

# ASSIGNMENT – 1

---

## 1. To study the basics of system call and system library.

### System Call:

In computing, a system call is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on. A system call is a way for programs to interact with the operating system. A computer program makes a system call when it makes a request to the operating system's kernel. System call provides the services of the operating system to the user programs via Application Program Interface (API). It provides an interface between a process and operating system to allow user-level processes to request services of the operating system. System calls are the only entry points into the kernel system. All programs needing resources must use system calls.

### Types of System Calls:

There are 5 different categories of system calls:

#### Process Control

These system calls deal with processes such as process creation, process termination etc.

#### File Management

These system calls are responsible for file manipulation such as creating a file, reading a file, writing into a file etc.

#### Device Management

These system calls are responsible for device manipulation such as reading from device buffers, writing into device buffers etc.

#### Information Maintenance

These system calls handle information and its transfer between the operating system and the user program.

#### Communication

These system calls are useful for interprocess communication. They also deal with creating and deleting a communication connection.

### System Library:

System libraries are special functions or programs using which application programs or system utilities accesses Kernel's features. These libraries implement most of the

functionalities of the operating system and do not require kernel module's code access rights.

## 2. Study the following system calls:

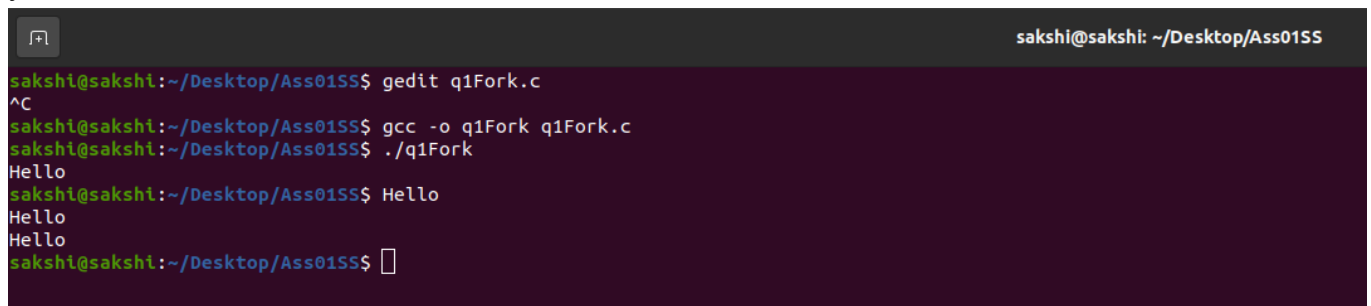
### Fork()

Fork system call is used for creating a new process, which is called child process, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork () system call.

#### Source Code:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{

    fork();
    fork();
    printf("Hello\n");
    return 0;
}
```

A terminal window with a dark background and light green text. The prompt is 'sakshi@sakshi: ~/Desktop/Ass01SS'. The user enters 'gedit q1Fork.c', followed by '^C' to cancel. Then they enter 'gcc -o q1Fork q1Fork.c' and './q1Fork'. The output shows 'Hello' printed three times, once from each of the three processes created by two consecutive fork() calls. The prompt returns after the third 'Hello' is printed.

