**Linux Experiment 2**

**CHOWN Command -22,29 (PR)**

soni@LAPTOP-71SEP48I:~$ dfghdghd

dfghdghd: command not found

touch file22.txt

dsfdsfsdf 2> file22.txt

cat < file22.txt

dsfdsfsdf: command not found

echo "sdfjsdhfsdjklfjdsflsfsdf" (Print)

sdfjsdhfsdjklfjdsflsfsdf

echo "sdfjsdhfsdjklfjdsflsfsdf" > file22.txt (Output file)

cat < file22.txt (Input)

sdfjsdhfsdjklfjdsflsfsdf

1. **Create .sh file**

Create- touch script.sh

Permission- chmod +x script.sh

Write- nano script.sh

**#This is a Comment**

**echo This is my first shell script**

**echo shell script test**

**ls**

**echo End of my shell script**

After adding this code to script.sh, it's time to save and exit the file. With "nano", you do this by pressing ctrl+o to write the changes, followed by ctrl+x to exit the editor.

Run- ./script.sh

1. **Assessing element**

**Create-var.sh file**

# Accessing an Environment Variable

echo $USER

# Creating and accessing User defined Variable

variable\_name="LPU"

echo $variable\_name

**Output:**

**Soni**

**LPU**

1. **Flow Control**

**Create-flow.sh file**

#!/bin/sh

x=10

y=11

if [ $x -ne $y ]

then

echo "Not equal to this"

fi

**Output:**

**Not equal to this**

#!/Syntaxbin/sh

x=10

y=10

if [ $x -ne $y]

then

echo "Not equal"

else

echo "They are equal"

fi

1. **Loops in Shell Scripting**

**Create-loop.sh**

#!/bin/sh

x=2

while [ $x -lt 6 ]

do

echo $x

x=`expr $x + 1`

done

**Output:**

**2**

**3**

**4**

**5**

#!/bin/sh

for var in 2 4 5 8

do

echo $var

done

1. **Nested Loop**

nested.sh

#!/bin/sh

for i in 0 1 2 3 4 5 6 7 8 9

do

for j in 0 1 2 3 4 5 6 7 8 9

do

echo "$i$j"

done

done

**Experiment -3**

**sudo apt install gcc**

**sudo apt-get update**

**read**() and write() system calls are used to read and write data respectively to a file descriptor. To understand the concept of write()/read() system calls let us first start with write() system call.

**write()** system call is used to write to a file descriptor. In other words write() can be used to write to any file (all hardware are also referred as file in Linux) in the system but rather than specifying the file name, you need to specify its file descriptor.

#### **Syntax:**

**#include<unistd.h>**

**ssize\_t write(int fd, const void \*buf, size\_t count);**

The first parameter (fd) is the file descriptor where you want to write. The data that is to be written is specified in the second parameter. Finally, the third parameter is the total bytes that are to be written.

**/////Write operation**

**Touch write.c**

**Chmod +rwx write.c**

**nano write.c**

#include<stdio.h>

#include<unistd.h>

int main()

{

int count;

count=write(1,"hello\n",6);

printf("Total bytes written: %d\n",count);

}

**To run the code**

**gcc write.c //compile**

**gcc write.c -o write**

**./write ///to execute**

The write() system call takes three parameters: “1” which is the file descriptor of the file where we want to write. Since we want to write on standard output device which is the screen, hence the file descriptor, in this case, is ‘1’, which is fixed (0 is the file descriptor for standard input device (e.g. keyboard) and 2 is for standard error device)).  
Next thing is what we want to write on the screen. In this case its “hello\n” i.e. hello and newline(\n), so a total of 6 characters, which becomes the third parameter. The third parameter is how much you want to write, which may be less than the data specified in the second parameter.

**//Read operation**

**Touch read.c**

**Chmod +rwx read.c**

**nano read.c**

#include<unistd.h>

int main()

{

char buff[20];

read(0,buff,10);//read 10 bytes from standard input device(keyboard), store in buffer (buff)

write(1,buff,10);//print 10 bytes from the buffer on the screen

}

**To run the code**

**gcc read.c //compile**

**gcc read.c -o read**

**./read ///to execute**

**Enter 10 number**

The read() system call reads the input typed by the user via the keyboard (file descriptor 0) and stores it in the buffer (buff) which is nothing but a character array. It will read a maximum of 10 bytes (because of the third parameter). This can be less than or equal to the buffer size. No matter how much the user types only first 10 characters will be read.

Finally, the data is printed on the screen using the write() system call. It prints the same 10 bytes from the buffer (buff) on the screen (file descriptor 1).

1. **Write a program using open() system call to read the first 10 characters of an xisting file “test.txt” and print them on screen.**

**//open.c**

**#include<stdio.h>**

**#include<unistd.h>**

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

**#include<sys/stat.h>**

**#include<fcntl.h>**

**int main()**

**{**

**int n, fd;**

**char buf[50];**

**fd=open("lpu.txt", O\_RDONLY);**

**read(fd, buf,10);**

**write(1,buf,10);**

**}**

**Step1:**create the file **lpu.txt** and write “1234567890abcdefghij54321” into it  
**$nano lpu.txt  
Step2: compile the program  
$ gcc read.c -o read  
Step3: run**

**./read**

**Program2:** To read 10 characters from file “test.txt” and write them into non-existing file “towrite.txt”

**//open2.c**

**#include<unistd.h>**

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

**#include<sys/stat.h>**

**#include<fcntl.h>**

**int main()**

**{**

**int n,fd,fd1;**

**char buff[50];**

**fd=open("test.txt",O\_RDONLY);**

**n=read(fd,buff,10);**

**fd1=open("towrite.txt",O\_WRONLY|O\_CREAT,0642);//use the pipe symbol (|) to separate O\_WRONLY and O\_CREAT**

**write(fd1,buff,n);**

**}**

**How it works?**

In this we open the “towrite.txt” in write mode and also use O\_CREAT to create it with read and write permission for user, read for the group and write for others. The data is read from test.txt using file descriptor fd and stored in buff. It is then written from buff array into file towrite.txt using file descriptor fd1.

