**Experiment No. 9**

**Title:** File Handling System Calls (Low level system calls)

**Aim:** Implement an assignment using File Handling System Calls (Low level system calls like open, read, write, etc).

# **Theory:**

# **UNIX System Calls**

A system call is just what its name implies -- a request for the operating system to do something on behalf of the user's program. The system calls are functions used in the kernel itself. To the

programmer, the system call appears as a normal C function call. However since a system call executes code in the kernel, there must be a mechanism to change the mode of a process from user mode to kernel mode.

The C compiler uses a predefined library of functions (the C library) that have the names of the system calls. The library functions typically invoke an instruction that changes the process execution mode to kernel mode and causes the kernel to start executing code for system calls. The instruction that causes the mode change is often referred to as an "operating system trap" which is a software generated interrupt.

The library routines execute in user mode, but the system call interface is a special case of an interrupt handler. The library functions pass the kernel a unique number per system call in a machine dependent way either as a parameter to the operating system trap, in a particular register, or on the stack -- and the kernel thus determines the specific system call the user is invoking. In handling the operating system trap, the kernel looks up the system call number in a table to find the address of the appropriate kernel routine that is the entry point for the system call and to find the number of parameters the system call expects. The kernel calculates the (user) address of the first parameter to the system call by adding (or subtracting, depending on the direction of stack growth) an offset to the user stack pointer, corresponding to the number of the parameters to the system call.

Finally, it copies the user parameters to the "u area" and call the appropriate system call routine. After executing the code for the system call, the kernel determines whether there was an error. If so, it adjusts register locations in the saved user register context, typically setting the "carry" bit for the PS (processor status) register and copying the error number into register 0 location. If there were no errors in the execution of the system call, the kernel clears the "carry" bit in the PS register and copies the appropriate return values from the system call into the locations for registers 0 and 1 in the saved user register context. When the kernel returns from the operating system trap to user mode, it returns to the library instruction after the trap instruction. The library interprets the return values from the kernel and returns a value to the user program.

UNIX system calls are used to manage the file system, control processes, and to provide interprocess communication. The UNIX system interface consists of about 80 system calls (as UNIX evolves this number will increase).

The following table lists some important system call:

**GENERAL CLASS SYSTEM CALL**

**---------------------------------------------------------------------**

File Structure creat()

Related Calls open() close()

Input /Output read() write()

Random Access lseek()

Channel Duplication dup()

Aliasing and Removing link()

Files unlink()

File Status stat() fstat()

Access Control access()

chmod()

chown()

umask()

Device Control ioctl()

Following are some examples in the use of the most often used system calls.

**File Structure Related System Calls**

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.

When doing I/O, a process specifies the file descriptor for an I/O channel, a buffer to be filled or emptied, and the maximum size of data to be transferred. An I/O channel may allow input, output, or both.

Furthermore, each channel has a read/write pointer. Each I/O operation starts where the last operation finished and advances the pointer by the number of bytes transferred. A process can access a channel's data randomly by changing the read/write pointer.

All input and output operations start by opening a file using either the "creat()" or "open()" system calls. These calls return a file descriptor that identifies the I/O channel. Recall that file

descriptors 0, 1, and 2 refer to standard input, standard output, and standard error files respectively, and that file descriptor 0 is a channel to your terminal's keyboard and file descriptors 1 and 2 are channels to your terminal's display screen.

**creat()**

The prototype for the creat() system call is:

int creat(file\_name, mode)

char \*file\_name;

int mode;

where file\_name is pointer to a null terminated character string that names the file and mode defines the file's access permissions. The mode is usually specified as an octal number such as 0666 that would mean read/write permission for owner, group, and others or the mode may also

be entered using manifest constants defined in the "/usr/include/sys/stat.h" file. If the file named by file\_name does not exist, the UNIX system creates it with the specified mode permissions.

However, if the file does exist, its contents are discarded and the mode value is ignored. The permissions of the existing file are retained. Following is an example of how to use creat():

/\* creat.c \*/

#include <stdio.h>

#include <sys/types.h> /\* defines types used by sys/stat.h \*/

#include <sys/stat.h> /\* defines S\_IREAD & S\_IWRITE \*/

int main()

{

int fd;

fd = creat("datafile.dat", S\_IREAD | S\_IWRITE);

if (fd == -1)

printf("Error in opening datafile.dat\n");

else

{

printf("datafile.dat opened for read/write access\n");

printf("datafile.dat is currently empty\n");

}

close(fd);

exit (0);

}

The following is a sample of the manifest constants for the mode argument as defined in /usr/include/sys/stat.h:

#define S\_IRWXU 0000700 /\* -rwx------ \*/

#define S\_IREAD 0000400 /\* read permission, owner \*/

#define S\_IRUSR S\_IREAD

#define S\_IWRITE 0000200 /\* write permission, owner \*/

#define S\_IWUSR S\_IWRITE

#define S\_IEXEC 0000100 /\* execute/search permission, owner \*/

#define S\_IXUSR S\_IEXEC

#define S\_IRWXG 0000070 /\* ----rwx--- \*/

#define S\_IRGRP 0000040 /\* read permission, group \*/

#define S\_IWGRP 0000020 /\* write " " \*/

#define S\_IXGRP 0000010 /\* execute/search " " \*/

#define S\_IRWXO 0000007 /\* -------rwx \*/

#define S\_IROTH 0000004 /\* read permission, other \*/

#define S\_IWOTH 0000002 /\* write " " \*/

#define S\_IXOTH 0000001 /\* execute/search " " \*/

Multiple mode values may be combined by or'ing (using the | operator) the values together as demonstrated in the above sample program.