```
sakshi@sakshi: ~/Desktop/Ass01SS$ gedit q1Fork.c
^C
sakshi@sakshi: ~/Desktop/Ass01SS$ gcc -o q1Fork q1Fork.c
sakshi@sakshi: ~/Desktop/Ass01SS$ ./q1Fork
Hello
sakshi@sakshi: ~/Desktop/Ass01SS$ Hello
Hello
Hello
sakshi@sakshi: ~/Desktop/Ass01SS$
```

### Exec()

This system call runs an executable file in the context of an already running process. It replaces the previous executable file. This is known as an overlay. The original process identifier remains since a new process is not created but data, heap, stack etc. of the process are replaced by the new process.

#### Source Code:

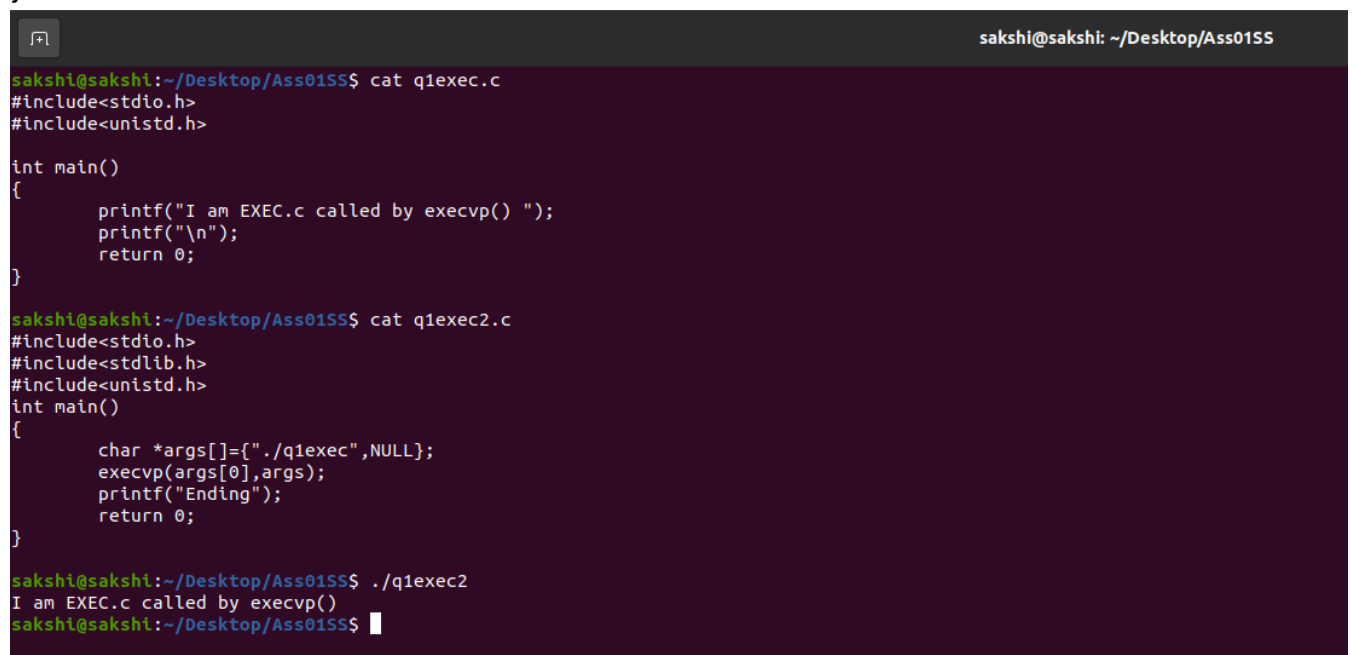
##### exec.c file

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

```
int main()
{
    printf("I am EXEC.c called by execvp() ");
    printf("\n");
    return 0;
}
```

### **execDemo.c file**

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main()
{
    char *args[]={"/q1exec",NULL};
    execvp(args[0],args);
    printf("Ending");
    return 0;
}
```



```
sakshi@sakshi: ~/Desktop/Ass01SS$ cat q1exec.c
#include<stdio.h>
#include<unistd.h>

int main()
{
    printf("I am EXEC.c called by execvp() ");
    printf("\n");
    return 0;
}

sakshi@sakshi:~/Desktop/Ass01SS$ cat q1exec2.c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main()
{
    char *args[]={"/q1exec",NULL};
    execvp(args[0],args);
    printf("Ending");
    return 0;
}

sakshi@sakshi:~/Desktop/Ass01SS$ ./q1exec2
I am EXEC.c called by execvp()
sakshi@sakshi:~/Desktop/Ass01SS$
```

### **getpid()**

Returns the process ID of the calling process. This is often used by routines that generate unique temporary filenames.

