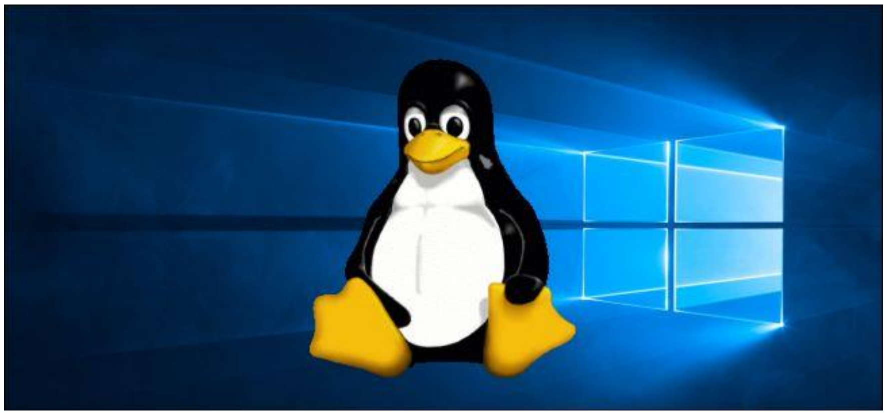
UNIVERSITY OF ENGINEERING AND TECHNOLOGY, TAXILA

FACULTY OF TELECOMMUNICATION AND INFORMATION ENGINEERING

SOFTWARE ENGINEERING DEPARTMENT



Operating Systems

Experiment 7

System Calls In UNIX

CLO 2. Use modern tools and languages.
CLO 3. Demonstrate an original solution of problem under discussion.
CLO 4. Work individually as well as in teams

System Call:

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.**

Process related system calls:

- fork()
- wait()
- exec()
- exit()

File Structure related System Calls:

- open()
- creat()
- read()
- write()
- close()

Creat() System Call:

The prototype for the creat() system call is:

```
int creat(file_name, mode)
char *file_name;
int mode;
```

- filename is pointer to a **null terminated character string** that names the file.
- 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.

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

```
#define S_IREAD 0000400     /* read permission, owner */
#define S_IWRITE 0000200    /* write permission, owner */
#define S_IEXEC 0000100     /* execute/search permission, owner */
```

Code for Creat System Call

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

   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);
   }
```

open()

- open() lets you open a file for reading, writing, or reading and writing.
- In most modern operating systems, a program that needs access to a file stored in a file system uses the open system call. After using the file, the **close system** call should be used. In several operating systems this system call can also be used to create a non-existent file.

```
int open(pathname, oflag , /* mode_t mode */);
```

- The **pathname** argument is a file path indicating in which place of file system, the file should be found (or created if that is requested) or a **filename**; file_name is a pointer to the character string that names the file
- The second argument **oflag** specifies the status of flags.
  '**oflag**' value is constructed by **Oring various flags**: O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_CREAT, etc.
- **mode** specifies permission on the file if it is to be created.

Example:

- open("hello.c",O_RDWR) // hello.c is opened and is ready for reading and writing.
- If **successful open returns non negative integer** else return -1.

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_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 */

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**.

read() and write()

UNIX provides sequential access to files and other devices through the read and write function. 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**. Both read() and write() take **three arguments**.

File Descriptors:
Standard Input: 0
Standard Output: 1

read() System Call:

int read(int fildes,void *buf,ssize_t nbytes);

- fildes identifies the I/O channel.
- buf points to the area in memory where the data is stored for a read().
- ssize_t nbytes defines the maximum number of bytes to read.

The read function actually attempts to retrieve nbyte (third argument) from the file represented by fildes (first argument) into the user variable buf (second argument).You must actually provide a buffer that is large enough to hold nbytes of data.

- If successful, **read returns the number of bytes actually read**.
- If unsuccessful, read returns -1.

Example:

