OPERATING SYSTEMS

RECORD

->Karedla Anantha Sashi Sekhar

->1601-14-733-091

->BE(3/4) CSE-2

# FILE RELATED SYSTEM CALLS

1. **Program to demonstrate the use creat system call.**

**System calls required:**

**creat() system call :**

open, creat - open and possibly create a file or device

**Header files required:**

#include <[sys/types.h](https://linux.die.net/include/sys/types.h)>

#include <[sys/stat.h](https://linux.die.net/include/sys/stat.h)>

#include <[fcntl.h](https://linux.die.net/include/fcntl.h)>

**Syntax:**

int creat(const char \*pathname, mode\_t\* mode)

**Description:**

Given a *pathname* for a file, open() returns a file descriptor, a small, nonnegative integer for use in subsequent system calls. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

A call to open() creates a new *open file description*, an entry in the system-wide table of open files. This entry records the file offset and the file status flags. A file descriptor is a reference to one of these entries; this reference is unaffected if *pathname* is subsequently removed or modified to refer to a different file. The new open file description is initially not shared with any other process, but sharing may arise via [***fork***](https://linux.die.net/man/2/fork)*(2)*.

The argument *flags* must include one of the following *access modes*: O\_RDONLY, O\_WRONLY, or O\_RDWR. These request opening the file read-only, write-only, or read/write, respectively.

**Return Value:**

creat() return the new file descriptor, or -1 if an error occurred.

**close() system call :**

close - close a file descriptor

**Header files required:**

#include <[unistd.h](https://linux.die.net/include/unistd.h)>

**Syntax:**

int close(int fd);

**Description:**

close() closes a file descriptor, so that it no longer refers to any file and may be reused. Any record locks held on the file it was associated with, and owned by the process, are removed (regardless of the file descriptor that was used to obtain the lock).

If *fd* is the last file descriptor referring to the underlying open file description, the resources associated with the open file description are freed; if the descriptor was the last reference to a file which has been removed using [*unlink*](https://linux.die.net/man/2/unlink)*(2)* the file is deleted.

**Return Value:**

close() returns zero on success. On error, -1 is returned.

**Program:**

#include<stdio.h> /\*header file for main function\*/

#include<sys/types.h>

#include<sys/stat.h> /\*header files for creat() system call\*/

#include<fcntl.h>

int main()

{

int fd; /\*creating 2 file descriptors\*/

int fd1;

fd=creat("first.txt",S\_IREAD|S\_IWRITE); /\*creating 2 files which \*/

fd1=creat("second.txt",S\_IREAD|S\_IWRITE); //returns file descriptors

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

printf("%d\n",fd1);

if(fd==-1) /\*checking whether file descriptor is negative or not\*/

printf("ERROR\n");

else

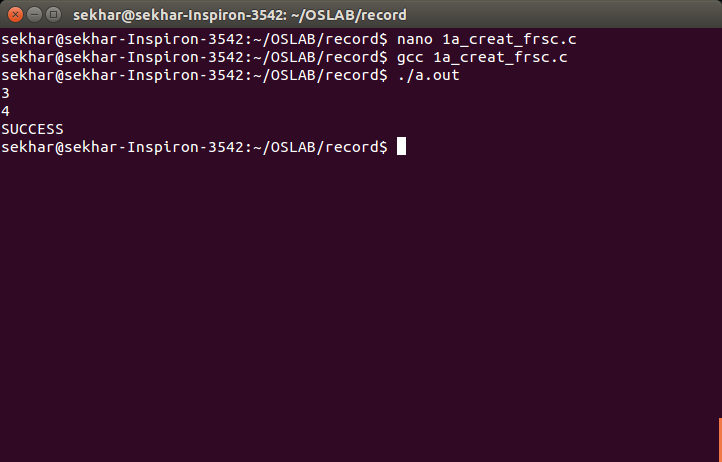
printf("SUCCESS\n");

close(fd); /\*closing the file descriptors\*/

close(fd1);

}

**OUTPUT:**



1. **Program to write contents from file to console**

**System calls required:**

**open() system call:**

open - open and possibly create a file or device

**Header files required:**

#include <[sys/types.h](https://linux.die.net/include/sys/types.h)>

#include <[sys/stat.h](https://linux.die.net/include/sys/stat.h)>

#include <[fcntl.h](https://linux.die.net/include/fcntl.h)>

**Syntax:**

int open(const char \*pathname, int flags);

int open(const char \*pathname,int flags, mode\_t mode);

**Description:**

Given a *pathname* for a file, open() returns a file descriptor, a small, nonnegative integer for use in subsequent system calls. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

A call to open() creates a new *open file description*, an entry in the system-wide table of open files. This entry records the file offset and the file status flags. A file descriptor is a reference to one of these entries; this reference is unaffected if *pathname* is subsequently removed or modified to refer to a different file. The new open file description is initially not shared with any other process, but sharing may arise via [***fork***](https://linux.die.net/man/2/fork)*(2)*.

The argument *flags* must include one of the following *access modes*: O\_RDONLY, O\_WRONLY, or O\_RDWR. These request opening the file read-only, write-only, or read/write, respectively.

**Return Value:**

open() return the new file descriptor, or -1 if an error occurred.

**Read() system call:**

read - read from a file descriptor

**Header file required:**

#include <[unistd.h](https://linux.die.net/include/unistd.h)>

**Syntax:**

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

**Description:**

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

**Return Value:**

On success, the number of bytes read is returned,On error, -1 is returned.

**Write() system call:**

write - write to a file descriptor

**Header file required:**

#include <[unistd.h](https://linux.die.net/include/unistd.h)>

**Syntax:**

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

**Description:**

write() writes up to *count* bytes from the buffer pointed *buf* to the file referred to by the file descriptor *fd*.

**Return Value:**

On success, the number of bytes written is returned (zero indicates nothing was written). On error, -1 is returned.

If *count* is zero and *fd* refers to a regular file, then write() may return a failure status if one of the errors is detected. If no errors are detected, 0 will be returned without causing any other effect. If *count* is zero and *fd* refers to a file other than a regular file, the results are not specified.

**Lseek() system call:**

lseek - reposition read/write file offset

**Header files required:**

#include<[sys/types.h](https://linux.die.net/include/sys/types.h)>  
#include <[unistd.h](https://linux.die.net/include/unistd.h)>

**Syntax:**

off\_t lseek(int *fd*, off\_t *offset*, int *whence*);

**Description:**

The lseek() function repositions the offset of the open file associated with the file descriptor *fd* to the argument *offset* according to the directive *whence* as follows:

SEEK\_SET

The offset is set to *offset* bytes.

SEEK\_CUR

The offset is set to its current location plus *offset* bytes.

SEEK\_END

The offset is set to the size of the file plus *offset* bytes.

**Return Value:**

Upon successful completion, lseek() returns the resulting offset location as measured in bytes from the beginning of the file. On error, the value *(off\_t) -1* is returned.

**Exit() system call:**

\_exit, \_Exit - terminate the calling process

**Header files required:**

#include <[unistd.h](https://linux.die.net/include/unistd.h)>

#include <[stdlib.h](https://linux.die.net/include/stdlib.h)>

**Syntax:**

void \_exit(int *status*);

void \_Exit(int *status*);

**Description:**

The function \_exit() terminates the calling process "immediately". Any open file descriptors belonging to the process are closed; any children of the process are inherited by process 1, *init*, and the process's parent is sent a SIGCHLD signal.

The value *status* is returned to the parent process as the process's exit status, and can be collected using one of the [wait](https://linux.die.net/man/2/wait)(2) family of calls.

The function \_Exit() is equivalent to \_exit().

**Return Value:**

These functions do not return.

**Program:**

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

int main(int argc,char \*argv[])

{

int fd;

int n\_char=0;

char buffer[1];

fd=open(argv[1],O\_RDONLY);

if(fd==-1)

{

exit(-1);

}

while((n\_char=read(fd,buffer,1))!=0)

{

//printf("%d",n\_char);

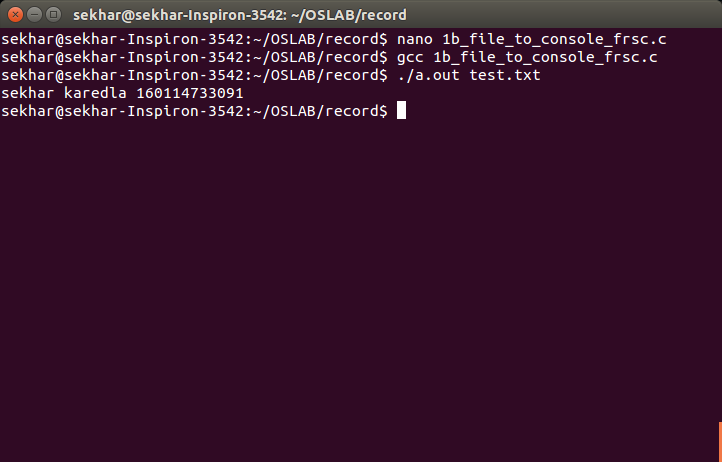
write(1,buffer,n\_char);

}

return 0;

}

**Output:**



**3. Program to read from one file and write to another file**

**System calls required:**

**Creat(),open() system call**

open, creat - open and possibly create a file or device

**Header files required:**

#include <[sys/types.h](https://linux.die.net/include/sys/types.h)>

#include <[sys/stat.h](https://linux.die.net/include/sys/stat.h)>

#include <[fcntl.h](https://linux.die.net/include/fcntl.h)>

**Syntax:**

int creat(const char \*pathname, mode\_t mode);

int open(const char \*pathname, int flags);

int open(const char \*pathname,int flags, mode\_t mode);

**Description:**

Given a *pathname* for a file, open() returns a file descriptor, a small, nonnegative integer for use in subsequent system calls. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

A call to open() creates a new *open file description*, an entry in the system-wide table of open files. This entry records the file offset and the file status flags. A file descriptor is a reference to one of these entries; this reference is unaffected if *pathname* is subsequently removed or modified to refer to a different file. The new open file description is initially not shared with any other process, but sharing may arise via [***fork***](https://linux.die.net/man/2/fork)*(2)*.

The argument *flags* must include one of the following *access modes*: O\_RDONLY, O\_WRONLY, or O\_RDWR. These request opening the file read-only, write-only, or read/write, respectively.

**Return Value:**

creat() and open() return the new file descriptor, or -1 if an error occurred.

**Read() system call:**

read - read from a file descriptor

**Header file required:**

#include <[unistd.h](https://linux.die.net/include/unistd.h)>

**Syntax:**

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

**Description:**

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

**Return Value:**

On success, the number of bytes read is returned,On error, -1 is returned.

**Write() system call:**

write - write to a file descriptor

**Header file required:**

#include <[unistd.h](https://linux.die.net/include/unistd.h)>

**Syntax:**

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

**Description:**

write() writes up to *count* bytes from the buffer pointed *buf* to the file referred to by the file descriptor *fd*.

**Return Value**

On success, the number of bytes written is returned (zero indicates nothing was written). On error, -1 is returned.

If *count* is zero and *fd* refers to a regular file, then write() may return a failure status if one of the errors is detected. If no errors are detected, 0 will be returned without causing any other effect. If *count* is zero and *fd* refers to a file other than a regular file, the results are not specified.

**Close(2) system call:**

close - close a file descriptor

**Header files required:**

#include <[unistd.h](https://linux.die.net/include/unistd.h)>

**Syntax:**

int close(int fd);

**Description:**

close() closes a file descriptor, so that it no longer refers to any file and may be reused. Any record locks held on the file it was associated with, and owned by the process, are removed (regardless of the file descriptor that was used to obtain the lock).

If *fd* is the last file descriptor referring to the underlying open file description, the resources associated with the open file description are freed; if the descriptor was the last reference to a file which has been removed using [*unlink*](https://linux.die.net/man/2/unlink)*(2)* the file is deleted.

**Return Value:**

close() returns zero on success. On error, -1 is returned.

**Exit() system call:**

\_exit, \_Exit - terminate the calling process

**Header files required:**

#include <[unistd.h](https://linux.die.net/include/unistd.h)>

#include <[stdlib.h](https://linux.die.net/include/stdlib.h)>

**Syntax:**

void \_exit(int *status*);

void \_Exit(int *status*);

**Description**

The function \_exit() terminates the calling process "immediately". Any open file descriptors belonging to the process are closed; any children of the process are inherited by process 1, *init*, and the process's parent is sent a SIGCHLD signal.