**open()**

Next is the open() system call. open() lets you open a file for reading, writing, or reading and writing.

The prototype for the open() system call is:

#include <fcntl.h>

int open(file\_name, option\_flags [, mode])

char \*file\_name;

int option\_flags, mode;

where file\_name is a pointer to the character string that names the file, option\_flags represent the type of channel, and mode defines the file's access permissions if the file is being created.

The allowable option\_flags as defined in "/usr/include/fcntl.h" are:

#define O\_RDONLY 0 /\* Open the file for reading only \*/

#define O\_WRONLY 1 /\* Open the file for writing only \*/

#define O\_RDWR 2 /\* Open the file for both reading and writing\*/

#define O\_NDELAY 04 /\* Non-blocking I/O \*/

#define O\_APPEND 010 /\* append (writes guaranteed at the end) \*/

#define O\_CREAT 00400 /\*open with file create (uses third open arg) \*/

#define O\_TRUNC 01000 /\* open with truncation \*/

#define O\_EXCL 02000 /\* exclusive open \*/

Multiple values are combined using the | operator (i.e. bitwise OR). Note: some combinations are mutually exclusive such as: O\_RDONLY | O\_WRONLY and will cause open() to fail. If the O\_CREAT flag is used, then a mode argument is required. The mode argument may be specified in the same manner as in the creat() system call.

**close()**

To close a channel, use the close() system call. The prototype for the close() system call is:

int close(file\_descriptor)

int file\_descriptor;

where file\_descriptor identifies a currently open channel. close() fails if file\_descriptor does not identify a currently open channel.

**read() write()**

The read() system call does all input and the write() system call does all output. When used together, they provide all the tools necessary to do input and output sequentially. When used with the lseek() system call, they provide all the tools necessary to do input and output randomly.

Both read() and write() take three arguments.

Their prototypes are:

int read(file\_descriptor, buffer\_pointer, transfer\_size)

int file\_descriptor;

char \*buffer\_pointer;

unsigned transfer\_size;

int write(file\_descriptor, buffer\_pointer, transfer\_size)

int file\_descriptor;

char \*buffer\_pointer;

unsigned transfer\_size;

where file\_descriptor identifies the I/O channel, buffer\_pointer points to the area in memory where the data is stored for a read() or where the data is taken for a write(), and transfer\_size defines the maximum number of characters transferred between the file and the buffer.

read() and write() return the number of bytes transferred.

There is no limit on transfer\_size, but you must make sure it's safe to copy transfer\_size bytes to or from the memory pointed to by buffer\_pointer. A transfer\_size of 1 is used to transfer a byte at a time for so-called "unbuffered" input/output. The most efficient value for transfer\_size is the size of the largest physical record the I/O channel is likely to have to handle. Therefore, 1K bytes -- the disk block size -- is the most efficient general-purpose buffer size for a standard file. However, if you are writing to a terminal, the transfer is best handled in lines ending with a newline.

For an example using read() and write(), see the above example of open().

**lseek()**

The UNIX system file system treats an ordinary file as a sequence of bytes. No internal structure is imposed on a file by the operating system. Generally, a file is read or written sequentially -- that is, from beginning to the end of the file. Sometimes sequential reading and writing is not appropriate. It may be inefficient, for instance, to read an entire file just to move to the end of the file to add characters. Fortunately, the UNIX system lets you read and write anywhere in the file. Known as "random access", this capability is made possible with the lseek() system call. During file I/O, the UNIX system uses a long integer, also called a File Pointer, to keep track of the next byte to read or write. This long integer represents the number of bytes from the beginning of the file to that next character. Random access I/O is achieved by changing the value of this file pointer using the lseek() system call.

The prototype for lseek() is:

long lseek(file\_descriptor, offset, whence)

int file\_descriptor;

long offset;

int whence;

where file\_descriptor identifies the I/O channel and offset and whence work together to describe how to change the file pointer according to the following table:

whence new position

---------------------------------------------------------------

0 offset bytes into the file

1 current position in the file plus offset

2 current end-of-file position plus offset

If successful, lseek() returns a long integer that defines the new file pointer value measured in bytes from the beginning of the file. If unsuccessful, the file position does not change. Certain devices are incapable of seeking, namely terminals and the character interface to a tape drive. lseek() does not change the file pointer to these devices.

Following is an example using lseek():

/\* lseek.c \*/

#include <stdio.h>

#include <fcntl.h>

int main()

{

int fd;

long position;

fd = open("datafile.dat", O\_RDONLY);

if ( fd != -1)

{

position = lseek(fd, 0L, 2); /\* seek 0 bytes from end-of-file \*/

if (position != -1)

printf("The length of datafile.dat is %ld bytes.\n", position);

else

perror("lseek error");

}

else

printf("can't open datafile.dat\n");

close(fd);

}

**Steps for programming:**

Implement an assignment using File Handling System Calls (Low level system calls like open, read, write, etc).

1. Create File
2. Open File
3. Insert values
4. View File

**OUTPUT:**

1. Enter the name of the file you wish to create
2. File will be Successfully created for reading and writing
3. File is currently empty
4. Enter the name of file you wish to write
5. Enter the message
6. Message will be successfully written in the file
7. Enter the name of file you wish to read
8. File opened for reading

**FAQs**

1. Explain difference between high level system call and low level system calls.
2. Explain open() system call with all parameters.
3. Explain read() system call with all parameters.
4. Explain write() system call with all parameters.
5. Explain file creation flags O\_CREAT, O\_EXCL, O\_NOCTTY, O\_TRUNC, O\_APPEND.
6. Explain different modes with open() system calls.
7. Explain lseek().

**Reference**

http://www.tutorialspoint.com/unix\_system\_calls