**cat lpu.txt**

**gcc open2.c -o open2**

**./open**

**Cat towrite.txt**

# **lseek() system call**

**lseek()** system call repositions the read/write file offset i.e., it changes the positions of the read/write pointer within the file. In every file any read or write operations happen at the position pointed to by the pointer. lseek() system call helps us to manage the position of this pointer within a file.  
e.g., let’s suppose the content of a file F1 is “1234567890” but you want the content to be “12345hello”. You simply can’t open the file and write “hello” because if you do so then “hello” will be written in the very beginning of the file. This means you need to reposition the pointer after ‘5’ and then start writing “hello”. lseek() will help to reposition the pointer and write() will be used to write “hello”

**Program1: Program using lseek() system call that reads 10 characters from file “seeking” and print on screen. Again read 10 characters and write on screen.**

**#include<unistd.h>**

**#include<fcntl.h>**

**#include<sys/types.h**

**#include<sys/stat.h>**

**#int main()**

**{**

**int n,f;**

**char buff[10];**

**f=open("seeking",O\_RDWR);**

**read(f,buff,10);**

**write(1,buff,10);**

**read(f,buff,10);**

**write(1,buff,10);**

**}**

#### Output

#### 1234567890abcdefghij Because when the file opens the pointer is in the start by default. read() reads 10 characters “1234567890”. The pointer is now positioned at ‘a’ i.e., 10 positions ahead. write() then writes the characters on screen. read() again reads next 10 characters “abcdefghij” and write() writes on the screen.

Program2: Program using lseek() system call that reads 10 characters from file “seeking” and print on screen. Skip next 5 characters and again read 10 characters and write on screen.

**#include<unistd.h>**

**#include<fcntl.h>**

**#include<sys/types.h**

**#include<sys/stat.h>**

**int main()**

**{**

**int n,f;**

**char buff[10];**

**f=open("seeking",O\_RDWR);**

**read(f,buff,10);**

**write(1,buff,10);**

**lseek(f,5,SEEK\_CUR);//skips 5 characters from the current position**

**read(f,buff,10);**

**write(1,buff,10);**

**}**

Output

This time the output will be the first 10 characters “1234567890” followed by “fghijxxxxx”. The inbetween 5 characters are skipped because we used lseek to reposition the pointer 5 characters ahead from the current (SEEK\_CUR) position.

**Program2: Program using lseek() system call that reads 10 characters from file “seeking” and print on screen. Skip next 5 characters and again read 10 characters and write on screen.**

**#include<unistd.h>**

**#include<fcntl.h>**

**#include<sys/types.h**

**#include<sys/stat.h>**

**int main()**

**{**

**int n,f;**

**char buff[10];**

**f=open("seeking",O\_RDWR);**

**read(f,buff,10);**

**write(1,buff,10);**

**lseek(f,5,SEEK\_CUR);//skips 5 characters from the current position**

**read(f,buff,10);**

**write(1,buff,10);**

**}**

**Output**

This time the output will be the first 10 characters “1234567890” followed by “fghij54321”. The inbetween 5 characters are skipped because we used lseek to reposition the pointer 5 characters ahead from the current (SEEK\_CUR) position.

**Program3: Write a program to print 10 characters starting from the 10th character from a file “lpu.txt”.**

//Let the contents of the file F1 be “1234567890abcdefghijxxxxxxxx”. This means we want the output to be “abcdefghij”.

//Note: the first character ‘1’ is at 0th position

**#include<unistd.h>**

**#include<fcntl.h>**

**#include<sys/types.h**

**#include<sys/stat.h>**

**#include<stdio.h>**

**int main()**

**{**

**int n,f,f1;**

**char buff[10];**

**f=open("seeking",O\_RDWR);**

**f1=lseek(f,10,SEEK\_SET);**

**printf("Pointer is at %d position\n",f1);**

**read(f,buff,10);**

**write(1,buff,10);**

**}**

**How it works?**

lseek is used to position the cursor at 10th position from the starting, hence the use of SEEK\_SET. f1 saves the current position of the pointer which is printed. Then the next 10 characters are read using read() and printed on screen using write().

**Linux Experiment 4 (**Directory Manipulation Using System Calls)

**//Program for Opendir() Readdir() Closedir()**

#include<sys/types.h>

#include<sys/stat.h>

#include<stdio.h>

#include<dirent.h>

#include<unistd.h>

int main()

{

DIR \*dp;

struct dirent \*direntPt;

dp=opendir("soni");

if(dp==NULL)

{

printf("error\n");

}

while((direntPt=readdir(dp)) != NULL)

{

printf("%s\n", direntPt->d\_name);

}

closedir(dp);

return 0;

}

**///Creating new directory**

#include <stdio.h>

#include <sys/stat.h>

#include <sys/types.h>

int main() {

**// Attempt to create a new directory named "NewDirectory"**

int n = mkdir("NewDirectory", 0777);

// Check if the directory creation was successful

if (n == 0) {

printf("Directory created successfully.\n");

} else {

printf("Failed to create directory.\n");

}

return 0;

}

**///Write a program that uses the getcwd system call to retrieve the current working directory and displays it to the user.**

#include <stdio.h>

#include <unistd.h>

#define MAX\_PATH\_LEN 4096 // Maximum length of a file path

int main() {

char cwd[MAX\_PATH\_LEN]; **// Buffer to store the current working directory**

**// Attempt to retrieve the current working directory**

if (getcwd(cwd, sizeof(cwd)) != NULL) {

printf("Current working directory: %s\n", cwd);

} else {

perror("getcwd() error");

return 1; // Return an error status

}

return 0;

}

* **getcwd** retrieves the current working directory and stores it in the provided buffer (**cwd**).
* If **getcwd** returns **NULL**, it indicates an error, so **perror** prints an error message.
* Otherwise, it prints the current working directory to the user.