The value *status* is returned to the parent process as the process's exit status, and can be collected using one of the [wait](https://linux.die.net/man/2/wait)(2) family of calls.

The function \_Exit() is equivalent to \_exit().

**Return Value**

These functions do not return.

**Program:**

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

int main()

{

int fd1,fd2;

char ch[1];

fd1=open("first.txt",O\_RDONLY);

printf("%d\n",fd1);

fd2=creat("second.txt",S\_IREAD|S\_IWRITE);

printf("%d\n",fd2);

if(fd1<0||fd2<0)

{

printf("Error");

exit(-1);

}

while((read(fd1,ch,1))>0)

{

write(fd2,ch,1);

printf("%c",ch[0]);

}

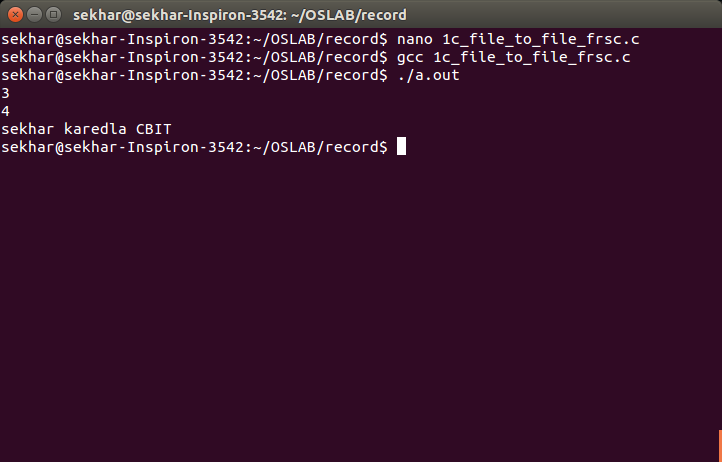
close(fd1);

close(fd2);

return 0;

}

**Output:**



**4.Program to show the working of lseek function.**

**Program:**

#include<unistd.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

#include<stdio.h>

#include<stdlib.h>

int main()

{

int fd1=open("1.txt",O\_RDWR);

printf("%d",fd1);

char buffer[1];

buffer[0]='1';

printf("enter the data : (press # to exit)");

do

{

scanf("%c",&buffer[0]);

if(buffer[0]!='#')

write(fd1,buffer,1);

}while(buffer[0]!='#');

close(fd1);

int fd2=open("1.txt",O\_RDWR);

lseek(fd2,2\*sizeof(char),0);int k;

do

{

k=read(fd2,&buffer[0],1);

if(k!=0)

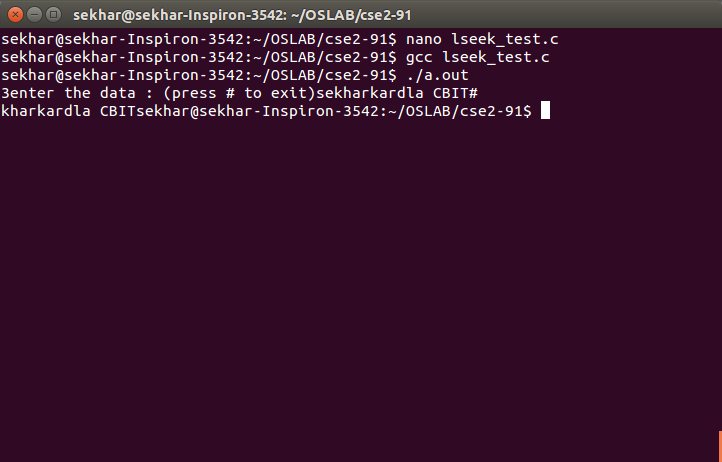
printf("%c",buffer[0]);

}while(k!=0);

return 0;

}

**Output:-**

****

# PROCESS RELATED SYSTEM CALLS

**5. Program to demonstrate fork system call**

**System calls required:**

**fork() system call:**

fork - create a child process

**Header file required:**

#include <[unistd.h](https://linux.die.net/include/unistd.h)>

**Syntax:**

pid\_t fork(void);

**Description:**

fork() creates a new process by duplicating the calling process. The new process, referred to as the *child*, is an exact duplicate of the calling process, referred to as the *parent*.

**Return Value:**

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created.

**open(2) system call:**

open, creat - open and possibly create a file or device

**Header files required:**

#include <[sys/types.h](https://linux.die.net/include/sys/types.h)>

#include <[sys/stat.h](https://linux.die.net/include/sys/stat.h)>

#include <[fcntl.h](https://linux.die.net/include/fcntl.h)>

**Syntax:**

int open(const char \*pathname, int flags);

int open(const char \*pathname,int flags, mode\_t mode);

**Description:**

Given a *pathname* for a file, open() returns a file descriptor, a small, nonnegative integer for use in subsequent system calls. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

A call to open() creates a new *open file description*, an entry in the system-wide table of open files. This entry records the file offset and the file status flags. A file descriptor is a reference to one of these entries; this reference is unaffected if *pathname* is subsequently removed or modified to refer to a different file. The new open file description is initially not shared with any other process, but sharing may arise via [***fork***](https://linux.die.net/man/2/fork)*(2)*.

The argument *flags* must include one of the following *access modes*: O\_RDONLY, O\_WRONLY, or O\_RDWR. These request opening the file read-only, write-only, or read/write, respectively.

**Return Value:**

open() return the new file descriptor, or -1 if an error occurred.

**Read() system call:**

read - read from a file descriptor

**Header file required:**

#include <[unistd.h](https://linux.die.net/include/unistd.h)>

**Syntax:**

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

**Description:**

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

**Return Value:**

On success, the number of bytes read is returned,On error, -1 is returned.

**Program:**

#include<stdio.h>

#include<unistd.h>

int main()

{

int a=2;

pid\_t pid;

pid=fork();

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

if(pid<0)

{

printf("fork failed");

}

else if(pid==0)

{

printf("child process \t a is : ");

printf("%d\n",++a);

}

else

{

printf("parent process \t a is : ");

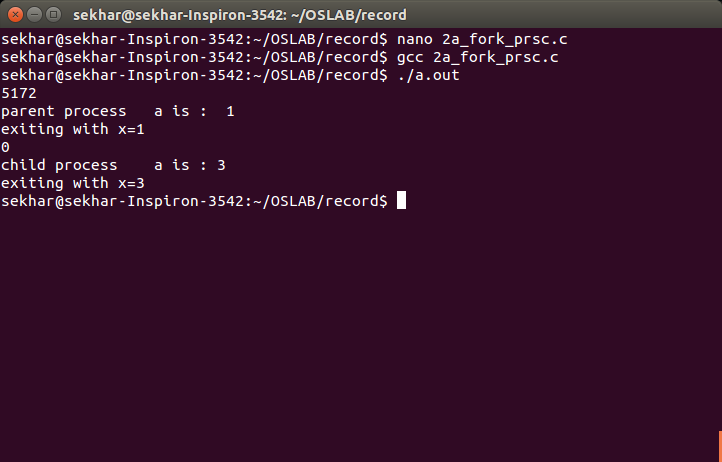
printf("%d\n",--a);

}

printf("exiting with x=%d\n",a);

}

**Output:**



**6. Program to demonstrate getpid(),getppid() system calls**

**System calls required:**

**fork() system call:**

fork - create a child process

**Header file required:**

#include <[unistd.h](https://linux.die.net/include/unistd.h)>

**Syntax:**

pid\_t fork(void);

**Description:**

fork() creates a new process by duplicating the calling process. The new process, referred to as the *child*, is an exact duplicate of the calling process, referred to as the *parent*.

**Return Value:**

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created.

**getpid(),getppid() system calls:**

getpid, getppid - get process identification

**Header files required:**

#include <[**sys/types.h**](https://linux.die.net/include/sys/types.h)>  
#include <[**unistd.h**](https://linux.die.net/include/unistd.h)>

**Syntax:**

pid\_t getpid(void);  
pid\_t getppid(void);

**Description:**

getpid() returns the process ID of the calling process. (This is often used by routines that generate unique temporary filenames.)

getppid() returns the process ID of the parent of the calling process.

**Program:**

#include<stdio.h>

#include<unistd.h>

int main()

{

int a=2;

pid\_t pid;

pid=fork();

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

if(pid<0)

{

printf("Error");

}

else if(pid==0)

{

sleep(10);

printf("child process \t a is : ");

printf("%d\n",++a);

printf("I am the child and my process id is %d\n",getpid());

printf("I am the child and my parent process id is %d\n",getpid());

}

else

{

printf("parent process \t a is : ");

printf("%d\n",--a);

printf("I am the parent and my process id is %d\n",getpid());

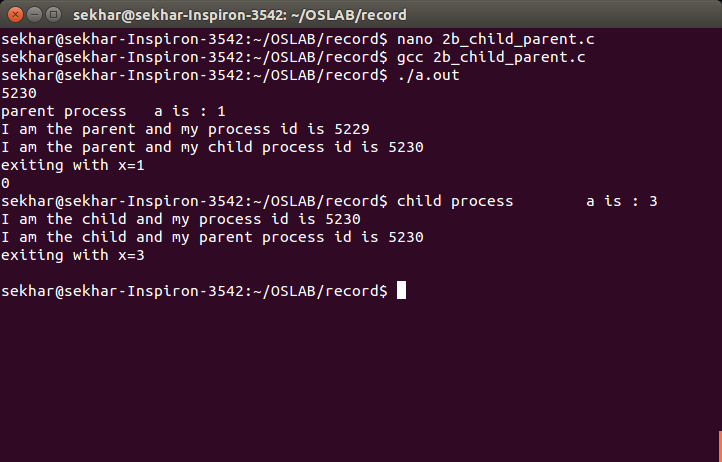
printf("I am the parent and my child process id is %d\n",pid);

}

printf("exiting with x=%d\n",a);

}

**Output:**



**7. Program to demonstrate getpid(),getppid() system calls without sleep**

**System calls required:**

**fork() system call:**

fork - create a child process

**Header file required:**

#include <[unistd.h](https://linux.die.net/include/unistd.h)>

**Syntax:**

pid\_t fork(void);

**Description:**

fork() creates a new process by duplicating the calling process. The new process, referred to as the *child*, is an exact duplicate of the calling process, referred to as the *parent*.

**Return Value:**

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created.

**getpid(),getppid() system calls:**

getpid, getppid - get process identification

**Header files required:**

#include <[**sys/types.h**](https://linux.die.net/include/sys/types.h)>  
#include <[**unistd.h**](https://linux.die.net/include/unistd.h)>

**Syntax:**

pid\_t getpid(void);  
pid\_t getppid(void);

**Description:**

getpid() returns the process ID of the calling process. (This is often used by routines that generate unique temporary filenames.)

getppid() returns the process ID of the parent of the calling process.

**Program:**

#include<stdio.h>

#include<unistd.h>

int main()

{

int a=2;

pid\_t pid;

pid=fork();

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

if(pid<0)

{

printf("Error");

}

else if(pid==0)

{

printf("child process");

printf("%d\n",++a);

printf("I am the child and my process id is %d\n",getpid());

printf("I am the child and my parent process id is %d\n",getppid());

}

else

{

printf("parent process");

printf("%d\n",--a);

printf("I am the parent and my process id is %d\n",getpid());

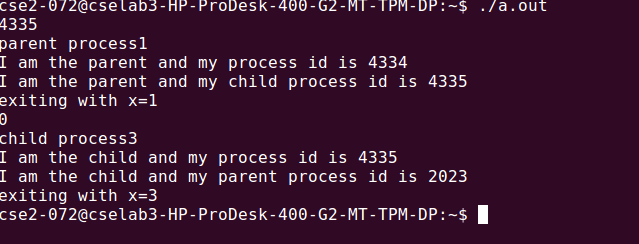
printf("I am the parent and my child process id is %d\n",pid);

}

printf("exiting with x=%d\n",a);

}

**Output:**



**8. Program to demonstrate execlp and wait system calls**

**System calls required:**

**Wait() system calls:**

wait, waitpid, waitid - wait for process to change state

**Header files required:**

#include<[**sys/types.h**](https://linux.die.net/include/sys/types.h)>  
#include <[**sys/wait.h**](https://linux.die.net/include/sys/wait.h)>

**Syntax:**

pid\_t wait(int \**status*);

pid\_t waitpid(pid\_t *pid*, int \**status*, int *options*);

int waitid(idtype\_t *idtype*, id\_t *id*, siginfo\_t \**infop*, int *options*);

**Description:**

The wait() system call suspends execution of the calling process until one of its children terminates. The call *wait(&status)* is equivalent to: waitpid(-1, &status, 0);

The waitpid() system call suspends execution of the calling process until a child specified by *pid* argument has changed state. By default, waitpid() waits only for terminated children.