### **Source Code:**

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

int main()
```

```

{
    int pid = fork();
    if (pid == 0)
        printf("\nCurrent process id of Process : %d\n",
            getpid());
    return 0;
}

```

```

sakshi@sakshi: ~/Desktop/Ass01SS
sakshi@sakshi:~/Desktop/Ass01SS$ gcc -o q1getpid q1getpid.c
sakshi@sakshi:~/Desktop/Ass01SS$ ./q1getpid
sakshi@sakshi:~/Desktop/Ass01SS$
Current process id of Process : 5745
sakshi@sakshi:~/Desktop/Ass01SS$ █

```

## Exit()

The exit() system call is used by a program to terminate its execution. In a multithreaded environment, this means that the thread execution is complete. The operating system reclaims resources that were used by the process after the exit() system call.

### Source Code:

```

#include <stdio.h>
#include <stdlib.h>

int main ()
{

    printf("Exit 1\n");
    exit(EXIT_SUCCESS);
    printf("Exit 2\n");
    return 0;
}

```

```

sakshi@sakshi: ~/Desktop/Ass01SS
sakshi@sakshi:~/Desktop/Ass01SS$ gcc -o q1exit q1exit.c
sakshi@sakshi:~/Desktop/Ass01SS$ ./q1exit
Exit 1
sakshi@sakshi:~/Desktop/Ass01SS$ █

```

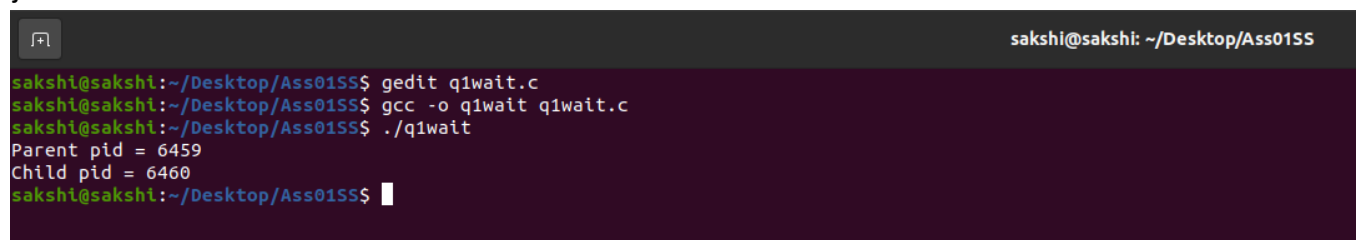
## wait()

In some systems, a process may wait for another process to complete its execution. This happens when a parent process creates a child process and the execution of the parent process is suspended until the child process executes. The suspending of the parent process occurs with a wait() system call. When the child process completes execution, the control is returned back to the parent process.

### Source Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>

int main()
{
    pid_t cpid;
    if (fork()== 0)
        exit(0);          /* terminate child */
    else
        cpid = wait(NULL); /* reaping parent */
    printf("Parent pid = %d\n", getpid());
    printf("Child pid = %d\n", cpid);
    return 0;
}
```

A terminal window with a dark background and light green text. The prompt is 'sakshi@sakshi: ~/Desktop/Ass01SS'. The user enters 'gedit q1wait.c', then 'gcc -o q1wait q1wait.c', and finally './q1wait'. The output shows 'Parent pid = 6459' and 'Child pid = 6460'.

```
sakshi@sakshi:~/Desktop/Ass01SS$ gedit q1wait.c
sakshi@sakshi:~/Desktop/Ass01SS$ gcc -o q1wait q1wait.c
sakshi@sakshi:~/Desktop/Ass01SS$ ./q1wait
Parent pid = 6459
Child pid = 6460
sakshi@sakshi:~/Desktop/Ass01SS$
```

### stat()

Stat() is a system call that is used to determine information about a file based on its file path.

### Source Code:

```
#include<stdio.h>
#include<sys/stat.h>
int main()
{
    struct stat sfile; //pointer to stat struct
    stat("q1wait.c", &sfile); //stat system call
    printf("st_mode = %o\n", sfile.st_mode); //accessing st_mode (data member of stat struct)
    return 0;
}
```

```
sakshi@sakshi: ~/Desktop/Ass01SS
sakshi@sakshi:~/Desktop/Ass01SS$ gedit q1stat.c
sakshi@sakshi:~/Desktop/Ass01SS$ gcc -o q1stat q1stat.c
sakshi@sakshi:~/Desktop/Ass01SS$ ./q1stat
st_mode = 100664
sakshi@sakshi:~/Desktop/Ass01SS$
```

## opendir()

The opendir() function shall open a directory stream corresponding to the directory named by the dirname argument. The directory stream is positioned at the first entry. If the type DIR is implemented using a file descriptor, applications shall only be able to open up to a total of {OPEN\_MAX} files and directories.