**Experiment 5: Process Management using System Calls**

1. **A program to create a child process using fork system call.**

#include <stdio.h>

#include <unistd.h>

int main() {

pid\_t child\_pid;

**// Create a child process**

child\_pid = fork();

**// Check for errors**

if (child\_pid < 0) {

**// Fork failed**

perror("fork failed");

return 1;

} else if (child\_pid == 0) {

**// Child process**

printf("This is the child process. PID: %d\n", getpid());

} else {

**// Parent process**

printf("This is the parent process. PID: %d\n", getpid());

printf("Child process created. Child PID: %d\n", child\_pid);

}

return 0;

}

* **fork()** is called to create a new process. After **fork()**, the process is split into two: the parent process and the child process. Both continue execution from the point of the **fork()** call.
* The return value of **fork()** determines which process is currently running. If **fork()** returns a negative value, an error occurred. If it returns 0, the process is the child process. If it returns a positive value, the process is the parent process.
* In the child process, **fork()** returns 0. In the parent process, it returns the child's process ID (**PID**).
* Both the parent and child processes continue execution from the point after the **fork()** call, but they have separate memory spaces.

1. **C program to demonstrates the creation of an orphan process.**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main() {

pid\_t child\_pid;

printf("Parent process PID: %d\n", getpid());

**// Create a child process**

child\_pid = fork();

if (child\_pid < 0) {

**// Fork failed**

perror("fork failed");

return 1;

} else if (child\_pid == 0) {

**// Child process**

printf("Child process PID: %d\n", getpid());

**// Sleep to keep the child process running for demonstration**

printf("Child process sleeping for 10 seconds...\n");

sleep(10);

printf("Child process completed.\n");

} else {

**// Parent process**

printf("Parent process continuing execution.\n");

**// Parent process terminates immediately, becoming an orphan**

}

return 0;

}

* The parent process prints its PID and then forks a child process.
* If **fork()** succeeds, the child process prints its PID and then sleeps for 10 seconds to simulate some work.
* Meanwhile, the parent process doesn't wait for the child and immediately continues execution. This causes it to terminate before the child completes its work.
* Since the parent process terminates before the child, the child becomes an orphan process.
* Eventually, the orphan process is adopted by the init process, which has PID 1 on Unix-like systems.
* When the child process completes its sleep and finishes execution, it prints a completion message.

1. **/// fork creation // zombie process and orphan process**

#include<unistd.h>

#include<sys/types.h>

#include<stdio.h>

#include<wait.h> //fork creation // zombie process and orphan process

int main() // wait() sleep getpid getppid

{

pid\_t q;

printf("before fork\n");

q=fork();

if(q==0)

{ //sleep(5);

printf("child id is %d\n", getpid());

printf("parent id is %d\n", getppid());

}

else

{ // wait(NULL);

printf("in parent domain child id is %d\n",q);

printf(" parent domain parent id is %d\n",getpid());

}

printf("common variables\n");

}

Experiment 6

(Creation of Multithreaded Processes using Pthread Library)

**#include<stdio.h>**

**#include<unistd.h>**

**#include<pthread.h>**

**void \*thread\_function();**

**int main()**

**{**

**pthread\_t t1; //Thread initialization**

**pthread\_create(&t1,NULL,thread\_function,NULL); //Thread Creation**

**pthread\_join(t1,NULL); //Thread Join**

**printf("inside main program\n");**

**for(int i=1;i<6;i++)**

**{**

**printf("%d\n",i);**

**sleep(1);**

**}**

**}**

**void \*thread\_function()**

**{**

**printf("inside thread function\n");**

**for(int j =31;j<36;j++)**

**{**

**printf("%d\n",j);**

**sleep(1);**

**Output:**

**inside thread function**

**31**

**32**

**33**

**34**

**35**

**inside main program**

**1**

**2**

**3**

**4**

**5**

#include <stdio.h>

#include <string.h>

#include <pthread.h>

**// Global variable:**

int i = 2;

void\* foo(void\* p){

**// Print value received as argument:**

printf("Value recevied as argument in starting routine: ");

printf("%i\n", \* (int\*)p);

**// Return reference to global variable:**

pthread\_exit(&i);

}

int main(void){

**// Declare variable for thread's ID:**

pthread\_t id;

int j = 1;

pthread\_create(&id, NULL, foo, &j);

int\* ptr;

**// Wait for foo() and retrieve value in ptr;**

pthread\_join(id, (void\*\*)&ptr);

printf("Value recevied by parent from child: ");

printf("%i\n", \*ptr);

}

**Output:**

Value recevied as argument in starting routine: 1

Value recevied by parent from child: 2

**#include<pthread.h>**

**int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*start\_routine) (void \*), void arg);**

The first parameter is the buffer which will contain the ID of the new thread, if pthread\_create is successful. The second parameter specifies the attributes of the thread. This parameter is generally NULL until you want to change the default settings. The third parameter is the name the function which the thread will execute. Hence, everything that you want the thread to do should be defined in this function. Lastly, the fourth parameter is the input to the function in the third parameter. If the function in the third parameter does not take any input then the fourth parameter is NULL

* **Lines 1-3:**We defined all the necessary header files for standard input-output operations, string manipulation, and pthread library for threading functionalities.
* **Lines 8-15:**We defined a thread function, which takes the p a void pointer as its parameter, which is being printed. It then exits the thread and returns a reference to the global variable i using pthread\_exit.
* **Lines 17-30:**We defined the main function in which we manage the thread's execution. We declare a variable id to store the thread ID and initialize a local variable j to 1. Using pthread\_create, we create a new thread with foo as the function and j's address as the argument. We then wait for the thread to finish using pthread\_join and print the value returned by the child thread.

##### Program to create a thread. The thread prints numbers from zero to n, where value of n is passed from the main process to the thread. The main process also waits for the thread to finish first and then prints from 20-24.

