1.A) Fork() System Call:

AIM:To write the program to implement fork () system call.

DESCRIPTION:

Used to create new processes. The new process consists of a copy of the address space of the original process. The value of process id for the child process is zero, whereas the value of process id for the parent is an integer value greater than zero.

Syntax:

Fork ( );

ALGORITHM:

Step 1: Start the program.

Step 2: Declare the variables pid and child id.

Step 3: Get the child id value using system call fork().

Step 4: If child id value is greater than zero then print as "i am in the parent process".

Step 5: If child id! = 0 then using getpid() system call get the process id.

Step 6: Print "i am in the parent process" and print the process id.

Step 7: If child id! = 0 then using getppid() system call get the parent process id.

Step 8: Print "i am in the parent process" and print the parent process id.

Step 9: Else If child id value is less than zero then print as "i am in the child process".

Step 10: If child id! = 0 then using getpid() system call get the process id.

Step 11: Print "i am in the child process" and print the process id.

Step 12: If child id! = 0 then using getppid() system call get the parent process id.

Step 13: Print "i am in the child process" and print the parent process id.

Step 14: Stop the program.

PROGRAM :

SOURCE CODE:

```
/* fork system call */
#include<stdio.h>
#include <unistd.h>
#include<sys/types.h>
int main()
{
        int id,childid;
        id=getpid();
        if((childid=fork())>0)
        {
                printf("\n i am in the parent process %d",id);
                printf("\n i am in the parent process %d",getpid());
                printf("\n i am in the parent process %d\n",getppid());
        }
        else
        {
                printf("\n i am in child process %d",id);
                printf("\n i am in the child process %d",getpid());
                printf("\n i am in the child process %d",getppid());
        }
}
```


OUTPUT:

```
$ vi fork.c
$ cc fork.c
$ ./a.out
 i am in child process 3765
 i am in the child process 3766
 i am in the child process 3765
 i am in the parent process 3765
 i am in the parent process 3765
 i am in the parent process 3680
```

RESULT:

Thus the program was executed and verified successfully.

1.B) Wait () and Exit () System Calls:

AIM:To write the program to implement the system calls wait ( ) and exit ( ).

DESCRIPTION:

i. fork ( )

Used to create new process. The new process consists of a copy of the address space of the original process. The value of process id for the child process is zero, whereas the value of process id for the parent is an integer value greater than zero.

Syntax: fork ( );

ii. wait ( )

The parent waits for the child process to complete using the wait system call. The wait system call returns the process identifier of a terminated child, so that the parent can tell which of its possibly many children has terminated.

Syntax: wait (NULL);

iii. exit ( )

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit system call. At that point, the process may return data (output) to its parent process (via the wait system call).

Syntax: exit (0);

ALGORITHM:

Step 1: Start the program.

Step 2: Declare the variables pid and i as integers.

Step 3: Get the child id value using the system call fork ().

Step 4: If child id value is less than zero then print "fork failed".

Step 5: Else if child id value is equal to zero, it is the id value of the child and then start the child process to execute and perform Steps 7 & 8.

Step 6: Else perform Step 9.

Step 7: Use a for loop for almost five child processes to be called.

Step 8: After execution of the for loop then print "child process ends".

Step 9: Execute the system call wait ( ) to make the parent to wait for the child process to get over.

Step 10: Once the child processes are terminated, the parent terminates and hence prints "Parent process ends".

Step 11: After both the parent and the child processes get terminated it execute the wait ( ) system call to permanently get deleted from the OS.

Step 12: Stop the program.

