OPERATING

SYSTEMS

RECORD

Nymisha Kandi

CSE2-160114733072

Index

|  |  |  |  |
| --- | --- | --- | --- |
| S.No | Program | Page no. | Date |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14. | **File related system calls**  Program to create a file  Program to write contents from file to console  Program to write contents from file to another  **Process related system calls**  Program to demonstrate fork system call  Program to demonstrate getpid(),getppid() system calls  Program to demonstrate getpid(),getppid() system calls without sleep  Program to demonstrate execlp(),wait system calls  **CPU scheduling algorithms**  FCFS(First Come First Serve)  SJF(Shortest Job First)  Round Robbin  **Deadlock detection and avoidance**  Bankers algorithm  **Page replacement algorithms**  FIFO LRU **Echo server**  Using pipes | 5  9  15  22  26  29  32  37  40  43  48  56  59  64 | July19  July19  July26  July26  Aug2  Aug2  Aug9  Aug16  Aug23  Aug30  Sep6  Sep13  Sep20  Aug16 |

File

Related

System Calls

1. Program to create a File

**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");

else

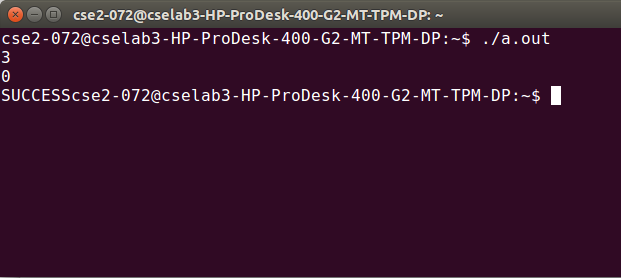
printf("SUCCESS");

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

close(fd1);

}

**Output:**



2. 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[10];

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

if(fd==-1)

{

exit(-1);

}

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

{

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

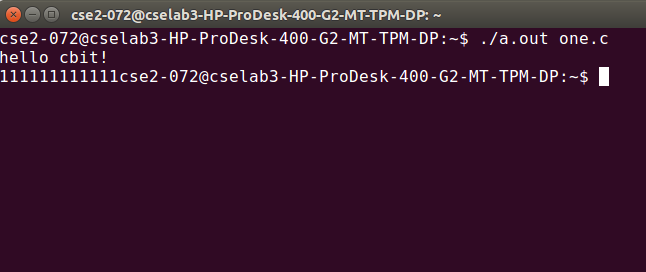
write(1,buffer,n\_char);

}

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,n;

char ch[11];

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((ch==read(fd1,ch,10))!=0)

write(fd2,ch,n);

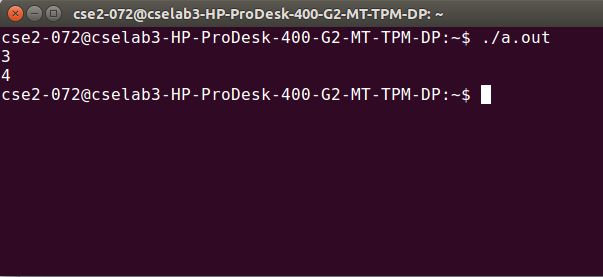
close(fd1);

close(fd2);

return 0;

}

**Output:**



Process

Related

System Calls

4. 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");

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

}

else

{

printf("parent process");

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

}

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

}

**Output:**



5. 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");

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());

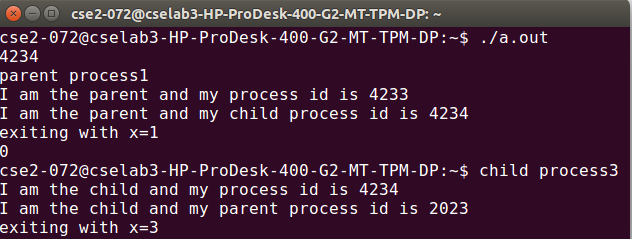
printf("I am the parent and my child process id is %d\n",pid);

}

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

}

Output:



**6. 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());

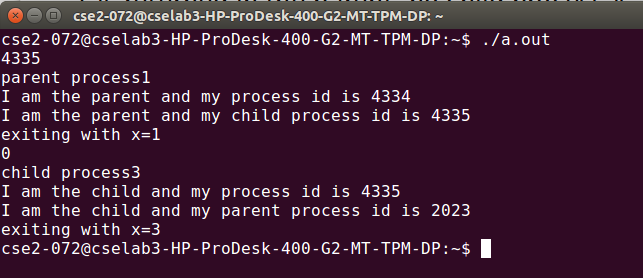
printf("I am the parent and my child process id is %d\n",pid);

}

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

}

Output:



7.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);

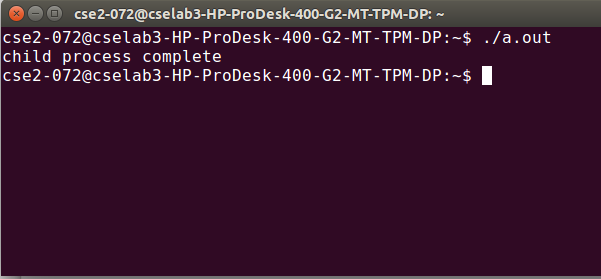
printf("child process complete\n");

}

return 0;

}

**Output:**



CPU

Scheduling

8.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<stdio.h>

int main()

{

int n,i,sum,pt,exe;

printf("Enter the number of processes: ");

scanf("%d",&n);

int AT[n],BT[n],CT[n],TAT[n],WT[n],temp[n];

float AWT,ATAT,wsum,tsum;

printf("Enter the burst times of the processes: ");

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

scanf("%d",&BT[i]);

printf("Enter the arrival times of the processes: ");

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

scanf("%d",&AT[i]);

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

temp[i]=0;

pt=0;

exe=0;

sum=0;

while(exe<n)

{

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

{

if(pt>=AT[i]&&temp[i]!=-1)

temp[i]=1;

}

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

{

if(temp[i]==1)

{

sum=sum+BT[i];

CT[i]=sum;

TAT[i]=CT[i]-AT[i];

WT[i]=TAT[i]-BT[i];

pt=CT[i];

exe++;

temp[i]=-1;

}

}

}

wsum=0.0;

tsum=0.0;

printf("The output is: ");

printf("\nPID AT BT CT TAT WT");

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

{

printf("\n%d %d %d %d %d %d",i,AT[i],BT[i],CT[i],TAT[i],WT[i]);

wsum=wsum+(float)WT[i];

tsum=tsum+(float)TAT[i];

}

AWT=(float)(wsum/n);

ATAT=(float)(tsum/n);

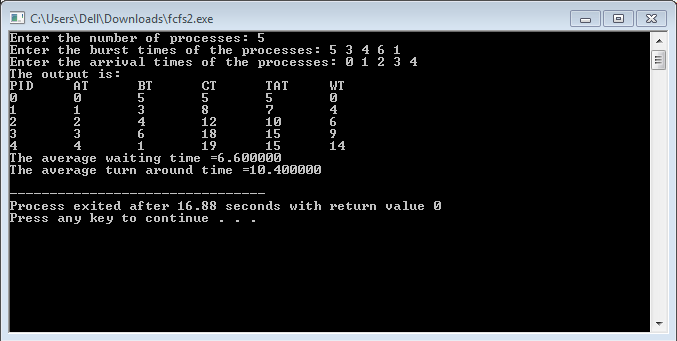
printf("\nThe average waiting time =%f",AWT);

printf("\nThe average turn around time =%f\n",ATAT);

return 0;

}

**Output:**



9.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<stdio.h>

int main()

{

int n,j,temp,temp1,temp2,pr[10],b[10],t[10],w[10],p[10],i;

float att=0,awt=0;

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

{

b[i]=0;

w[i]=0;

}

printf("enter the number of process");

scanf("%d",&n);

printf("enter the burst times");

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

{

scanf("%d",&b[i]);

p[i]=i;

}

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

{

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

{

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

{

temp=b[i];

temp1=p[i];

b[i]=b[j];

p[i]=p[j];

b[j]=temp;

p[j]=temp1;

}

}

}

w[0]=0;

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

w[i+1]=w[i]+b[i];

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

{

t[i]=w[i]+b[i];

awt=awt+w[i];

att=att+t[i];

}

awt=awt/n;

att=att/n;

printf("\n\tP\tBT\tTAT\tWT\n");