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<pthread.h>

#include<string.h>

void \*thread\_function(void \*arg);

int i,n,j;

int main() {

char \*m="5";

pthread\_t a\_thread; //thread declaration

void \*result;

pthread\_create(&a\_thread, NULL, thread\_function, m); //thread is created

pthread\_join(a\_thread, &result);

printf("Thread joined\n");

for(j=20;j<25;j++)

{

printf("%d\n",j);

sleep(1);

}

printf("thread returned %s\n",(char \*)result);

}

void \*thread\_function(void \*arg) {

int sum=0;

n=atoi(arg);

for(i=0;i<n;i++)

{

printf("%d\n",i);

sleep(1);

}

pthread\_exit("Done"); // Thread returns "Done"

}

Experiment 7

Process Synchronization using Semaphore/- Mutex

**//Race Condition**

#include<unistd.h>

#include<stdio.h>

#include<pthread.h>

int ab=5;

void \*thread\_function1();

void \*thread\_function2();

int main()

{

pthread\_t t1,t2;

pthread\_create(&t1,NULL,thread\_function1,NULL);

pthread\_create(&t2,NULL,thread\_function2,NULL);

pthread\_join(t1,NULL);

pthread\_join(t2,NULL);

printf("final value of shared variable is %d\n",ab);

}

void \*thread\_function1()

{

int a=ab;

printf("value of thread 1 is %d\n",a);

a++;

printf(" thread 1 value updated to %d\n",a);

sleep(1);

ab=a;

printf("shared variable is %d\n",ab);

**Output:**

**value of thread 1 is 5**

**thread 1 value updated to 6**

**value of thread 2 is 5**

**thread 2 value updated to 4**

**shared variable is 6**

**shared variable is 4**

**final value of shared variable is 4**

**Q. Program creates two threads: one to increment the value of a shared variable and second to decrement the value of the shared variable. Both the threads make use of semaphore variable so that only one of the threads is executing in its critical section.**

**#include<pthread.h>**

**#include<stdio.h>**

**#include<semaphore.h>**

**#include<unistd.h>**

**void \*fun1();**

**void \*fun2();**

**int shared=1; //shared variable**

**sem\_t s; //semaphore variable**

**int main()**

**{**

**sem\_init(&s,0,1); //initialize semaphore variable - 1st argument is address of variable, 2nd is number of processes sharing semaphore, 3rd argument is the initial value of semaphore variable**

**pthread\_t thread1, thread2;**

**pthread\_create(&thread1, NULL, fun1, NULL);**

**pthread\_create(&thread2, NULL, fun2, NULL);**

**pthread\_join(thread1, NULL);**

**pthread\_join(thread2,NULL);**

**printf("Final value of shared is %d\n",shared); //prints the last updated value of shared variable**

**}**

**void \*fun1()**

**{**

**int x;**

**sem\_wait(&s); //executes wait operation on s**

**x=shared;//thread1 reads value of shared variable**

**printf("Thread1 reads the value as %d\n",x);**

**x++; //thread1 increments its value**

**printf("Local updation by Thread1: %d\n",x);**

**sleep(1); //thread1 is preempted by thread 2**

**shared=x; //thread one updates the value of shared variable**

**printf("Value of shared variable updated by Thread1 is: %d\n",shared);**

**sem\_post(&s);**

**}**

**void \*fun2()**

**{**

**int y;**

**sem\_wait(&s);**

**y=shared;//thread2 reads value of shared variable**

**printf("Thread2 reads the value as %d\n",y);**

**y--; //thread2 increments its value**

**printf("Local updation by Thread2: %d\n",y);**

**sleep(1); //thread2 is preempted by thread 1**

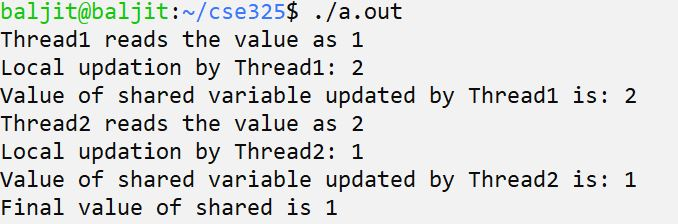
**shared=y; //thread2 updates the value of shared variable**

**printf("Value of shared variable updated by Thread2 is: %d\n",shared);**

**sem\_post(&s);**

**}**

**Output:**

****

The process initializes the semaphore variable s to ‘1’ using the sem\_init() function. The initial value is set to ‘1’ because binary semaphore is used here. If you have multiple instances of the resource then counting semaphores can be used. Next, the process creates two threads. thread1 acquires the semaphore variable by calling sem\_wait(). Next, it executes statements in its critical section part. We use sleep(1) function to preempt thread1 and start thread2. This simulates a real-life scenario. Now, when thraed2 executes sem\_wait() it will not be able to do so because thread1 is already in the critical section. Finally, thread1 calls sem\_post() function. Now thread2 will be able to acquire s using sem\_wait(). This ensures synchronization among threads.

/\* Program to show the race condition.  
2. Program to create two threads: one to increment the value of a shared variable and second to decrement the value of shared variable. Both the threads are executed, so the final value of shared variable should be same as its initial value. But due to race condition it would not be same. \*/

