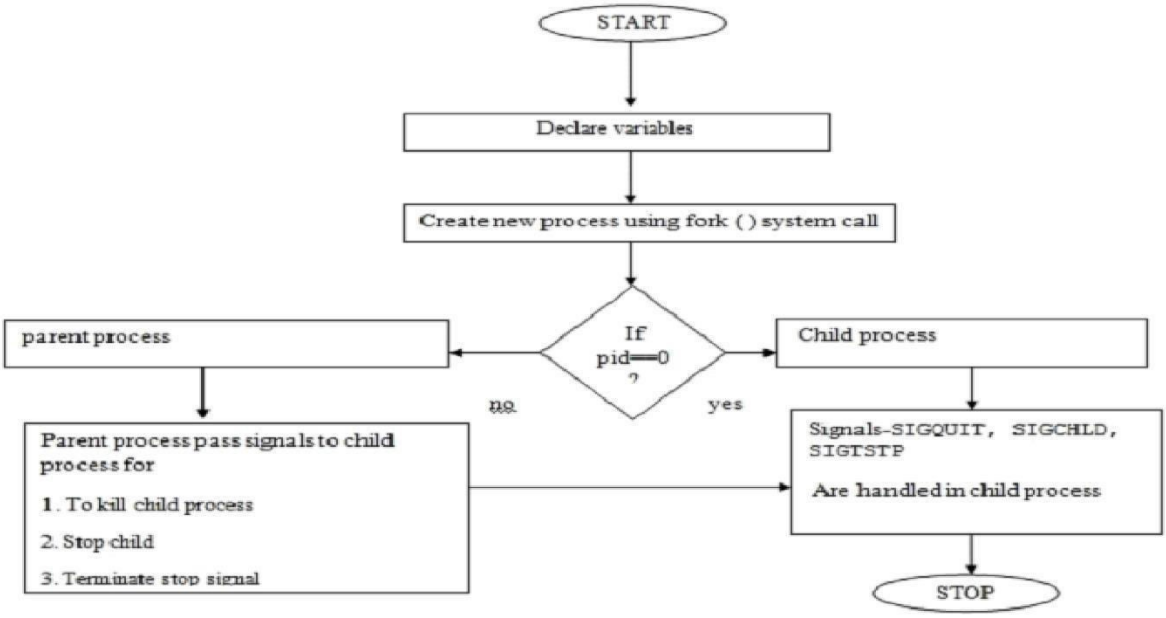


Fig: 3.1 Flowchart

Program :

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

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

Output :

```
sumit@sumit-15:~/Documents/UOS$ ./a.out
--------------------------
File Size:              0 bytes
Number of Links:        2
File inode:             5919243
File Permissions:       -rw-rw-r--

The file is not a symbolic link
```

Conclusion :

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

References :

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

File system Internals

3b: File system Internals: stat, fstat, ustat,

Subject:- Unix Operating
System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

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

Objectives:

1. To learn about File system Internals.

Theory:

Name:

stat, fstat, lstat - get file status

Syntax:

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

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

Description:

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

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

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

File system Internals

3b: File system Internals: stat, fstat, ustat,

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

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

```
struct stat {

dev_t     st_dev;     /* ID of device containing file */ino_t     st_ino;  /* inode number */ mode_t
                      st_mode;    /* protection */
nlink_t   st_nlink;   /* number of hard links */

uid_t     st_uid;     /* user ID of owner */

gid_t     st_gid;     /* group ID of owner */

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

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

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

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

time_t    st_atime;    /* time of last access */

time_t    st_mtime;   /* time of last modification */

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

 };
```

Data Dictionary:

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

File system Internals

3b: File system Internals: stat, fstat, ustat,

Program:

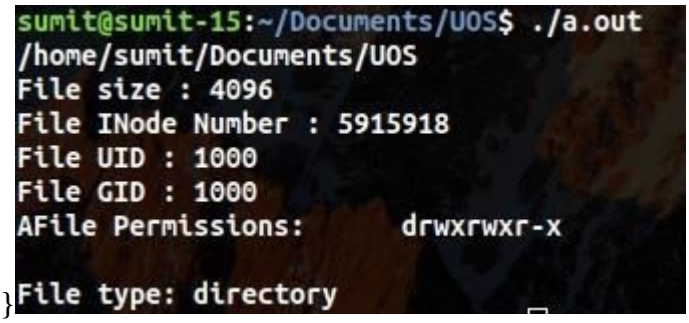
```
#include<stdio.h>
#include<stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h> int
int main(){
    char s[100]; gets(s);
    //printf("%s",s);
    FILE *fp;
    if((fp=fopen(s,"r"))==NULL) return 1;
    int fn=0; fn=fileno(fp);
    struct stat sta;
    if(fstat(fn,&sta) < 0) return 1;
    printf("File size : %ld\n",(long)sta.st_size); printf("File
    INode Number : %ld\n",sta.st_ino); printf("File UID :
    %ld\n",(long)sta.st_uid); printf("File GID : %ld\n",
    (long)sta.st_gid);
    printf("File Permissions: \t");
    printf( (S_ISDIR(sta.st_mode)) ? "d" : "-");
    printf( (sta.st_mode & S_IRUSR) ? "r" : "-");
    printf( (sta.st_mode & S_IWUSR) ? "w" : "-");
    printf( (sta.st_mode & S_IXUSR) ? "x" : "-");
    printf( (sta.st_mode & S_IRGRP) ? "r" : "-");
    printf( (sta.st_mode & S_IWGRP) ? "w" : "-");
    printf( (sta.st_mode & S_IXGRP) ? "x" : "-");
    printf( (sta.st_mode & S_IROTH) ? "r" : "-");
    printf( (sta.st_mode & S_IWOTH) ? "w" : "-");
    printf( (sta.st_mode & S_IXOTH) ? "x" : "-"); printf("\n\
    n");
    printf("File type: ");
```

File system Internals

3b: File system Internals: stat, fstat, ustat,

```
    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:



}

File system Internals

3b: File system Internals: stat, fstat, ustat,

File GID : 1000

File Permissions: -rw-rw-r--

File type: regular file

Conclusion:

Stats of file like UID, GIDfile 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

File system Internals
3c: File system Internals: stat, fstat, ustat, link/unlink,dup

Subject:- Unix Operating System Lab
Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

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

Objectives -

To learn about File system Internals.

Theory -

A) **link** - make a new name for a file

Syntax-

#include <unistd.h>

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

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

If newpath exists it will not be overwritten.

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

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

Hard links, as created by link(), cannot span filesystems. Use symlink() if this is required.

POSIX.1-2001 says that link() should dereference oldpath if it is a symbolic link. However, Linux does not do so: if oldpath is a symbolic link, then newpath is created as a (hard) link to the same symbolic link file (i.e., newpath becomes a symbolic link to the same file that oldpath refers to). Some other implementations behave in the same manner as Linux.

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

Syntax-

#include <unistd.h>

int unlink(const char *pathname);

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

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

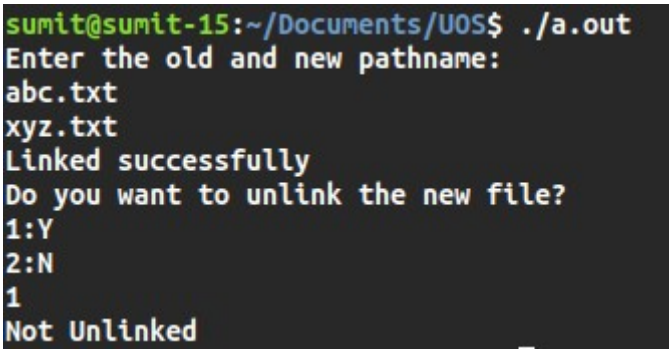
If the name referred to a symbolic link the link is removed. If the name referred to a socket, fifo or device the name for it is removed but processes which have the object open may continue to use it.

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

Program-

```
#include<stdio.h>
#include<unistd.h>
#include<string.h>
int main()
{
  char old[100];
  char new[100];
  char ch;
  printf("Enter the old and new pathname: \n");
  gets(old);
  gets(new);
  int n = link(old,new);
  if(n==0)
  {
     printf("Linked successfully\n");
  }
  else
 {
   printf("Linked unsuccessfully\n");
 }
 printf("Do you want to unlink the new file?\n1:Y\n2:N\n");
 scanf("%c",&ch);
 if(ch=='Y'||ch=='y')
 {
   int m = unlink(new);
   if(m==0)
   {
      printf("Unlinked successfully\n");
   }
   else
   {
     printf("Unlinked unsuccessfully\n");
   }
 }
 else
 {
   printf("Not Unlinked\n");
 }
}
```

Output:-



Conclusion:-

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

4a) Write a multi-threaded program in Java/c for chatting (multiuser and multi-terminal) using threads

Subject:- Unix Operating System
System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

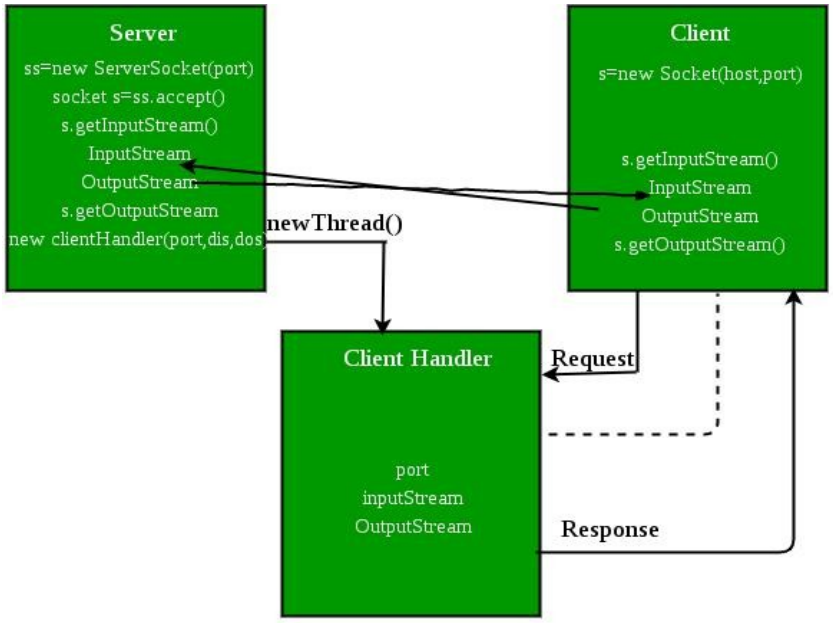
Objectives:

1. To learn about threading in Linux/Unix and Java and difference between them

2. Use of system call/library to write effective programs

Theory:

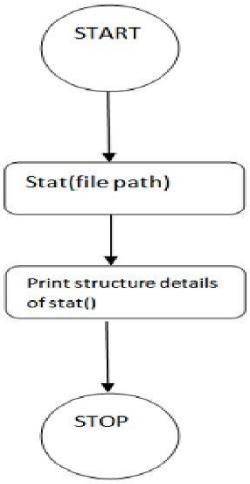
As normal, we will create two Java files, **Server.java** and **Client.java**. Server file contains two classes namely **Server** (public class for creating server) and **ClientHandler** (for handling any client using multi-threading). Client file contain only one public class **Client** (for creating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



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

1. **Establishing the Connection:** Server socket object is initialized and inside a while loop a socket object continuously accepts incoming connection.
2. **Obtaining the Streams:** The inputstream object and outputstream object is extracted from the current requests' socket object.
3. **Creating a handler object:** After obtaining the streams and port number, a new ClientHandler object (the above class) is created with these parameters.
4. **Invoking the start() method :** The start() method is invoked on this newly created thread object.
   ClientHandler class : As we will be using separate threads for each request, lets understand the working and implementation of the ClientHandler class extending Threads. An object of this class will be instantiated each time a request comes.
5. First of all this class extends Thread so that its objects assumes all properties of Threads.
6. Secondly, the constructor of this class takes three parameters, which can uniquely identify any incoming request, i.e. a **Socket**, a D**ataInput Stream** to read from and a DataOutputStream to write to. Whenever we receive any request of client, the server extracts its port number, the DataInputStream object and DataOutputStream object and creates a new thread object of this class and invokes start() method on it.
   Note : Every request will always have a triplet of socket, input stream and output stream. This ensures that each object of this class writes on one specific stream rather than on multiple streams.
7. Inside the **run()** method of this class, it performs three operations: request the user to specify whether time or date needed, read the answer from input stream object and accordingly write the output on the output stream object.

Flowchart:



Data Dictionary:

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

Program:

For Server:

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

```
	public static void main(String[] args) throws IOException
	{
		ServerSocket ss = new ServerSocket(1234);
		Socket s;
		while (true)
		{
			s = ss.accept();
			System.out.println("New client request received : " + s);
			DataInputStream dis = new DataInputStream(s.getInputStream());
			DataOutputStream dos = new DataOutputStream(s.getOutputStream());
			System.out.println("Creating a new handler for this client...");
			ClientHandler mtch = new ClientHandler(s,"client " + i, dis, dos);
			Thread t = new Thread(mtch);
			System.out.println("Adding this client to active client list");
			ar.add(mtch);
			t.start();
			i++;
		}
	}
}
class ClientHandler implements Runnable
{
	Scanner scn = new Scanner(System.in);
	private String name;
	final DataInputStream dis;
	final DataOutputStream dos;
	Socket s;
```

```
boolean isloggedin;


public ClientHandler(Socket s, String name, DataInputStream
dis, DataOutputStream dos)
{
	this.dis = dis;
	this.dos = dos;
	this.name = name;
	this.s = s;
	this.isloggedin=true;
}


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

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

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

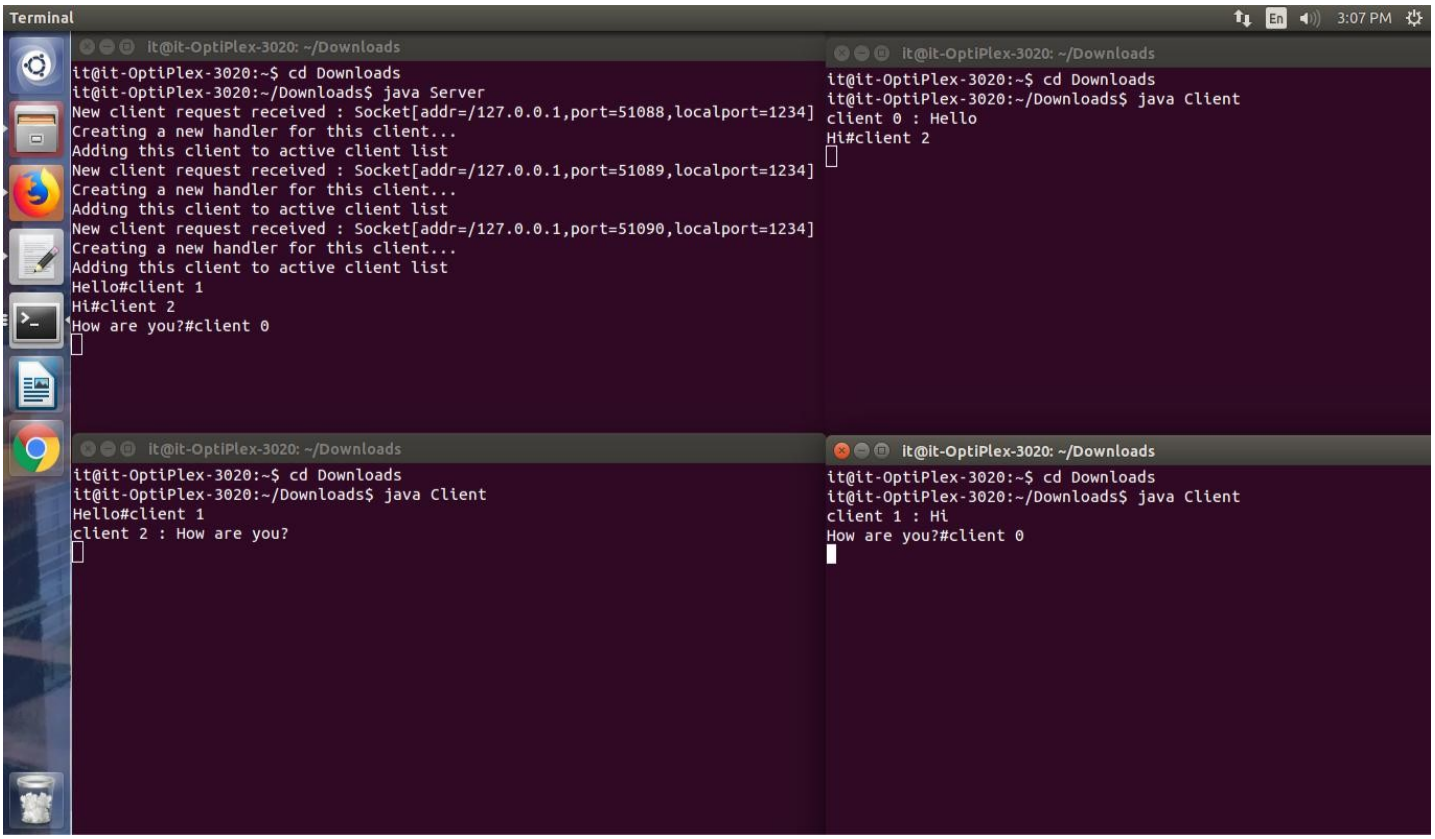
For Client:

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

```
final static int ServerPort = 1234;
public static void main(String args[]) throws UnknownHostException, IOException
{
	final Scanner scn = new Scanner(System.in);
	InetAddress ip = InetAddress.getByName("localhost");
	Socket s = new Socket(ip, ServerPort);
	final DataInputStream dis = new DataInputStream(s.getInputStream());
	final DataOutputStream dos = new
	DataOutputStream(s.getOutputStream()); Thread sendMessage = new
	Thread(new Runnable() {
		@Override
		public void run()
		{
			while (true)
			{
				String msg = scn.nextLine();
			try
			{
				dos.writeUTF(msg);
			} catch (IOException e){e.printStackTrace();}
		}
	}
});
Thread readMessage = new Thread(new Runnable()
{
@Override
public void run()
```

```
        {
        while (true)
        {
        try
        {
                String msg = dis.readUTF();
                System.out.println(msg);
        }catch (IOException e){e.printStackTrace();}
}
sendMessage.start();
readMessage.start();
}
}
```

Output:



Conclusion:

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

References:

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

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

Subject:- Unix Operating System
System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

Objectives:

1. To learn about threading in Linux/Unix and Java and difference between them

2. Use of system call/library to write effective programs

Theory:

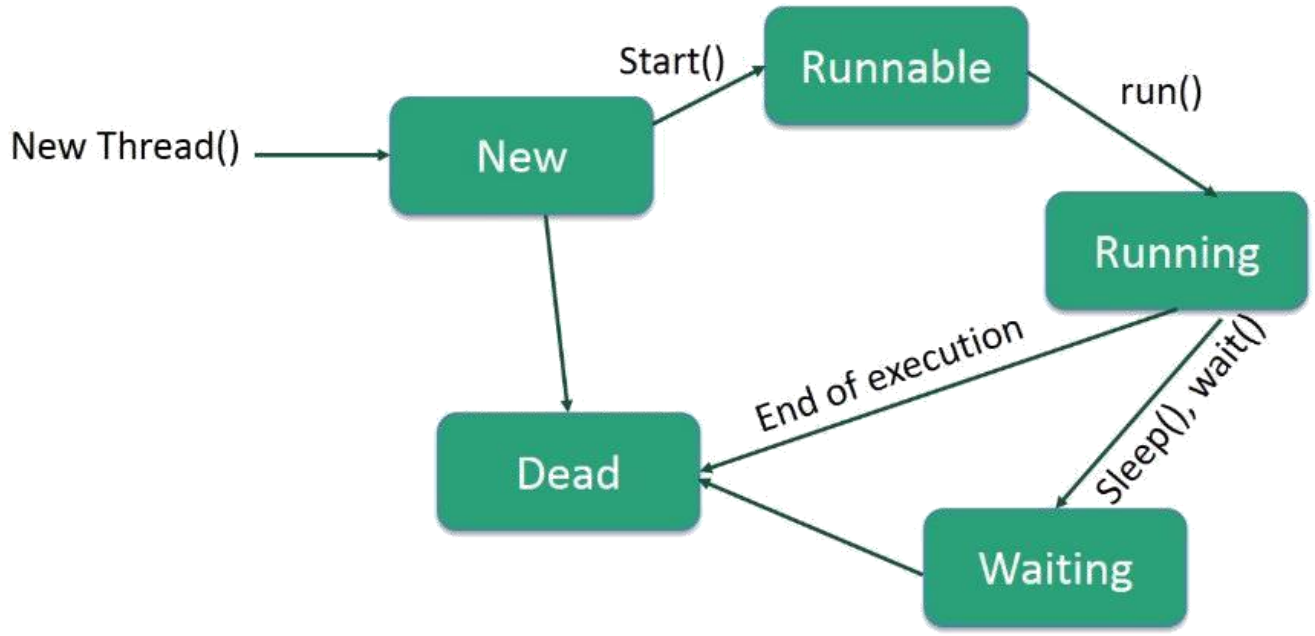
Java is a *multi-threaded programming language* which means we can develop multi-threaded program using Java. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially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

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

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

Life Cycle of a Thread:

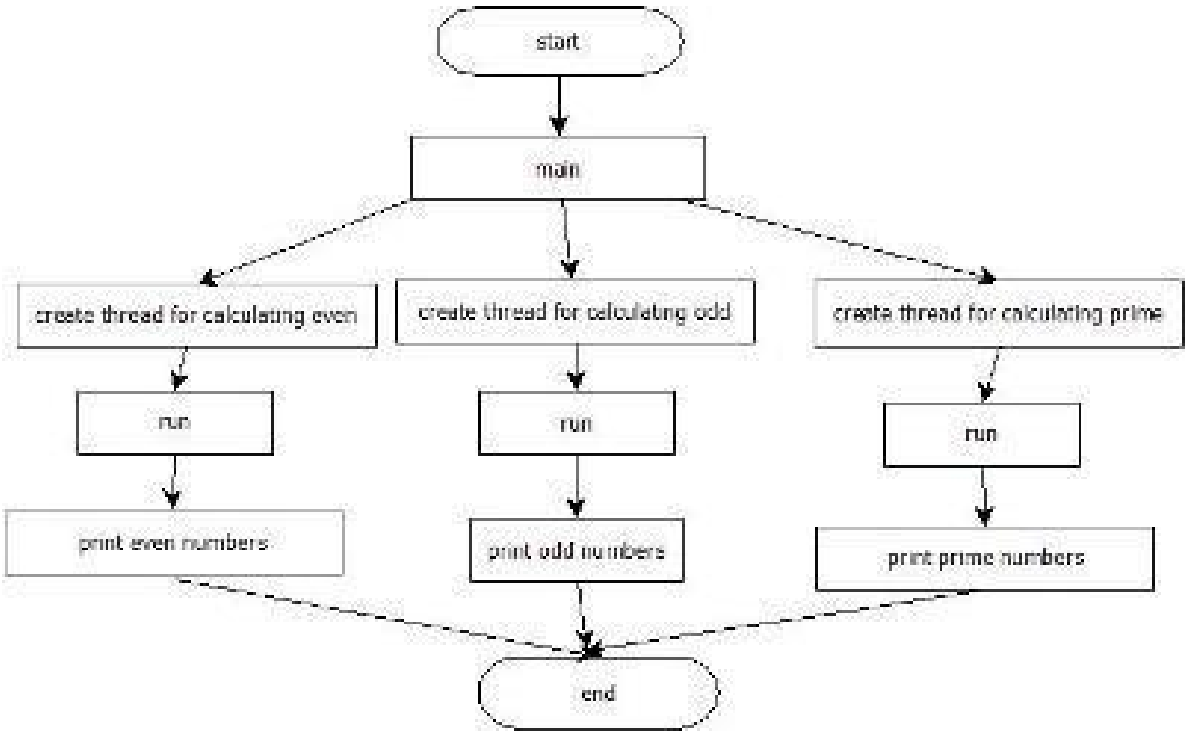
A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread.



Following are the stages of the life cycle:

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

Flowchart:



Data Dictionary:

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

Program:

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

```
                tm.start();
                t1.start();
                t2.start();
        }
        public void run()
        {
                for(int i=0;i<=30;i=i+2)
                {
                        System.out.println("Even: "+i);
                }
        }
}

class Thread1 extends Thread
{
public void run()
{
for(int i=1;i<=31;i=i+2)
{
System.out.println("Odd: "+i);
}
}
}
```

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

Output:

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

Prime: 5

Prime: 7

Even: 0

Even: 2

Odd: 1

Even: 4

Even: 6

Even: 8

Prime: 11

Even: 10

Odd: 3

Even: 12

Prime: 13

Even: 14

Odd: 5

Even: 16

Prime: 17

Even: 18

Odd: 7

Even: 20

Prime: 19

Even: 22

Odd: 9

Even: 24

Prime: 23

Even: 26

Odd: 11

Even: 28

Prime: 29

Even: 30

Odd: 13

Prime: 31

Odd: 15

Prime: 37

Odd: 17

Prime: 41

Odd: 19

Prime: 43

Odd: 21

Prime: 47

Odd: 23

Prime: 53

Odd: 25

Prime: 59

Odd: 27

Prime: 61

Odd: 29

Prime: 67

Odd: 31

Prime: 71

Prime: 73

Prime: 79

Prime: 83

Prime: 89

Prime: 97

Conclusion:

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

References:

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

Thread Concept

Subject:- Unix Operating System

System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

Assignment No - 4c

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

Objectives –

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

Theory-

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

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

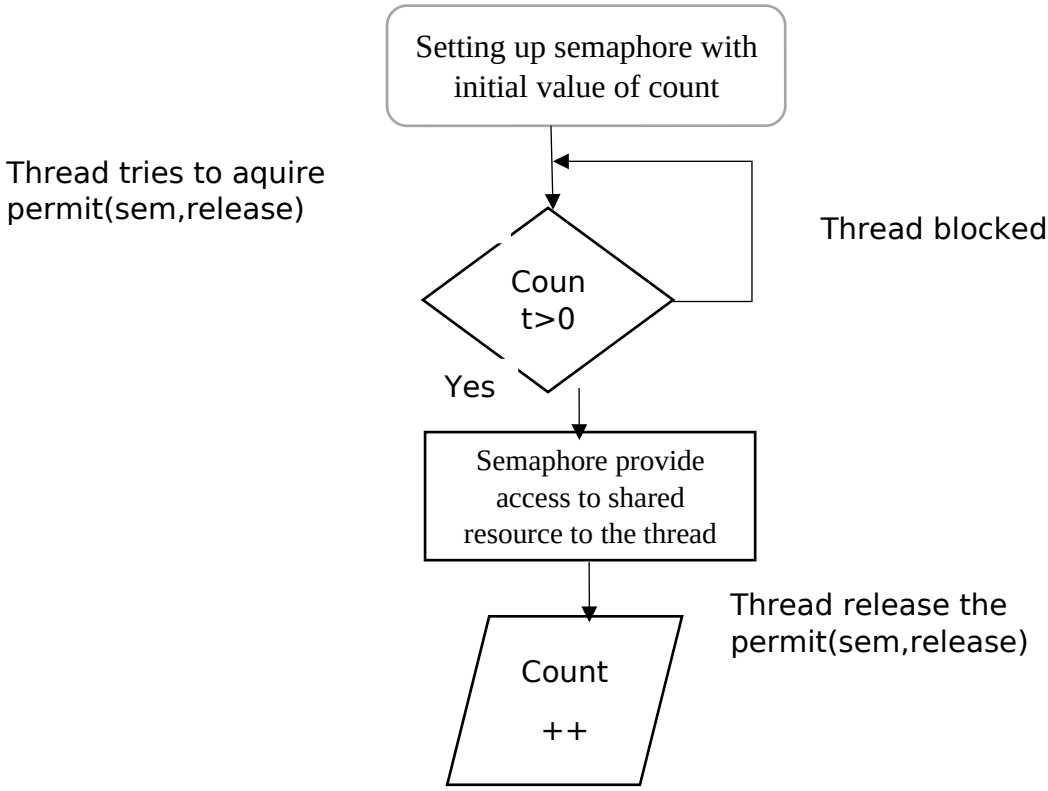
• If the semaphore's count is greater than zero, then the thread acquires a permit, which causes the semaphore's count to be decremented.

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

• When the thread no longer needs an access to the shared resource, it releases the permit, which causes the semaphore's count to be incremented.

• If there is another thread waiting for a permit, then that thread will acquire a permit at that time. Java provide Semaphore class in java.util.concurrent package that implements this mechanism, so you don't have to implement your own semaphores.

Flowchart-

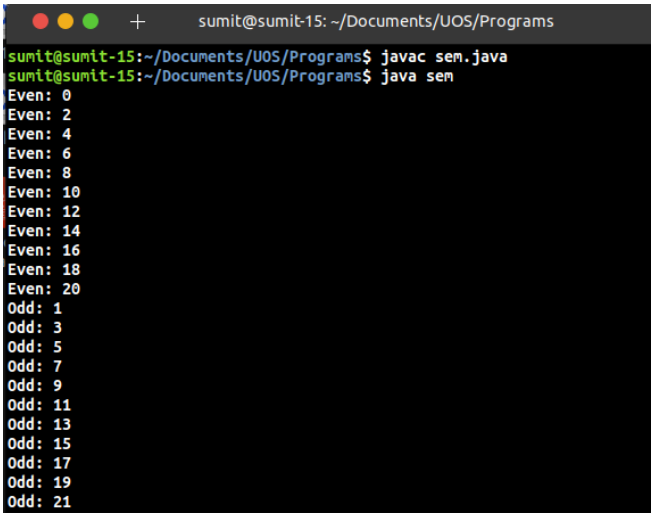


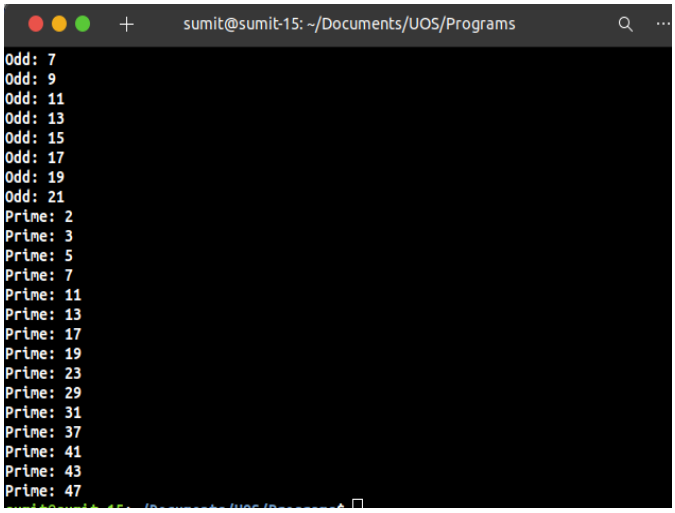
Program-

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

```
catch(InterruptedException exc){}
}
static class Thread1 extends Thread
{
public void run()
{
try
{
semaphore.acquire();
for(int i=1;i<=21;i=i+2)
{
System.out.println("Odd: "+i);
}
semaphore.release();
}
catch(InterruptedException exc){}
}
}
static class Thread2 extends Thread
{
public void run()
{
try
{
semaphore.acquire();
for(int i=2;i<50;i++)
{
int count = 0;
for(int j=2;j<i;j++)
{
if(i%j==0)
{
count++;
}
}
if(count==0)
System.out.println("Prime: "+i);
}
semaphore.release();
}
catch(InterruptedException exc){}
}
}
}
```

Output-





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

Shell Programming: Shell Scripts

Subject:- Unix Operating System

System Lab Class :- TYIT

Name	**PRN**
Sumit Sunil Koundanya	**2019BTEIT00023**
Aniket Shivnath Sable	**2019BTEIT00020**

Assignment No - 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.

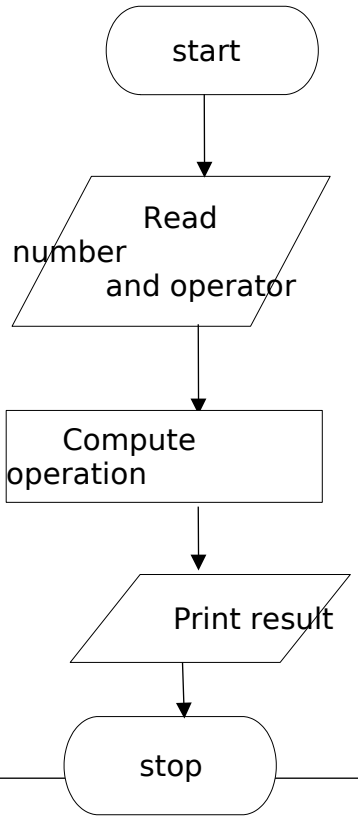
Theory-

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.

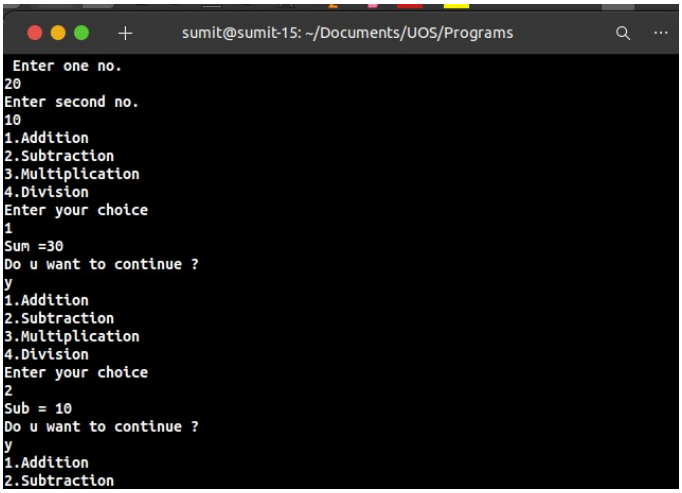
Flowchart-

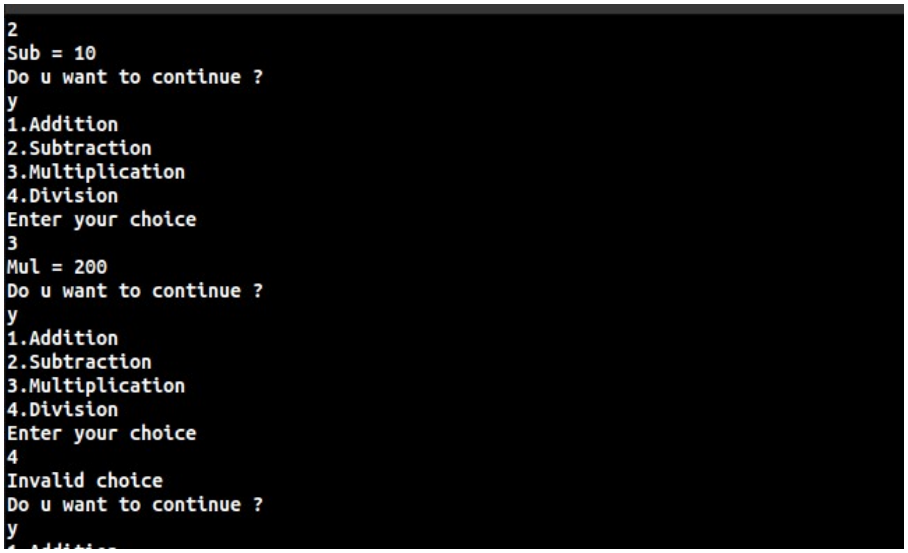


Program-

```
clear
sum=0
i="y"
echo " Enter one no."
read n1
echo "Enter second no."
read n2
while [ $i = "y" ]
do
echo "1.Addition"
echo "2.Subtraction"
echo "3.Multiplication"
echo "4.Division"
echo "Enter your choice"
read ch
case $ch in
1)sum=`expr $n1 + $n2`
echo "Sum ="$sum;;
2)sum=`expr $n1 - $n2`
echo "Sub = "$sum;;
3)sum=`expr $n1 \* $n2`
echo "Mul = "$sum;;
1284)sum=`expr $n1 / $n2`
echo "Div = "$sum;;
*)echo "Invalid choice";;
esac
echo "Do u want to continue ?"
read i
if [ $i != "y" ]
then
exit
fi
done
```

Output-





Conclusion:

Calculator constructed using shell programming

References:

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

Shell Programming: Shell Scripts

Subject:- Unix Operating System

System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

Assignment No - 5b

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

Objectives –

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

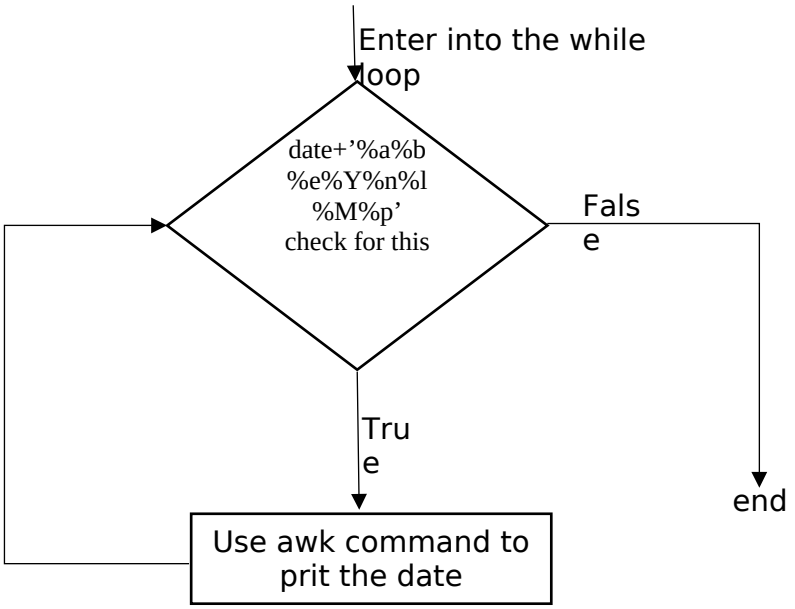
Theory-

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.