### Source Code:

```
#include<stdio.h>
#include<dirent.h>
#include<stdlib.h>
void main()
{
    DIR *dirp;
    if((dirp=opendir("../Ass01SS"))==NULL) /* trying to open directory*/
    {
        printf("cannot open\n");
        exit(1);
    }

    printf("opened\n");
    closedir(dirp);
}
```

```
sakshi@sakshi: ~/Desktop/Ass01SS
sakshi@sakshi:~/Desktop/Ass01SS$ gcc -o q1opendir q1opendir.c
sakshi@sakshi:~/Desktop/Ass01SS$ ./q1opendir
opened
sakshi@sakshi:~/Desktop/Ass01SS$
```

## readdir()

Returns a pointer to a dirent structure describing the next directory entry in the directory stream associated with dir.

Tag	Description
EBADF	Invalid file descriptor <i>fd</i> .
EFAULT	Argument points outside the calling process's address space.
EINVAL	Result buffer is too small.
ENOENT	No such directory.
ENOTDIR	
	File descriptor does not refer to a directory.

### Source Code:

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>
#include <errno.h>
#include <string.h>

int main(int c, char *v[]) {
    DIR *myDirectory;
    struct dirent *myFile;

    if (c == 2) {
        myDirectory = opendir(v[1]);
        if (myDirectory) {
            puts("OK the directory is opened, let's see its files:");
            while ((myFile = readdir(myDirectory)))
                printf("%s\n", myFile->d_name);
            if (closedir(myDirectory) == 0)
                puts("The directory is now closed.");
            else
                puts("The directory can not be closed.");
        } else if (errno == ENOENT)
            puts("This directory does not exist.");
        else if (errno == ENOTDIR)
            puts("This file is not a directory.");
        else if (errno == EACCES)
            puts("You do not have the right to open this folder.");
        else
            puts("That's a new error, check the manual.");
    } else
        puts("Sorry we need exactly 2 arguments.");
    return (0);
}
```

```
sakshi@sakshi: ~/Desktop/Ass01SS
sakshi@sakshi:~/Desktop/Ass01SS$ gcc -o q1readdir q1readdir.c
sakshi@sakshi:~/Desktop/Ass01SS$ ./q1readdir ../Ass01SS/
OK the directory is opened, let's see its files:
..
q1readdir
q1stat
q1getpid
q1exec
q1exec2
q1Fork.c
q1exec2.c
.
q1wait.c
q1opendir.c
q1exit
q1opendir
q1getpid.c
q1Fork
q1readdir.c
q1stat.c
q1wait
q1exit.c
q1exec.c
The directory is now closed.
sakshi@sakshi:~/Desktop/Ass01SS$
```

## chdir()

The chdir command is a system function (system call) which is used to change the current working directory. On some systems, this command is used as an alias for the shell command cd. chdir changes the current working directory of the calling process to the directory specified in path.

### Source Code:

```
#include<stdio.h>
#include<unistd.h> //chdir declared in this header
int main()
{
    char s[100];

    // printing current working directory
    printf("%s\n", getcwd(s, 100));

    chdir("../");

    // printing current working directory
    printf("%s\n", getcwd(s, 100));
    return 0;
}
```