**Return value:**

If *wait*() or *waitpid*() returns because the status of a child process is available, these functions shall return a value equal to the process ID of the child process for which *status* is reported. If *wait*() or *waitpid*() returns due to the delivery of a signal to the calling process, -1 shall be returned. If *waitpid*() was invoked with WNOHANG set in *options*, it has at least one child process specified by *pid* for which *status* is not available, and *status* is not available for any process specified by *pid*, 0 is returned. Otherwise, (pid\_t)-1 shall be returned.

**Execlp() system call:**

execl, execlp, execle, execv, execvp, execvpe - execute a file

**Header file required:**

#include <[**unistd.h**](https://linux.die.net/include/unistd.h)>

**Syntax:**

Int execl(const char \**path*, const char\**arg*,...);  
int execlp(const char \**file*, const char \**arg*, ...);  
int execle(const char \**path*, const char \**arg*,..., char \* const *envp*[]);  
int execv(const char \**path*, char \*const *argv*[]);  
int execvp(const char \**file*, char \*const *argv*[]);  
int execvpe(const char \**file*, char \*const *argv*[],char \*const *envp*[]);

**Description:**

The execlp(), execvp(), and execvpe() functions duplicate the actions of the shell in searching for an executable file if the specified filename does not contain a slash (/) character.

**Return Value**

The exec() functions only return if an error has occurred. The return value is -1.

**Program:**

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<string.h>

#include<errno.h>

int main(int argc,char \*argv[])

{

int pid,childpid,status;

pid=fork();

if(pid<0)

{

fprintf(stderr,"fork failed");

return 1;

}

else if(pid==0)

{

execlp("/bin/ls","ls",NULL);

\_exit(0);

}

else

{

wait(NULL);

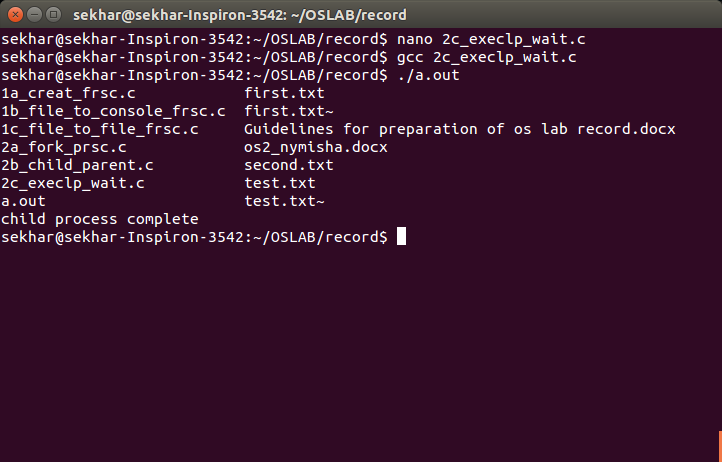
printf("child process complete\n");

}

return 0;

}

**Output:**



**9.Program to show the working of execv():**

The execv(), execvp(), and execvpe() functions provide an array of

pointers to null-terminated strings that represent the argument list

available to the new program. The first argument, by convention,

should point to the filename associated with the file being executed.

The array of pointers must be terminated by a NULL pointer.

**Program:-**

#include<unistd.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

#include<stdio.h>

#include<stdlib.h>

int main(int argc,char \*argv[])

{

int pid;

int status;

pid=fork();

if(pid<0)

exit(-1);

else if(pid==0)

{

execv("/bin/echo",argv);

exit(1);

}

else

{

int childid=wait(&status);

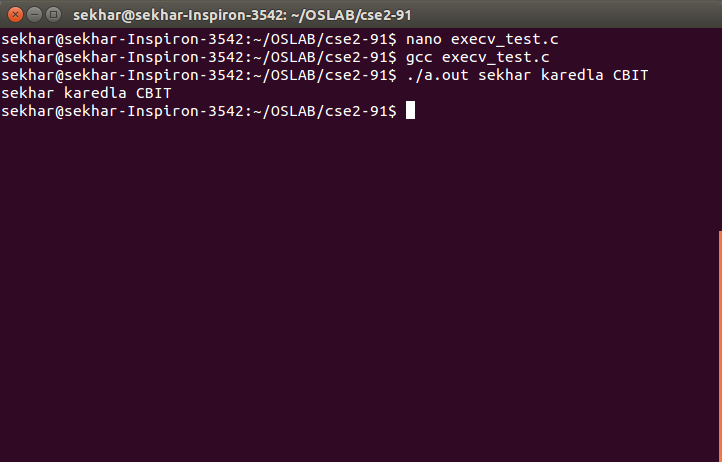
exit(2);

}

return 0;

}

**Output:**



# CPU SCHEDULING

**10. FCFS(First Come First Serve)**

**Algorithm:**

1. Start the process

2. Declare the array size

3. Get the number of processes to be inserted

4. Get the value

5. Start with the first process from it’s initial position let other process to be in queue

6. Calculate the total number of burst time

7.  Display the values

8. Stop the process

**Program:**

#include<iostream>

using namespace std;

int shortest(int at[],int done[],int n)

{

int k=0;

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

{

if(at[k]>at[i]&&done[i]!=1)

k=i;

else if(at[k]==at[i]&&done[i]!=1)

k=i;

}

return k;

}

int main()

{

int n;

cout<<"enter no of processes :";

cin>>n;

int \*bt=new int[n];

cout<<"enter burst times : ";

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

cin>>bt[i];

cout<<"enter arrival times : ";

int \*at=new int[n];

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

cin>>at[i];

int ct=0;

int \*wt=new int[n];

int \*tat=new int[n];

cout<<"PID\tAT\tBT\tCT\tWT\tTAT\n";

int p=0;

int \*done=new int[n];

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

done[i]=0;

while(p<n)

{

int i=shortest(at,done,n);

wt[i]=ct-at[i];

ct+=bt[i];

tat[i]=ct-at[i];

cout<<i<<"\t"<<at[i]<<"\t"<<bt[i]<<"\t"<<ct<<"\t"<<wt[i]<<"\t"<<tat[i]<<endl;p++;

at[i]=99999;

done[i]=1;

}

float awt,atat;awt=atat=0.0;

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

{

awt+=wt[i];

atat+=tat[i];

}

awt=float(awt/n);

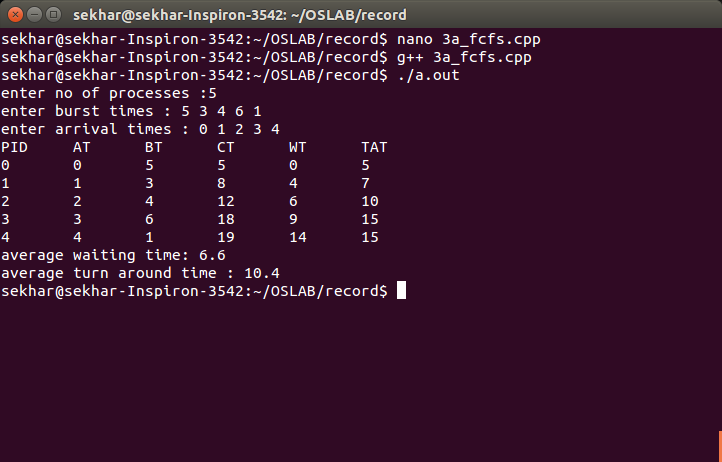
atat=float(atat/n);

cout<<"average waiting time: "<<awt<<endl;

cout<<"average turn around time : "<<atat<<endl;

}

**Output:**



**11. SJF(Shortest Job First)**

**Algorithm:**

1. Start the process

2. Declare the array size

3. Get the number of elements to be inserted

4. Select the process which have shortest burst will execute first

5. If two process have same burst length then FCFS scheduling algorithm used

6. Make the average waiting the length of next process

7.  Start with the first process from it’s selection as above and  let other process to be in

     queue

6. Calculate the total number of burst time

7.  Display the values

8. Stop the process

**Program:**

#include<iostream>

using namespace std;

int shortest(int bt[],int done[],int n,int ct,int at[])

{

int k=0;

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

{

if(bt[k]>bt[i]&&at[i]<=ct)

k=i;

else if(bt[k]==bt[i]&&i<k&&at[i]<=ct)

k=i;

}

return k;

}

int main()

{

int n;

cout<<"enter no of processes :";

cin>>n;

int \*bt=new int[n];

cout<<"enter burst times : ";

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

cin>>bt[i];

cout<<"enter arrival times : ";

int \*at=new int[n];

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

cin>>at[i];

int ct=0;

int \*wt=new int[n];

int \*tat=new int[n];

cout<<"PID\tAT\tBT\tCT\tWT\tTAT\n";

int p=0;

int \*done=new int[n];

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

done[i]=0;

while(p<n)

{

int k=shortest(bt,done,n,ct,at);

wt[k]=ct-at[k];

ct+=bt[k];

tat[k]=ct-at[k];

cout<<k<<"\t"<<at[k]<<"\t"<<bt[k]<<"\t"<<ct<<"\t"<<wt[k]<<"\t"<<tat[k]<<endl;

bt[k]=999999;

done[k]=1;p++;

}

float awt,atat;awt=atat=0.0;

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

{

awt+=wt[i];

atat+=tat[i];

}

awt=float(awt/n);

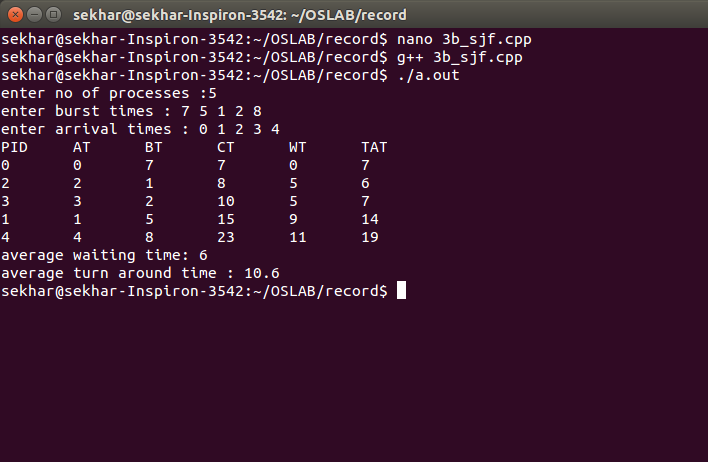
atat=float(atat/n);

cout<<"average waiting time: "<<awt<<endl;

cout<<"average turn around time : "<<atat<<endl;

}

**Output:**



**12. Round Robin**

**Algorithm:**

1. The queue structure in ready queue is of First In First Out (FIFO) type.

2. A fixed time is allotted to every process that arrives in the queue. This fixed time is known as time slice or time quantum.

3. The first process that arrives is selected and sent to the processor for execution. If it is not able to complete its execution within the time quantum provided, then an interrupt is generated using an automated timer.

4. The process is then stopped and is sent back at the end of the queue. However, the state is saved and context is thereby stored in memory. This helps the process to resume from the point where it was interrupted.

5. The scheduler selects another process from the ready queue and dispatches it to the processor for its execution. It is executed until the time Quantum does not exceed.

6. The same steps are repeated until all the process are finished.

The round robin algorithm is simple and the overhead in decision making is very low. It is the best scheduling algorithm for achieving better and evenly distributed response time.