**#include<pthread.h>**

**#include<stdio.h>**

**#include<unistd.h>**

**void \*fun1();**

**void \*fun2();**

**int shared=1; //shared variable**

**int main()**

**{**

**pthread\_t thread1, thread2;**

**pthread\_create(&thread1, NULL, fun1, NULL);**

**pthread\_create(&thread2, NULL, fun2, NULL);**

**pthread\_join(thread1, NULL);**

**pthread\_join(thread2,NULL);**

**printf("Final value of shared is %d\n",shared); //prints the last updated value of shared variable**

**}**

**void \*fun1()**

**{**

**int x;**

**x=shared;//thread one reads value of shared variable**

**printf("Thread1 reads the value of shared variable as %d\n",x);**

**x++; //thread one increments its value**

**printf("Local updation by Thread1: %d\n",x);**

**sleep(1); //thread one is preempted by thread 2**

**shared=x; //thread one updates the value of shared variable**

**printf("Value of shared variable updated by Thread1 is: %d\n",shared);**

**}**

**void \*fun2()**

**{**

**int y;**

**y=shared;//thread two reads value of shared variable**

**printf("Thread2 reads the value as %d\n",y);**

**y--; //thread two increments its value**

**printf("Local updation by Thread2: %d\n",y);**

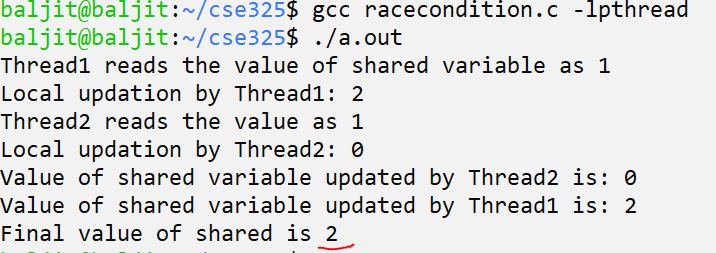
**sleep(1); //thread two is preempted by thread 1**

**shared=y; //thread one updates the value of shared variable**

**printf("Value of shared variable updated by Thread2 is: %d\n",shared);**

**}**

Output**:**

****

The final value of shared variable should have been 1 but it will be either 2 or 0 depending upon which thread executes first. This happened because the two processes were not synchronized. When one thread was modifying the value of shared variable the other thread must not have read its value for modification. This can be achieved using locks or semaphores.

Thread1 reads the value of shared variable as 1 and then increments it to 2. Now, before it could update the shared variable, Thread1 is preempted (using sleep()) by Thread2 which reads the unstable value of shared variable as 1. Thread2 then decrements it to 0. Then, both the threads prints the updated value of shared variable. Since, the final updation is done by Thread1, so the final value comes out to be 2.

1. Program create two threads: one to increment the value of a shared variable and second to decrement the value of shared variable. Both the threads make use of locks so that only one of the threads is executing in its critical section \*/.

**#include<pthread.h>**

**#include<stdio.h>**

**#include<unistd.h>**

**void \*fun1();**

**void \*fun2();**

**int shared=1; //shared variable**

**pthread\_mutex\_t l; //mutex lock**

**int main()**

**{**

**pthread\_mutex\_init(&l, NULL); //initializing mutex locks**

**pthread\_t thread1, thread2;**

**pthread\_create(&thread1, NULL, fun1, NULL);**

**pthread\_create(&thread2, NULL, fun2, NULL);**

**pthread\_join(thread1, NULL);**

**pthread\_join(thread2,NULL);**

**printf("Final value of shared is %d\n",shared); //prints the last updated value of shared variable**

**}**

**void \*fun1()**

**{**

**int x;**

**printf("Thread1 trying to acquire lock\n");**

**pthread\_mutex\_lock(&l); //thread one acquires the lock. Now thread 2 will not be able to acquire the lock //until it is unlocked by thread 1**

**printf("Thread1 acquired lock\n");**

**x=shared;//thread one reads value of shared variable**

**printf("Thread1 reads the value of shared variable as %d\n",x);**

**x++; //thread one increments its value**

**printf("Local updation by Thread1: %d\n",x);**

**sleep(1); //thread one is preempted by thread 2**

**shared=x; //thread one updates the value of shared variable**

**printf("Value of shared variable updated by Thread1 is: %d\n",shared);**

**pthread\_mutex\_unlock(&l);**

**printf("Thread1 released the lock\n");**

**}**

**void \*fun2()**

**{**

**int y;**

**printf("Thread2 trying to acquire lock\n");**

**pthread\_mutex\_lock(&l);**

**printf("Thread2 acquired lock\n");**

**y=shared;//thread two reads value of shared variable**

**printf("Thread2 reads the value as %d\n",y);**

**y--; //thread two increments its value**

**printf("Local updation by Thread2: %d\n",y);**

**sleep(1); //thread two is preempted by thread 1**

**shared=y; //thread one updates the value of shared variable**

**printf("Value of shared variable updated by Thread2 is: %d\n",shared);**

**pthread\_mutex\_unlock(&l);**

**printf("Thread2 released the lock\n");**

**}**

**Output:**

****

The final value of shared variable will be 1. When any one of the threads acquires the lock and is making changes to shared variable the other thread (even if it preempts the running thread) is not able to acquire the lock and thus not able to read the inconsistent value of shared variable. Thus only one of the thread is running in its critical section at any given time \*/

Experiment 8

Inter Process Communication (IPC)

# **Program for IPC using popen()**

IPC (Inter-process communication) can be achieved using popen/pclose functions. popen() function opens a process by creating a pipe, forking and invoking the shell. This pipe is a unidirectional pipe. Hence, it can be used either for reading or writing purpose only. We understand this concept with the help of a Program for IPC using popen()

## Syntax

#include<stdio.h>

FILE \*popen(const char \*command, const char \*type)

The first argument specifies the name of the process with which the communication is to take place and the second argument tells whether your current process is going to send data (writing into pipe) or receive data (reading from pipe).

We will demonstrate the use of popen using two different programs. One to send data to another process and second to receive data from the other process.