PROGRAM:

1.B.1) SOURCE CODE:

```
#include<stdio.h>
#include<unistd.h>
int main( )
{
        int i, pid;
        pid=fork( );
        if(pid== -1)
        {
                printf("fork failed");
                exit(0);
        }
        else if(pid==0)
        {
                printf("\n Child process starts");
                for(i=0; i<5; i++)
                {
                        printf("\n Child process %d is called", i);
                }
                printf("\n Child process ends");
        }
        else
        {
                wait(0);
                printf("\n Parent process ends");
        }
        exit(0);
}
```

OUTPUT:

$ vi waitexit.c

$ cc waitexit.c

$ ./a.out

Child process starts

Child process 0 is called

Child process 1 is called

Child process 2 is called

Child process 3 is called

Child process 4 is called

Child process ends

Parent process ends

RESULT:

Thus the program was executed and verified successfully

1.B.2) WAIT () AND EXIT () SYSTEM CALLS

PROGRAM:

SOURCE CODE:

```
/* wait system call */
#include <stdlib.h>
#include <errno.h>
#include<stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
main()
{
        pid_t pid;
        int rv;
        switch(pid=fork())
        {
                case -1:
                                perror("fork");
                                exit(1);
                case 0:
                                printf("\n CHILD: This is the child process!\n");
                                fflush(stdout);
                                printf("\n CHILD: My PID is %d\n", getpid());
                                printf("\n CHILD: My parent's PID is %d\n",getppid());
                                printf("\n CHILD: Enter my exit status (make it small):\n ");
                                printf("\n CHILD: I'm outta here!\n");
                                scanf(" %d", &rv);
                                exit(rv);
                default:
                                printf("\nPARENT: This is the parent process!\n");
                                printf("\nPARENT: My PID is %d\n", getpid());
                                fflush(stdout);
                                wait(&rv);
                                fflush(stdout);
                                printf("\nPARENT: My child's PID is %d\n", pid);
```

```
                printf("\nPARENT: I'm now waiting for my child to exit()...\n");
                fflush(stdout);
                printf("\nPARENT: My child's exit status is:
                %d\n",WEXITSTATUS(rv));
                printf("\nPARENT: I'm outta here!\n");
        }
}
```

OUTPUT:

```
$ vi wait.c
$ cc wait.c
$ ./a.out
CHILD: This is the child process!
CHILD: My PID is 3821
CHILD: My parent's PID is 3820
CHILD: Enter my exit status (make it small):
CHILD: I'm outta here!
PARENT: This is the parent process!
PARENT: My PID is 3820
10
PARENT: My child's PID is 3821
PARENT: I'm now waiting for my child to exit()...
PARENT: My child's exit status is: 10
PARENT: I'm outta here!
```

RESULT:

Thus the program was executed and verified successfully

1. C) EXECL SYSTEM CALL

AIM: To write a program to implement the system call execl ( ).

DESCRIPTION:

In execl () system function takes the path of the executable binary file (i.e. **/bin/ls**) as the first and second argument. Then, the arguments (i.e. **-lh, /home**) that you want to pass to the executable followed by **NULL**. Then execl () system function runs the command and prints the output. If any error occurs, then execl () returns -1. Otherwise, it returns nothing.

Syntax:

int execl (const char *path, const char *arg, ..., NULL);

ALGORITHM

Step 1: Start the program.

Step 2: Include the necessary header files.

Step 3: Print execution of exec system call for the ls Unix command.

Step 4: Execute the execl function using the appropriate syntax for the Unix command ls.

Step 5: The list of all files and directories of the system is displayed.

Step 6: Stop the program.

PROGRAM :

SOURCE CODE:

```
/* execl system call */
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
main()
{
        printf("Before execl \n");
        execl("/bin/ls","ls",(char*)0);
        printf("After Execl\n");
}
```

OUTPUT:

$ vi execl.c

$ cc execl.c

$ ./a.out

Before execl

a1 aaa aaa.txt abc a.out b1 b2 comm.c db db1 demo2 dir1 direc.c execl.c f1.txt fflag.c file1 file2 fork.c m1 m2 wait.c xyz

RESULT:

Thus the program was executed and verified successfully

1.D) EXECV SYSTEM CALL

AIM: To write a program to implement the system call execv ( ).