```
sakshi@sakshi: ~/Desktop/Ass01SS
sakshi@sakshi:~/Desktop/Ass01SS$ gcc -o q1chdir q1chdir.c
sakshi@sakshi:~/Desktop/Ass01SS$ ./q1chdir
/home/sakshi/Desktop/Ass01SS
/home/sakshi/Desktop
sakshi@sakshi:~/Desktop/Ass01SS$
```



## chmod()

In Unix and Unix-like operating systems, chmod is the command and system call used to change the access permissions of file system objects (files and directories) sometimes known as modes. It is also used to change special mode flags such as setuid and setgid flags and a 'sticky' bit. Modes are specified by *or'ing* the following:

Tag	Description
S_ISUID	04000 set user ID on execution
S_ISGID	02000 set group ID on execution
S_ISVTX	01000 sticky bit
S_IRUSR	00400 read by owner
S_IWUSR	00200 write by owner
S_IXUSR	00100 execute/search by owner
S_IRGRP	00040 read by group
S_IWGRP	00020 write by group
S_IXGRP	00010 execute/search by group
S_IROTH	00004 read by others
S_IWOTH	00002 write by others
S_IXOTH	00001 execute/search by others

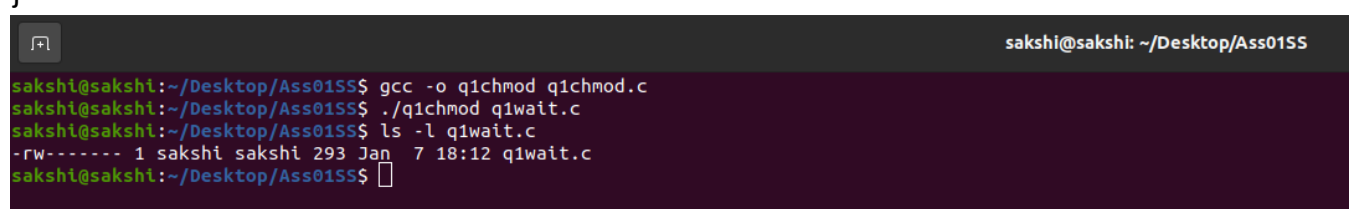
### Source Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>

int main( int argc, char **argv )
{
    int i;

    for( i = 1; i < argc; i++ ) {
        if( chmod( argv[i], S_IRUSR | S_IWUSR ) == -1 ) {
            perror( argv[i] );
        }
    }

    return 0;
}
```



```
sakshi@sakshi: ~/Desktop/Ass01SS
sakshi@sakshi:~/Desktop/Ass01SS$ gcc -o q1chmod q1chmod.c
sakshi@sakshi:~/Desktop/Ass01SS$ ./q1chmod q1wait.c
sakshi@sakshi:~/Desktop/Ass01SS$ ls -l q1wait.c
-rw----- 1 sakshi sakshi 293 Jan  7 18:12 q1wait.c
sakshi@sakshi:~/Desktop/Ass01SS$
```

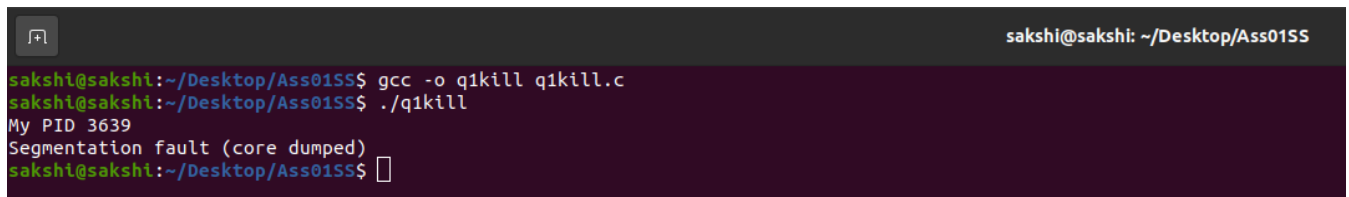
## kill()

The kill() system call is used by the operating system to send a termination signal to a process that urges the process to exit. However, kill system call does not necessarily mean killing the process and can have various meanings.

### Source Code:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>

int main(int argc, char *argv[]){
    printf("My PID %d\n",getpid());
    sleep (1);
    kill(getpid(),SIGSEGV);
    return 0;
}
```



```
sakshi@sakshi: ~/Desktop/Ass01SS
sakshi@sakshi:~/Desktop/Ass01SS$ gcc -o q1kill q1kill.c
sakshi@sakshi:~/Desktop/Ass01SS$ ./q1kill
My PID 3639
Segmentation fault (core dumped)
sakshi@sakshi:~/Desktop/Ass01SS$
```

## read()

The read() system call is used to access data from a file that is stored in the file system. The file to read can be identified by its file descriptor and it should be opened using open() before it can be read. In general, the read() system calls takes three arguments i.e. the file descriptor, buffer which stores read data and number of bytes to be read from the file.

### Source Code:

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

int main()
{
    int fd, sz;
```

```

char c[100];
fd = open("file1", O_RDONLY);
if (fd < 0) { perror("r1"); exit(1); }
sz = read(fd, c, 10);
printf("%d bytes were read.\n",sz);
c[sz] = '\0';
printf("Those bytes are as follows: %s\n", c);
}

```

### **write()**

The write() system calls writes the data from a user buffer into a device such as a file. This system call is one of the ways to output data from a program. In general, the write system calls takes three arguments i.e. file descriptor, pointer to the buffer where data is stored and number of bytes to write from the buffer.