**Q. Program to write into a pipe i.e to send data from one process to another process.  
//ipc1.c**

**#include<stdio.h>  
#include<stdlib.h>  
#include<unistd.h>  
#include<string.h>**

**int main()**

**{**

**FILE \*rd;**

**char buffer[50];**

**sprintf(buffer,"name first");**

**rd=popen("wc -c","w"); // wc -c -> is the process which counts the number of characters passed. 2nd parameter is "w" which means pipe is opened in writing mode**

**fwrite(buffer,sizeof(char),strlen(buffer),rd); // to write the data into the pipe**

**pclose(rd);**

}

##### How it Works?

There are two programs “ipc1.c” (which will send the data) and “wc” command(which will receive the data). As the data will be sent to another process, so the mode of opening the pipe is writing mode “w”. popen() establishes the pipe between ipc1.c and wc. fwrite() function writes data into this pipe.  
ipc1.c stores some data in the buffer, then it connects with “wc” using popen. Finally, the fwrite() function writes data into the pipe.  
The data is received by “wc” which then counts the number of characters in the input and prints it

##### Output:

The output will be  
**10**  
because ipc1.c passed “name first” which contains 10 characters. So, when wc -c reads this it prints the count of number of characters which is 10

##### Program 2:

//Q. Program to read from a pipe i.e. to receive data from another process  
// ipc2.c

**#include<stdio.h>**

**#include<stdlib.h>**

**#include<unistd.h>**

**#include<string.h>**

**int main()**

**{**

**FILE \*rd;**

**char buffer[50];**

**rd=popen("ls","r"); //pipe opened in reading mode**

**fread(buffer, 1, 40, rd); //will read only 50 characters**

**printf("%s\n", buffer);**

**pclose(rd);**

**}**

#### How it works?

In this case, we establish the pipe between “ipc2.c” and “ls”. Since, the data will be read, hence, the pipe is opened in reading mode “r”. ls sends the data through the pipe. This data will be read by ipc2.c. So, this time our program is the one receiving the data. ls will send the list of files in current working directory. ipc2.c will read that, save it in buffer and then finally print it.

**Remember:** The **ls** command generates the list of all files in the current working directory but our process will read only first 40 character as specified in the fread() function. So, the output will consist of 40 characters only.

#### Output

1.c

123.txt

NewDirectory

NewDirectory22

D

# **Program for IPC using pipe() function**

The second method for IPC is using the pipe() function. Before writing a program for IPC using pipe() function let us first understand its working.

##### Syntax:

#include<unistd.h>  
int pipe(int pipefd[2]);

pipe() function creates a unidirectional pipe for IPC. On success it return two file descriptors pipefd[0] and pipefd[1]. pipefd[0] is the reading end of the pipe. So, the process which will receive the data should use this file descriptor. pipefd[1] is the writing end of the pipe. So, the process that wants to send the data should use this file descriptor.

**The program below creates a child process. The parent process will establish a pipe and will send the data to the child using writing end of the pipe and the child will receive that data and print on the screen using the reading end of the pipe.**

**Q. Program to send a message from parent process to child process using pipe()**

**#include<stdio.h>  
#include<unistd.h>  
#include<sys/types.h>  
#include<sys/wait.h>  
int main()  
{  
int fd[2],n;  
char buffer[100];  
pid\_t p;  
pipe(fd); //creates a unidirectional pipe with two end fd[0] and fd[1]  
p=fork();  
if(p>0) //parent  
{  
printf("Parent Passing value to child\n");  
write(fd[1],"hello\n",6); //fd[1] is the write end of the pipe  
int status;**

**wait(&status); // Wait for child process to terminate  
}  
else // child  
{  
printf("Child printing received value\n");  
n=read(fd[0],buffer,100); //fd[0] is the read end of the pipe  
write(1,buffer,n);  
}  
}**

#### **Output:**

#### **Parent Passing value to child**

#### **Child printing received value**

#### **Hello**

#### How it works?

The parent process create a pipe using pipe(fd) call and then creates a child process using fork(). Then the parent sends the data by writing to the writing end of the pipe by using the fd[1] file descriptor. The child then reads this using the fd[0] file descriptor and stores it in buffer. Then the child prints the received data from the buffer onto the screen.

# **Program for IPC using named pipes (mkfifo())**

The third method for IPC is using mkfifo() function. mkfifo() creates a named pipe which can be used exactly like a file. So, if you know how to read/write in a file this is a convenient method for IPC

#### Syntax:

#include<sys/types.h>  
#include<sys/stat.h>  
int mkfifo(const char \*pathname, mode\_t mode);

* mkfifo() makes a FIFO special file with the name specified by pathname and the permissions are specified by mode. On success mkfifo() returns 0 while on error it returns -1.
* The advantage is that this FIFO special file can be used by any process for reading or writing just like a normal file. This means to sender process can use the write() system call to write data into the pipe and the receiver process can use the read() system call to read data from the pipe, hence, completing the communication.
* It is same as a pipe except that it is accessed as part of the filesystem. Multiple process can access it for writing and reading. When the FIFO special files is used for exchange of data by process, the entire data is passed internally without writing it on the filesystem. Hence, if you open this special file there will be no content written in it.

**Note:** The FIFO pipe works in blocked mode(by default) i.e., the writing process must be present on one end while the reading process must be present on the other side at the same time else the communication will not happen. Operating the FIFO special file in non-blocking mode is also possible.

The entire IPC process will consist of three programs:

**Program1: to create a named pipe  
Program2: process that will write into the pipe (sender process)  
Program3: process that will receive data from pipe (receiver process)**

//Program1: Creating fifo/named pipe ( 1.c )

#include<stdio.h>

#include<sys/types.h>

#include<sys/stat.h>

int main()

{

int res;

res = mkfifo("fifo1",0777); //creates a named pipe with the name fifo1

printf("named pipe created\n");

}

#### How it works?

This will simply create a named pipe (fifo1) with read, write and execute permission for all users. You can change this to whatever you prefer

**Step 2** is to create a process which will use this pipe to send data. The below program will do that.

//**Program2:**Writing to a fifo/named pipe ( 2.c )

#include<unistd.h>

#include<stdio.h>

#include<fcntl.h>

int main()

{

int res,n;

res=open("fifo1",O\_WRONLY);

write(res,"Message",7);

printf("Sender Process %d sent the data\n",getpid());

}

$gcc -o 2 2.c  
//Note: If you run this you will not see any output

#### How it works?

The above code opens the pipe created previously in writing mode (because it wants to send data). Then it uses “write” system call to write some data into it. Finally, it prints a message using printf. But when you compile and run it, it won’t run because by default the sender runs in BLOCKING mode which means that until the receiver is not there the sender process gets blocked. Hence, you need a receiver process also.