**Program:**

#include<iostream>

using namespace std;

int main()

{

int n;

cout<<"enter no of processes :";

cin>>n;

int \*bt=new int[n];

cout<<"enter burst times : ";

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

cin>>bt[i];

cout<<"enter arrival times : ";

int \*at=new int[n];

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

cin>>at[i];

int ct=0;

int \*wt=new int[n];

int \*tat=new int[n];

int \*cta=new int[n];

int tq;

cout<<"enter time quantum ";

cin>>tq;

int \*temp=new int[n];

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

temp[i]=bt[i];

int flag=0;

while(flag<n)

{

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

{

if(at[i]<=ct&&bt[i]!=0)

{

if(bt[i]<tq)

{

ct+=bt[i];

bt[i]=0;

cta[i]=ct;

tat[i]=cta[i]-at[i];

wt[i]=tat[i]-temp[i];

flag++;

}

else

{

ct+=tq;

bt[i]-=tq;

if(bt[i]==0)

{

flag++;

cta[i]=ct;

tat[i]=cta[i]-at[i];

wt[i]=tat[i]-temp[i];

}

}

}

}

}

cout<<"\nPID\tAT\tBT\tCT\tTAT\tWT";

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

{

cout<<endl<<i<<"\t"<<at[i]<<"\t"<<temp[i]<<"\t"<<cta[i]<<"\t"<<tat[i]<<"\t"<<wt[i];

}

cout<<endl;

float awt,atat;awt=atat=0.0;

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

{

awt+=wt[i];

atat+=tat[i];

}

awt=float(awt/n);

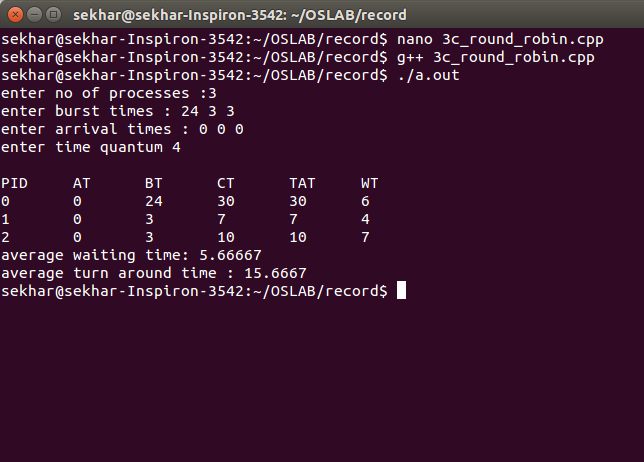
atat=float(atat/n);

cout<<"average waiting time: "<<awt<<endl;

cout<<"average turn around time : "<<atat<<endl;

}

**Output:**



**Bankers Algorithm**

**Deadlock Detection &** Deadlock **Avoidance**

**13. Resource Allocation Algorithm:**

P puts request vector

Pi Requesti

1. if Requesti<Needi then goto 2 else error

2. if Requesti<Available then goto 3 else wait

3. Available=Available-Requesti

Allocationi=Allocationi-Requesti

Needi=Needi-Requesti

4. Check if this new state is safe and if safe sequence exists.

**Program:**

#include<iostream>

#include<stdlib.h>

#include<stdbool.h>

using namespace std;

void display(int \*\*a,int n,int m)

{

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

{

cout<<endl;

for(int j=0;j<m;j++)

cout<<a[i][j]<<" ";

}

}

bool isSmall(int \*a,int \*b,int n)

{

int flag=0;

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

{

if(a[i]>b[i])

{

flag=1;

break;

}

}

if(flag==0)

return true;

else

return false;

}

int main()

{

cout<<"\nenter no of variety of resources : ";

int nr;

cin>>nr;

cout<<"enter the instances of resources : ";

int \*r=new int[nr];

for(int i=0;i<nr;i++)

cin>>r[i];

cout<<"enter the no of processes : ";

int p;

cin>>p;

cout<<"enter the allocation matrix : ";

int \*\*am=new int\*[p];

for(int i=0;i<p;i++)

am[i]=new int[nr];

for(int i=0;i<p;i++)

for(int j=0;j<nr;j++)

cin>>am[i][j];

cout<<"enter the max matrix : ";

int \*\*mm=new int\*[p];

for(int i=0;i<p;i++)

mm[i]=new int[nr];

for(int i=0;i<p;i++)

for(int j=0;j<nr;j++)

cin>>mm[i][j];

int \*\*nm=new int\*[p];

for(int i=0;i<p;i++)

nm[i]=new int[nr];

for(int i=0;i<p;i++)

for(int j=0;j<nr;j++)

nm[i][j]=mm[i][j]-am[i][j];

display(nm,p,nr);

int \*avai=new int[nr];

int \*temp=new int[nr];

cout<<"\nenter the process that is requesting :";

int rp;

cin>>rp;

cout<<"\nenter the request : ";

int \*request=new int[nr];

for(int i=0;i<nr;i++)

cin>>request[i];

if(isSmall(request,nm[rp],nr)&&isSmall(request,avai,nr))

{

for(int i=0;i<nr;i++)

{

avai[i]-=request[i];

nm[rp][i]-=request[i];

am[rp][i]+=request[i];

}

}

for(int i=0;i<nr;i++)

{

int sum=0;

for(int j=0;j<p;j++)

{

sum+=am[j][i];

}

temp[i]=sum;

}

int \*work=new int[nr];

for(int i=0;i<nr;i++)

{

avai[i]=r[i]-temp[i];

work[i]=avai[i];

}

bool \*finish=new bool[p];

for(int i=0;i<p;i++)

finish[i]=false;

int trap=0;

int count=0;

int disp=1;

//cout<<"\nsafe sequence :";

while(trap<p)

{

if(count>p\*p)

{cout<<"unsafe"<<endl;

break;

}

for(int i=0;i<p;i++)

{count++;

if(finish[i]==false&&isSmall(nm[i],work,nr))

{

if(disp==1)

cout<<"\nsafe sequence :";

disp++;

for(int j=0;j<nr;j++)

work[j]=work[j]+am[i][j];

trap++;

finish[i]=true;

cout<<i<<" ";

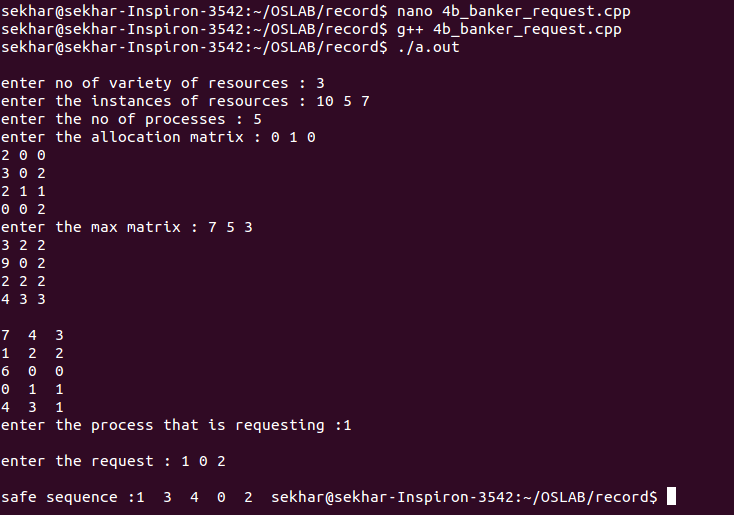
}

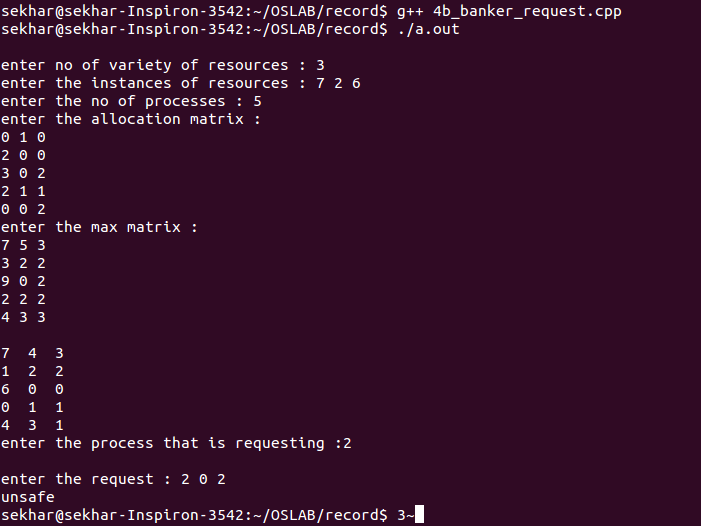
}

}

}

**Output:**





**14. Safety Algorithm:**

1. Work=Available

Finish=false

2. find p1 such that

Finish[i]=false

Needi<Work if no goto step 4

3. Work=Work+Allocation

Finish[i]=true

go to step 2

4. if Finish[i]=true for all i

Then system is safe

**Program:**

#include<iostream>

#include<stdlib.h>

#include<stdbool.h>

using namespace std;

void display(int \*\*a,int n,int m)

{

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

{

cout<<endl;

for(int j=0;j<m;j++)

cout<<a[i][j]<<" ";

}

}

bool isSmall(int \*a,int \*b,int n)

{

int flag=0;

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

{

if(a[i]>b[i])

{

flag=1;

break;

}

}

if(flag==0)

return true;

else

return false;

}

int main()

{

cout<<"\nenter no of variety of resources : ";

int nr;

cin>>nr;

cout<<"enter the instances of resources : ";

int \*r=new int[nr];

for(int i=0;i<nr;i++)

cin>>r[i];

cout<<"enter the no of processes : ";

int p;

cin>>p;

cout<<"enter the allocation matrix : ";

int \*\*am=new int\*[p];

for(int i=0;i<p;i++)

am[i]=new int[nr];

for(int i=0;i<p;i++)

for(int j=0;j<nr;j++)

cin>>am[i][j];

cout<<"enter the max matrix : ";

int \*\*mm=new int\*[p];

for(int i=0;i<p;i++)

mm[i]=new int[nr];

for(int i=0;i<p;i++)

for(int j=0;j<nr;j++)

cin>>mm[i][j];

int \*\*nm=new int\*[p];

for(int i=0;i<p;i++)

nm[i]=new int[nr];

for(int i=0;i<p;i++)

for(int j=0;j<nr;j++)

nm[i][j]=mm[i][j]-am[i][j];

display(nm,p,nr);

int \*avai=new int[nr];

int \*temp=new int[nr];

for(int i=0;i<nr;i++)

{

int sum=0;

for(int j=0;j<p;j++)

{

sum+=am[j][i];

}

temp[i]=sum;

}

int \*work=new int[nr];

for(int i=0;i<nr;i++)

{

avai[i]=r[i]-temp[i];

work[i]=avai[i];

}

bool \*finish=new bool[p];

for(int i=0;i<p;i++)

finish[i]=false;

int trap=0;

cout<<"\nsafe sequence :";

while(trap<p)

{

for(int i=0;i<p;i++)

{

if(finish[i]==false&&isSmall(nm[i],work,nr))

{

for(int j=0;j<nr;j++)

work[j]=work[j]+am[i][j];

trap++;

finish[i]=true;

cout<<i<<" ";

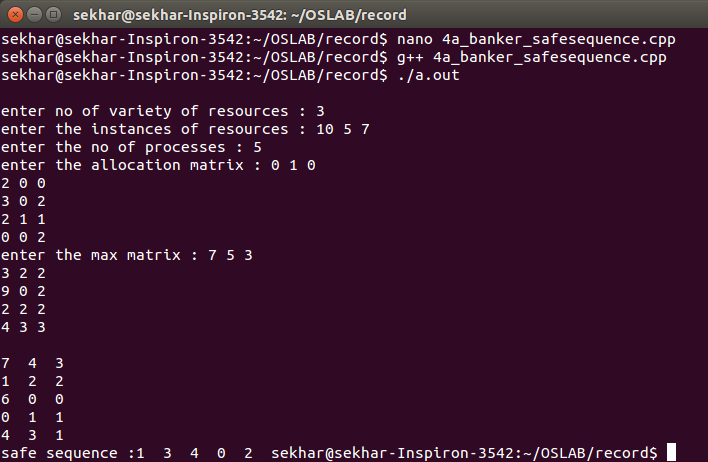
}

}

}

}

**Output:**



# PAGE REPLACEMENT ALGORITHMS

**15. Program to implement page replacement using FIFO algorithm**

**Algorithm**

1. Start the process

2. Declare the size with respect to page length

3. Check the need of replacement from the page to memory

4. Check the need of replacement from old page to new page in memory

5. Form a queue to hold all pages

6. Insert the page require memory into the queue

7. Check for bad replacement and page fault

8. Get the number of processes to be inserted

9. Display the values

10. Stop the process

**Program:**

#include<iostream>

#include<stdbool.h>

using namespace std;

void display(int \*s,int n,bool flag)

{

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

{

cout<<'\t'<<s[i];

}

if(flag)

cout<<"\tHIT"<<endl;

else

cout<<endl;

}

int main()

{

cout<<"\nenter no of sequences : ";

int n;

cin>>n;

int \*s=new int[n];

cout<<"\nenter the sequences:";

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

{

cin>>s[i];

}

int f;

cout<<"\nenter the frame size : ";

cin>>f;

int \*fr=new int[f];

for(int i=0;i<f;i++)

fr[i]=-1;

int size=0;int hit=0;int insert=0;int flag;

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

{

insert=0;flag=0;

for(int j=0;j<f;j++)

{

if(fr[j]==s[i])