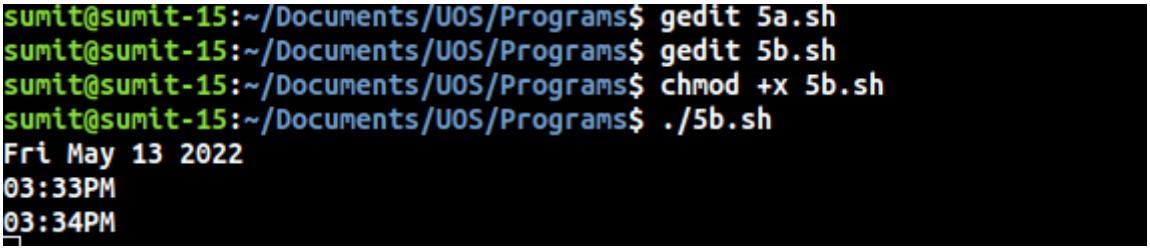
Flowchart-



Program-

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

Output-



Conclusion:

System date is retrieve using shell programming

References:

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

Shell Programming: Shell Scripts

Subject:- Unix Operating System

System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

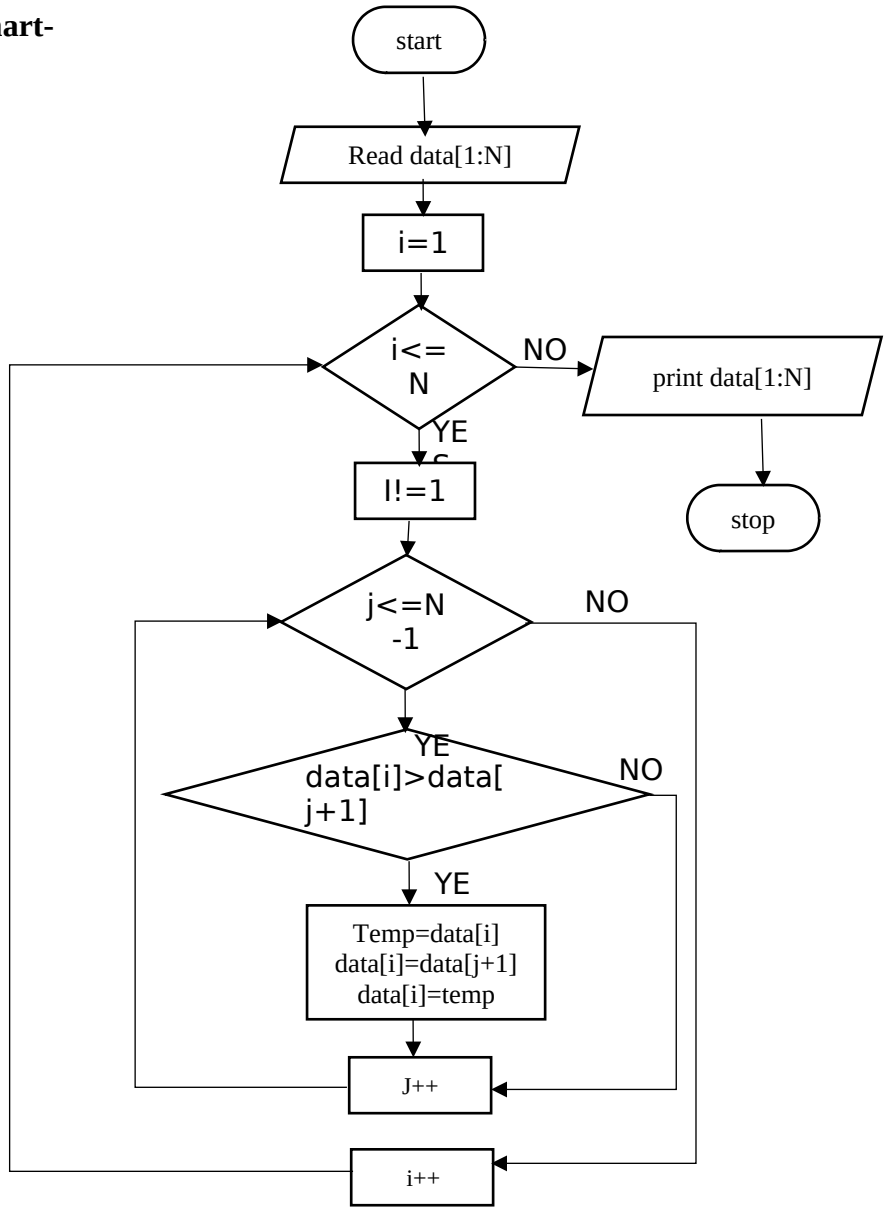
Assignment No - 5c

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

Objectives –

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

Flowchart-



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.

Program-

```
#!/bin/bash
echo "enter numbers"
read n;
declare -a a;
for((i=0;i<n;i++))
 do
   read a[$i];
 done
for((i=0;i<n;i++))
do
 for((j=i+1;j<n;j++))
  do
     {
       if((a[i]>a[j]))
       then
       temp=${a[i]};
        a[$i]=${a[j]};
        a[$j]=$temp;
       fi
     }
done
done
for((i=0;i<n;i++))
do
echo ${a[i]}
done
```

Output-

```
sumit@sumit-15:~/Documents/UOS/Programs$ bash ./5c.sh
enter numbers
10
5
2
1
3
6
10
10
12
36
52
1
2
3
5
6
10
10
12
36
52
```

Conclusion:

Array is sorted using shell programming

References:

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

IPC:Semaphores

Assignment No 6.a

Subject:- Unix Operating System

System Lab Class :- TYIT

Name PRN

Sumit Sunil Koundanya 2019BTEIT00023

Aniket Shivnath Sable 2019BTEIT00020

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

Program-

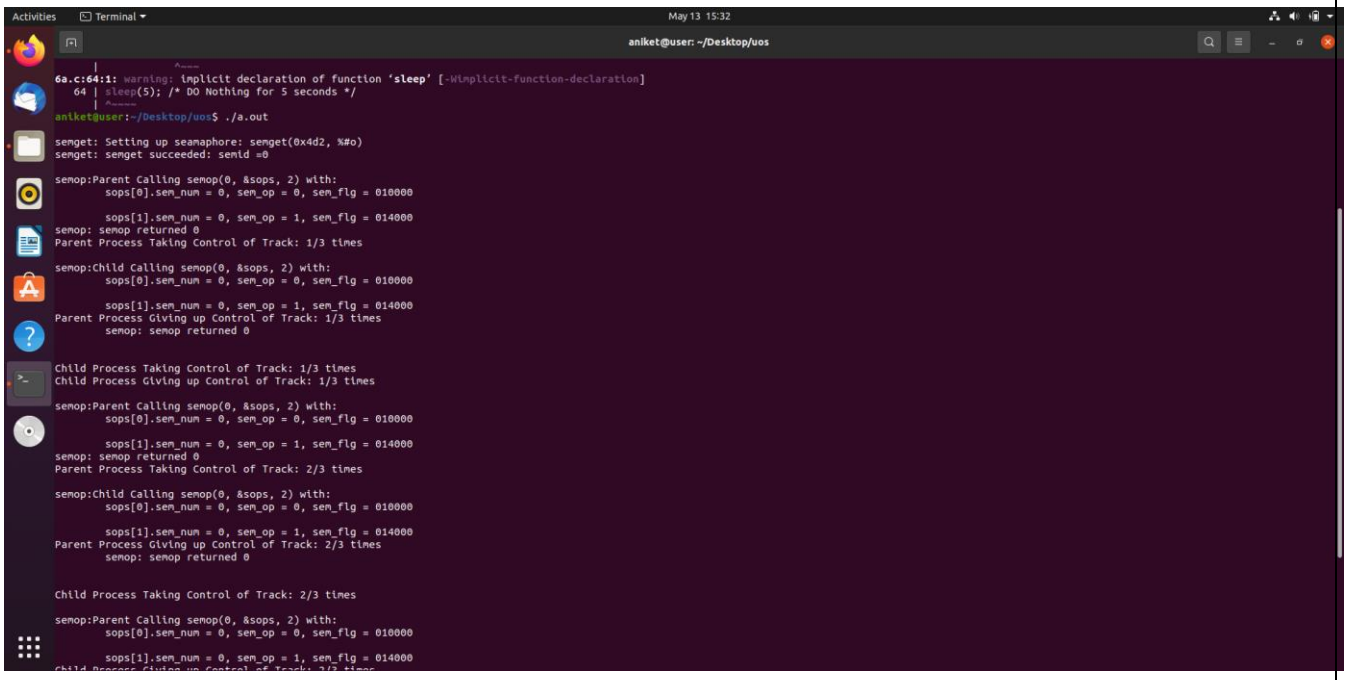
```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include<stdlib.h>
union semun {
int val;
struct semid_ds *buf;
ushort *array;
};
main()
{ int i,j;
int pid;
int semid; /* semid of semaphore set */
key_t key = 1234; /* key to pass to semget() */
int semflg = IPC_CREAT | 0666; /* semflg to pass to semget() */
int nsems = 1; /* nsems to pass to semget() */
```

```
int nsops; /* number of operations to do */
struct sembuf *sops = (struct sembuf *) malloc(2*sizeof(struct sembuf));
/* ptr to operations to perform */
/* set up semaphore */
(void) fprintf(stderr, "\nsemget: Setting up seamaphore: semget(%#lx, %\
%#o)\n",key, nsems, semflg);
if ((semid = semget(key, nsems, semflg)) == -1) {
perror("semget: semget failed");
exit(1);
}
else
(void) fprintf(stderr, "semget: semget succeeded: semid =\
%d\n", semid);
/* get child process */
if ((pid = fork()) < 0) {
perror("fork");
exit(1);
}
if (pid == 0)
{ /* child */
i = 0;
while (i < 3) {/* allow for 3 semaphore sets */
nsops = 2;
/* wait for semaphore to reach zero */
sops[0].sem_num = 0; /* We only use one track */
sops[0].sem_op = 0; /* wait for semaphore flag to become zero */
sops[0].sem_flg = SEM_UNDO; /* take off semaphore asynchronous */
sops[1].sem_num = 0;
sops[1].sem_op = 1; /* increment semaphore -- take control of track */
sops[1].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore */
/* Recap the call to be made. */
(void) fprintf(stderr,"\nsemop:Child Calling semop(%d, &sops, %d) with:", semid,
nsops);
for (j = 0; j < nsops; j++)
{
(void) fprintf(stderr, "\n\tsops[%d].sem_num = %d, ", j, sops[j].sem_num);
(void) fprintf(stderr, "sem_op = %d, ", sops[j].sem_op);
(void) fprintf(stderr, "sem_flg = %#o\n", sops[j].sem_flg);
}
/* Make the semop() call and report the results. */
if ((j = semop(semid, sops, nsops)) == -1) {
perror("semop: semop failed");
}
else
{
(void) fprintf(stderr, "\tsemop: semop returned %d\n", j);
(void) fprintf(stderr, "\n\nChild Process Taking Control of Track: %d/3 times\n", i+1);
sleep(5); /* DO Nothing for 5 seconds */
nsops = 1;
/* wait for semaphore to reach zero */
```

```
sops[0].sem_num = 0;
sops[0].sem_op = -1; /* Give UP COntrol of track */
sops[0].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore, asynchronous
*/
if ((j = semop(semid, sops, nsops)) == -1) {
perror("semop: semop failed");
}
else
(void) fprintf(stderr, "Child Process Giving up Control of Track: %d/3 times\n", i+1);
sleep(5); /* halt process to allow parent to catch semaphor change first */
}
++i;
}
}
else /* parent */
{ /* pid hold id of child */
i = 0;
while (i < 3) { /* allow for 3 semaphore sets */
nsops = 2;
/* wait for semaphore to reach zero */
sops[0].sem_num = 0;
sops[0].sem_op = 0; /* wait for semaphore flag to become zero */
sops[0].sem_flg = SEM_UNDO; /* take off semaphore asynchronous */
sops[1].sem_num = 0;
sops[1].sem_op = 1; /* increment semaphore -- take control of track */
sops[1].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore */
/* Recap the call to be made. */
(void) fprintf(stderr,"\nsemop:Parent Calling semop(%d, &sops, %d) with:", semid,
nsops);
for (j = 0; j < nsops; j++)
{
(void) fprintf(stderr, "\n\tsops[%d].sem_num = %d, ", j, sops[j].sem_num);
(void) fprintf(stderr, "sem_op = %d, ", sops[j].sem_op);
(void) fprintf(stderr, "sem_flg = %#o\n", sops[j].sem_flg);
}
/* Make the semop() call and report the results. */
if ((j = semop(semid, sops, nsops)) == -1) {
perror("semop: semop failed");
}
else
{
(void) fprintf(stderr, "semop: semop returned %d\n", j);
(void) fprintf(stderr, "Parent Process Taking Control of Track: %d/3 times\n", i+1);
sleep(5); /* Do nothing for 5 seconds */
nsops = 1;
/* wait for semaphore to reach zero */
sops[0].sem_num = 0;
sops[0].sem_op = -1; /* Give UP COntrol of track */
if ((j = semop(semid, sops, nsops)) == -1) {
perror("semop: semop failed");
```

```
}
else
(void) fprintf(stderr, "Parent Process Giving up Control of Track: %d/3 times\n", i+1);
sleep(5); /* halt process to allow child to catch semaphor change first */
}
++i;
}
}
}
```

Output-



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

6 b. IPC:Semaphores

Subject:- Unix Operating System

System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

Problem Statement :

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

Objectives:

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

Theory:

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

Data Dictionary:

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

7	producer	void	Used to regulate producer action

Table: 6.2 Data Dictionary

Program :

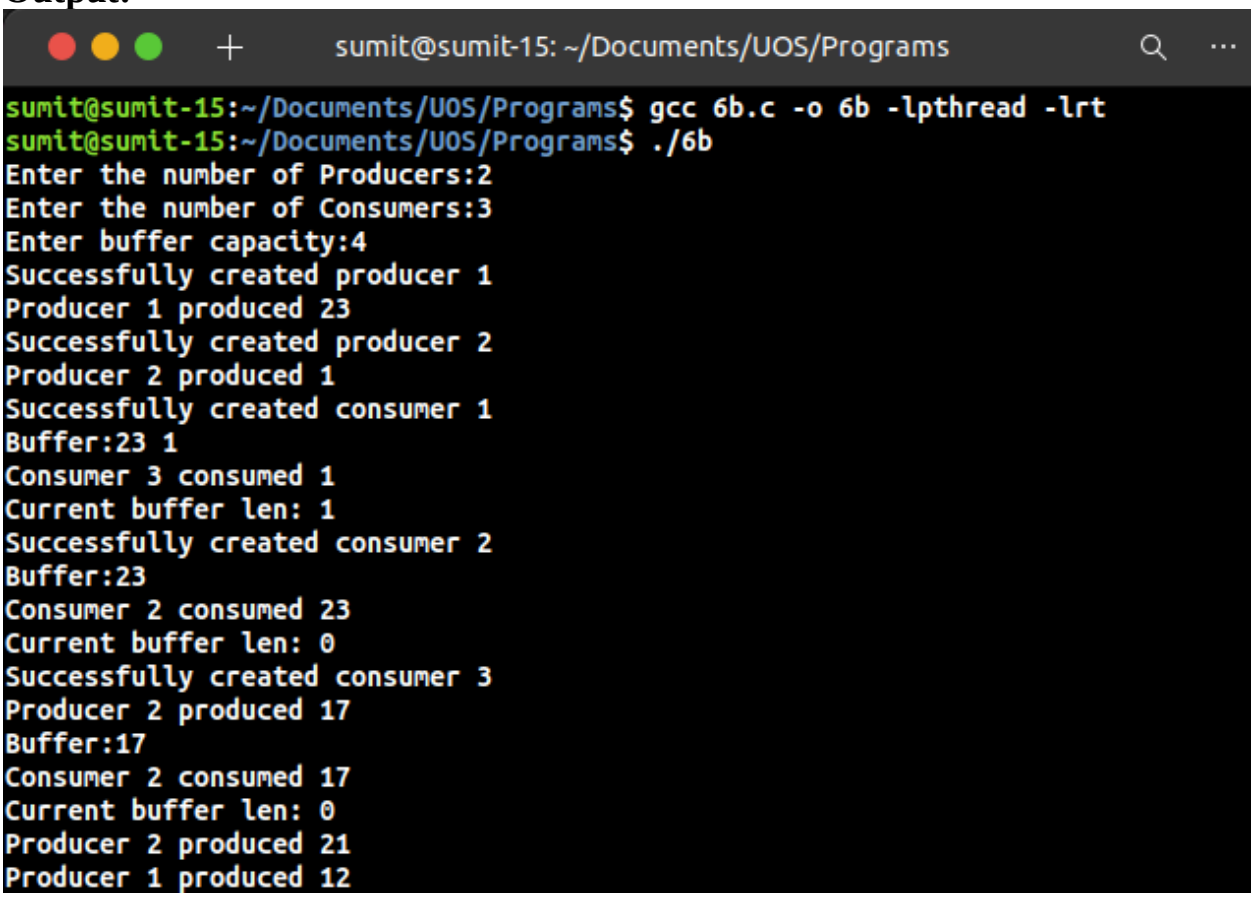
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>
/* use the pthread flag with gcc to compile this code
~$ gcc -pthread producer_consumer.c -o producer_consumer */
pthread_t *producers;
pthread_t *consumers;
sem_t buf_mutex,empty_count,fill_count;
int *buf,buf_pos=-
1,prod_count,con_count,buf_len; int
produce(pthread_t self){ int i = 0;