### **Source Code:**

```

#include<stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#include<string.h>

int main()
{
    int sz;
    int fd = open("file1", O_WRONLY);
    if (fd < 0)
    {
        perror("r1");
        exit(1);
    }

    write(fd, "hello geeks\n", strlen("hello geeks\n"));
    close(fd);
    return 0;
}

```

```
sakshi@sakshi: ~/Desktop/Ass0155
sakshi@sakshi:~/Desktop/Ass0155$ cat file1
abcdefghijklmnopqrstuvwxyz
sakshi@sakshi:~/Desktop/Ass0155$ gcc -o q1write q1write.c
sakshi@sakshi:~/Desktop/Ass0155$ ./q1write
sakshi@sakshi:~/Desktop/Ass0155$ cat file1
hello geeks
6789009
sakshi@sakshi:~/Desktop/Ass0155$
```

## open()

The open() system call is used to provide access to a file in a file system. This system call allocates resources to the file and provides a handle that the process uses to refer to the file. A file can be opened by multiple processes at the same time or be restricted to one process. It all depends on the file organisation and file system.

### Source Code:

```
#include<stdio.h>
#include<fcntl.h>
#include<errno.h>
extern int errno;
int main()
{
    int fd = open("xz.txt", O_RDONLY | O_CREAT);

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

    if (fd == -1)
    {
        // print which type of error have in a code
        printf("Error Number % d\n", errno);

        // print program detail "Success or failure"
        perror("Program");
    }
    return 0;
}
```

```
sakshi@sakshi: ~/Desktop/Ass0155
sakshi@sakshi:~/Desktop/Ass0155$ gcc -o q1open q1open.c
sakshi@sakshi:~/Desktop/Ass0155$ ./q1open
fd = 3
sakshi@sakshi:~/Desktop/Ass0155$
```

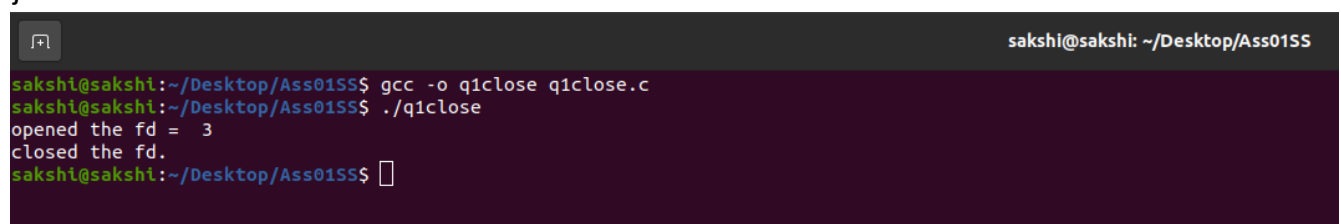
## close()

The close() system call is used to terminate access to a file system. Using this system call means that the file is no longer required by the program and so the buffers are flushed, the file metadata is updated and the file resources are de-allocated.

### Source Code:

```
#include<stdio.h>
#include <fcntl.h>
#include<stdlib.h>
#include <unistd.h>
#include<errno.h>
int main()
{
    int fd1 = open("xz.txt", O_RDONLY);
    if (fd1 < 0)
    {
        perror("c1");
        exit(1);
    }
    printf("opened the fd = % d\n", fd1);

    // Using close system Call
    if (close(fd1) < 0)
    {
        perror("c1");
        exit(1);
    }
    printf("closed the fd.\n");
}
```

A terminal window with a dark background. The title bar shows 'sakshi@sakshi: ~/Desktop/Ass01SS'. The terminal content shows the following commands and output:

```
sakshi@sakshi:~/Desktop/Ass01SS$ gcc -o q1close q1close.c
sakshi@sakshi:~/Desktop/Ass01SS$ ./q1close
opened the fd = 3
closed the fd.
sakshi@sakshi:~/Desktop/Ass01SS$
```

### lseek()

lseek is a system call that is used to change the location of the read/write pointer of a file descriptor. The location can be set either in absolute or relative terms.