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

printf("\tp[%d]\t%d\t%d\t%d\n",p[i],b[i],t[i],w[i]);

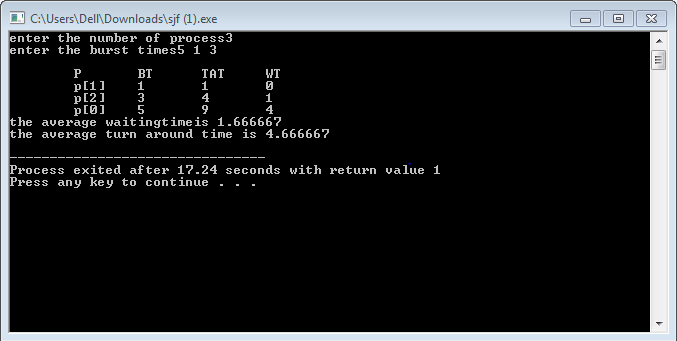
printf("the average waitingtimeis %f\n",awt);

printf("the average turn around time is %f\n",att);

return 1;

}

**Output:**



10.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<stdio.h>

int main()

{

int count,j,n,time,remain,flag=0, time\_quantum;

int wait\_time=0,turnaround\_time=0, at[10],bt[10],rt[10];

printf("Enter Total Process: ");

scanf("%d",&n);

remain=n;

printf("Enter process arrival time\n");

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

scanf("%d",&at[count]);

printf("Enter process burst time\n");

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

{

scanf("%d",&bt[count]);

rt[count]=bt[count];

}

printf("Enter Time Quantum:\t");

scanf("%d",&time\_quantum);

printf("\n\nProcess\t| Turnaround Time|Waiting Time\n\n");

for(time=0,count=0;remain!= 0;)

{

if(rt[count]<=time\_quantum && rt[count]>0)

{

time+=rt[count];

rt[count]=0;

flag=1;

}

else if(rt[count]>0)

{

rt[count]-=time\_quantum;

time+=time\_quantum;

}

if(rt[count]==0 && flag==1)

{

remain--;

printf("P[%d]\t%d\t%d\n",count+1,time-at[count],time-at[count]-bt[count]);

wait\_time+=time-at[ count]-bt[count];

turnaround\_time+=time- at[count];

flag=0;

}

if(count==n-1)

count=0;

else if(at[count+1]<=time)

count++;

else

count=0;

}

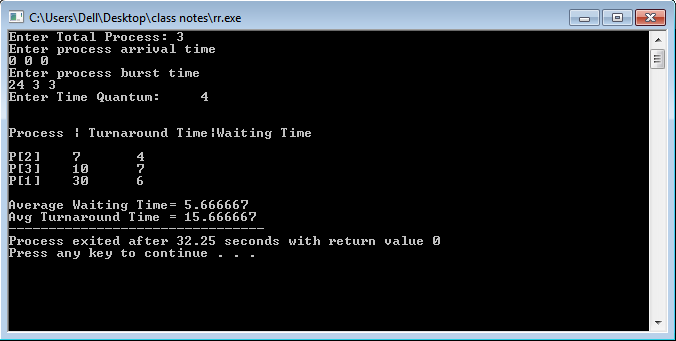
printf("\nAverage Waiting Time= %f\n",wait\_time\*1.0/n);

printf("Avg Turnaround Time = %f",turnaround\_time\*1.0/n);

return 0;

}

**Output:**



Bankers Algorithm

Deadlock Detection

&

Deadlock Avoidance

11. Bankers algorithm

**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.

**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

**Deadlock detection Algorithm:**

1.Work=Available

Finish[i]=false if Allocation!=0

=true if Allocation=0

2.Find i such that

Finish[i]=false and

Request[i]< work

If no such I then goto step 4

3.Finish[i]=true

Work=Work+Allocation

goto step 2

4.If Finish[i]=false for some I 1<i<n then system is in deadlock.