DESCRIPTION:

In execl() function, the parameters of the executable file is passed to the function as different arguments. With execv(), you can pass all the parameters in a NULL terminated array **argv**. The first element of the array should be the path of the executable file. Otherwise, execv() function works just as execl() function.

Syntax:int execv (const char *path, char *const argv[]);

ALGORITHM:

Step 1: Start the program.

Step 2: Include the necessary header files.

Step 3: Print execution of exec system call for the ls Unix command.

Step 4: Execute the execv function using the appropriate syntax for the Unix command ls.

Step 5: The list of all files and directories of the system is displayed.

Step 6: Stop the program.

PROGRAM :

SOURCE CODE:

```
/* execv system call */
#include<stdio.h>
#include<sys/types.h>
main(int argc,char *argv[])
{
        printf("before execv\n");
        execv("/bin/ls",argv);
        printf("after execv\n");
}
```

OUTPUT:

```
$ vi execv.c
$ cc execv.c
$ ./a.out
before execv
a1  aaa  aaa.txt  abc  a.out  b1  b2  comm.c  db  db1  demo2  dir1  direc.c  execl.c  execv.c  f1.txt
fflag.c  file1  file2  fork.c  m1  m2  wait.c  xyz
```

RESULT:

Thus the program was executed and verified successfully.

1.E)Directory Management (Directory Hierarchy):

AIM:To write the program to implement the system calls opendir ( ), readdir ( ),closedir ().

DESCRIPTION:

UNIX offers a number of system calls to handle a directory. The following are most commonly used system calls.

SYSTEM CALLS USED:

i. **opendir ( )**

Open a directory.

Syntax:

DIR * opendir (const char * dirname);

opendir () takes dirname as the path name and returns a pointer to a DIR structure. On error returns NULL.

ii. **readdir ( )**

Read a directory.

Syntax:

struct dirent * readdir (DIR *dp) ;

A directory maintains the inode number and filename for every file in its fold. This function returns a pointer to a dirent structure consisting of inode number and filename. 'dirent' structure is defined in <dirent.h> to provide at least two members – inode number and directory name.

```
struct dirent
{
ino_t d_ino; // directory inode number
char d_name[]; // directory name
}
```

iii. **closedir ():**

Close a directory.

Syntax:

int closedir (DIR * dp);

Closes a directory pointed by dp. It returns 0 on success and -1 on error.

ALGORITHM:

Step 1: Start the program.

Step 2: In the main function pass the arguments.

Step 3: Create structure as stat buff and the variables as integer.

Step 4: Open the directory.

Step 5: Read the contents of the directory (filenames).

Step 6: Display the contents of the directory.

Step 7: Close the directory.

Step 4: Stop the program.

PROGRAM:

SOURCE CODE:

```
/* Recursively descend a directory hierarchy pointing a file */
#include<stdio.h>
#include<dirent.h>
#include<errno.h>
#include<fcntl.h>
#include<unistd.h>
int main(int argc,char *argv[])
{
        struct dirent *direntp; DIR *dirp; if(argc!=2)
        {
                printf("ussage %s directory name \n",argv[0]);
                return 1;

        }
        if((dirp=opendir(argv[1]))==NULL)
        {
                perror("Failed to open directory \n");
                return 1;
        }
        while((direntp=readdir(dirp))!=NULL)
                printf("%s\n",direntp->d_name);
        while((closedir(dirp)==-1)&&(errno==EINTR));
                return 0;
}
```

OUTPUT:

```
$ vi direc.c
$ cc direc.c
$ ./a.out ./
f1.txt
fflag.c
.
..
b2
comm.c
dir1
demo2
a1
m1
db
a.out
direc.c
db1
file1
b1
xyz
abc
aaa
file2
aaa.txt
m2
```

RESULT:

Thus the program was executed and verified successfully.