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

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

while(1){
int p = produce(pthread_self());
sem_wait(&empty_count);
sem_wait(&buf_mutex);
++buf_pos;                    // critical section
*(buf + buf_pos) = p;
sem_post(&buf_mutex);
sem_post(&fill_count);
sleep(1 + rand()%3);
}
return NULL;
}
void* consumer(void *args){
int c;
while(1){
sem_wait(&fill_count);
sem_wait(&buf_mutex);
```

```
c = *(buf+buf_pos);
consume(c,pthread_self());
--buf_pos;
sem_post(&buf_mutex);
sem_post(&empty_count);
sleep(1+rand()%5);
}
return NULL;
}
int main(void){
int i,err;
srand(time(NULL));
sem_init(&buf_mutex,0,1);
sem_init(&fill_count,0,0);
printf("Enter the number of Producers:");
scanf("%d",&prod_count);
producers = (pthread_t*)
malloc(prod_count*sizeof(pthread_t)); printf("Enter
the number of Consumers:");
scanf("%d",&con_count);
consumers = (pthread_t*) malloc(con_count*sizeof(pthread_t));
printf("Enter buffer capacity:");
scanf("%d",&buf_len);
buf = (int*) malloc(buf_len*sizeof(int));
sem_init(&empty_count,0,buf_len);
for(i=0;i<prod_count;i++){
```

```
err =
pthread_create(producers+i,NULL,&producer,NU
LL); if(err != 0){
printf("Error creating producer %d: %s\
n",i+1,strerror(err)); }else{
printf("Successfully created producer %d\n",i+1);
}
}
for(i=0;i<con_count;i++){
err =
pthread_create(consumers+i,NULL,&consumer,NU
LL); if(err != 0){
printf("Error creating consumer %d: %s\
n",i+1,strerror(err)); }else{
printf("Successfully created consumer %d\
n",i+1); }
}
for(i=0;i<prod_count;i++){
pthread_join(*(producers+i),NULL);
}
for(i=0;i<con_count;i++){
pthread_join(*(consumers+i),NULL);
}
return 0;
}
```

Output:



Conclusion:

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

References:

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

6 c. IPC:Semaphores

Subject:- Unix Operating System

System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

Problem Statement :

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

Objectives:

1.To learn about IPC through semaphore.

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

Theory:

A semaphore is a resource that contains an integer value, and allows processes to synchronize by testing and setting this value in a single atomic operation. This means that the process that tests the value of a semaphore and sets it to a different value (based on the test), is guaranteed no other process will interfere with the operation in the middle.

Two types of operations can be carried on a semaphore: wait and signal. A set operation first checks if the semaphore's value equals some number. If it does, it decreases its value and returns. If it does not, the operation blocks the calling process until the semaphore's value reaches the desired value. A signal operation increments the value of the semaphore, possibly awakening one or more processes that are waiting on the semaphore. How this mechanism can be put to practical use will be exlained later.

A semaphore set is a structure that stores a group of semaphores together, and possibly allows the process to commit a transaction on part or all of the semaphores in the set together. Here, a transaction means that we are guaranteed that either all operations are done successfully, or none is done at all. Note that a semaphore set is not a general parallel programming concept, it's just an extra mechanism supplied by SysV IPC.

Data Dictionary:

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

Program:

```
#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;
}
```

Compilation and Execution Steps

```
Reading process: shared memory: counter is 11000
Reading process: Reading done, Detaching shared memory
Reading Process: Complete
```

```
#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;
}
```

Compilation and Execution Steps

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
Writing Process: Complete
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:

Dave's Programming in C Tutorials

IPC: Message Queue

Subject:- Unix Operating System

System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

Assignment No 7a

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

Objective:

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

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	DataType	Use
1.	msqid	int	For Socket Tuple
2.	msgflg	int	For Semaphore
3.	key	key_t	Semaphore id

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-

```
sumit@sumit-15:~/Documents/UOS/Programs$ ./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
sumit@sumit-15:~/Documents/UOS/Programs$
```

```
sumit@sumit-15:~/Documents/UOS/Programs$ ./a.out
Did you get this?
sumit@sumit-15:~/Documents/UOS/Programs$
```

Conclusion-

Use of message queue functions like msgget, msgsend, and msgrecv to implement message passing mechanism between server and client studied and implemented it to intoduce concept of chatting.

Reference-

Dave’s Programming in C Tutorials

IPC: Message Queue

Subject:- Unix Operating System

System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

Assignment No 7b

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

Objective:

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

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	DataType	Use
1.	msqid	int	For Socket Tuple
2.	msgflg	int	For Semaphore
3.	key	key_t	Semaphore id
4.	sbuf	struct msgbuf	

Program-

Messagesend.c

```
#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;
```

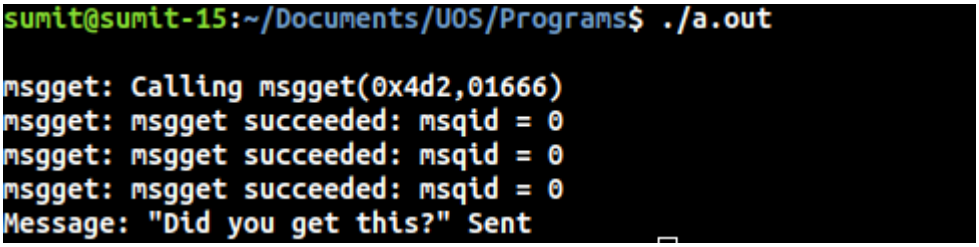
```
main()
{
   int msqid;
   int msgflg = IPC_CREAT | 0666;
   key_t key;
   message_buf sbuf;
   size_t buf_length;
   key = 1234;

(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);
}
```

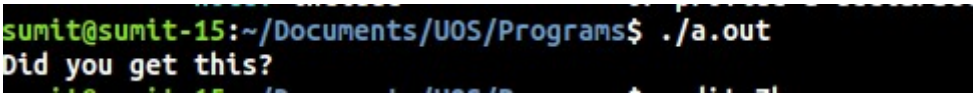
Messagreceive.c

```
#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;
   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-





Conclusion-

Use of message queue functions like msgget, msgsend, and msgrecv to implement message passing mechanism between server and client studied and implemented it to intoduce concept of chatting.

Reference-

Dave’s Programming in C Tutorials

1a: Processing Environment

Name:- Sumit Sunil Koundanya

PRN:-2019BTEIT00023 **Class:- TYIT**

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 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:

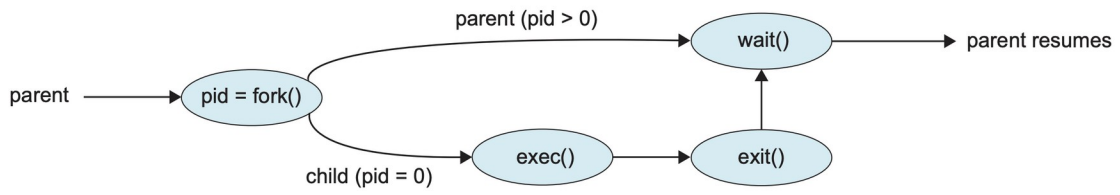


Fig: 1.1 Flowchart of fork

Data Dictionary:

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

Fig:1.1 Data Dictionary

Program:

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

int main(){

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

Output:

```
sumit@sumit-15:~/Documents/UOS$ gedit fork.c

sumit@sumit-15:~/Documents/UOS$ gcc fork.c

sumit@sumit-15:~/Documents/UOS$ ./a.out

Beginning

Parent process = 1

Parent process = 2

Parent process = 3

Parent process = 4

Parent process = 5

Parent Ended

Child process = 1

Child process = 2

Child process = 3

Child process = 4

Child process = 5

Child Ended
```

Conclusion:

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

References:

[1] www.tutorialspoint.com/unix_system_calls/

[2] https://users.cs.cf.ac.uk/Dave.Marshall/C/node22.html

IPC: Message Queue

Subject:- Unix Operating System

System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

Assignment No 7c

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

Objective:

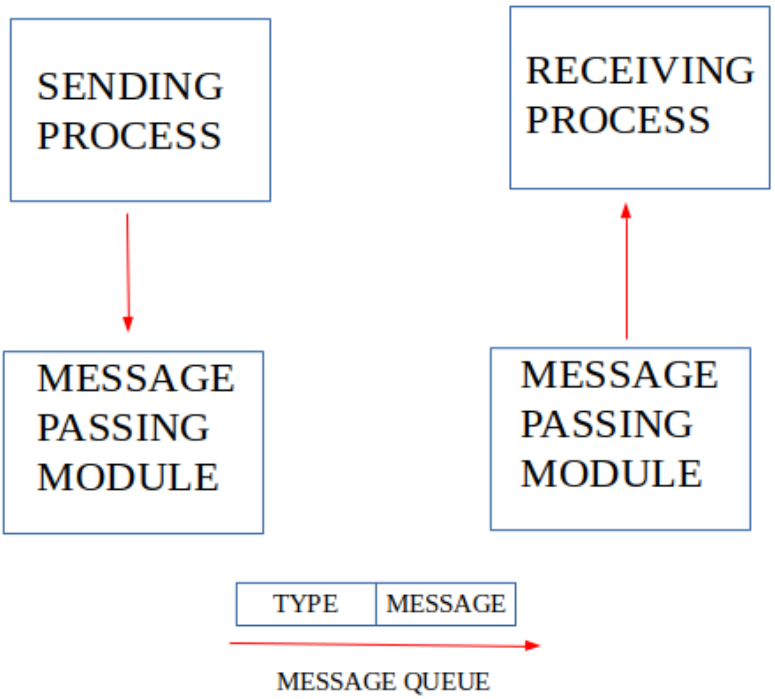
1. To learn about IPC through message queue.
2. Use of system call and IPC mechanism to write effective application programs

Theory:

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. A new queue is created or an existing queue opened by **msgget()**.

New messages are added to the end of a queue by **msgsnd()**. Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to msgsnd() when the message is added to a queue. Messages are fetched from a queue by **msgrcv()**. We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

All processes can exchange information through access to a common system message queue. The sending process places a message (via some (OS) message-passing module) 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.



Program-

Server-

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
extern void exit();
extern void perror();
main()
{
key_t key; /* key to be passed to msgget() */
int msgflg, /* msgflg to be passed to msgget() */
msqid; /* return value from msgget() */
(void) fprintf(stderr,"All numeric input is expected to follow C conventions:\
n");
(void) fprintf(stderr,"\t0x... is interpreted as hexadecimal,\n");
(void) fprintf(stderr, "\t0... is interpreted as octal,\n");
(void) fprintf(stderr, "\totherwise, decimal.\n");
(void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
(void) fprintf(stderr, "Enter key: ");
(void) scanf("%li", &key);
(void) fprintf(stderr, "\nExpected flags for msgflg argument are:\n");
(void) fprintf(stderr, "\tIPC_EXCL =\t%#8.8o\n", IPC_EXCL);
(void) fprintf(stderr, "\tIPC_CREAT =\t%#8.8o\n", IPC_CREAT);
(void) fprintf(stderr, "\towner read =\t%#8.8o\n", 0400);
(void) fprintf(stderr, "\towner write =\t%#8.8o\n", 0200);
(void) fprintf(stderr, "\tgroup read =\t%#8.8o\n", 040);
(void) fprintf(stderr, "\tgroup write =\t%#8.8o\n", 020);
(void) fprintf(stderr, "\tother read =\t%#8.8o\n", 04);
```

```
(void) fprintf(stderr, "\tother write =\t%#8.8o\n", 02);
(void) fprintf(stderr, "Enter msgflg value: ");
(void) scanf("%i", &msgflg);
(void) fprintf(stderr, "\nmsgget: Calling msgget(%#lx, %#o)\n",key, msgflg);if
((msqid = msgget(key, msgflg)) == -1)
{
perror("msgget: msgget failed");
exit(1);
}
else {
(void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);
exit(0);
}
}
```

Client 1-

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include<stdlib.h>
#include <time.h>
static void do_msgctl();
extern void exit();
extern void perror();
static char warning_message[] = "If you remove read permission for yourself,
this program will fail frequently!";
main()
{
struct msqid_ds buf;
int cmd,msqid;
(void) fprintf(stderr,"All numeric input is expected to follow C conventions:\
n");
(void) fprintf(stderr,"\t0x... is interpreted as hexadecimal,\n");
(void) fprintf(stderr, "\t0... is interpreted as octal,\n");
(void) fprintf(stderr, "\totherwise, decimal.\n");
(void) fprintf(stderr,"Please enter arguments for msgctls() as requested.");
(void) fprintf(stderr, "\nEnter the msqid: ");
(void) scanf("%i", &msqid);
(void) fprintf(stderr, "\tIPC_RMID = %d\n", IPC_RMID);
(void) fprintf(stderr, "\tIPC_SET = %d\n", IPC_SET);
(void) fprintf(stderr, "\tIPC_STAT = %d\n", IPC_STAT);
(void) fprintf(stderr, "\nEnter the value for the command: ");
(void) scanf("%i", &cmd);
switch (cmd) {
case IPC_SET:
(void) fprintf(stderr, "Before IPC_SET, get current values:");
case IPC_STAT:
do_msgctl(msqid,IPC_STAT,&buf);
(void) fprintf(stderr,"msg_perm.uid = %d\n",buf.msg_perm.uid);
(void) fprintf(stderr,"msg_perm.gid = %d\n",buf.msg_perm.gid);
```

```
(void) fprintf(stderr,"msg_perm.cuid = %d\n", buf.msg_perm.cuid);
(void) fprintf(stderr, "msg_perm.cgid = %d\n", buf.msg_perm.cgid);
(void) fprintf(stderr, "msg_perm.mode = %#o, ", buf.msg_perm.mode);
(void) fprintf(stderr, "access permissions = %#o\n", buf.msg_perm.mode & 0777);
(void) fprintf(stderr, "msg_cbytes = %d\n", buf.msg_cbytes);
(void) fprintf(stderr, "msg_qbytes = %d\n", buf.msg_qbytes);
(void) fprintf(stderr, "msg_qnum = %d\n", buf.msg_qnum);
(void) fprintf(stderr, "msg_lspid = %d\n", buf.msg_lspid);
(void) fprintf(stderr, "msg_lrpid = %d\n", buf.msg_lrpid);
(void) fprintf(stderr, "msg_stime = %s", buf.msg_stime ? ctime(&buf.msg_stime) : "Not Set\n");
(void) fprintf(stderr, "msg_rtime = %s", buf.msg_rtime ? ctime(&buf.msg_rtime) : "Not Set\n");
(void) fprintf(stderr, "msg_ctime = %s",ctime(&buf.msg_ctime));
if (cmd == IPC_STAT)
break;
(void) fprintf(stderr, "Enter msg_perm.uid: ");
(void) scanf ("%hi", &buf.msg_perm.uid);
(void) fprintf(stderr, "Enter msg_perm.gid: ");
(void) scanf("%hi", &buf.msg_perm.gid);
(void) fprintf(stderr, "%s\n", warning_message);
(void) fprintf(stderr, "Enter msg_perm.mode: ");
(void) scanf("%hi", &buf.msg_perm.mode);
(void) fprintf(stderr, "Enter msg_qbytes: ");
(void) scanf("%hi", &buf.msg_qbytes);
do_msgctl(msqid, IPC_SET, &buf);
break;
case IPC_RMID:
default:
do_msgctl(msqid, cmd, (struct msqid_ds *)NULL);
break;
}
exit(0);
}
static void
do_msgctl(msqid, cmd, buf)
struct msqid_ds *buf;
int cmd,msqid;
{
register int rtrn;
(void) fprintf(stderr, "\nmsgctl: Calling msgctl(%d,%d, %s)\n",msqid, cmd, buf ? "&buf" : "(struct msqid_ds*)NULL");
 rtrn = msgctl(msqid, cmd, buf);
if (rtrn == -1)
{
perror("msgctl: msgctl failed");
exit(1);
}
else
```

```
{
(void) fprintf(stderr, "msgctl: msgctl returned %d\n",rtrn);
}
}
```

Client 2-

```
#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);
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);
}
```

Output-

```
sumit@sumit-15:~$ ./a.out
All numeric input is expected to follow C conventions:
        0x... is interpreted as hexadecimal,
        0... is interpreted as octal,
        otherwise, decimal.
IPC_PRIVATE == 0
Enter key: 1234

Expected flags for msgflg argument are:
        IPC_EXCL =      00002000
        IPC_CREAT =     00001000
        owner read =    00000400
        owner write =   00000200
        group read =    00000040
        group write =   00000020
        other read =    00000004
        other write =   00000002
Enter msgflg value: 00002000

msgget: Calling msgget(0x4d2, 02000)
msgget: msgget succeeded: msqid = 0
```

```
sumit@sumit-15:~/Documents/UOS/Programs$ ./a.out
All numeric input is expected to follow C conventions:
        0x... is interpreted as hexadecimal,
        0... is interpreted as octal,
        otherwise, decimal.
Please enter arguments for msgctls() as requested.
Enter the msqid: 00002000
        IPC_RMID = 0
        IPC_SET = 1
        IPC_STAT = 2

Enter the value for the command: 1
Before IPC_SET, get current values:
msgctl: Calling msgctl(1024,2, &buf)
msgctl: msgctl failed: Invalid argument
```

```
sumit@sumit-15:~/Documents/UOS/Programs$ ./a.out

msgget: Calling msgget(0x4d2, 01666)
msgget: msgget succeeded: msqid = 0
0, 0, , 437096424
msgsnd: Invalid argument
```

Conclusion-

Use of message queue functions like msgget, msgsend, and msgrecv to implement message passing mechanism between server and client studied. Various clients communicated independently with server

Reference-

Dave's Programming in C Tutorials

IPC: Shared Memory

Subject:- Unix Operating System

System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

Assignment No 8a

Title-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.