{j=f;hit++;flag=1;display(fr,f,true);}

if(fr[j]==-1)

{fr[j]=s[i];insert=1;size++;j=f;display(fr,f,false);}

}

if(insert==1)

continue;

else if(flag==0)

{

fr[f-size]=s[i];

size--;

if(size==0)

size=f;

}

}

for(int i=0;i<f;i++)

cout<<'\t'<<s[i];

cout<<endl;

cout<<"\nnumber of faults:"<<(n-hit);

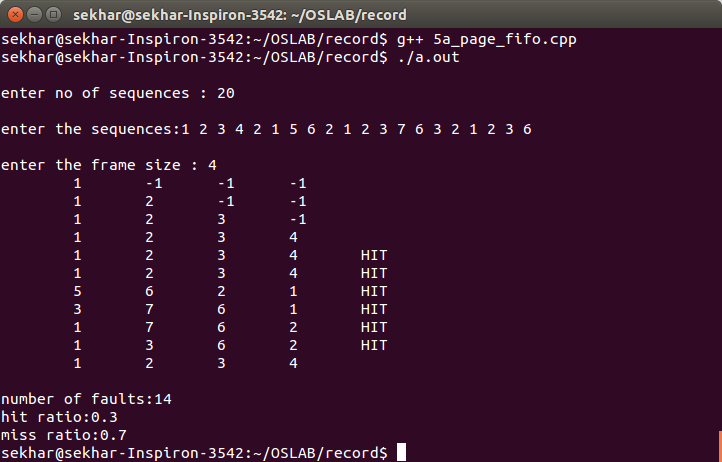
float temp=float(hit)/float(n);

cout<<"\nhit ratio:"<<temp;

cout<<"\nmiss ratio:"<<(1.0-temp)<<endl;

}

**Output:**



**16.Program to implement Page replacement algorithm using LRU algorithm.(Least Recently Used)**

**Program:-**

#include<iostream>

using namespace std;

int main()

{

int nop,nof,page[20],i,count=0;

cout<<"\n\tEnter the No. of Pages:";

cin>>nop; //Store the no of Pages

cout<<"\n\t Enter the Reference String:";

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

{

cout<<"\t";

cin>>page[i]; //Store the pages

}

cout<<"\n\t Enter the No of frames:-";

cin>>nof;

int frame[nof],fcount[nof];

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

{

frame[i]=-1; //Store the frames

fcount[i]=0; //Track when the page is last used

}

i=0;

while(i<nop)

{

int j=0,flag=0;

while(j<nof)

{

if(page[i]==frame[j]){ //Checking whether page already exist in frames or not

flag=1;

fcount[j]=i+1;

}

j++;

}

j=0;

cout<<"\n\t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

cout<<"\t"<<page[i]<<"-->";

if(flag==0)

{

int min=0,k=0;

while(k<nof-1)

{

if(fcount[min]>fcount[k+1]) //Calculating the page which is least recently used

min=k+1;

k++;

}

frame[min]=page[i]; //Replacing it

fcount[min]=i+1; //Increasing the time

count++; //counting Page Fault

while(j<nof)

{

cout<<"\t|"<<frame[j]<<"|";

j++;

}

}

i++;

}

cout<<"\n\t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

cout<<"\n\tPage Fault is:"<<count;

float hitratio=float(count)/nop;

float missratio=1.0-hitratio;

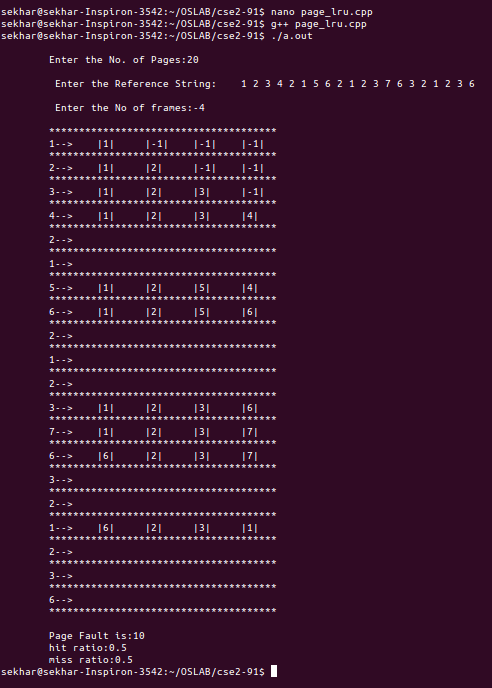
cout<<"\n\thit ratio:"<<hitratio;

cout<<"\n\tmiss ratio:"<<missratio<<endl;

return 0;

}

**Output:-**

****

# ECHO SERVER USING PIPES

**17. Echo server using pipes**

**System calls used:**

**Pipe():**

Pipe,pipe2 – create pipe

**Header files required:**

#include<unistd.h>

**Syntax:**

int pipe(int pipefd[2]);

#define \_GNU\_SOURCE

**Header files required:**

#include<fcntl.h>

#include<unitsd.h>

**Syntax:**

int pipe2(int pipefd[2],int flags);

**Description:**

Pipe() creates a pipe,a unidirectional data channel that can be used for interprocess communication. The array pipefd is used to reform a file descriptor referring to the ends of the pipe.

Pipefd[0] refers to the read end of the pipe.

Pipefd[1] refers to the write end of the pipe.

If flag is 0,then pipe() is same as pipe()

**Return value:**

On success, zero is returned, on error -1 is returned.

**Program:**

#include<stdlib.h>

#include<stdio.h>

#include<unistd.h>

void main(int argc,char \*argv[])

{

int fd1[2],fd2[2];

pipe(fd1);

pipe(fd2);

pid\_t pid;

char s[]="sekhar karedla";

char s1[100];

pid=fork();

char buf[100];

if(pid<0)

{

printf("error");

exit(-1);

}

else if(pid==0)

{

printf("enter the data : ");

scanf("%s",s1);

printf("child process writing\n");

close(fd1[0]);

write(fd1[1],s1,sizeof(s1));

wait(NULL);

close(fd2[1]);

read(fd2[0],buf,sizeof(buf));

printf("child process received %s\n",buf);

exit(0);

}

else

{

//buf[0]=' ';

close(fd1[1]);

//wait(NULL);

printf("parent process reading\n");

read(fd1[0],buf,sizeof(buf));

printf("confirming data at parent %s\n",buf);

close(fd2[0]);

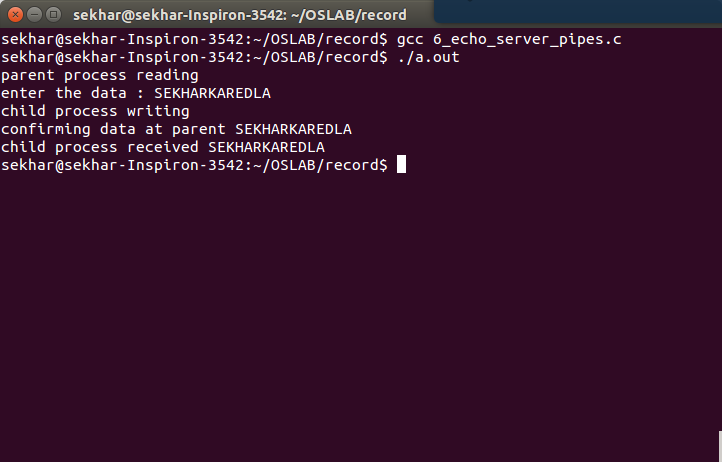
write(fd2[1],buf,sizeof(buf));

exit(0);

}

}

**Output:**



# Named Pipe

**18.Program of reader and writer in Named pipe.**

A named pipe works much like a regular pipe, but does have some noticeable differences.

* Named pipes exist as a device special file in the file system.
* Processes of different ancestry can share data through a named pipe.
* When all I/O is done by sharing processes, the named pipe remains in the file system for later use.

There are several ways of creating a named pipe. The first two can be done directly from the shell.

mknod MYFIFO p

mkfifo a=rw MYFIFO

The above two commands perform identical operations, with one exception. The mkfifo command provides a hook for altering the permissions on the FIFO file directly after creation. With mknod, a quick call to the chmod command will be necessary.

**Program(Writer):-**

#include<stdio.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<unistd.h>

#include<fcntl.h>

void main(int argc,char \*argv[])

{

char \*s="/tmp/testfifo";

int fd;

mkfifo(s,0666);

fd=open(s,O\_WRONLY);

write(fd,argv[1],sizeof(argv[1]));

close(fd);

unlink(s);

}

**Program(Reader):-**

#include<stdio.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<unistd.h>

#include<fcntl.h>

#define BUF 1024

void main(int argc,char \*argv[])

{

char r[BUF];

char \*s="/tmp/testfifo";

int fd;

fd=open(s,O\_RDONLY);

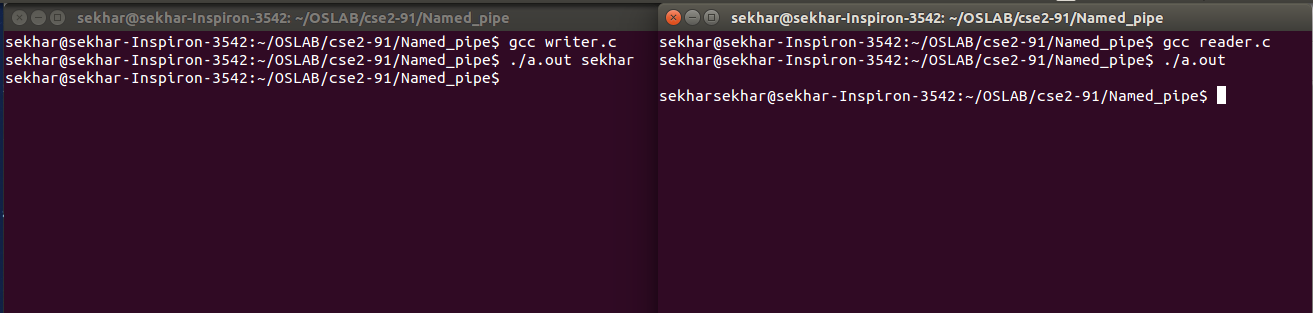
read(fd,r,BUF);

printf("\n%s",r);

close(fd);

}

**Output:-**

****

# Shell Script

**THEORY:-**

->WHAT IS SHELL SCRIPTING?

                  Shells are interactive i.e shell accept command from users and execute them. User can also write series of commands to execute a specific task in a text file and execute them instead of giving commands in command prompt.

-> DECLARE A VARIABLE ?

a=0

b=1

->SYNTAX:-

if syntax:-

if test <condition>

then

statements

elif test <condition>

then

statements

else

statements

fi

while syntax:-

while test <condition>

do

statements

done

comparison of integers:-

|  |  |
| --- | --- |
| *n1* -eq *n2* | Integer *n1* is equal to integer *n2*. |
| *n1* -ge *n2* | Integer *n1* is greater than or equal to integer *n2*. |
| *n1* -gt *n2* | Integer *n1* is greater than integer *n2*. |
| *n1* -le *n2* | Integer *n1* is less than integer *n2*. |
| *n1* -lt *n2* | Integer *n1* is less than or equal to integer *n2*. |
| *n1* -ne *n2* | Integer *n1* is not equal to integer *n2*. |
|  |  |

switch case syntax:-

case item in

part1)

statements

;;

part2)

statements

;;

part3)

statements

;;

part4)

statements

;;

esac

until loop syntax:-

until test <condition>

do

statements

done

for loop syntax:-

for((initialization;condition;modification))

do

statements

done

ex:-

for((i=0;i<10;i++))

do

echo "$i"

done

**19.Program to find the factorial.**

**Program:-**

read n

f=1

for((i=1;i<=n;i++))