### Source Code:

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

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

```
int main(){
    int n, f, f1;
    char buf[5];
    f=open("file1",O_RDWR);
    read(f, buf, 5);
    write(1, buf, 5);
    lseek(f,5,SEEK_CUR);
    read(f, buf, 5);
    write(1, buf, 5);
    printf("\n");
}
```

```
sakshi@sakshi: ~/Desktop/Ass01SS
sakshi@sakshi:~/Desktop/Ass01SS$ cat file1
abcdefghijk56789009
sakshi@sakshi:~/Desktop/Ass01SS$ gcc -o q1lseek q1lseek.c
sakshi@sakshi:~/Desktop/Ass01SS$ ./q1lseek
abcdek5678
sakshi@sakshi:~/Desktop/Ass01SS$
```

## time()

The time() function is defined in time.h (ctime in C++) header file. This function returns the time since 00:00:00 UTC, January 1, 1970 (Unix timestamp) in seconds. If second is not a null pointer, the returned value is also stored in the object pointed to by second.

### Source Code:

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

```
int main ()
{
    time_t seconds;
    seconds = time(NULL);
    printf("Seconds since January 1, 1970 = %ld\n", seconds);
    return(0);
}
```

```
sakshi@sakshi: ~/Desktop/Ass01SS
sakshi@sakshi:~/Desktop/Ass01SS$ gcc -o q1time q1time.c
sakshi@sakshi:~/Desktop/Ass01SS$ ./q1time
Seconds since January 1, 1970 = 1641648337
sakshi@sakshi:~/Desktop/Ass01SS$
```

## mount()

mount() attaches the filesystem specified by source (which is often a device name, but can also be a directory name or a dummy) to the directory specified by target.

### Syntax:

```
#include<sys/mount.h>

/* ... */

void mount_sys() {
    if (0 != mount("none", "/sys", "sysfs", 0, "")) {
        /* handle error */
    }
}
```

### chown()

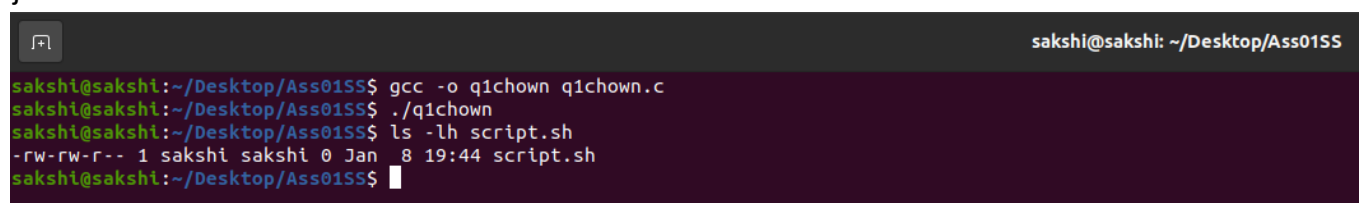
These system calls change the owner and group of the file specified by path or by fd. Only a privileged process (Linux: one with the CAP\_CHOWN capability) may change the owner of a file. The owner of a file may change the group of the file to any group of which that owner is a member.

### Source Code:

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

int main(int argc, char const *argv[]) {
    uid_t owner = 01000;
    uid_t group = 01000;

    chown("./script.sh", owner, group);
    return 0;
}
```



```
sakshi@sakshi: ~/Desktop/Ass01SS
sakshi@sakshi:~/Desktop/Ass01SS$ gcc -o q1chown q1chown.c
sakshi@sakshi:~/Desktop/Ass01SS$ ./q1chown
sakshi@sakshi:~/Desktop/Ass01SS$ ls -lh script.sh
-rw-rw-r-- 1 sakshi sakshi 0 Jan  8 19:44 script.sh
sakshi@sakshi:~/Desktop/Ass01SS$
```