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

Data Dictionary:

SR.NO	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

Program-

Server-

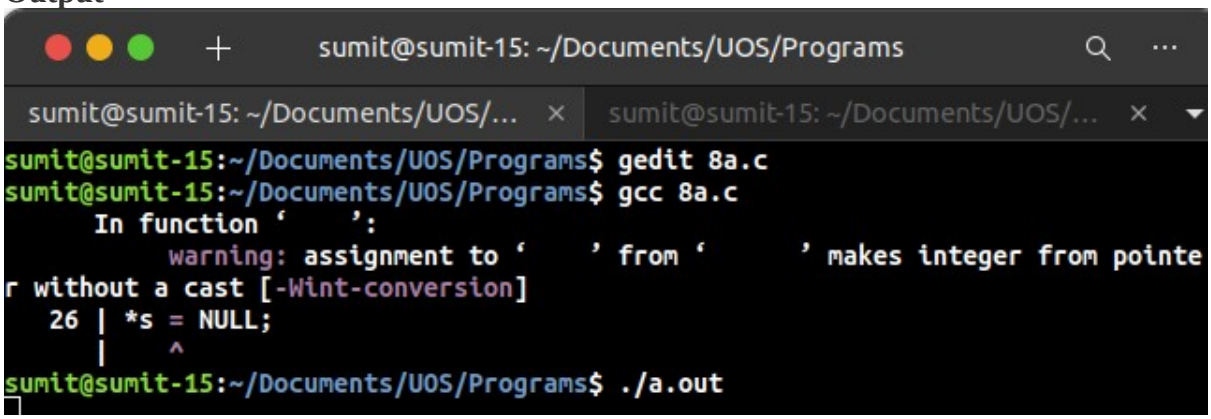
```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define SHMSZ 30
void main()
{
char c;
int shmid;
key_t key;
char *shm, *s;
key = 5858;
if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
perror("shmget");
exit(1);
}
if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
perror("shmat");
exit(1);
}
s = shm;
for (c = 'a'; c <= 'z'; c++)
*s++ = c;
*s = NULL;
while (*shm != '#')
sleep(1);
exit(0);
}
```

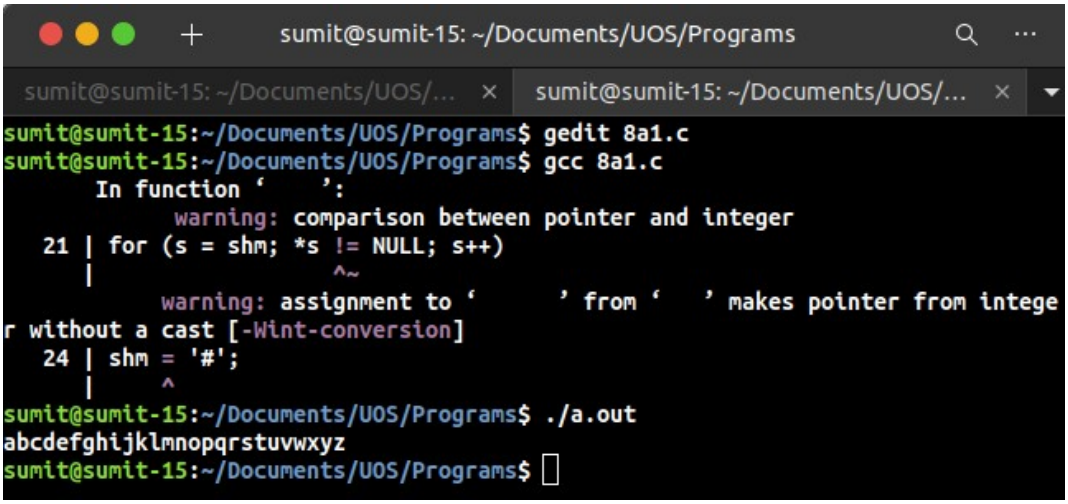
Client-

```
#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;
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);
}
```

Output-





Conclusion-

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

Reference-

Dave’s Programming in C Tutorials

IPC: Shared Memory

Subject:- Unix Operating System

System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

Assignment No 8b

Title-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.

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

Data Dictionary:

SR.NO	Variable/Function	Data Type	Use
1.	operations	struct sembuf	Used to Store Operations
2.	shm_address	void	Shared memory address
3.	shmid	int	Used to store Shared memory id
4.	semid	int	Used to store semaphore id
5.	semkey	Key_t	Store key of semaphores
6.	shmkey	key_t	Store key of SHM

Program-

Server-

```
#include <stdio.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#define SEMKEYPATH "/dev/null"
#define SEMKEYID 1
#define SHMKEYPATH "/dev/null"
#define SHMKEYID 1
#define NUMSEMS 2
#define SIZEOFSHMSEG 50
#define NUMMSG 2
int main(int argc, char *argv[])
{
int rc, semid, shmid, i;
key_t semkey, shmkey;
void *shm_address;
struct sembuf operations[2];
struct shmid_ds shmid_struct;
short sarray[NUMSEMS];
semkey = ftok(SEMKEYPATH,SEMKEYID);
if ( semkey == (key_t)-1 )
{
printf("main: ftok() for sem failed\n");
```

```
return -1;
}
shmkey = ftok(SHMKEYPATH,SHMKEYID);
if ( shmkey == (key_t)-1 )
{
printf("main: ftok() for shm failed\n");
return -1;
}
semid = semget( semkey, NUMSEMS, 0666 | IPC_CREAT | IPC_EXCL );
if ( semid == -1 )
{
printf("main: semget() failed\n");
return -1;
}
sarray[0] = 0;
sarray[1] = 0;
rc = semctl( semid, 1, SETALL, sarray);
if(rc == -1)
{
printf("main: semctl() initialization failed\n");
return -1;
}
shmid = shmget(shmkey, SIZEOFSHMSEG, 0666 | IPC_CREAT |
IPC_EXCL);
if (shmid == -1)
{
printf("main: shmget() failed\n");
return -1;
}
shm_address = shmat(shmid, NULL, 0);
if ( shm_address==NULL )
{
printf("main: shmat() failed\n");
return -1;
}
printf("Ready for client jobs\n");
for (i=0; i < NUMMSG; i++)
{
operations[0].sem_num = 1;
operations[0].sem_op = -1;
operations[0].sem_flg = 0;
operations[1].sem_num = 0;
operations[1].sem_op = 1;
operations[1].sem_flg = IPC_NOWAIT;
rc = semop( semid, operations, 2 );
if (rc == -1)
{
printf("main: semop() failed\n");
return -1;
}
```

```
printf("Server Received : \"%s\"\n", (char *) shm_address);
operations[0].sem_num = 0;
operations[0].sem_op = -1;
operations[0].sem_flg = IPC_NOWAIT;
rc = semop( semid, operations, 1 );
if (rc == -1)
{
printf("main: semop() failed\n");
return -1;
}
}
rc = semctl( semid, 1, IPC_RMID );
if (rc==-1)
{
printf("main: semctl() remove id failed\n");
return -1;
}
rc = shmdt(shm_address);
if (rc==-1)
{
printf("main: shmdt() failed\n");
return -1;
}
rc = shmctl(shmid, IPC_RMID, &shmid_struct);
if (rc==-1)
{
printf("main: shmctl() failed\n");
return -1;
}
return 0;
}
```

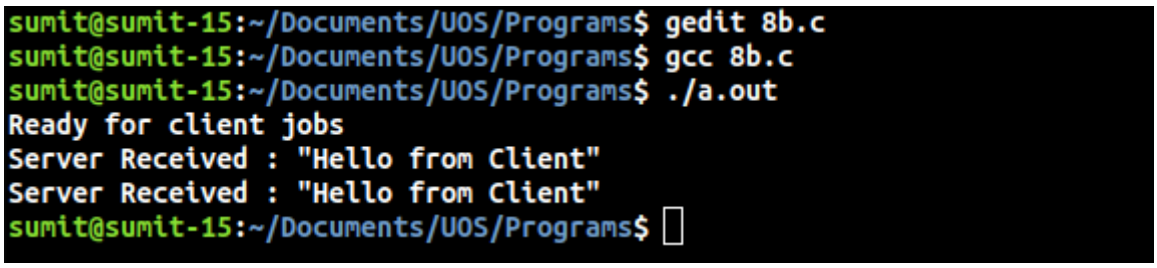
Client-

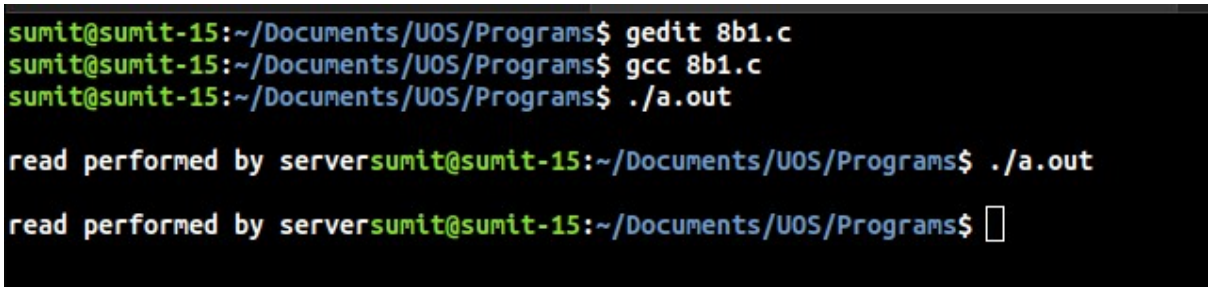
```
#include <stdio.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include<stdlib.h>
#define SEMKEYPATH "/dev/null"
#define SEMKEYID 1
#define SHMKEYPATH "/dev/null"
#define SHMKEYID 1
#define NUMSEMS 2
#define SIZEOFSHMSEG 50
int main(int argc, char *argv[])
{
struct sembuf operations[2];
void *shm_address;
int semid, shmid, rc;
key_t semkey, shmkey;
```

```
semkey = ftok(SEMKEYPATH,SEMKEYID);
if ( semkey == (key_t)-1 )
{
printf("main: ftok() for sem failed\n");
return -1;
}
shmkey = ftok(SHMKEYPATH,SHMKEYID);
if ( shmkey == (key_t)-1 )
{
printf("main: ftok() for shm failed\n");
return -1;
}
semid = semget( semkey, NUMSEMS, 0666);
if ( semid == -1 )
{
printf("main: semget() failed\n");
return -1;
}
shmid = shmget(shmkey, SIZEOFSHMSEG, 0666);
if (shmid == -1)
{
printf("main: shmget() failed\n");
return -1;
}
shm_address = shmat(shmid, NULL, 0);
if ( shm_address==NULL )
{
printf("main: shmat() failed\n");
return -1;
}
operations[0].sem_num = 0;
operations[0].sem_op = 0;
operations[0].sem_flg = 0;
operations[1].sem_num = 0;
operations[1].sem_op = 1;
operations[1].sem_flg = 0;
rc = semop( semid, operations, 2 );
if (rc == -1)
{
printf("main: semop() failed\n");
return -1;
}
strcpy((char *) shm_address, "Hello from Client");
operations[0].sem_num = 0;
operations[0].sem_op = -1;
operations[0].sem_flg = 0;
operations[1].sem_num = 1;
operations[1].sem_op = 1;
operations[1].sem_flg = 0;
rc = semop( semid, operations, 2 );
```

```
if (rc == -1)
{
printf("main: semop() failed\n");
return -1;
}
rc = shmdt(shm_address);
if (rc==-1)
{
printf("main: shmdt() failed\n");
return -1;
}
if(operations[0].sem_op == -1)
printf("\nread performed by server");
return 0;
}
```

Output-





Conclusion-

Communication using Shared Memory IPC between client and server established and implemented using shm functions.

Reference-

Dave’s Programming in C Tutorials

IPC: Shared Memory

Subject:- Unix Operating System

System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

Assignment No 8c

Title- 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.

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

Data Dictionary:

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

Program-

Server-

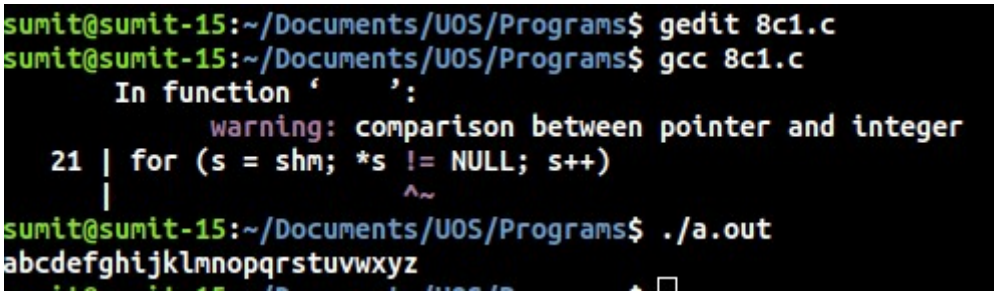
```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define SHMSZ 30
void main()
{
char c;
int shmid;
key_t key;
char *shm, *s;
key = 5858;
if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
perror("shmget");
exit(1);
}
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;
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);
}
```

Output-



Conclusion-

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

Reference-

Dave’s Programming in C Tutorials

IPC: Sockets

Subject:- Unix Operating System

System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

Assignment No 9a

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

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

Data Dictionary:

SR.NO	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	DataInputStream	Input Stream.
5.	str	String	String to display message from clients.

Program-

Server-

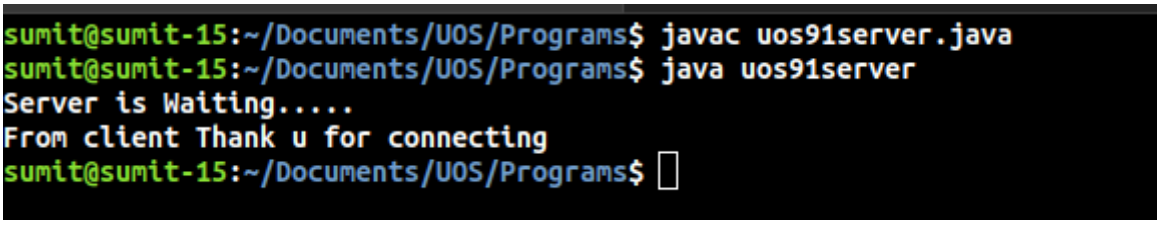
```
import java.net.*;
import java.io.*;
class uos91server
{
public static void main(String []args)throws Exception
{
ServerSocket ss=new ServerSocket(5050);
System.out.println("Server is Waiting.....");
Socket s=ss.accept();
DataOutputStream dos=new DataOutputStream(s.getOutputStream());
DataInputStream 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();
}
}
```

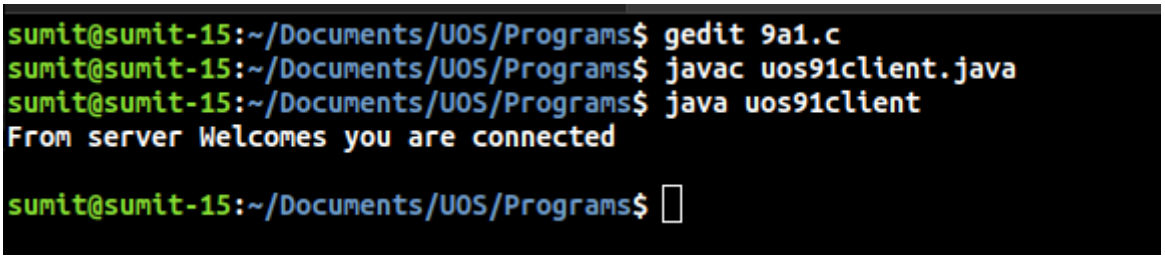
Client-

```
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-





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

IPC: Sockets

Subject:- Unix Operating System

System Lab Class :- TYIT

Name	**PRN**
Sumit Sunil Koundanya	**2019BTEIT00023**
Aniket Shivnath Sable	**2019BTEIT00020**

Assignment No 9b

Title- 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.(C or Java only) (use process concept for c server and thread for java server)

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

Data Dictionary:

SR.NO	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	DataInputStream	Input Stream.
5.	str	String	String to display message from clients.
6.	br	BufferedReader	Input data

Program-

Server-

```
import java.net.*;
import java.io.*;
class uos92server
{
public static void main(String []args)throws Exception
{
ServerSocket ss=new ServerSocket(5000);
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;
```

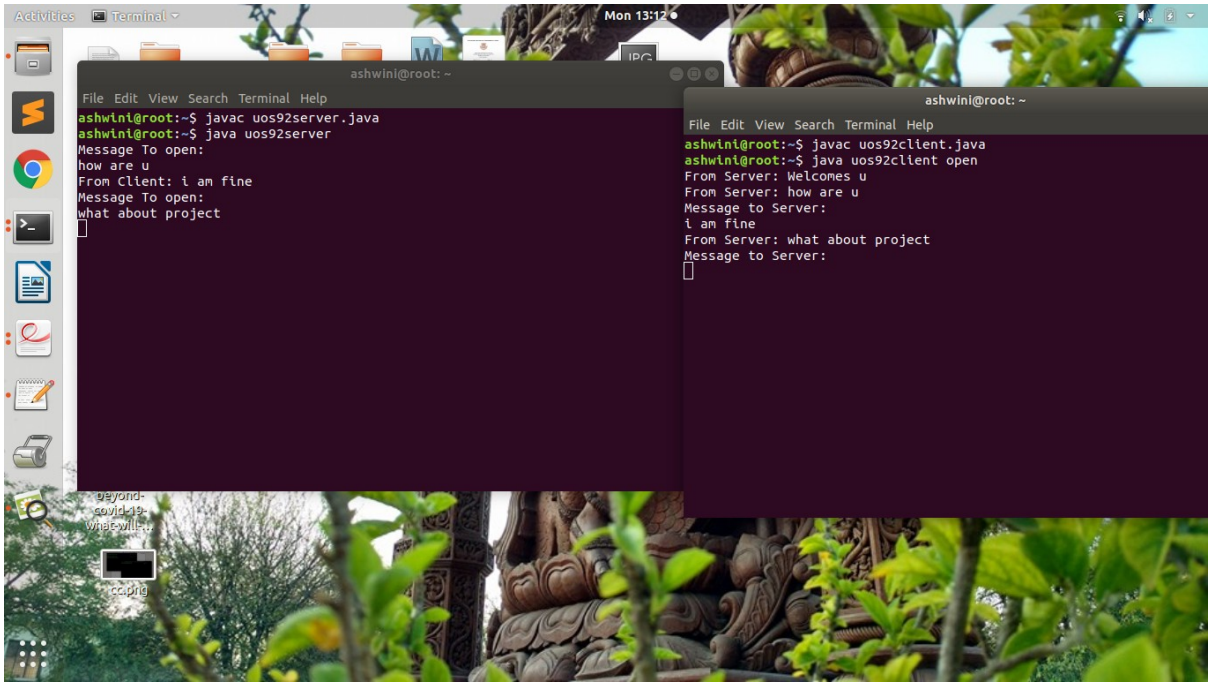
```
BufferedReader br;
String str,cnm;
ThrdComm(DataOutputStream dos,DataInputStream dis,String cnm)throws
Exception
{
super(cnm);
this.dos=dos;
this.dis=dis;
this.cnm=cnm;
br=new BufferedReader(new InputStreamReader(System.in));
start();
}
public void run()
{
while(true)
{
try
{
talk();
}
catch(Exception 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 uos92client extends Thread
{
public static void main(String []args)throws Exception
{
if(args.length!=1)
return;
uos92client a=new uos92client(args[0]);
}
uos92client(String s1)throws Exception
{
super(s1);//naming to thread
s=new Socket("localhost",5000);
dos=new DataOutputStream(s.getOutputStream());
dis=new DataInputStream(s.getInputStream());
```

```
br=new BufferedReader(new
InputStreamReader(System.in));
cnm=s1;//set aurgument as
String  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.