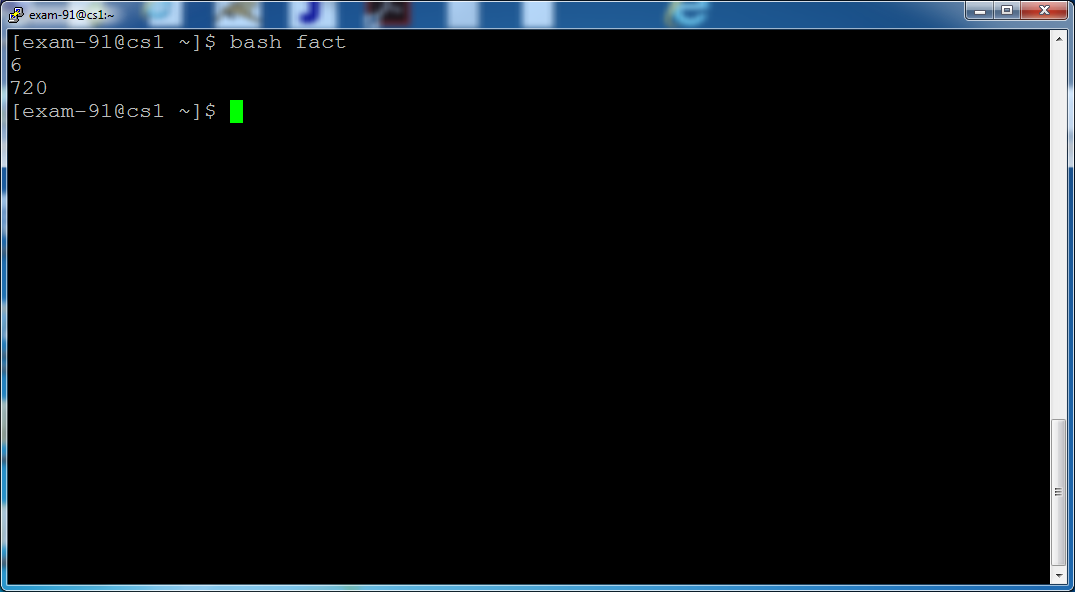
do

f=`expr $f \\* $i`

done

echo "$f"

**Output:-**

****

**20.Program to print the Fibonacci Series.**

**Program:-**

read n

a=0

b=1

c=0

for((i=1;i<=n;i++))

do

echo "$c"

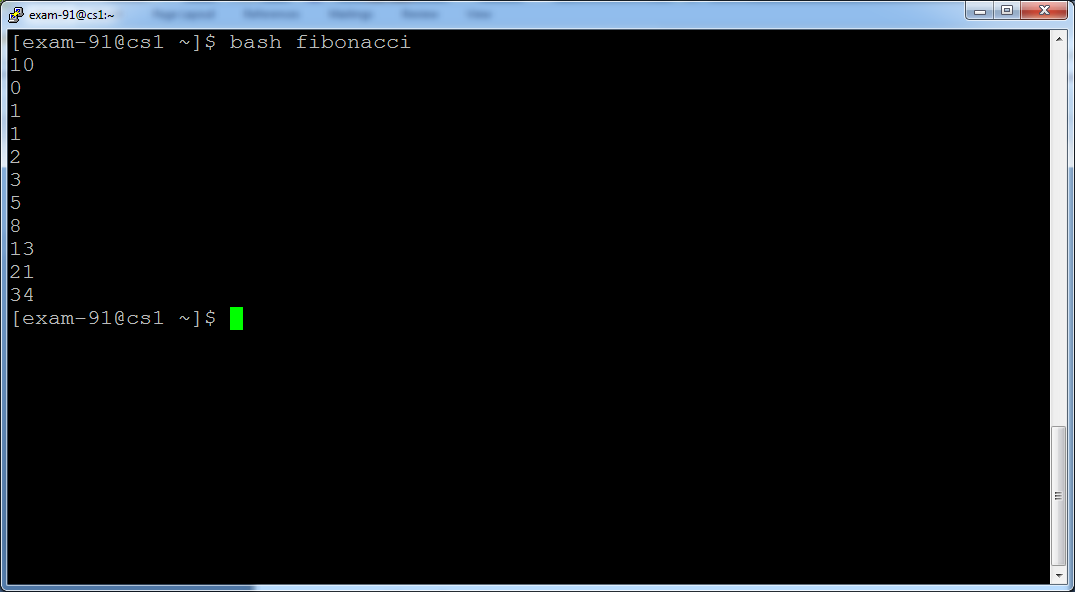
a=$b

b=$c

c=`expr $a + $b`

done

**OUTPUT:-**

****

**21.Program to find the Reverse of a number.**

**Program:-**

read n

s=0

r=0

while test $n -ne 0

do

r=`expr $n % 10`

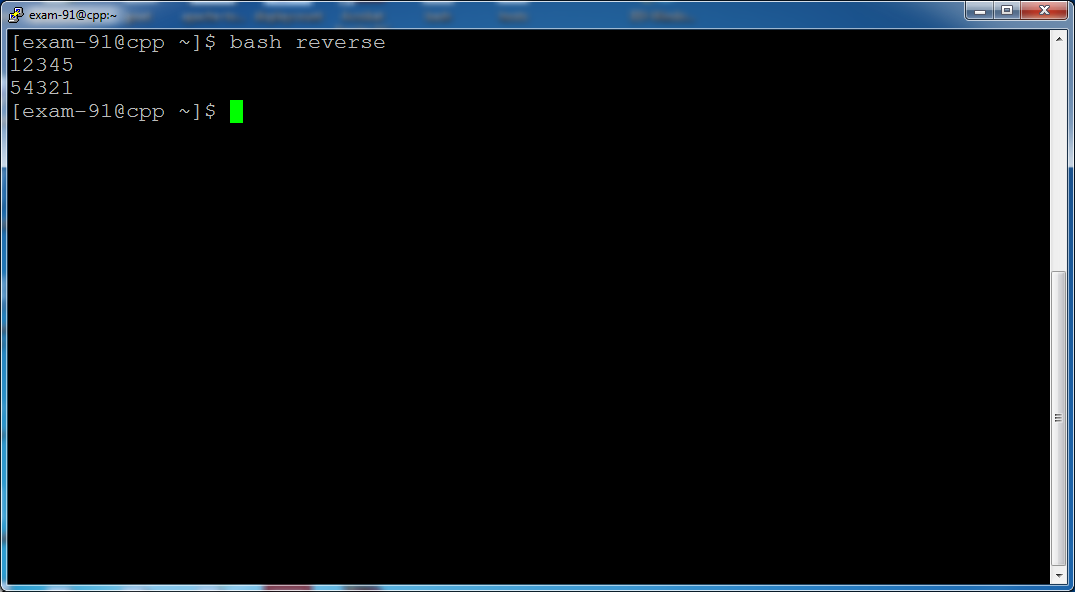
s=`expr $s \\* 10`

s=`expr $s + $r`

n=`expr $n / 10`

done

echo "$s"

**OUTPUT:-**

**22.Program to simulate simple Calculator.**

**Program:-**

cho -e " press 1 to add \n press 2 to subtract \n press 3 to multiply \n press 4 to divide"

read ch

echo -e "\n enter two numbers :\n"

read a

read b

if test $ch -eq 1

then

expr $a + $b

elif test $ch -eq 2

then

expr $a - $b

elif test $ch -eq 3

then

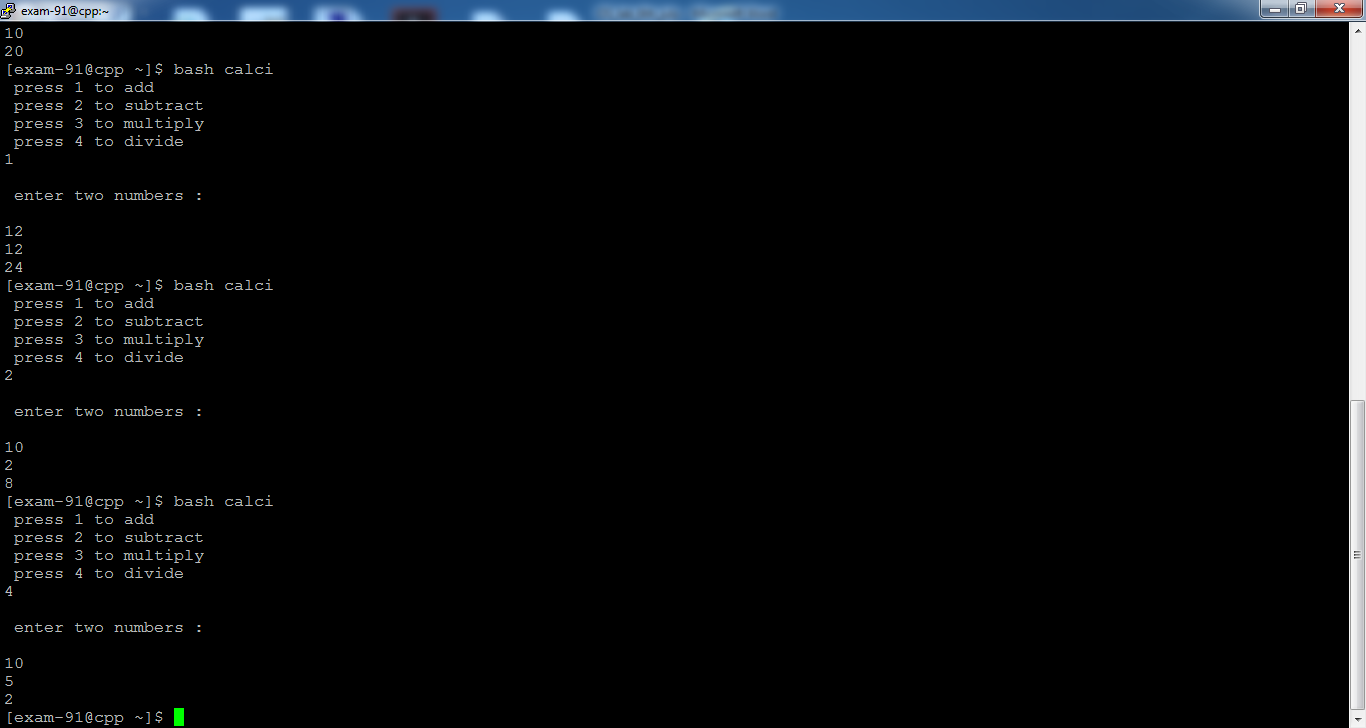
expr $a \\* $b

else

expr $a / $b

fi

**OUTPUT:-**



**23.Program to find whether a number is armstrong or not.**

**Program:-**

read n

k=$n

s=0

r=0

ar=0

while test $n -ne 0

do

r=`expr $n % 10`

s=`expr $r \\* $r`

s=`expr $s \\* $r`

ar=`expr $ar + $s`

n=`expr $n / 10`

done

if test $ar -eq $k

then

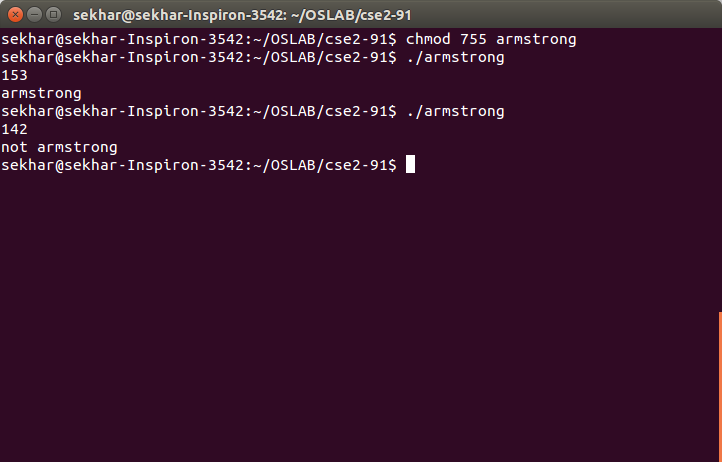
echo "armstrong"

else

echo "not armstrong"

fi

**Output:-**

****

**24.Program to find whether a number is Prime or not.**

read n

flag=1

for((i=2;i<n;i++))

do

m=`expr $n % $i`

if test $m -eq 0

then

flag=0

#echo "$flag"

fi

done

if test $flag -eq 0

then

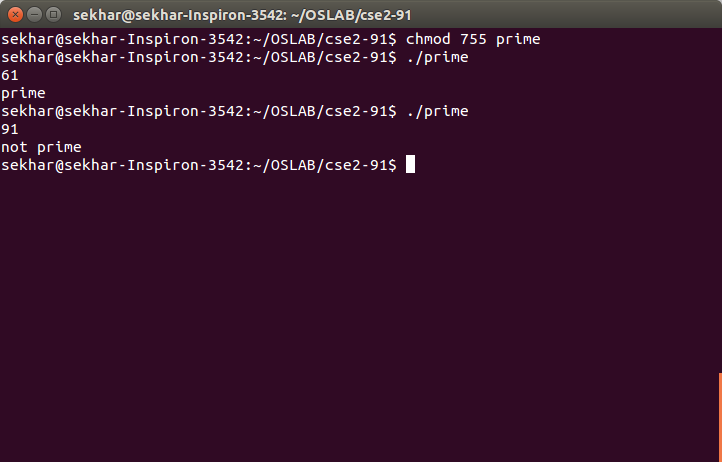
echo "not prime"

else

echo "prime"

fi

**Output:-**

****

# Threads

POSIX.1 specifies a set of interfaces (functions, header files) for

threaded programming commonly known as POSIX threads, or Pthreads. A

single process can contain multiple threads, all of which are

executing the same program. These threads share the same global

memory (data and heap segments), but each thread has its own stack

(automatic variables).