**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<<"\nenter the instances of resources : \n";

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 : \n";

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 : \n";

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";

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<<" ";

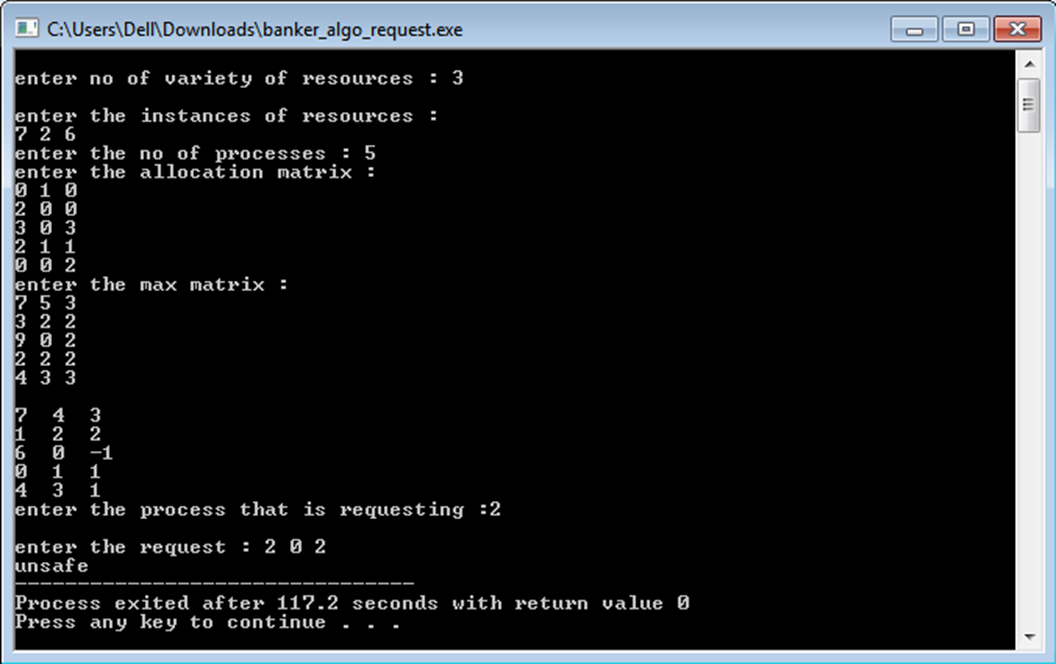
}

}

}

}

**Output:**



Page

Replacement

Algorithms

12. 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<stdio.h>

int main()

{

int i,j=0,k=0,l,array[50],arr[10],ele,f=0,size,c=0,x;

printf("Enter number of pages : ");

scanf("%d",&ele);

printf("Enter %d elements\n",ele);

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

scanf("%d",&array[i]);

printf("Enter frame size : ");

scanf("%d",&size);

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

arr[i]=-1;

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

{

for(x=0;x<size;x++)

{

if(arr[x]==array[i])

c=1;

}

if(c==1)

printf("No change\n");

else

{

arr[j]=array[i];

j=(j+1)%size;

f++;

for(l=0;l<size;l++)

printf("%d\t",arr[l]);

printf("\n");

}

c=0;

}

printf("Number of faults=%d\n",f);

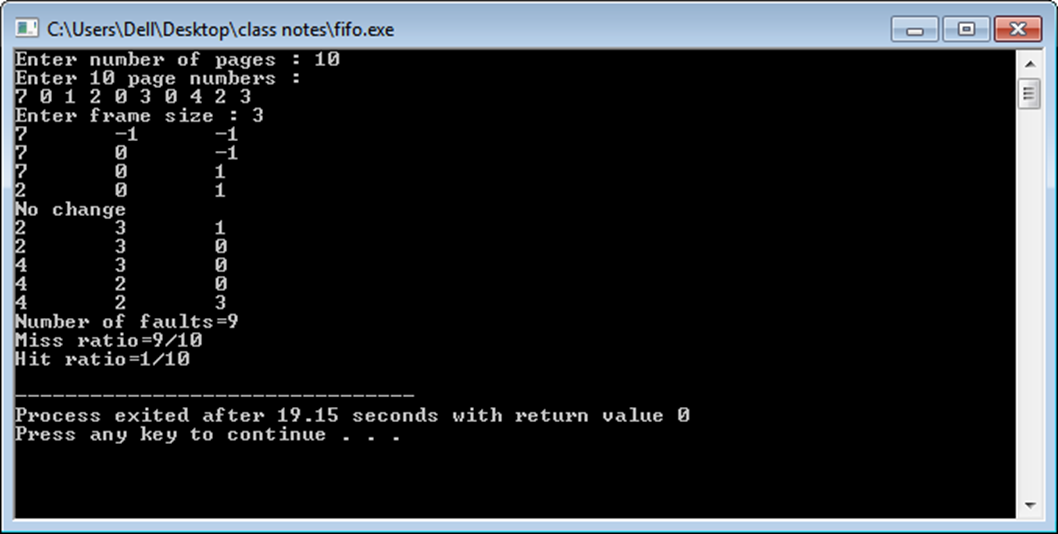
printf("Miss ratio=%d/%d\n",f,ele);