Reference-

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

IPC: Sockets

Subject:- Unix Operating System

System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

Assignment No 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;
```

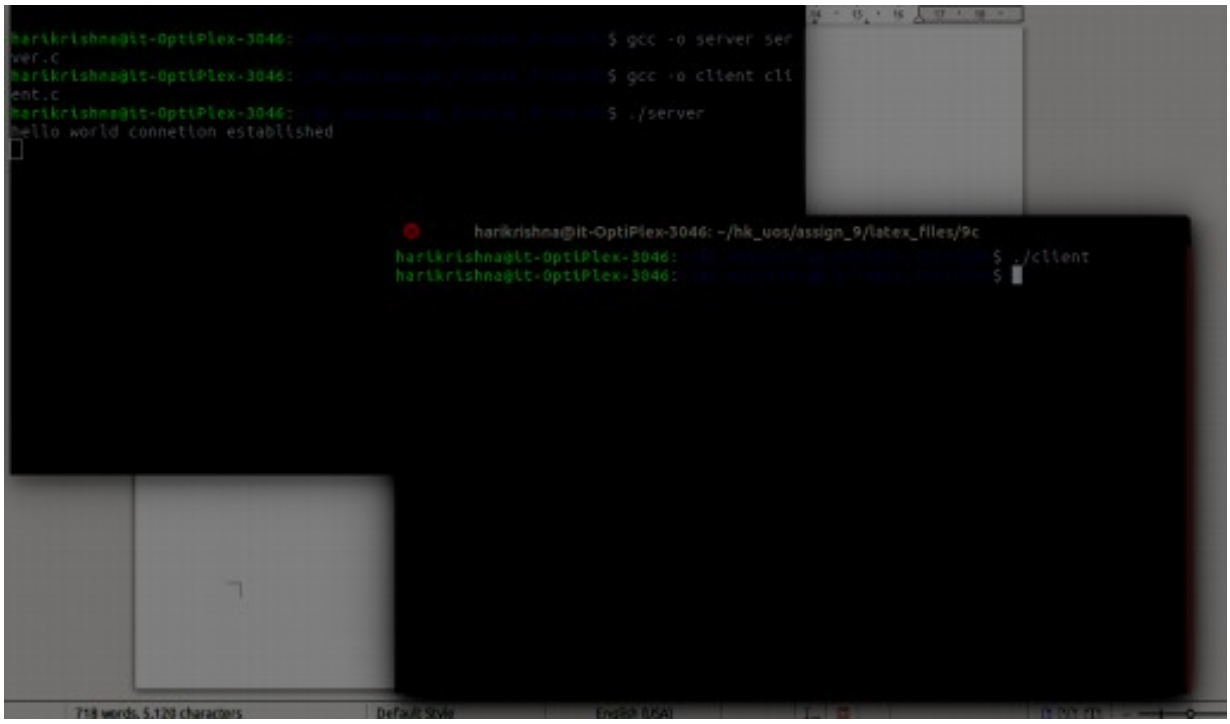
```
printf("%s\n", string);
close(newsockfd);
}
}
```

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

Python:As a scripting language

Subject:- Unix Operating System

System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

Assignment No 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.

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:

Subprocess Management:

The subprocess module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes. This module intends to replace several older modules and functions:

os.system
os.spawn*
os.popen*
popen2.*
commands.*

Using the subprocess module:

The recommended way to launch subprocesses is to use the following convenience functions. For more advanced use cases when these do not meet your needs, use the underlying Popen interface.

subprocess.call(args, *, stdin=None, stdout=None, stderr=None, shell=False)

Run the command described by args. Wait for command to complete, then return the returncode attribute.

The arguments shown above are merely the most common ones, described below in Frequently Used Arguments (hence the slightly odd notation in the abbreviated signature). The full function signature is the

same as that of the Popen constructor - this functions passes all supplied arguments directly through to that interface.

Program-

```
import os
rows, columns = os.popen('stty size', 'r').read().split()
r=int(rows)
c=int(columns)
n = int(input("Enter number of rows:"))
for i in range(int(r/2-n/2)):
print()
for i in range(n):
for k in range(int(c/2)-int(n/2)):
print(" ",end="")
for k in range(n-i-1):
print(" ",end="")
for k in range(2*i+1):
print("*",end="")
print("\n",end="")
for i in range(int(r/2-n/2)):
print()
```

Output-

```
Enter number of rows:6
 *
 ***
 *****
 *******
 *********
 ***********
```

Conclusion-

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

Reference-

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

Python:As a scripting language

Subject:- Unix Operating System

System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

Assignment No 10b

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.

Objective:

To learn about python as scripting option

Theory:

How do for loops work?

Many languages have conditions in the syntax of their for loop, such as a relational expression to determine if the loop is done, and an increment expression to determine the next loop value. In Python this is controlled instead by generating the appropriate sequence. Basically, any object with an iterable method can be used in a for loop. Even strings, despite not having an iterable method - but we'll not get on to that here. Having an iterable method basically means that the data can be presented in list form, where there are multiple values in an orderly fashion. You can define your own iterables by creating an object with next() and iter() methods.

Nested loops:

When you have a block of code you want to run x number of times, then a block of code within that code which you want to run y number of times, you use what is known as a "nested loop". In Python, these are heavily used whenever someone has a list of lists - an iterable object within.

Early Exits:

Like the while loop, the for loop can be made to exit before the given object is finished. This is done using the break statement, which will immediately drop out of the loop and contine execution at the first statement after the block. You can also have an optional else clause, which will run should the for loop exit cleanly - that is, without breaking.

Program-

```
lower = int(input("Enter lower range: "))
upper = int(input("Enter upper range: "))

for num in range(lower,upper + 1):
   if num > 1:
       for i in range(2,num):
           if (num % i) == 0:
               break
       else:
           print(num)
```

Output-

```
sumit@sumit-15:~/Documents/UOS/Programs$ python3 10b.py
Enter lower range: 3
Enter upper range: 15
3
5
7
11
13
sumit@sumit-15:~/Documents/UOS/Programs$
```

Conclusion-

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

Reference-

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

10.3 Take any txt file and count word frequencies in a file. (hint: file handling+basics)

Subject:- Unix Operating System
System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

Objectives:

1. To learn about python as scripting option.

Theory:

File handling:

A file is some information or data which stays in the computer storage devices. You already know about different kinds of file , like your music files, video files, text files. Python gives you easy ways to manipulate these files. Generally we divide files in two categories, text file and binary file. Text files are simple text where as the binary files contain binary data which is only readable by computer.

File opening:

To open a file we use open() function. It requires two arguments, first the file path or file name, second which mode it should open. Modes are like “r” -> open read only, you can read the file but can not edit / delete anything inside

- “w” -> open with write power, means if the file exists then delete all content and open it to write
- “a” -> open in append mode

  The default mode is read only, ie if you do not provide any mode it will open the file as read only. Let us open a file

```
> fobj = open("love.txt")

> fobj

<_io.TextIOWrapper name='love.txt' mode='r' encoding='UTF-8'>
```

Closing a file:

After opening a file one should always close the opened file. We use method close() for this.

```
> fobj = open("love.txt")

> fobj
```

<_io.TextIOWrapper name='love.txt' mode='r' encoding='UTF-8'>

>>> fobj.close()

Reading a file:

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

> fobj = open("sample.txt")

> fobj.read()

'I love Python\nPradeepto loves KDE\nSankarshan loves Openoffice\n'

Program:

```
f=open("10cfile.txt")
d={}
for line in f:
l=line.split()
#print(line)
#print(l)
for word in l:
#print(word)
if word in d:
d[word]=d[word]+1
else:
d[word]=1
for key in d:
print key," : ",d[key]
```

Output:

```
sarita@HP-Laptop-15g-dr0xxx:~$  python uox10.py
a : 2
A : 1
Peter : 4
```

of : 4

Piper : 4

pickled : 4

Where's : 1

picked : 4

peppers : 4

the : 1

peck : 4

If : 1

Conclusion:

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

References:

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

11.a IPC: Implement the program IPC/IPS using MPI library. Communication in processes of users.

Subject:- Unix Operating System
System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

Objectives:

1. To learn about IPC through MPI.
2. Use of IPC mechanism to write effective application programs.
3. configure cluster and experiment MPI program on it.

Theory:

Inter-process communication or **interprocess communication** (**IPC**) refers specifically to the mechanisms an operating system provides to allow the processes to manage shared data. Typically, applications can use IPC, categorized as clients and servers, where the client requests data and the server responds to client requests.[1] Many applications are both clients and servers, as commonly seen in distributed computing.

IPC is very important to the design process for microkernels and nanokernels, which reduce the number of functionalities provided by the kernel. Those functionalities are then obtained by communicating with servers via IPC, leading to a large increase in communication when compared to a regular monolithic kernel. IPC interfaces generally encompass variable analytic framework structures. These processes ensure compatibility between the multi-vector protocols upon which IPC models rely.[2]

An IPC mechanism is either synchronous or asynchronous. Synchronization primitives may be used to have synchronous behavior with an asynchronous IPC mechanism.

MPI-2 defines three one-sided communications operations, `MPI_Put`, `MPI_Get`, and `MPI_Accumulate`, being a write to remote memory, a read from remote memory, and a reduction operation on the same memory across a number of tasks, respectively. Also defined are three different methods to synchronize this communication (global, pairwise, and remote locks) as the specification does not guarantee that these operations have taken place until a synchronization point.

These types of call can often be useful for algorithms in which synchronization would be inconvenient (e.g. distributed matrix multiplication), or where it is desirable for tasks to be able to balance their load while other processors are operating on data.

Program:

```
/*
  "Hello World" MPI Test Program for IPC
*/
#include <assert.h>
```

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

int main(int argc, char **argv)
{
   char buf[256];
   int my_rank, num_procs;

   /* Initialize the infrastructure necessary for communication */
   MPI_Init(&argc, &argv);

   /* Identify this process */
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   /* Find out how many total processes are active */
   MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

   /* Until this point, all programs have been doing exactly the same.
     Here, we check the rank to distinguish the roles of the programs */
   if (my_rank == 0) {
      int other_rank;
      printf("We have %i processes.\n", num_procs);

      /* Send messages to all other processes */
      for (other_rank = 1; other_rank < num_procs; other_rank++)
      {
         sprintf(buf, "Hello %i!", other_rank);
         MPI_Send(buf, sizeof(buf), MPI_CHAR, other_rank,
               0, MPI_COMM_WORLD);
      }

      /* Receive messages from all other process */
      for (other_rank = 1; other_rank < num_procs; other_rank++)
      {
         MPI_Recv(buf, sizeof(buf), MPI_CHAR, other_rank,
               0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         printf("%s\n", buf);
      }

   } else {

      /* Receive message from process #0 */
      MPI_Recv(buf, sizeof(buf), MPI_CHAR, 0,
            0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
        assert(memcmp(buf, "Hello ", 6) == 0);

        /* Send message to process #0 */
        sprintf(buf, "Process %i reporting for duty.", my_rank);
        MPI_Send(buf, sizeof(buf), MPI_CHAR, 0,
                 0, MPI_COMM_WORLD);

    }

    /* Tear down the communication infrastructure */
    MPI_Finalize();
    return 0;
}
```

Output:

$ mpicc example.c && mpiexec -n 4 ./a.out

```
We have 4 processes.
Process 1 reporting for duty.
Process 2 reporting for duty.
Process 3 reporting for duty.
```

Conclusion:

1. Implemented the program IPC/IPS using MPI library

References:

https://en.wikipedia.org/wiki/Message_Passing_Interface

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

Subject:- Unix Operating System
System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

Objectives:

1. To learn about IPC through MPI.
2. Use of IPC mechanism to write effective application programs.
3. configure cluster and experiment MPI program on it.

Theory:

What is MPI ? (Message Passing Interface)

- MPI is a **specification** for the developers and users of message passing libraries. By itself, it is NOT a library - but rather the specification of what such a library should be.
- MPI primarily addresses the **message-passing parallel programming model:** data is moved from the address space of one process to that of another process through cooperative operations on each process.
- Simply stated, the goal of the Message Passing Interface is to provide a widely used standard for writing message passing programs.

Programming Model:

- Originally, MPI was designed for distributed memory architectures, which were becoming increasingly popular at that time (1980s - early 1990s).

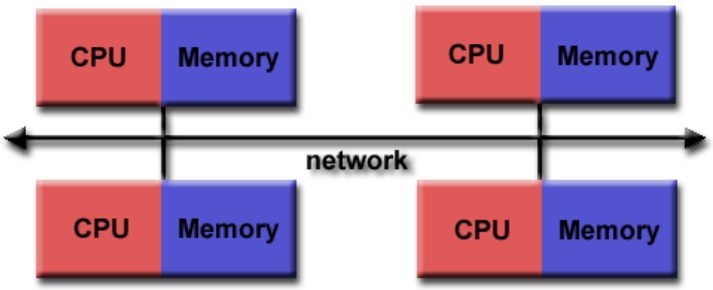


Fig 11.2 MPI

- As architecture trends changed, shared memory SMPs were combined over networks creating hybrid distributed memory / shared memory systems.
- MPI implementors adapted their libraries to handle both types of underlying memory architectures seamlessly. They also adapted/developed ways of handling different

interconnects and protocols.

- Today, MPI runs on virtually any hardware platform:
  - Distributed Memory
  - Shared Memory
  - Hybrid
- The programming model clearly remains a distributed memory model however, regardless of the underlying physical architecture of the machine.
- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs.

Reasons for Using MPI:

- Standardization
- Portability
- Performance Opportunities
- Functionality
- Availability

Program:

```
#include <boost/mpi.hpp>
#include <iostream>

int main(int argc, char* argv[])
{
 boost::mpi::environment env(argc, argv);
 boost::mpi::communicator world;

 if (world.rank() == 0) {
  world.send(1, 9, 32);
  world.send(2, 9, 33);
 } else {
  int data;
  world.recv(0, 9, data);
  std::cout << "In process " << world.rank( ) << "with data " << data
            << std::endl;
 }
 return 0;
}
```

Output:

In process 1 with data 32

In process 2 with data 33

Conclusion:

1. From a usability standpoint, IPC is easy to use. However, the MPI library is dependant on native MPI implementations,

References:

[1] http://mpitutorial.com/tutorials/mpi-introduction/

11.3 IPC: MPI(C library for message passing between processes of different systems) Distributed memory programming

Subject:- Unix Operating System
System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

Objectives:

1. To learn about IPC through MPI.
2. Use of IPC mechanism to write effective application programs.
3. configure cluster and experiment MPI program on it.

Theory:

Message Passing Interface (MPI) is a standardized and portable message-passing standard designed by a group of researchers from academia and industry to function on a wide variety of
parallel computing architectures. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in C, C++, and Fortran. There are several well-tested and efficient implementations of MPI, many of which are open-source or in the public domain. These fostered the development of a parallel software industry, and encouraged development of portable and scalable large-scale parallel applications.
The MPI interface is meant to provide essential virtual topology, synchronization, and communication functionality between a set of processes (that have been mapped to nodes/servers/computer instances) in a language-independent way, with language-specific syntax
(bindings), plus a few language-specific features. MPI programs always work with processes, but
programmers commonly refer to the processes as processors. Typically, for maximum performance, each CPU (or core in a multi-core machine) will be assigned just a single process. This assignment happens at runtime through the agent that starts the MPI program, normally called mpirun or
mpiexec

Program:

```
//MPI PI calculation using area of circle.
#include <stdio.h>
#include<math.h>
#include <mpi.h>
#define N 1E7
#define d 1E-7
#define d2 1E-14
int main (int argc, char* argv[])
```

```
{
 int rank, size, error, i;
 double pi=0.0, begin=0.0, end=0.0, result=0.0, sum=0.0, x2;
 error=MPI_Init (&argc, &argv);
 //Get process ID
 MPI_Comm_rank (MPI_COMM_WORLD, &rank);
 //Get processes Number
 MPI_Comm_size (MPI_COMM_WORLD, &size);
 //Synchronize all processes and get the begin time
 MPI_Barrier(MPI_COMM_WORLD);
 begin = MPI_Wtime();
 //Each process calculates a part of the sum
 for (i=rank; i<N; i+=size)
 {
 x2=d2*i*i;
 result+=sqrt(1-x2);
 }
 //Sum up all results
 MPI_Reduce(&result, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
 //Synchronize all processes and get the end time
 MPI_Barrier(MPI_COMM_WORLD);
 end = MPI_Wtime();
 //Calculate and print PI
 if (rank==0)
 {
 pi=4*d*sum;
 printf("np=%2d; Time=%fs; PI=%lf\n", size, end-begin, pi);
 }
 error=MPI_Finalize();
return 0;
}
```

Output:

```
pi = 3.1415
Time = 0.02554
```

Conclusion:

1. The MPI program run successfully with time less than sequential execution.

References:

[1] http://mpitutorial.com/tutorials/mpi-introduction/

12.a - Implement the program for threads using Open MP library. Print number of core.

Subject:- Unix Operating System

System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

Objectives:

1 To learn about openMP for better use of multicore system.

Theory:

An OpenMP program has sections that are sequential and sections that are parallel. In general an OpenMP program starts with a sequential section in which it sets up the environment, initializes the variables, and so on.

When run, an OpenMP program will use one thread (in the sequential sections), and several threads (in the parallel sections).

There is one thread that runs from the beginning to the end, and it's called the *master thread*. The parallel sections of the program will cause additional threads to fork. These are called the *slave* threads.

A section of code that is to be executed in parallel is marked by a special directive (omp pragma). When the execution reaches a parallel section (marked by omp pragma), this directive will cause slave threads to form. Each thread executes the parallel section of the code independently. When a thread finishes, it joins the master. When all threads finish, the master continues with code following the parallel section.

Each thread has an ID attached to it that can be obtained using a runtime library function (called omp_get_thread_num()). The ID of the master thread is 0.