**25.Program to demonstrate the use of threads.**

#include<stdio.h>

#include<string.h>

#include <pthread.h>

void \*myfunc (void \*mydata);

pthread\_t theThread;

main()

{

pthread\_t thread1;

char \*my1="first thread";

int ret1;

ret1=pthread\_create(&thread1,NULL,&myfunc,(void\*)my1);

printf("main function after pthread\n");

pthread\_join(thread1,NULL);

printf("first thread ret1=%d\n",ret1);

if(pthread\_equal(thread1,theThread))

{

printf("success\n");

}

}

void \*myfunc (void \*mydata)

{

char \*msg;

msg=(char\*)mydata;

int i;

for(i=0;i<10;i++)

{

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

sleep(1);

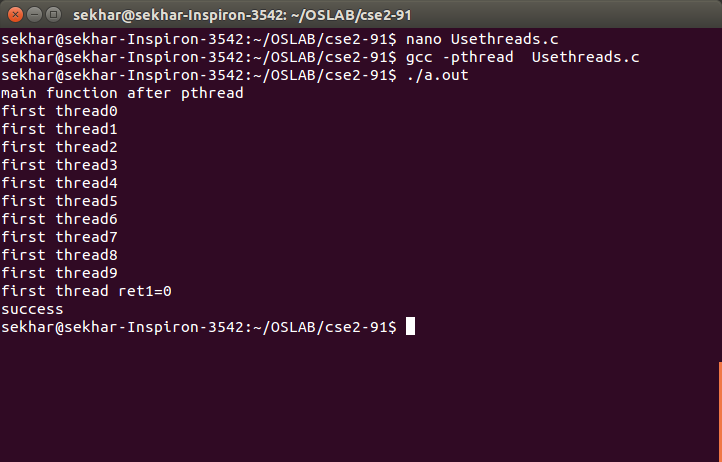
theThread=pthread\_self();

}

pthread\_exit(NULL);

}

**Output:-**

****

**26.Another Program to demonstrate the use of threads.**

**Program:-**

#include<stdio.h>

#include<string.h>

#include <pthread.h>

void \*myfunc (void \*mydata);

main()

{

pthread\_t thread1,thread2;

char \*my1="first thread";

char \*my2="second thread";

int ret1,ret2;

ret1=pthread\_create(&thread1,NULL,&myfunc,(void\*)my1);

ret2=pthread\_create(&thread2,NULL,&myfunc,(void\*)my2);

printf("main function after pthread\n");

pthread\_join(thread1,NULL);

pthread\_join(thread2,NULL);

printf("first thread ret1=%d\n",ret1);

printf("second thread ret1=%d\n",ret2);

}

void \*myfunc (void \*mydata)

{

char \*msg;

msg=(char\*)mydata;

int i;

for(i=0;i<10;i++)

{

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

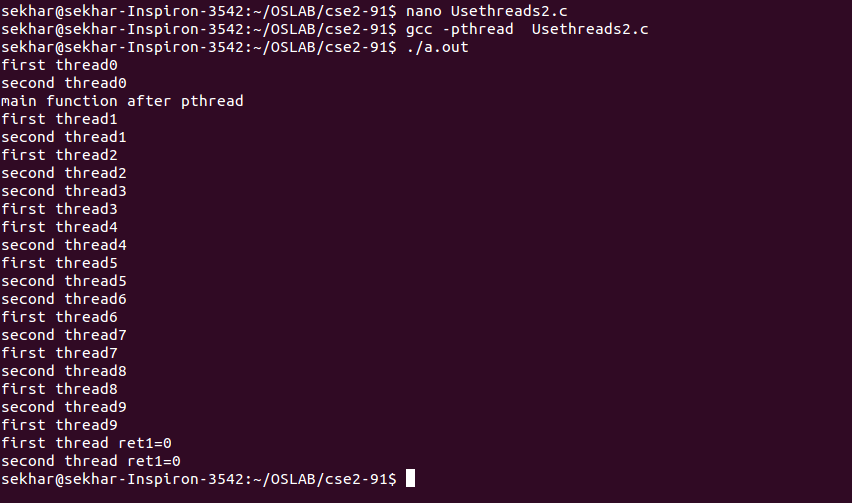
sleep(1);

}

return NULL;

}

**Output:-**

****

**27.Program to demonstrate the stat function.**

Header Files:

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

struct stat {

dev\_t st\_dev; /\* ID of device containing file \*/

ino\_t st\_ino; /\* inode number \*/

mode\_t st\_mode; /\* file type and mode \*/

nlink\_t st\_nlink; /\* number of hard links \*/

uid\_t st\_uid; /\* user ID of owner \*/

gid\_t st\_gid; /\* group ID of owner \*/

dev\_t st\_rdev; /\* device ID (if special file) \*/

off\_t st\_size; /\* total size, in bytes \*/

blksize\_t st\_blksize; /\* blocksize for filesystem I/O \*/

blkcnt\_t st\_blocks; /\* number of 512B blocks allocated \*/

/\* Since Linux 2.6, the kernel supports nanosecond

precision for the following timestamp fields.

For the details before Linux 2.6, see NOTES. \*/

struct timespec st\_atim; /\* time of last access \*/

struct timespec st\_mtim; /\* time of last modification \*/

struct timespec st\_ctim; /\* time of last status change \*/

#define st\_atime st\_atim.tv\_sec /\* Backward compatibility \*/

#define st\_mtime st\_mtim.tv\_sec

#define st\_ctime st\_ctim.tv\_sec

};

RETURN VALUE

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

set appropriately.

**Program:-**

#include<stdio.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<unistd.h>

#include<string.h>

#include<time.h>

main()

{

struct stat buf;

int i;

i=stat("file.txt",&buf);

printf("st\_dev=%ld\n",buf.st\_dev);

printf("st\_ino=%ld\n",buf.st\_ino);

printf("st\_mode=%o\n",buf.st\_mode);

printf("st\_nlink=%ld\n",buf.st\_nlink);

printf("st\_atime=%s\n",ctime(&buf.st\_atime));

printf("st\_size=%ld\n",buf.st\_size);

printf("st\_uid=%d\n",buf.st\_uid);

printf("st\_gid=%d\n",buf.st\_gid);

printf("st\_mtime=%s\n",ctime(&buf.st\_mtime));

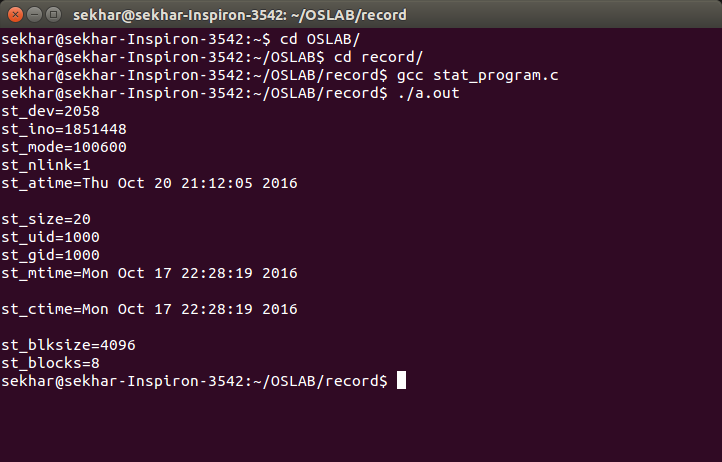
printf("st\_ctime=%s\n",ctime(&buf.st\_ctime));

printf("st\_blksize=%ld\n",buf.st\_blksize);

printf("st\_blocks=%ld\n",buf.st\_blocks);

}

**Output:-**

****

# CLASSICAL SYNCHRONIZATION PROBLEMS

**29.To use semaphores to solve Reader Writer Problem.**

Description:-

In computer science, the readers-writers problems are examples of a common computing problem in concurrency. There are at least three variations of the problems, which deal with situations in which many threads try to access the same shared resource at one time. Some threads may read and some may write, with the constraint that no process may access the share for either reading or writing, while another process is in the act of writing to it. (In particular, it is allowed for two or more readers to access the share at the same time.) A readers-writer lock is a data structure that solves one or more of the readers-writers problems.

**Program:-**

#include<stdio.h>

#include<semaphore.h>

sem\_t mutex;

sem\_t db;

int readercount=0;

pthread\_t reader1,reader2,writer1,writer2;

void \*reader(void \*);

void \*writer(void \*);

void main()

{

sem\_init(&mutex,0,1);

sem\_init(&db,0,1);

while(1)

{

pthread\_create(&reader1,NULL,reader,"1");

pthread\_create(&reader2,NULL,reader,"2");

pthread\_create(&writer1,NULL,writer,"1");

pthread\_create(&writer2,NULL,writer,"2");

}

}

void \*reader(void \*p)

{

int temp;

sem\_getvalue(&mutex,&temp);

printf("in reader : %s prevoius value %d\n",(char \*)p,temp);

sem\_wait(&mutex);

sem\_getvalue(&mutex,&temp);

printf("Mutex acquired by reader %d\n",temp);

readercount++;

if(readercount==1) sem\_wait(&db);

sem\_post(&mutex);

sem\_getvalue(&mutex,&temp);

printf("Mutex returned by reader %d\n",temp);

printf("Reader %s is Reading\n",(char \*)p);

sleep(3);

sem\_wait(&mutex);

printf("Reader %s Completed Reading\n",(char \*)p);

readercount--;

if(readercount==0) sem\_post(&db);

sem\_post(&mutex);

}

void \*writer(void \*p)

{

printf("Writer : %s is Waiting\n",(char \*)p);

sem\_wait(&db);

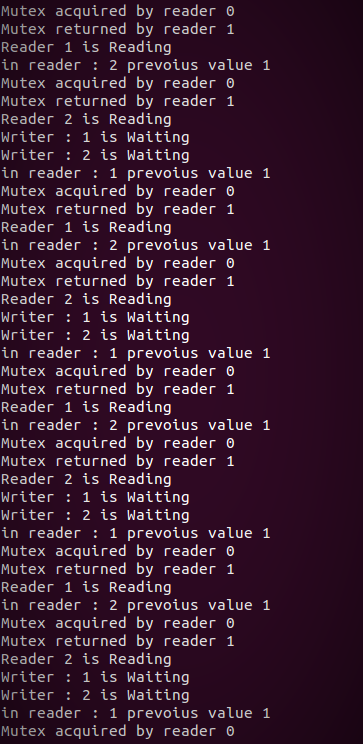
printf("Writer %s is writing\n",(char \*)p);

sem\_post(&db);

sleep(2);

}

**Output:-**

****

**30.To demonstrate producer consumer problem using semaphores.**

Description:

In computing, the producer–consumer problem[1][2] (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. 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.

