

Lab 3: System Calls

Introduction

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

- `open()`
- `read()`
- `write()`
- `close()`
- `wait()`
- `fork()`
- `exec()`

Objective

- To be familiar with system calls.
- To be able to use system calls.
- To be able to read/write.
- To be able to open/close.
- To be able to fork and execute.

Concept Map

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

open()

`open()` lets you open a file for reading, writing, or reading and writing. Example of this system call is as following.

`int open(file_name, mode)`

Where `file_name` is a pointer to the character string that names the file and `mode` defines the file's access permissions if the file is being created.

Read() and 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. 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 data which is going to be written.

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;

```

Implementation of open(), read(), write() and close() functions:

```

#include<stdio.h>
#include<unistd.h>
#include<fcntl.h>
int main()
{
int fd;
char buffer[80];
static char message[] = "Hello,
world";
fd = open("myfile.txt",O_RDWR);
if (fd != -1)
{
printf("myfile opened for read/write access
\n");
write(fd, message, sizeof(message));
lseek(fd, 5, 0);
/* go back to the beginning of the file */
read(fd, buffer, sizeof(message));
printf(" %s was written to myfile \n",
buffer);
close (fd);
}
else
{
printf(" Error in openng myfile \n");
}
}

```

fork()

When the fork system call is executed, a new process is created which consists of a copy of the address space of the parent.

The return code for fork is zero for the child process and the process identifier of child is returned to the parent process. On success, both processes continue execution at the instruction after the fork call. On failure, -1 is returned to the parent process.

Implementing fork system call using C program

```
#include <sys/types.h>
void main()
{
    pid_t pid;
    pid = fork();
    if (pid == 0)
        printf("\n I'm the child process");
    else if (pid > 0)
        printf("\n I'm the parent process. My child pid is %d", pid);
    else
        perror("error in fork");
}
```

wait()

The wait system call suspends the calling process until one of its immediate children terminates. If the call is successful, the process ID of the terminating child is returned.

Zombie process—a process that has terminated but whose exit status has not yet been received by its parent process. The process will remain in the operating system's process table as a zombie process, indicating that it is not to be scheduled for further execution.

But that it cannot be completely removed (and its process ID cannot be reused)

```
pid_t wait(int *status);
```

Where status is an integer value where the UNIX system stores the value returned by child process.

Implementing wait system call using C program

```
#include <stdio.h>
void main()
{
    int pid, status;
    pid = fork();

    if (pid == -1)
    {
        printf("fork failed\n");
        exit(1);
    }
    if (pid == 0)
```

```

{
/* Child */
printf("Child here!\n");
}
else
{
/* Parent */
wait(&status);
printf("Well done kid!\n");
}
}

```

exec()

Typically, the exec system call is used after a fork system call by one of the two processes to replace the process' memory space with a new executable program. The new process image is constructed from an ordinary, executable file. There can be no return from a successful exec because the calling process image is overlaid by the new process image.

Takes the path name of an executable program (binary file) as its first argument. The rest of the arguments are a list of command line arguments to the new program (argv[]). The list is terminated with a null pointer: `execl("a.out", "a.out", NULL)`

Procedure & Tools

In this section, you will study how to setup and VMware.

Tools

- Download and install Virtual Box
- Download and install Ubuntu

Walkthrough Task

This section will provide a practice task which you need to finish during the lab. You need to finish the tasks in the required time.

Task:

Make a text file with "myfile" name in a folder by typing the following command:

cat > myfile.txt

Then make a new c type file to code, to do so type the following command:

gedit code.c

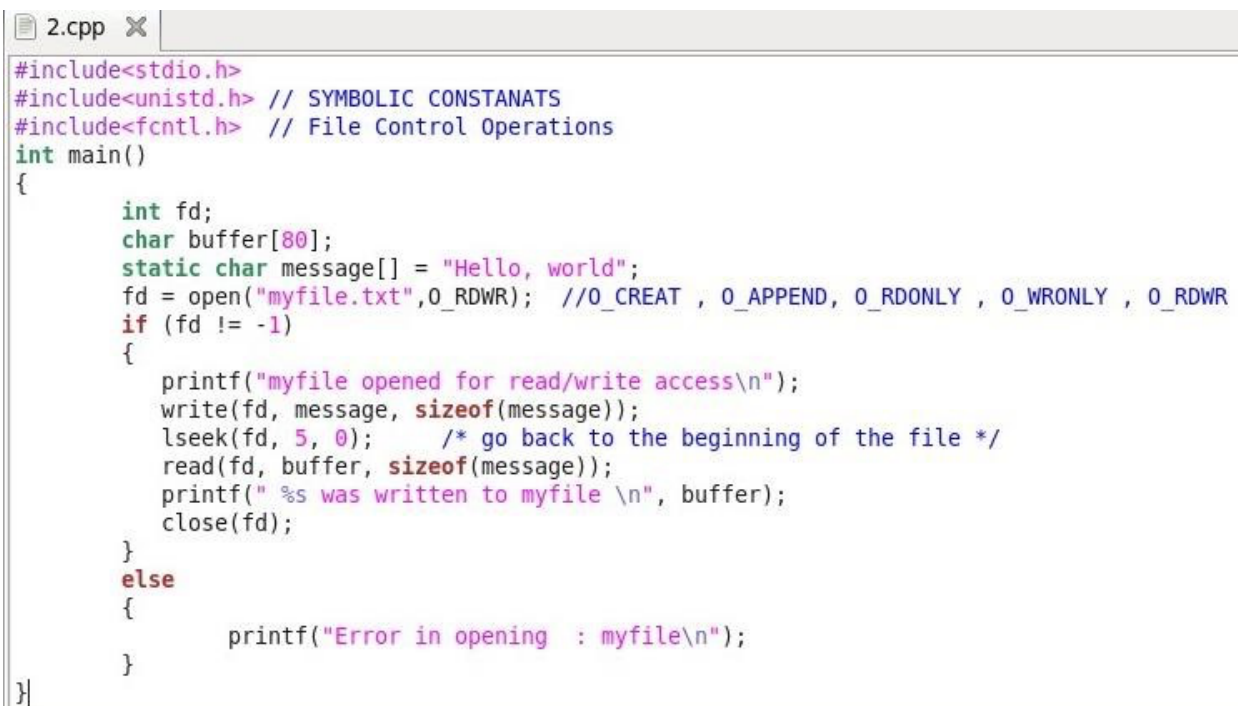
Now type all the following code into that file.