Program:

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
#define CHUNKSIZE   10
#define N       100

int main (int argc, char *argv[])
{
  int nthreads, tid, i, chunk;
  float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;
```

```
#pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
  {
  tid = omp_get_thread_num();
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  printf("Thread %d starting...\n",tid);

  #pragma omp for schedule(dynamic,chunk)
  for (i=0; i < N; i++)
       c[i] = a[i] + b[i];
    printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
    }

  }  /* end of parallel section */

}
```

Output:

```
Number of threads = 2
 Thread 1 starting...
 Thread 0 starting...
 Thread 0: c[10]= 20.000000
 Thread 1: c[0]= 0.000000
 Thread 0: c[11]= 22.000000
 Thread 1: c[1]= 2.000000
 Thread 0: c[12]= 24.000000
 Thread 1: c[2]= 4.000000
Thread 0: c[13]= 26.000000
Thread 1: c[3]= 6.000000
Thread 0: c[14]= 28.000000
Thread 1: c[4]= 8.000000
Thread 0: c[15]= 30.000000
Thread 1: c[5]= 10.000000
Thread 0: c[16]= 32.000000
Thread 1: c[6]= 12.000000
Thread 0: c[17]= 34.000000
Thread 1: c[7]= 14.000000
Thread 0: c[18]= 36.000000
Thread 1: c[8]= 16.000000
Thread 0: c[19]= 38.000000
Thread 1: c[9]= 18.000000
Thread 0: c[20]= 40.000000
Thread 1: c[30]= 60.000000
Thread 0: c[21]= 42.000000
Thread 1: c[31]= 62.000000
Thread 0: c[22]= 44.000000
Thread 1: c[32]= 64.000000
Thread 0: c[23]= 46.000000
```

Thread 1: c[33]= 66.000000

Conclusion:

- We learned about the concepts of parallel programming using OpenMP.
- Use of OpenMP in Shared memory programming
- Efficient use of processor and thereby reducing time by using OpenMP.

References:

[1]https://www.codeproject.com/Articles/60176/A-Beginner-s-Primer-to-OpenMP
[2]https://www.researchgate.net/post/Is_there_a_way_to_specify_how_many_cores_a_program_should_run-in_other_words_can_I_control_where_the_threads_are_mapped
[3]https://www.embedded.com/design/mcus-processors-and-socs/4007155/Using-OpenMP-for-programming-parallel-threads-in-multicore-applications-Part-2
[4]http://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-basics.html

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

Subject:- Unix Operating System
System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

Objectives:

1. To learn about openMP for better use multicore system.
2. Implement the program OpenMP threads and print prime number task, odd number and Fibonacci series using three thread on core. Comment on performance CPU.

Theory:

OpenMP Is:
An Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism.
Comprised of three primary API components:
Compiler Directives
Runtime Library Routines
Environment Variables
An abbreviation for: Open Multi-Processing
Shared Memory Model:
OpenMP is designed for multi-processor/core, shared memory machines. The underlying architecture can be shared memory UMA or NUMA.

Program:

```
#include<stdio.h>
#include<stdlib.h>
#include<omp.h>
Int main()
{
int i, n, t1 = 0, t2 = 1, nextTerm=0,j=0,counter,i,a,count;
#pragma openmp parallel private(i,counter,a,count)
In ID = omp_get_thread_num();
if(ID==1)
{
   for (i = 1; i <= n; ++i)
   {
      printf("%dth Fib no: =%d \n", nextTerm);
      nextTerm = t1 + t2;
      t1 = t2;
      t2 = nextTerm;
```

```
   }
}
if(ID==2)
{
 for(counter = 1; counter <= 100; counter++) {
     if(counter%2 == 1) {
   printf("%dth Odd no: =%d \n", j,counter);

       }
}
if(ID==3)
{
for (i=100;i<500;i++)
{
          count=0;
                    for (a=1;a<=i;a++)
                    {
                              if (i%a==0)
                                        count++;
                    }
          if (count==2)
   printf("%dth Prime no: =%d \n", j,i);
j++;
}


}
}
```

Output:

```
1th Fib no: = 0
2th Fib no: = 1
3th Fib no: = 1
4th Fib no: = 2
5th Fib no: = 3
1th Prime no: = 2
6th Fib no: = 5
1th Odd no: = 1
2th Odd no: = 3
2th Prime no: = 3
7th Fib no: = 8
3th Odd no: = 5
3th Prime no: = 5
8th Fib no: = 13
4th Prime no: = 7
9th Fib no: = 21
4th Odd no: = 7
5th Prime no: = 11
```

10th Fib no: = 34
6th Prime no: = 13

Conclusion:

1. If compared with serial code, OpenMP works faster as work is performed by multiple threads simultaneously.

2. CPU is being more utilized.

References:

[1] http://mpitutorial.com/tutorials/mpi-introduction/

12.3 Write a program for Matrix Multiplication in OpenMP.

Subject:- Unix Operating System
System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

Objectives:

1. To learn about openMP for better use multicore system.
2. Implement the program OpenMP threads and print prime number task, odd number and Fibonacci series using three thread on core. Comment on performance CPU.

Theory:

OpenMP

- OpenMP is an Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism.
- OpenMP is a directive-based method to invoke parallel computations on share-memory multiprocessors.
- The omp for directive instructs the compiler to distribute loop iterations within the team of threads that encounters this work-sharing construct.
- OpenMP consists of a set of compiler #pragmas that control how the program works. The pragmas are designed so that even if the compiler does not support them, the program will still yield correct behavior, but without any parallelism. Here are two simple example programs demonstrating OpenMP.

Program:

```
#include<stdio.h>
void main()
{
	int m1[3][3], m2[3][3], m3[3][3], sum=0, i, j, k;
	printf("Enter first matrix element : ");
	for(i=0;i<3;i++)
	{
		for(j=0;j<3;j++)
		{
			scanf("%d",&m1[i][j]);
		}
	}
	printf("Enter second matrix element : ");
	for(i=0;i<3;i++)
	{
```

```
            for(j=0;j<3;j++)
            {
                scanf("%d",&m2[i][j]);
            }
        }
        #pragma omp parallel for
        for(i=0;i<3;i++)
        {
            for(j=0;j<3;j++)
            {
                sum=0;
                for(k=0;k<3;k++)
                {
                    sum=sum+m1[i][k] * m2[k][j];
                }
                m3[i][j]=sum;
            }
        }
        printf("\nMultiplication of two Matrices : \n");
        for(i=0;i<3;i++)
        {
            for(j=0;j<3;j++)
            {
                printf("%d\t",m3[i][j]);
            }

            printf("\n");
        }
}
```

Output:

```
Enter first matrix element:
4 6 8
2 1 4
6 9 3
Enter second matrix element:
1 5 8
4 3 2
7 6 5

Multiplication of two Matrices :
84 86 84
34 37 38
63 75 81
```

Conclusion:

1 Hence we learnt that the Sequential programs can be enhanced with OpenMP directives, leaving the original program essentially intact.

References:

[1] http://openmptutorial.com/tutorials/openmp-introduction/

STREAMS message/PIPEs/FIFO:pipe, popenand pcloseFunctions

Assignment No: 13_a

Subject:- Unix Operating System

System Lab Class :- TYIT

Name PRN

Sumit Sunil Koundanya 2019BTEIT00023

Aniket Shivnath Sable 2019BTEIT00020

Title:

Send data from parent to child over a pipe

Objectives:

1. To learn about STREAMS message/PIPEs/FIFO:pipe, popenand pcloseFunctions

Theory:

Pipes

Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems. Pipes have two limitations:

1. Historically, they have been half duplex (data flows in only one direction). Some systems now provide full-duplex pipes, but for maximum portability, we should never assume that this is the case.
2. Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child.

FIFOs (Section 15.5) get around the second limitation, and that UNIX domain sockets (Section 17.2) get around both limitations.

Despite these limitations, half-duplex pipes are still the most commonly used form of IPC. Every time you type a sequence of commands in a pipeline for the shell to execute, the shell creates a separate process for each command and links the standard output of one process to the standard input of the next using a pipe.

A pipe is created by calling the `pipe` function.

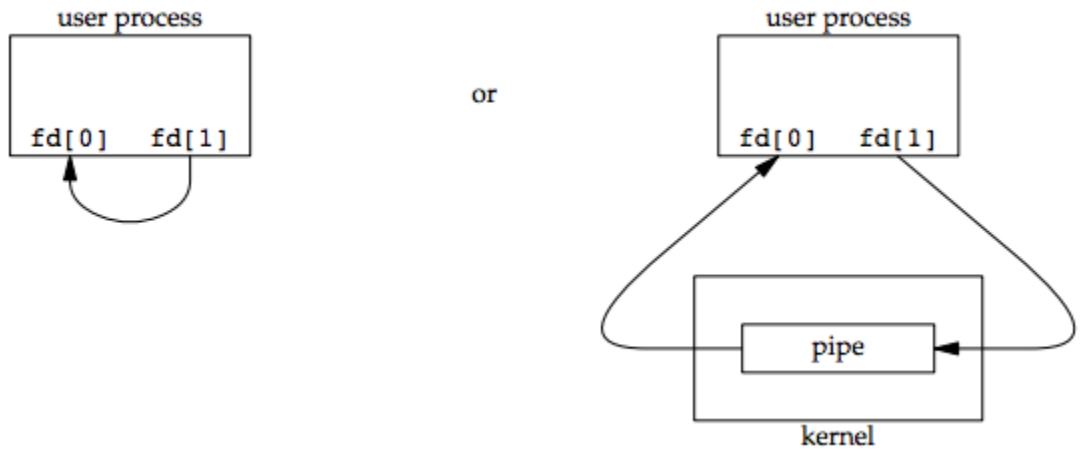
```
#include <unistd.h>
```

```
int pipe(int fd[2]);

/* Returns: 0 if OK, −1 on error */
```

Two file descriptors are returned through the *fd* argument: *fd[0]* is open for reading, and *fd[1]* is open for writing. The output of *fd[1]* is the input for *fd[0]*.

POSIX.1 allows for implementations to support full-duplex pipes. For these implementations, *fd[0]* and *fd[1]* are open for both reading and writing.

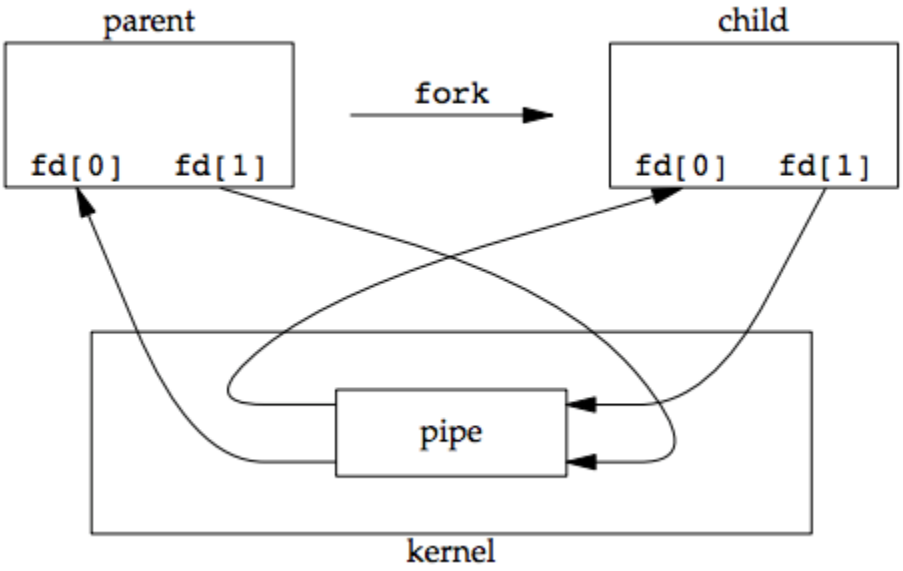
Two ways to picture a half-duplex pipe are shown in the figure below. The left half of the figure shows the two ends of the pipe connected in a single process. The right half of the figure emphasizes that the data in the pipe flows through the kernel.



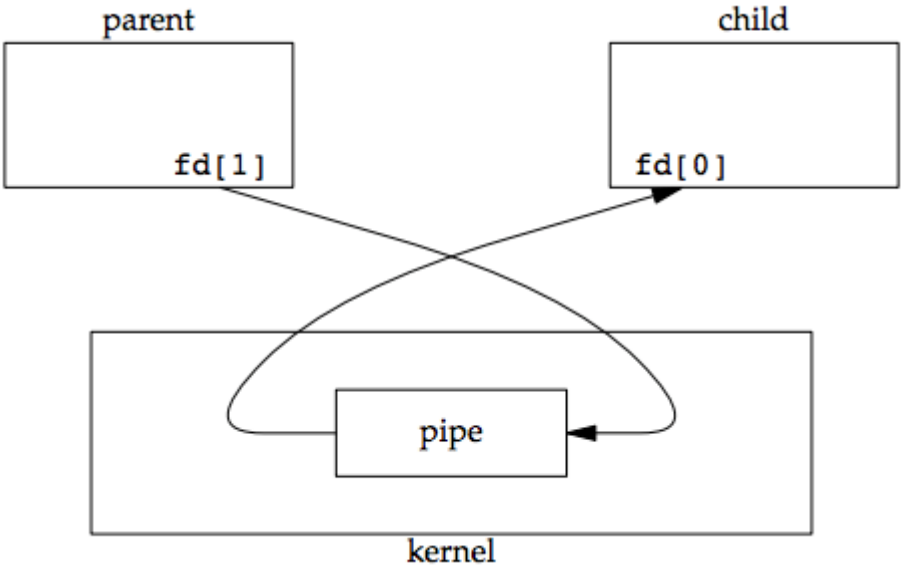
The `fstat` function returns a file type of FIFO for the file descriptor of either end of a pipe. We can test for a pipe with the `S_ISFIFO` macro.

POSIX.1 states that the `st_size` member of the stat structure is undefined for pipes. But when the `fstat` function is applied to the file descriptor for the read end of the pipe, many systems store in `st_size` the number of bytes available for reading in the pipe, which is nonportable.

A pipe in a single process is next to useless. Normally, the process that calls `pipe` then calls `fork`, creating an IPC channel from the parent to the child, or vice versa. The following figure shows this scenario:



What happens after the `fork` depends on which direction of data flow we want. For a pipe from the parent to the child, the parent closes the read end of the pipe (*fd[0]*), and the child closes the write end (*fd[1]*). The following figure shows the resulting arrangement of descriptors.



For a pipe from the child to the parent, the parent closes fd[1], and the child closes fd[0].

When one end of a pipe is closed, two rules apply:

1. If we read from a pipe whose write end has been closed, read returns 0 to indicate an end of file after all the data has been read.
    - Technically, we should say that this end of file is not generated until there are no more writers for the pipe.
    - It's possible to duplicate a pipe descriptor so that multiple processes have the pipe open for writing.
    - Normally, there is a single reader and a single writer for a pipe. (The FIFOs in the next section dicusses that there are multiple writers for a single FIFO.)

2. If we write to a pipe whose read end has been closed, the signal SIGPIPE is generated. If we either ignore the signal or catch it and return from the signal handler, write returns −1 with errno set to EPIPE.

When we're writing to a pipe (or FIFO), the constant `PIPE_BUF` specifies the kernel's pipe buffer size. A write of `PIPE_BUF` bytes or less will not be interleaved with the writes from other processes to the same pipe (or FIFO). But if multiple processes are writing to a pipe (or FIFO), and if we write more than `PIPE_BUF` bytes, the data might be interleaved with the data from the other writers. We can determine the value of `PIPE_BUF` by using `pathconf` or `fpathconf`.

Example: creating a pipe between a parent and its child

ipc1/pipe1.c

```
#include "apue.h"

int
main(void)
{
    int     n;
    int     fd[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("pipe error");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {       /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else {                    /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
```

```
    exit(0);
}
```

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
        int fd[2];
        int childID = 0;

        // create pipe descriptors
        pipe(fd);

        // fork() returns 0 for child process, child-pid for parent process.
        if (fork() != 0) {
                // parent: writing only, so close read-descriptor.
                close(fd[0]);

                // send the childID on the write-descriptor.
                childID = 1;
                write(fd[1], &childID, sizeof(childID));
                printf("Parent(%d) send childID: %d\n", getpid(), childID);

                // close the write descriptor
                close(fd[1]);
        } else {
                // child: reading only, so close the write-descriptor
```

```
                close(fd[1]);


                // now read the data (will block until it succeeds)
                read(fd[0], &childID, sizeof(childID));
                printf("Child(%d) received childID: %d\n", getpid(), childID);


                // close the read-descriptor
                close(fd[0]);
        }
        return 0;
}
```

Output:

sarita@hp:~/Desktop/UOS/13$ gedit A13_a.c

^C

sarita@hp:~/Desktop/UOS/13$ gcc A13_a.c

sarita@hp:~/Desktop/UOS/13$ ./a.out

Parent(23699) send childID: 1

Child(23700) received childID: 1

sarita@hp:~/Desktop/UOS/13$ ^C

sarita@hp:~/Desktop/UOS/13$

Conclusion:

The concept of pipe has been learned.

References:

- https://bytefreaks.net/programming-2/c-programming-2/cc-pass-value-from-parent-to-child-after-fork-via-a-pipe
- https://notes.shichao.io/apue/ch15/

STREAMS message/PIPEs/FIFO:pipe, popenand pcloseFunctions

Assignment No: 13_b

Subject:- Unix Operating System

System Lab Class :- TYIT

Name PRN

Sumit Sunil Koundanya 2019BTEIT00023

Aniket Shivnath Sable 2019BTEIT00020

Title:

Filter to convert uppercase characters to lowercase.

Objectives:

1. To learn about STREAMS message/PIPEs/FIFO:pipe, popenand pcloseFunctions

Theory:

Pipes

Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems. Pipes have two limitations:

1. Historically, they have been half duplex (data flows in only one direction). Some systems now provide full-duplex pipes, but for maximum portability, we should never assume that this is the case.
2. Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child.

FIFOs (Section 15.5) get around the second limitation, and that UNIX domain sockets (Section 17.2) get around both limitations.

Despite these limitations, half-duplex pipes are still the most commonly used form of IPC. Every time you type a sequence of commands in a pipeline for the shell to execute, the shell creates a separate process for each command and links the standard output of one process to the standard input of the next using a pipe.

A pipe is created by calling the pipe function.