The **third step** is to create the receiver process. The below program does so.

//**Program 3**: Reading from the named pipe ( 3.c )

#include<unistd.h>

#include<stdio.h>

#include<fcntl.h>

int main()

{

int res,n;

char buffer[100];

res=open("fifo1",O\_RDONLY);

n=read(res,buffer,100);

printf("Reader process %d started\n",getpid());

printf("Data received by receiver %d is: %s\n",getpid(), buffer);

}

Compile the program as  
$ gcc -o 3 3.c

#### How it works?

This program connects to the pipe in reading mode and reads the data into buffer and prints it. But again this program will not run. Because the receiver is BLOCKED until the sender is there.

Therefore, run both the object files simultaneously as  
$./2 & ./3  
and you will see the output as

[1] 157

Sender Process 157 sent the data

Reader process 158 started

Data received by receiver 158 is: Message

[1]+ Done ./3

# **Program for IPC using shared memory**

**Shared Memory** is the fastest inter-process communication (IPC) method. The operating system maps a memory segment in the address space of several processes so that those processes can read and write in that memory segment.The overview is as shown below:  
Two functions:shmget() and shmat() are used for IPC using shared memory. shmget() function is used to create the shared memory segment while shmat() function is used to attach the shared segment with the address space of the process.

#### Syntax (shmget()):

#include <sys/ipc.h>

#include <sys/shm.h>

int shmget(key\_t key, size\_t size, int shmflg);

The first parameter specifies the unique number (called key) identifying the shared segment. The second parameter is the size of the shared segment e.g. 1024 bytes or 2048 bytes. The third parameter specifies the permissions on the shared segment. On success the shmget() function returns a valid identifier while on failure it return -1.

#### Syntax (shmat()):

#include <sys/types.h>

#include <sys/shm.h>

void \*shmat(int shmid, const void \*shmaddr, int shmflg);

shmat() is used to attach the created shared segment with the address space of the calling process. The first parameter here is the identifier which shmget() function returns on success. The second parameter is the address where to attach it to the calling process. A NULL value of second parameter means that the system will automatically choose a suitable address. The third parameter is ‘0’ if the second parameter is NULL, otherwise, the value is specified by SHM\_RND.

**We will write two program for IPC using shared memory. Program 1 will create the shared segment, attach to it and then write some content into it. Then Program 2 will attach itself to the shared segment and read the value written by Program 1.**

**//Program 1: This program creates a shared memory segment, attaches itself to it and then writes some content into the shared memory segment.**

**#include<stdio.h>**

**#include<stdlib.h>**

**#include<unistd.h>**

**#include<sys/shm.h>**

**#include<string.h>**

**int main()**

**{**

**int i;**

**void \*shared\_memory;**

**char buff[100];**

**int shmid;**

**shmid=shmget((key\_t)2345, 1024, 0666|IPC\_CREAT); //creates shared memory segment with key 2345, having size 1024 bytes. IPC\_CREAT is used to create the shared segment if it does not exist. 0666 are the permisions on the shared segment**

**printf("Key of shared memory is %d\n",shmid);**

**shared\_memory=shmat(shmid,NULL,0); //process attached to shared memory segment**

**printf("Process attached at %p\n",shared\_memory); //this prints the address where the segment is attached with this process**

**printf("Enter some data to write to shared memory\n");**

**read(0,buff,100); //get some input from user**

**strcpy(shared\_memory,buff); //data written to shared memory**

**printf("You wrote : %s\n",(char \*)shared\_memory);**

}

#### 

#### How it works?

shmget() function creates a segment with key 2345, size 1024 bytes and read and write permissions for all users. It returns the identifier of the segment which gets store in shmid. This identifier is used in shmat() to attach the shared segment to the address space of the process. NULL in shmat() means that the OS will itself attach the shared segment at a suitable address of this process.  
Then some data is read from the user using read() system call and it is finally written to the shared segment using strcpy() function.

#### //**Program 2:** This program attaches itself to the shared memory segment created in Program 1. Finally, it reads the content of the shared memory

**#include<stdio.h>**

**#include<stdlib.h>**

**#include<unistd.h>**

**#include<sys/shm.h>**

**#include<string.h>**

**int main()**

**{**

**int i;**

**void \*shared\_memory;**

**char buff[100];**

**int shmid;**

**shmid=shmget((key\_t)2345, 1024, 0666);**

**printf("Key of shared memory is %d\n",shmid);**

**shared\_memory=shmat(shmid,NULL,0); //process attached to shared memory segment**

**printf("Process attached at %p\n",shared\_memory);**

**printf("Data read from shared memory is : %s\n",(char \*)shared\_memory);**

**}**

#### 

#### How it works?

shmget() here generates the identifier of the same segment as created in Program 1. Remember to give the same key value. The only change is, do not write IPC\_CREAT as the shared memory segment is already created. Next, shmat() attaches the shared segment to the current process.  
After that, the data is printed from the shared segment. In the output, you will see that it is the same data that you have written while executing the Program 1.

# **Program for IPC using Message Queues**

Program for IPC using Message queues are almost similar to named pipes with the exception that they do not require the opening and closing of pipes. But, they face one similar problem like named pipes; blocking on full pipes. Message queues send blocks of data from one process to another. Each block of data is considered to have a type. There is an upper limit on the maximum size of each block and also a limit on the maximum total size of all blocks on all queues in the system.