The solution for the producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer. The solution can be reached by means of inter-process communication, typically using semaphores. An inadequate solution could result in a deadlock where both processes are waiting to be awakened. The problem can also be generalized to have multiple producers and consumers.-

**Program:-**

#include<stdio.h>

#include<pthread.h>

#include<semaphore.h>

int buf[5],f,r;

sem\_t mutex,full,empty;

void \*produce(void \*arg)

{

int i;

for(i=0;i<10;i++)

{int temp;

sem\_wait(&empty);

sem\_wait(&mutex);

printf("produced item is %d\n",i);

buf[(++r)%5]=i;

sleep(1);

sem\_post(&mutex);

sem\_post(&full);

sem\_getvalue(&full,&temp);

printf("producer : full %u\n",temp);

}

}

void \*consume(void \*arg)

{

int item,i;

for(i=0;i<10;i++)

{int temp;

sem\_wait(&full);

sem\_getvalue(&full,&temp);

printf("consumer : full %u\n",temp);

sem\_wait(&mutex);

item=buf[(++f)%5];

printf("consumed item is %d\n",item);

sleep(1);

sem\_post(&mutex);

sem\_post(&empty);

}

}

void main()

{

pthread\_t tid1,tid2;

sem\_init(&mutex,0,1);

sem\_init(&full,0,1);

sem\_init(&empty,0,5);

pthread\_create(&tid1,NULL,produce,NULL);

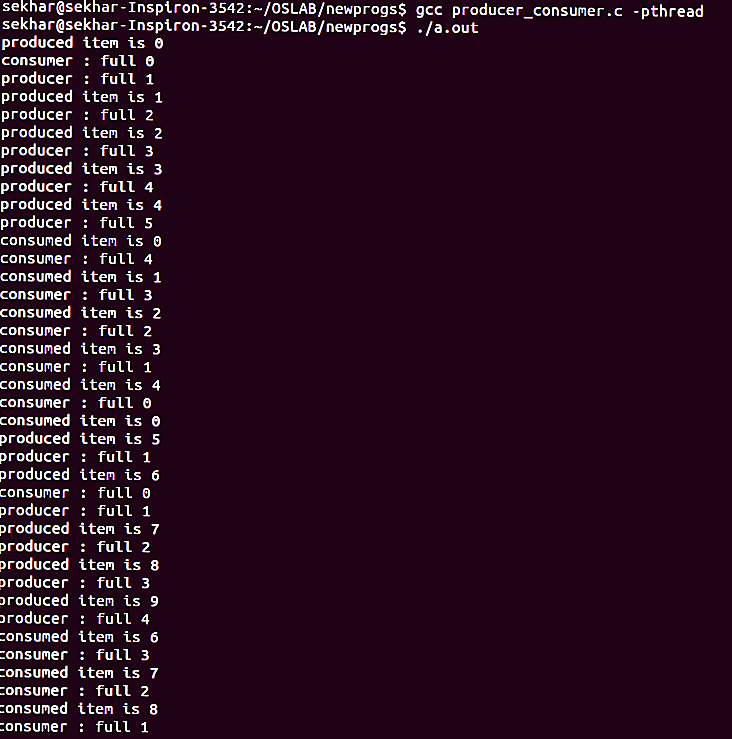
pthread\_create(&tid2,NULL,consume,NULL);

pthread\_join(tid1,NULL);

pthread\_join(tid2,NULL);

}

**Output:-**

****

**31.To demonstrate Dining Philosophers Problem.**

Description:-

The Dining Philosophers problems is a classic synchronization problem (Introduced by E. W. Dijkstra) introducing semaphores as a conceptual synchronization mechanism. The problem is discussed in just about every operating systems textbook.

Dining Philosophers. There is a dining room containing a circular table with five chairs. At each chair is a plate, and between each plate is a single chopstick. In the middle of the table is a bowl of spaghetti. Near the room are five philosophers who spend most of their time thinking, but who occasionally get hungry and need to eat so they can think some more.

In order to eat, a philosopher must sit at the table, pick up the two chopsticks to the left and right of a plate, then serve and eat the spaghetti on the plate.

Thus, each philosopher is represented by the following pseudocode:

process P[i]

while true do

{ THINK;

PICKUP(CHOPSTICK[i], CHOPSTICK[i+1 mod 5]);

EAT;

PUTDOWN(CHOPSTICK[i], CHOPSTICK[i+1 mod 5])

}

A philosopher may THINK indefinately. Every philosopher who EATs will eventually finish. Philosophers may PICKUP and PUTDOWN their chopsticks in either order, or non-deterministically, but these are atomic actions, and, of course, two philosophers cannot use a single CHOPSTICK at the same time.

The problem is to design a protocol to satisfy the liveness condition: any philosopher who tries to EAT, eventually does.

**Program:-**

#include<stdio.h>

#include<semaphore.h>

#include<pthread.h>

#define N 5

#define THINKING 0

#define HUNGRY 1

#define EATING 2

#define LEFT (ph\_num+4)%N

#define RIGHT (ph\_num+1)%N

sem\_t mutex;

sem\_t S[N];

void \* philospher(void \*num);

void take\_fork(int);

void put\_fork(int);

void test(int);

int state[N];

int phil\_num[N]={0,1,2,3,4};

int main()

{

int i;

pthread\_t thread\_id[N];

sem\_init(&mutex,0,1);

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

sem\_init(&S[i],0,0);

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

{

pthread\_create(&thread\_id[i],NULL,philospher,&phil\_num[i]);

printf("Philosopher %d is thinking\n",i+1);

}

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

pthread\_join(thread\_id[i],NULL);

}

void \*philospher(void \*num)

{

while(1)

{

int \*i = num;

sleep(1);

take\_fork(\*i);

sleep(0);

put\_fork(\*i);

}

}

void take\_fork(int ph\_num)

{

sem\_wait(&mutex);

state[ph\_num] = HUNGRY;

printf("Philosopher %d is Hungry\n",ph\_num+1);

test(ph\_num);

sem\_post(&mutex);

sem\_wait(&S[ph\_num]);

sleep(1);

}

void test(int ph\_num)

{

if (state[ph\_num] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)

{

state[ph\_num] = EATING;

sleep(2);

printf("Philosopher %d takes fork %d and %d\n",ph\_num+1,LEFT+1,ph\_num+1);

printf("Philosopher %d is Eating\n",ph\_num+1);

sem\_post(&S[ph\_num]);

}

}

void put\_fork(int ph\_num)

{

sem\_wait(&mutex);

state[ph\_num] = THINKING;

printf("Philosopher %d putting fork %d and %d down\n",ph\_num+1,LEFT+1,ph\_num+1);

printf("Philosopher %d is thinking\n",ph\_num+1);

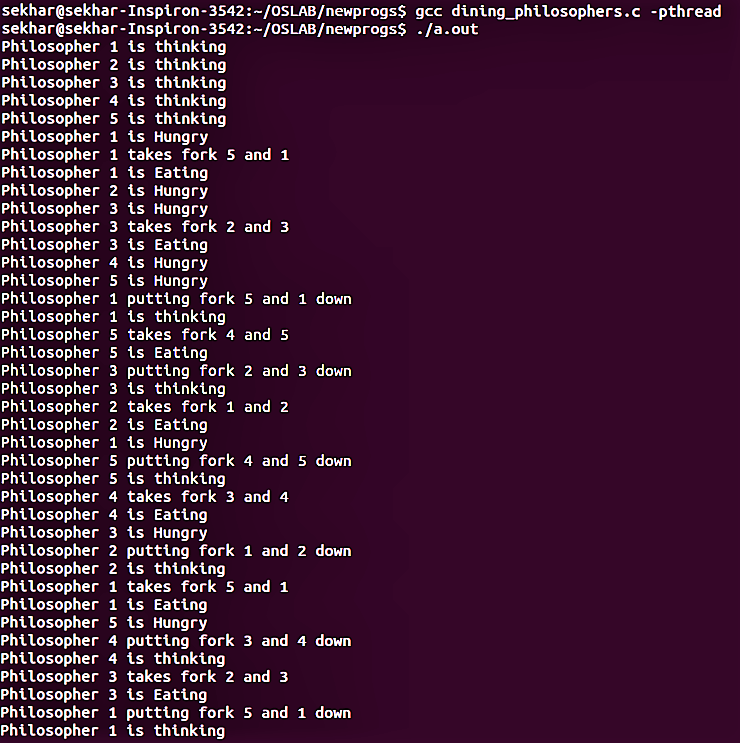
test(LEFT);

test(RIGHT);

sem\_post(&mutex);

}

**Output:-**



**32.Program to demonstrate Shared Memory.**

Description:-

Shared Memory is a type of IPC where the two processes share same memory chunk and use it for IPC. One process writes into that memory and other reads it.

**Program(SERVER):-**

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <stdio.h>

#include<stdlib.h>

#define SHMSZ 27

void main()

{

char c;

int shmid;

key\_t key;

char \*shm, \*s;

/\*

\* We'll name our shared memory segment

\* "5678".

\*/

key = 5678;

/\*

\* Create the segment.

\*/

if ((shmid = shmget(key, SHMSZ, IPC\_CREAT | 0666)) < 0) {

perror("shmget");

exit(1);

}

/\*

\* Now we attach the segment to our data space.

\*/

if ((shm = shmat(shmid, NULL, 0)) == (char \*) -1) {

perror("shmat");

exit(1);

}

/\*

\* Now put some things into the memory for the

\* other process to read.

\*/

s = shm;

for (c = 'a'; c <= 'z'; c++)

\*s++ = c;

\*s = '\0';

/\*

\* Finally, we wait until the other process

\* changes the first character of our memory

\* to '\*', indicating that it has read what

\* we put there.

\*/

while (\*shm != '\*')

sleep(1);

exit(0);

}

**Program(Client):-**

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <stdio.h>

#include<stdlib.h>

#define SHMSZ 27

void main()

{

int shmid;

key\_t key;

char \*shm, \*s;

/\*

\* We need to get the segment named

\* "5678", created by the server.

\*/

key = 5678;

/\*

\* Locate the segment.

\*/

if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {

perror("shmget");

exit(1);

}

/\*

\* Now we attach the segment to our data space.

\*/

if ((shm = shmat(shmid, NULL, 0)) == (char \*) -1) {

perror("shmat");

exit(1);

}

/\*

\* Now read what the server put in the memory.

\*/

for (s = shm; \*s != '\0'; s++)

putchar(\*s);

putchar('\n');

/\*

\* Finally, change the first character of the

\* segment to '\*', indicating we have read

\* the segment.

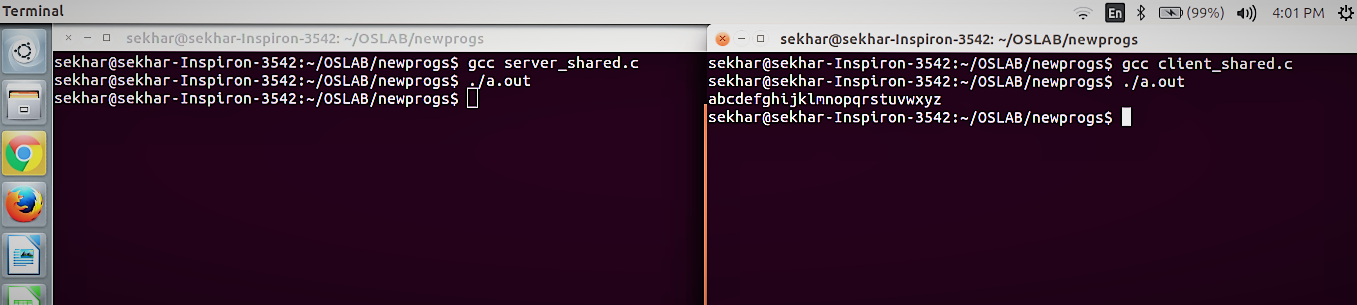
\*/

\*shm = '\*';

exit(0);

}

**Output:-**

****

**33.Program to demonstrate Message Queues.**

Description:-

A message queue provide an asynchronous communications protocol, a system that puts a message onto a message queue does not require an immediate response to continue processing. Email is probably the best example of asynchronous messaging. When an email is sent can the sender continue processing other things without an immediate response from the receiver. This way of handling messages decouple the producer from the consumer. The producer and the consumer of the message do not need to interact with the message queue at the same time.

**Program(Sender):-**

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define MAXSIZE 128

struct msgbuf

{

long mtype;

char mtext[MAXSIZE];

};

main()

{

int msqid;

int msgflg = IPC\_CREAT | 0666;

key\_t key;

struct msgbuf sbuf;

size\_t buflen;

key = 1234;

if ((msqid = msgget(key, msgflg )) < 0) //Get the message queue ID for the given key

{perror("msgget");exit(1);}

//Message Type

sbuf.mtype = 1;

printf("Enter a message to add to message queue : ");

scanf("%[^\n]",sbuf.mtext);

getchar();

buflen = strlen(sbuf.mtext) + 1 ;

if (msgsnd(msqid, &sbuf, buflen, IPC\_NOWAIT) < 0)

{

printf ("%d, %ld, %s, %d\n", msqid, sbuf.mtype, sbuf.mtext,(int)buflen);

perror("msgsnd");

exit(1);

}

else

printf("Message Sent\n");

exit(0);

}

**Program(Receiver):-**

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <stdio.h>

#include <stdlib.h>

#define MAXSIZE 128

struct msgbuf

{

long mtype;

char mtext[MAXSIZE];

};

main()

{

int msqid;

key\_t key;

struct msgbuf rcvbuffer;

key = 1234;

if ((msqid = msgget(key, 0666)) < 0)

{perror("msgget()");exit(1);}

//Receive an answer of message type 1.

if (msgrcv(msqid, &rcvbuffer, MAXSIZE, 1, 0) < 0)

{perror("msgrcv");exit(1);}

printf("%s\n", rcvbuffer.mtext);

exit(0);

}

**Output:-**