printf("Hit ratio=%d/%d\n",ele-f,ele);

return 0;

}

**Output:**



13.LRU(Least Recently Used)

**Algorithm :**

1. Start the process

2. Declare the size

3. Get the number of pages to be inserted

4. Get the value

5. Declare counter and stack

6. Select the least recently used page by counter value

7. Stack them according the selection.

8.  Display the values

9. Stop the process

**Program:**

#include<stdio.h>

main()

{

int q[20],p[50],c=0,c1,d,f,i,j,k=0,n,r,t,b[20],c2[20];

printf("Enter no of pages: ");

scanf("%d",&n);

printf("Enter the page numbers: \n");

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

scanf("%d",&p[i]);

printf("Enter no of frames:");

scanf("%d",&f);

q[k]=p[k];

printf("\n%d \n",q[k]);

c++;

k++;

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

{

c1=0;

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

{

if(p[i]!=q[j])

c1++;

}

if(c1==f)

{

c++;

if(k<f)

{

q[k]=p[i];

k++;

for(j=0;j<k;j++)

printf("%d ",q[j]);

printf("\n");

}

else

{

for(r=0;r<f;r++)

{

c2[r]=0;

for(j=i-1;j<n;j--)

{

if(q[r]!=p[j])

c2[r]++;

else

break;

}

}

for(r=0;r<f;r++)

b[r]=c2[r];

for(r=0;r<f;r++)

{

for(j=r;j<f;j++)

{

if(b[r]<b[j])

{

t=b[r];

b[r]=b[j];

b[j]=t;

}

}

}

for(r=0;r<f;r++)

{

if(c2[r]==b[0])

q[r]=p[i];

printf("%d ",q[r]);

}

printf("\n");

}

}

}

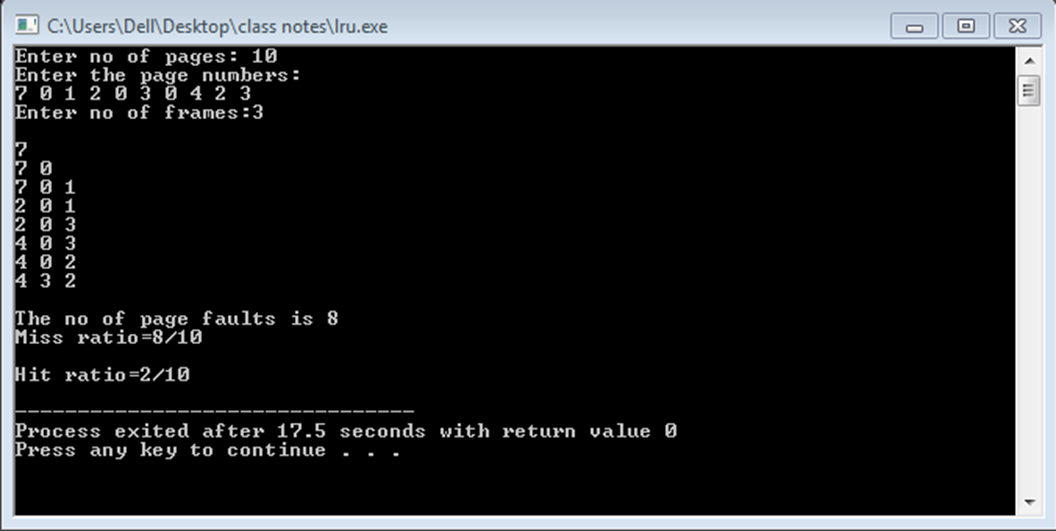
printf("\nThe no of page faults is %d",c);

printf("\nMiss ratio=%d/%d\n",c,n);

printf("\nHit ratio=%d/%d\n",n-c,n);