```
char buf[200];
open(“hello.c”,O_RDWR);
read(“hello.c”,buf,100);
```

100 bytes are read from the file named “hello.c” into the temporary buffer buf, whose capacity to save data is 200 bytes.

write() System Call:

int write(int fildes,void *buf,ssize_t nbytes);

- fildes identifies the I/O channel.

- buf points to the area in memory from where the data is taken for a write().
- ssize_t nbytes defines the maximum number of bytes to write.

The write function attempts to **output nbytes (third argument) from the user buffer buf(second argument) to the file represented by file descriptor fildes** (first argument).

- If unsuccessful, write returns negative value.
- If **successful, write returns the number of bytes actually written**, this value can be less then nbytes if OS buffer get full.

Example:

```
char buf[200];
int bytesread;
open("hello.c",O_RDWR);
bytesread=read("hello.c",buf,100);
int n= write(STDOUT_FILENO,buf,bytesread);
```

The number of bytes read is saved in integer variable "bytesread". The data is taken from "buf" and then displayed at standard output. n is number of bytes actually written.

close() System Call

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

int close(file_descriptor);

The close function has a single parameter, file_descriptor (single argument), representing the open file whose resources are to be released.

- If successful "close" returns 0 else return -1.

Implementation of open(), read(), write() and close() System Calls

```
#include<stdio.h>

#include<fcntl.h>

int main()
{
```

```
int fd;

char buffer[80];

static char message[] = "Hello, world";

fd = open("file.txt",O_RDWR);

if (fd != -1)

{

printf("file.txt opened for read/write access\n");

write(fd, message, sizeof(message));

lseek(fd, 0, 0);   /* go back to the beginning of the file */

read(fd, buffer, sizeof(message));

printf(" %s was written to file.txt \n", buffer);

close (fd);

}

}
```

lseek():

It is the **system call to change the location of read/write pointe**r of a given file descriptor.

off_t lseek(int file_descriptor, off_t offset, int whence);

1. file_descriptor (The file whose current file offset you want to change).
2. offset (The amount (positive or negative) the byte offset is to be changed. The sign indicates whether the offset is to be moved forward (positive) or backward (negative).
3. Whence: One of the following symbols:

1. SEEK_SET: The start of the file
2. SEEK_CUR: The current file offset in the file
3. SEEK_END: The end of the file

   Or

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.**

See what is the result???

```
#include<stdio.h>
#include<fcntl.h>
int main()
  {
     int fd;
     char buffer[80];
     static char message[] = "Hello, world";
     fd = open("myfile.txt",O_RDWR|O_CREAT);

     if (fd != -1)
     {
       printf("myfile opened for read/write access\n");
       write(fd, message, sizeof(message));
       lseek(fd, 0, 0);                    /* go back to the beginning of the file */
       read(fd, buffer, sizeof(message));
       printf(" %s was written to myfile \n", buffer);
       close (fd);
     }
 }
```

LAB TASKS

User Input in C:

```
char file_name[20];
printf("Enter a file_name:");
scanf("%s", file_name);
```

TASK 1:

Write "read" and "write" commands for the following conditions

- Read data from **standard input** and write it on **standard output.**
- Read data from the file, **given as an argument by the user** and write it on standard output.
- Read data from standard input and write it on the file **given as an argument by the user**.

TASK 2:

Examine the program, then translate and execute it.

Copy the contents of an input file to an output file using system calls.

```
#include<stdlib.h>
#include<unistd.h>
#include<strings.h>
#include<limits.h>
#include<fcntl.h>
#include<sys/stat.h>
#define BUFFER_SIZE 64
 int main()
{
  char buffer[BUFFER_SIZE];
  int in_fd;
  int in_bytes;
  int out_fd;
  in_fd=open("lab3.txt",O_RDONLY);
  out_fd=open("lab4.txt",O_RDWR|O_CREAT|O_TRUNC,S_IRUSR|S_IWUSR);
for(;;)
{
```

```
in_bytes=read(in_fd,buffer,BUFFER_SIZE);
if(in_bytes==0)break;
write(out_fd,buffer,in_bytes);
}
close(in_fd);
close(out_fd);
exit(0);}
```

i. What is the effect of executing this program?

ii. How end of files is detected by program?

iii. Describe the arguments used in the first call to “open”.
in_fd=open(“lab3.txt”,O_RDONLY);

iv. Describe the arguments used in second call to “open”.
out_fd=open(“lab4.txt”,O_RDWR|O_CREAT|O_TRUNC,S_IRUSR|S_IWUSR);