1.F) File Management:

AIM:To write the program to implement the system calls open ( ), read ( ), write ( ) & close ().

DESCRIPTION:

The file structure related system calls available in the UNIX system let you create, open, and close files, read and write files, randomly access files, alias and remove files, get information about files, check the accessibility of files, change protections, owner, and group of files, and control devices. These operations either use a character string that defines the absolute or relative path name of a file, or a small integer called a file descriptor that identifies the I/O channel. A channel is a connection between a process and a file that appears to the process as an unformatted stream of bytes. The kernel presents and accepts data from the channel as a process reads and writes that channel. To a process then, all input and output operations are synchronous and unbuffered.

SYSTEM CALLS USED:

System calls are functions that a programmer can call to perform the services of the operating system.

Open ():

Open () system call to open a file.

open () returns a file descriptor, an integer specifying the position of this open n file in the table of open files for the current process.

Close ():

Close () system call to close a file.

Read ():

Read () data from a file opened for reading.

Write ():

Write () data to a file opened for writing.

The open () system call:

#include<fcntl.h>

int open (const char *path, int oflag);

The return value is the descriptor of the file. Returns -1 if the file could not be opened. The first parameter is path name of the file to be opened and the second parameter is the opening mode specified by bitwise oring one or more of the following values

Value	Meaning
O_RDONLY	Open for reading only
O_WRONLY	Open for writing only
O_RDWR	Open for reading and writing
O_APPEND	Open at end of file for writing
O_CREAT	Create the file if it doesn't already exist
O_EXCL	If set and O_CREAT set will cause open() to fail if the file already exists
O_TRUNC	Truncate file size to zero if it already exists

close () system call:

The close () system call is used to close files.

#include <unistd.h>

int close (int fildes);

It is always good practices to close files when not needed as open files do consume resources and all normal systems impose a limit on the number of files that a process can hold open.

The read () system call:

The read () system call is used to read data from a file or other object identified by a file descriptor. The prototype is

#include<sys/types.h>

size_t read (int fildes, void *buf, size_t nbyte);

fildes is the descriptor, buf is the base address of the memory area into which the data is read and nbyte is the maximum amount of data to read. The return value is the actual amount of data read from the file. The pointer is incremented by the amount of data read. An attempt to read beyond the end of a file results in a return value of zero.

The write () system call:

The write () system call is used to write data to a file or other object identified by a file descriptor. The prototype is

#include<sys/types.h>

size_t write (int fildes, const void *buf, size_t nbyte);

ALGORITHM:

Step 1: Start the program.

Step 2: Declare the structure elements.

Step 3: Create a temporary file named temp1.

Step 4: Open the file named "test" in a write mode.

Step 5: Enter the strings for the file.

Step 6: Write those strings in the file named "test".

Step 7: Create a temporary file named temp2.

Step 8: Open the file named "test" in a read mode.

Step 9: Read those strings present in the file "test" and save it in temp2.

Step 10: Print the strings which are read.

Step 11: Stop the program.

PROGRAM :

SOURCE CODE:

```
#include<stdio.h>
#include<unistd.h>
#include<string.h>
#include<fcntl.h>
int main( )
{
        int fd[2];
        char buf1[25]= "just a test\n"; char
        buf2[50];
        fd[0]=open("file1", O_RDWR);
        fd[1]=open("file2", O_RDWR);
        write(fd[0], buf1, strlen(buf1));
        printf("\n Enter the text now....");
        gets(buf1);
        write(fd[0], buf1, strlen(buf1));
        lseek(fd[0], SEEK_SET, 0);
        read(fd[0], buf2, sizeof(buf1));
        write(fd[1], buf2, sizeof(buf2));
        close(fd[0]);
        close(fd[1]);
        printf("\n");
```

```
        return0;
}
```

OUTPUT:

Enter the text now….progress

Cat file1 Just a

test progress

Cat file2 Just a test progress

RESULT:

Thus the program was executed successfully.