}

**Output:**



Echo Server

Using

Pipes

14.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 <sys/wait.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

int

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

{

int pipefd[2];

pid\_t cpid;

char buf;

if (argc != 2) {

fprintf(stderr, "Usage: %s <string>\n", argv[0]);

exit(EXIT\_FAILURE);

}

if (pipe(pipefd) == -1) {

perror("pipe");

exit(EXIT\_FAILURE);

}

cpid = fork();

if (cpid == -1) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (cpid == 0) { /\* Child reads from pipe \*/

close(pipefd[1]); /\* Close unused write end \*/

while (read(pipefd[0], &buf, 1) > 0)

write(STDOUT\_FILENO, &buf, 1);

write(STDOUT\_FILENO, "\n", 1);

close(pipefd[0]);

\_exit(EXIT\_SUCCESS);

} else { /\* Parent writes argv[1] to pipe \*/

close(pipefd[0]); /\* Close unused read end \*/

write(pipefd[1], argv[1], strlen(argv[1]));

close(pipefd[1]); /\* Reader will see EOF \*/

wait(NULL); /\* Wait for child \*/

exit(EXIT\_SUCCESS);

}

}

#include <sys/wait.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

int

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

{

int pipefd[2];

pid\_t cpid;

char buf;

if (argc != 2) {

fprintf(stderr, "Usage: %s <string>\n", argv[0]);

exit(EXIT\_FAILURE);

}

if (pipe(pipefd) == -1) {

perror("pipe");

exit(EXIT\_FAILURE);

}

cpid = fork();

if (cpid == -1) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (cpid == 0) { /\* Child reads from pipe \*/

close(pipefd[1]); /\* Close unused write end \*/

while (read(pipefd[0], &buf, 1) > 0)

write(STDOUT\_FILENO, &buf, 1);

write(STDOUT\_FILENO, "\n", 1);

close(pipefd[0]);

\_exit(EXIT\_SUCCESS);

} else { /\* Parent writes argv[1] to pipe \*/

close(pipefd[0]); /\* Close unused read end \*/

write(pipefd[1], argv[1], strlen(argv[1]));

close(pipefd[1]); /\* Reader will see EOF \*/

wait(NULL); /\* Wait for child \*/

exit(EXIT\_SUCCESS);

}

}

#include <sys/wait.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

int

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

{

int pipefd[2];

pid\_t cpid;

char buf;

if (argc != 2) {

fprintf(stderr, "Usage: %s <string>\n", argv[0]);

exit(EXIT\_FAILURE);

}

if (pipe(pipefd) == -1) {

perror("pipe");

exit(EXIT\_FAILURE);

}

cpid = fork();

if (cpid == -1) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (cpid == 0) { /\* Child reads from pipe \*/

close(pipefd[1]); /\* Close unused write end \*/

while (read(pipefd[0], &buf, 1) > 0)

write(STDOUT\_FILENO, &buf, 1);

write(STDOUT\_FILENO, "\n", 1);

close(pipefd[0]);

\_exit(EXIT\_SUCCESS);

} else { /\* Parent writes argv[1] to pipe \*/

close(pipefd[0]); /\* Close unused read end \*/

write(pipefd[1], argv[1], strlen(argv[1]));

close(pipefd[1]); /\* Reader will see EOF \*/

wait(NULL); /\* Wait for child \*/

exit(EXIT\_SUCCESS);

}

#include <sys/wait.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <sys/types.h>

int main()

{

int fd[2],fd1[2],nbyte;

pid\_t cpid;

char buf[80];

char string[]="hello D";int ch;

pipe(fd);

pipe(fd1);

if ((cpid=fork())==-1)

{

perror("fork");

exit(1);

}

if (cpid==0)

{

printf("\nchild writing");

close(fd[0]);

write(fd[1],string,(strlen(string)+1));

wait(NULL);

close(fd1[1]);

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

printf("\n'C' received string:%s",buf);

exit(0);

}

else

{

close(fd[1]);

printf("\nparent reading");

nbyte = read(fd[0],buf,sizeof(buf));

printf("\n'P'received string:%s",buf);

close(fd1[0]);

printf("\n'P' writing:");

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

}

return 0;

}

**Output:**