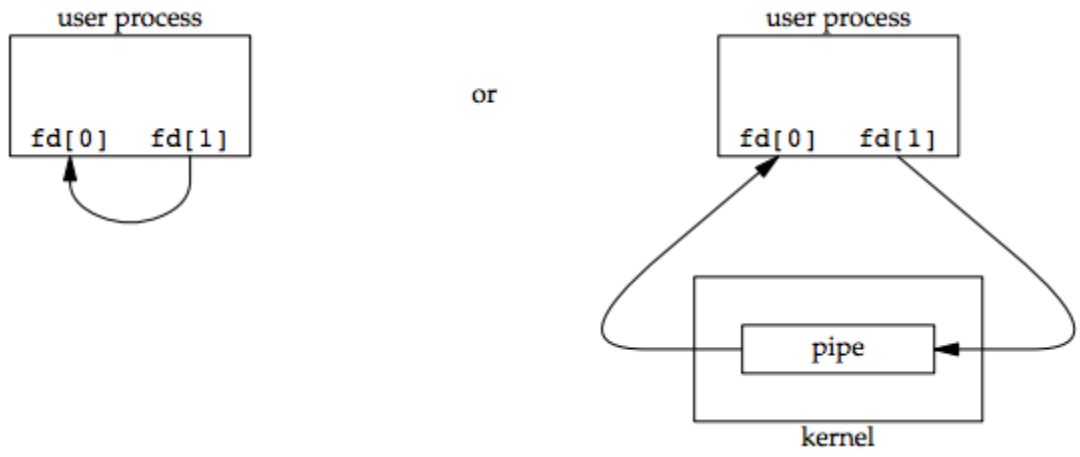
```
#include <unistd.h>
```

```
int pipe(int fd[2]);

/* Returns: 0 if OK, −1 on error */
```

Two file descriptors are returned through the *fd* argument: *fd[0]* is open for reading, and *fd[1]* is open for writing. The output of *fd[1]* is the input for *fd[0]*.

POSIX.1 allows for implementations to support full-duplex pipes. For these implementations, *fd[0]* and *fd[1]* are open for both reading and writing.

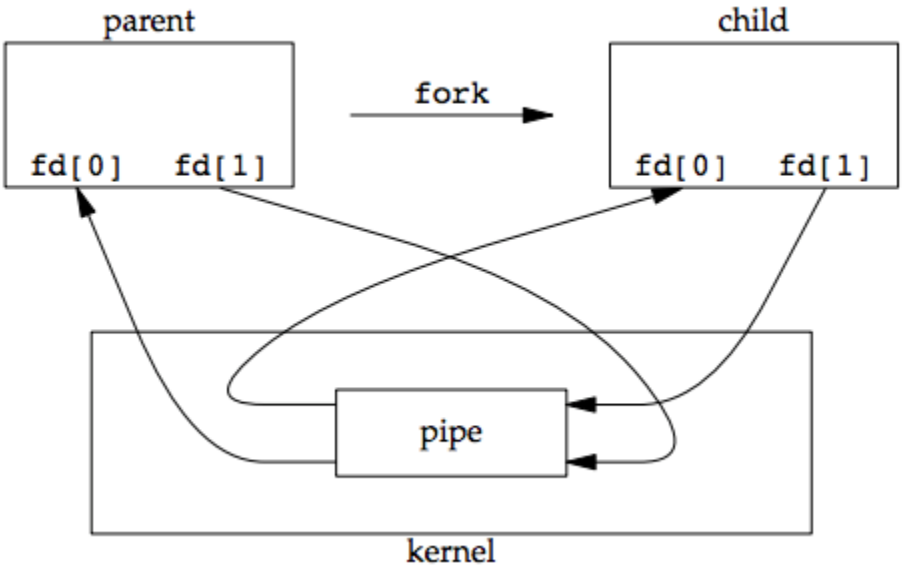
Two ways to picture a half-duplex pipe are shown in the figure below. The left half of the figure shows the two ends of the pipe connected in a single process. The right half of the figure emphasizes that the data in the pipe flows through the kernel.



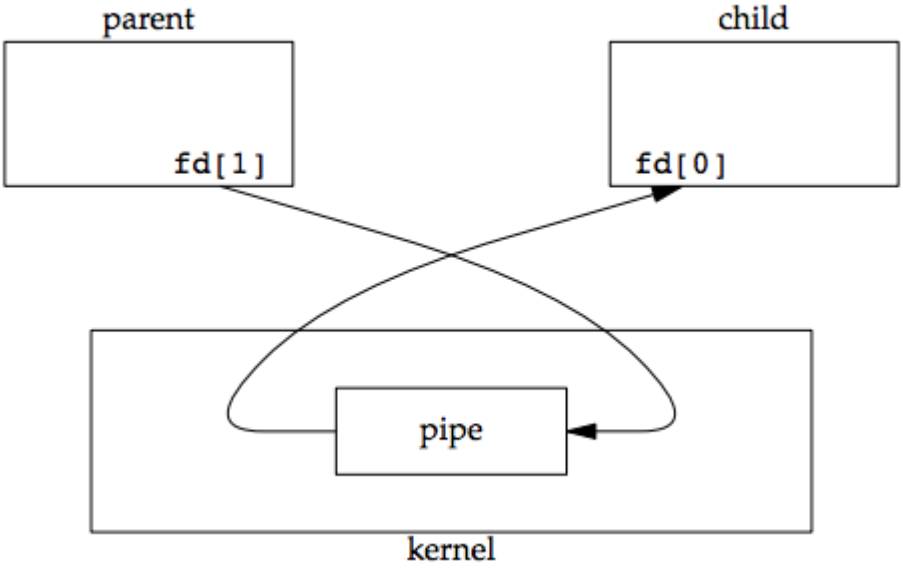
The fstat function returns a file type of FIFO for the file descriptor of either end of a pipe. We can test for a pipe with the S_ISFIFO macro.

POSIX.1 states that the st_size member of the stat structure is undefined for pipes. But when the fstat function is applied to the file descriptor for the read end of the pipe, many systems store in st_size the number of bytes available for reading in the pipe, which is nonportable.

A pipe in a single process is next to useless. Normally, the process that calls pipe then calls fork, creating an IPC channel from the parent to the child, or vice versa. The following figure shows this scenario:



What happens after the fork depends on which direction of data flow we want. For a pipe from the parent to the child, the parent closes the read end of the pipe (*fd[0]*), and the child closes the write end (*fd[1]*). The following figure shows the resulting arrangement of descriptors.



For a pipe from the child to the parent, the parent closes fd[1], and the child closes fd[0].

When one end of a pipe is closed, two rules apply:

1. If we read from a pipe whose write end has been closed, read returns 0 to indicate an end of file after all the data has been read.
    - Technically, we should say that this end of file is not generated until there are no more writers for the pipe.
    - It's possible to duplicate a pipe descriptor so that multiple processes have the pipe open for writing.
    - Normally, there is a single reader and a single writer for a pipe. (The FIFOs in the next section dicusses that there are multiple writers for a single FIFO.)

2. If we write to a pipe whose read end has been closed, the signal SIGPIPE is generated. If we either ignore the signal or catch it and return from the signal handler, write returns −1 with errno set to EPIPE.

When we're writing to a pipe (or FIFO), the constant PIPE_BUF specifies the kernel's pipe buffer size. A write of PIPE_BUF bytes or less will not be interleaved with the writes from other processes to the same pipe (or FIFO). But if multiple processes are writing to a pipe (or FIFO), and if we write more than PIPE_BUF bytes, the data might be interleaved with the data from the other writers. We can determine the value of PIPE_BUF by using pathconf or fpathconf.

Example: creating a pipe between a parent and its child

ipc1/pipe1.c

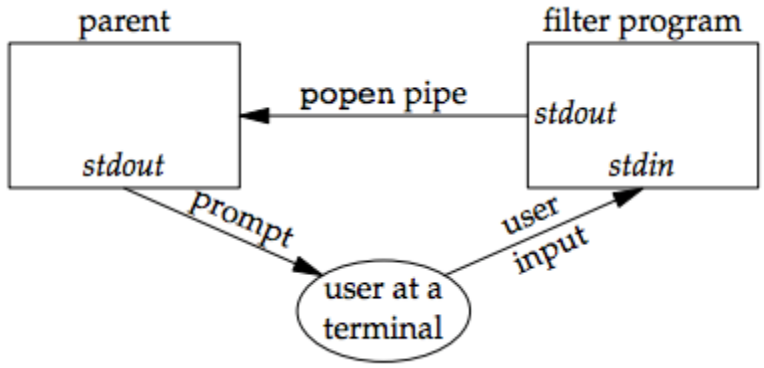
```
#include "apue.h"

int
main(void)
{
    int     n;
    int     fd[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("pipe error");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {       /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else {                    /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
```

```
    exit(0);
}
```

Example: transforming input using popen

One thing that popen is especially well suited for is executing simple filters to transform the input or output of the running command. Such is the case when a command wants to build its own pipeline.

Consider an application that writes a prompt to standard output and reads a line from standard input. With the popen function, we can interpose a program between the application and its input to transform the input. The following figure shows the arrangement of processes in this situation.



The following program is a simple filter to demonstrate this operation:

Code:

//file apue.h

```
/*
 * Our own header, to be included before all standard system headers.
 */
#ifndef _APUE_H
#define _APUE_H

#define _POSIX_C_SOURCE 200809L

#if defined(SOLARIS)            /* Solaris 10 */
#define _XOPEN_SOURCE 600
#else
#define _XOPEN_SOURCE 700
#endif
```

```
#include <sys/types.h>		/* some systems still require this */
#include <sys/stat.h>
#include <sys/termios.h>	/* for winsize */
#if defined(MACOS) || !defined(TIOCGWINSZ)
#include <sys/ioctl.h>
#endif


#include <stdio.h>		/* for convenience */
#include <stdlib.h>		/* for convenience */
#include <stddef.h>		/* for offsetof */
#include <string.h>		/* for convenience */
#include <unistd.h>		/* for convenience */
#include <signal.h>		/* for SIG_ERR */

#define MAXLINE	4096			/* max line length */

/*
 * Default file access permissions for new files.
 */
#define FILE_MODE	(S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

/*
 * Default permissions for new directories.
 */
#define DIR_MODE	(FILE_MODE | S_IXUSR | S_IXGRP | S_IXOTH)

typedef void	Sigfunc(int);	/* for signal handlers */

#define min(a,b)	((a) < (b) ? (a) : (b))
#define max(a,b)	((a) > (b) ? (a) : (b))

/*
 * Prototypes for our own functions.
 */
char	*path_alloc(size_t *);				/* {Prog pathalloc} */
```

```
long      open_max(void);                                   /* {Prog openmax} */

int               set_cloexec(int);                                 /* {Prog setfd} */
void      clr_fl(int, int);
void      set_fl(int, int);                         /* {Prog setfl} */

void      pr_exit(int);                                     /* {Prog prexit} */

void      pr_mask(const char *);                    /* {Prog prmask} */
Sigfunc *signal_intr(int, Sigfunc *);       /* {Prog signal_intr_function} */

void      daemonize(const char *);                  /* {Prog daemoninit} */

void      sleep_us(unsigned int);           /* {Ex sleepus} */
ssize_t   readn(int, void *, size_t);       /* {Prog readn_writen} */
ssize_t   writen(int, const void *, size_t);/* {Prog readn_writen} */

int               fd_pipe(int *);                           /* {Prog sock_fdpipe} */
int               recv_fd(int, ssize_t (*func)(int,
                        const void *, size_t));   /* {Prog recvfd_sockets} */
int               send_fd(int, int);                                /* {Prog sendfd_sockets} */
int               send_err(int, int,
                        const char *);            /* {Prog senderr} */
int               serv_listen(const char *);                /* {Prog servlisten_sockets} */
int               serv_accept(int, uid_t *);                /* {Prog servaccept_sockets} */
int               cli_conn(const char *);           /* {Prog cliconn_sockets} */
int               buf_args(char *, int (*func)(int,
                        char **));                /* {Prog bufargs} */

int               tty_cbreak(int);                          /* {Prog raw} */
int               tty_raw(int);                                     /* {Prog raw} */
int               tty_reset(int);                           /* {Prog raw} */
void      tty_atexit(void);                                 /* {Prog raw} */
struct termios    *tty_termios(void);               /* {Prog raw} */

int               ptym_open(char *, int);           /* {Prog ptyopen} */
```

```
int                 ptys_open(char *);                                  /* {Prog ptyopen} */
#ifdef  TIOCGWINSZ
pid_t    pty_fork(int *, char *, int, const struct termios *,
                        const struct winsize *);    /* {Prog ptyfork} */
#endif

int                 lock_reg(int, int, int, off_t, int, off_t); /* {Prog lockreg} */

#define read_lock(fd, offset, whence, len) \
                        lock_reg((fd), F_SETLK, F_RDLCK, (offset), (whence), (len))
#define readw_lock(fd, offset, whence, len) \
                        lock_reg((fd), F_SETLKW, F_RDLCK, (offset), (whence), (len))
#define write_lock(fd, offset, whence, len) \
                        lock_reg((fd), F_SETLK, F_WRLCK, (offset), (whence), (len))
#define writew_lock(fd, offset, whence, len) \
                        lock_reg((fd), F_SETLKW, F_WRLCK, (offset), (whence), (len))
#define un_lock(fd, offset, whence, len) \
                        lock_reg((fd), F_SETLK, F_UNLCK, (offset), (whence), (len))

pid_t   lock_test(int, int, off_t, int, off_t);             /* {Prog locktest} */

#define is_read_lockable(fd, offset, whence, len) \
                        (lock_test((fd), F_RDLCK, (offset), (whence), (len)) == 0)
#define is_write_lockable(fd, offset, whence, len) \
                        (lock_test((fd), F_WRLCK, (offset), (whence), (len)) == 0)

void    err_msg(const char *, ...);                         /* {App misc_source} */
void    err_dump(const char *, ...) __attribute__((noreturn));
void    err_quit(const char *, ...) __attribute__((noreturn));
void    err_cont(int, const char *, ...);
void    err_exit(int, const char *, ...) __attribute__((noreturn));
void    err_ret(const char *, ...);
void    err_sys(const char *, ...) __attribute__((noreturn));

void    log_msg(const char *, ...);                         /* {App misc_source} */
void    log_open(const char *, int, int);
```

```
void	log_quit(const char *, ...) __attribute__((noreturn));
void	log_ret(const char *, ...);
void	log_sys(const char *, ...) __attribute__((noreturn));
void	log_exit(int, const char *, ...) __attribute__((noreturn));

void	TELL_WAIT(void);		/* parent/child from {Sec race_conditions} */
void	TELL_PARENT(pid_t);
void	TELL_CHILD(pid_t);
void	WAIT_PARENT(void);
void	WAIT_CHILD(void);

#endif	/* _APUE_H */
```

//file myuclc.c

```c
#include "apue.h"
#include <ctype.h>

int
main(void)
{
    int     c;

    while ((c = getchar()) != EOF) {
        if (isupper(c))
            c = tolower(c);
        if (putchar(c) == EOF)
            err_sys("output error");
        if (c == '\n')
            fflush(stdout);
    }
    exit(0);
}
```

The filter copies standard input to standard output, converting any uppercase character to lowercase. The reason we're careful to `fflush` standard output after writing a newline is discussed in the next section when we talk about coprocesses.

We compile this filter into the executable file `myuclc`, which we then invoke from the program in the following code using `popen`:

//file popen1.c

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

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

    if ((fpin = popen("myuclc", "r")) == NULL)
        err_sys("popen error");
    for ( ; ; ) {
        fputs("prompt> ", stdout);
        fflush(stdout);
        if (fgets(line, MAXLINE, fpin) == NULL) /* read from pipe */
            break;
        if (fputs(line, stdout) == EOF)
            err_sys("fputs error to pipe");
    }
    if (pclose(fpin) == -1)
        err_sys("pclose error");
    putchar('\n');
    exit(0);
}
```

Conclusion:

The concept of pipe has been learned. File has converted from lowercase to upper case using filter.

References:

- https://notes.shichao.io/apue/ch15/

STREAMS message/PIPEs/FIFO:pipe, popenand pcloseFunctions

Assignment No: 13c

Subject:- Unix Operating System

System Lab Class :- TYIT

Name	PRN
Sumit Sunil Koundanya	2019BTEIT00023
Aniket Shivnath Sable	2019BTEIT00020

Objectives:

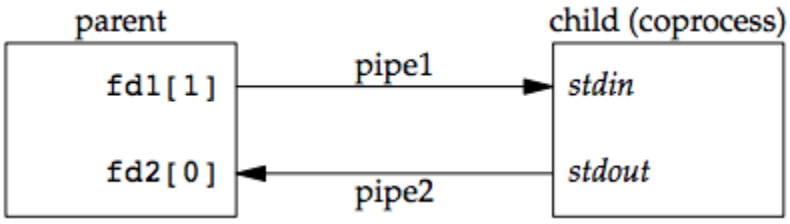
1.Simple filter to add two numbers. (B)

Theory:

A UNIX system filter is a program that reads from standard input and writes to standard output. Filters are normally connected linearly in shell pipelines. A filter becomes a coprocess when the same program generates the filter's input and reads the filter's output.

The Korn shell provides coprocesses. The Bourne shell, the Bourne-again shell, and the C shell don't provide a way to connect processes together as coprocesses. A coprocess normally runs in the background from a shell, and its standard input and standard output are connected to another program using a pipe. Although the shell syntax required to initiate a coprocess and connect its input and output to other processes is quite contorted, coprocesses are also useful from a C program.

Whereas popen gives us a one-way pipe to the standard input or from the standard output of another process, with a coprocess we have two one-way pipes to the other process: one to its standard input and one from its standard output. We want to write to its standard input, let it operate on the data, and then read from its standard output.



Example: invoking add2 as a coprocess

For example, the process creates two pipes: one is the standard input of the coprocess and the other is the standard output of the coprocess. The figure below shows this arrangement:

Figure 15.16 Driving a coprocess by writing its standard input and reading its standard output

The following program is a simple coprocess that reads two numbers from its standard input, computes their sum, and writes the sum to its standard output.

Program:

```
#include "apue.h"

int

main(void)
```

```
{
    int     n, int1, int2;
    char    line[MAXLINE];

    while ((n = read(STDIN_FILENO, line, MAXLINE)) > 0) {
        line[n] = 0;        /* null terminate */
        if (sscanf(line, "%d%d", &int1, &int2) == 2) {
            sprintf(line, "%d\n", int1 + int2);
            n = strlen(line);
            if (write(STDOUT_FILENO, line, n) != n)
                err_sys("write error");
        } else {
            if (write(STDOUT_FILENO, "invalid args\n", 13) != 13)
                err_sys("write error");
        }
    }
    exit(0);
}
```

Input:

10 12

Output:

22

Conclusion:

In this way two numbers are added.

References:

1.[https://notes.shichao.io/apue/ch15/