### **Message Queue Functions**

There are 4 important functions that we will use in the programs to achieve IPC using message queues

* int msgget(key\_t key, int msgflg);
* int msgsnd(int msqid, const void \*msg\_ptr, size\_t msg\_sz, int msgflg);
* int msgrcv(int msqid, void \*msg\_ptr, size\_t msg\_sz, long int msgtype, int msgflg);
* int msgctl(int msqid, int command, struct msqid\_ds \*buf);

Now let’s understand each of these functions in detail.

int msgget(key\_t key, int msgflg);

create and access a message queue, we use the msgget function. It takes two parameters. The first parameter is a key that names a message queue in the system. The second parameter is used to assign permission to the message queue and is ORed with IPC\_CREAT to create the queue if it doesn’t already exist. if the queue already exists then IPC\_CREAT is ignored. On success, the msgget function returns a positive number which is the queue identifier while on failure it returns -1.

The second functions is msgsnd

int msgsnd(int msqid, const void \*msg\_ptr, size\_t msg\_sz, int msgflg);

This function allows us to add a message to the message queue. The first parameter (msgid) is the message queue identifier returned by the msgget function. The second parameter, is the pointer to the message to be sent, which must start with a long int type. The third parameter, is the size of the message. It must not include the long int message type. The fourth and final parameter controls what happens if either the message queue is full or the system limit on queued messages is reached. The function on success returns 0 and place the copy of message data on the message queue. On failure it returns -1.

There are two constraints related to the structure of the message. First, it must be smaller than the system limit and,  
second, it must start with a long int. This long int is used as a message type in the receive function. The best structure of the message is:  
struct my\_message {  
long int message\_type;  
/\* The data you wish to transfer \*/  
}  
Since the message\_type is used in message reception, you can’t simply ignore it. Not only must you declare  
your data structure to include it, it’s also wise to initialize it, so that it contains a known value.

The third function is msgrcv

int msgrcv(int msqid, void \*msg\_ptr, size\_t msg\_sz, long int msgtype, int msgflg);

This function retrieves messages from a message queue. The first parameter (msgid) is the message queue identifier returned by the msgget function. The second parameter, is the pointer to the message to be received, which must start with a long int type as explained above. The third parameter is the size of the message.

The fourth parameter allows implementing priority. If the value is 0, the first available message in the queue is retrieved. But if the value is greater than 0 then the first message with the same message type is retrieved. If the value is less than 0 then the first message having the type value same as the absolute value of msgtype is retrieved. In simple words 0 value means to receive the messages in the order in which they were sent and non zero means receive the message with a specific message type. The final parameter controls what happens if either the message queue is full or the system limit on queued messages is reached. The function on success returns 0 and place the copy of message data on the message queue. On failure it returns -1.

The final function is msgctl, which is the control function.

int msgctl(int msqid, int command, struct msqid\_ds \*buf);

The first parameter is the identifier returned by msgget function. The second parameter can have one out of the below three values

|  |  |
| --- | --- |
| Command | Description |
| IPC\_STAT | Sets the data in the msqid\_ds structure to reflect the values associated with the message queue. |
| IPC\_SET | If the process has permission to do so, this sets the values associated with the message queue to those provided in the msqid\_ds data structure. |
| IPC\_RMID | Deletes the message queue. |

The msgctl function returns 0 on success and -1 on error. The send or receive function will fail if a message queue is deleted while a process is waiting in a msgsnd or msgrcv function.

#### **Program 1: Program for IPC using Message Queues To send data to a message queue**

#include<stdlib.h>

#include<stdio.h>

#include<string.h>

#include<unistd.h>

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/msg.h>

#define MAX\_TEXT 512 //maximum length of the message that can be sent allowed

struct my\_msg{

long int msg\_type;

char some\_text[MAX\_TEXT];

};

int main()

{

int running=1;

int msgid;

struct my\_msg some\_data;

char buffer[50]; //array to store user input

msgid=msgget((key\_t)14534,0666|IPC\_CREAT);

if (msgid == -1) // -1 means the message queue is not created

{

printf("Error in creating queue\n");

exit(0);

}

while(running)

{

printf("Enter some text:\n");

fgets(buffer,50,stdin);

some\_data.msg\_type=1;

strcpy(some\_data.some\_text,buffer);

if(msgsnd(msgid,(void \*)&some\_data, MAX\_TEXT,0)==-1) // msgsnd returns -1 if the message is not sent

{

printf("Msg not sent\n");

}

if(strncmp(buffer,"end",3)==0)

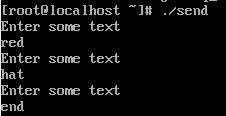
{

running=0;

}

}

}



### **How it Works?**

The structure my\_msg declares the long int variable and the char array to store the data to be sent to the message queue. Then the message queue is created using the msgget() function. Next, read data from the user into the buffer using fgets() and then copy it into the variable some\_text of the structure some\_data. Finally, send the data to the queue using the msgsnd() function. The strcmp function is used to stop sending the data by comparing the first three characters of the data. If the data starts with “end” this means no more data is to be sent.

#### **Program 2: Program for IPC using Message Queues To receive/read message from the above-created message queue**

#include<stdlib.h>

#include<stdio.h>

#include<string.h>

#include<unistd.h>

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/msg.h>

struct my\_msg{

long int msg\_type;

char some\_text[BUFSIZ];

};

int main()

{

int running=1;

int msgid;

struct my\_msg some\_data;

long int msg\_to\_rec=0;

msgid=msgget((key\_t)14534,0666|IPC\_CREAT);

while(running)

{

msgrcv(msgid,(void \*)&some\_data,BUFSIZ,msg\_to\_rec,0);

printf("Data received: %s\n",some\_data.some\_text);

if(strncmp(some\_data.some\_text,"end",3)==0)

{

running=0;

}

}

msgctl(msgid,IPC\_RMID,0);

}

#### t:

### How it works?

The msg\_to\_rec variable is set to 0 so that the data is received in the same order as sent (refer the theory above for more details). The while is used to continuous receive the data using the mgrcv() function until the text received is “end”, which we check using the strcmp function. The data is read using the structure my\_msg.