```

#include<stdio.h>
#include<unistd.h>
#include<fcntl.h>
int main()
{
int fd;
char
buffer[80];

static char message[] = "Hello, world";
fd = open("myfile.txt",O_RDWR);
if (fd != -1)
{
printf("myfile opened for read/write access \n");
write(fd, message, sizeof(message));
lseek(fd, 5, 0);
/* go back to the beginning of the file */
read(fd, buffer, sizeof(message));
printf(" %s was written to myfile \n", buffer);
close (fd);
}
else
{
printf(" Error in openng myfile \n");
}
}

```



```

2.cpp
#include<stdio.h>
#include<unistd.h> // SYMBOLIC CONSTANATS
#include<fcntl.h> // File Control Operations
int main()
{
    int fd;
    char buffer[80];
    static char message[] = "Hello, world";
    fd = open("myfile.txt",O_RDWR); //O_CREAT , O_APPEND, O_RDONLY , O_WRONLY , O_RDWR
    if (fd != -1)
    {
        printf("myfile opened for read/write access\n");
        write(fd, message, sizeof(message));
        lseek(fd, 5, 0); /* go back to the beginning of the file */
        read(fd, buffer, sizeof(message));
        printf(" %s was written to myfile \n", buffer);
        close(fd);
    }
    else
    {
        printf("Error in opening : myfile\n");
    }
}

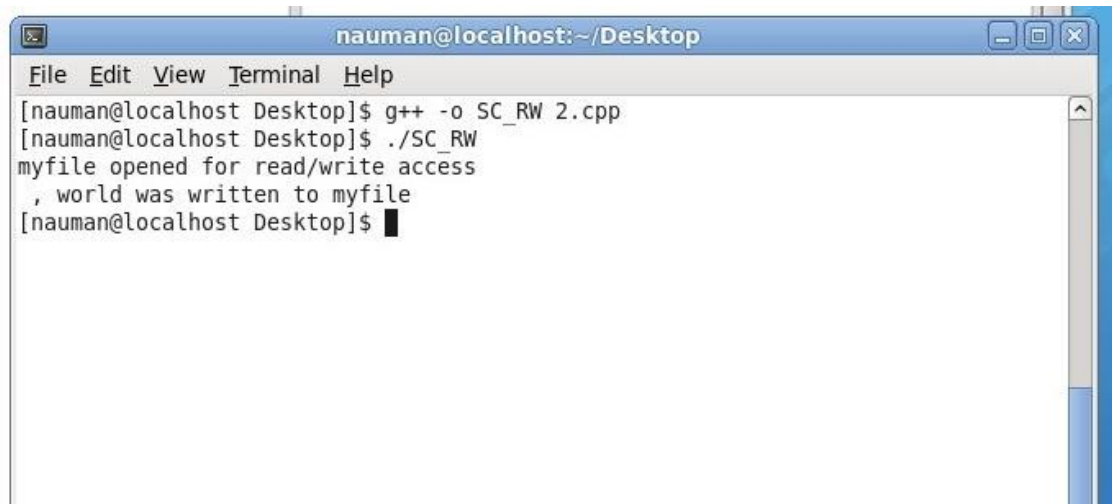
```

To run this .c file, you have to first install GCC library on your system. To install gcc, give the following command on terminal:

\$ sudo apt-get install build-essential

Give the following command on terminal:

gcc code.c -o code ./code

A terminal window titled 'nauman@localhost:~/Desktop' with a menu bar (File, Edit, View, Terminal, Help). The terminal shows the following commands and output:

```
[nauman@localhost Desktop]$ g++ -o SC_RW 2.cpp
[nauman@localhost Desktop]$ ./SC_RW
myfile opened for read/write access
, world was written to myfile
[nauman@localhost Desktop]$
```

Practice Tasks

This section will provide more practice exercises which you need to finish during the lab. You need to finish the tasks in the required time. When you finish them, put these tasks in the following folder:

GitHub Folder Name: "OS_3111_Lab_folder".

Practice Task 1

[Expected time = 15mins]

Write your university card's details in a text file using write system call. Now read the contents of the file using read system call.

Practice Task 2

[Expected time = 15mins]

Write a code to start a process and create its child processes. Now display details of these processes one by one using system calls.

Out comes

Evaluation Task (Unseen)

[Expected time = 30mins]

The lab instructor will give you unseen task depending upon the progress of the class.

Evaluation criteria

The evaluation criteria for this lab will be based on the completion of the following tasks. Each task is assigned the marks percentage which will be evaluated by the instructor in the lab whether the student has finished the complete/partial task(s).

Table 3: Evaluation of the Lab

Sr. No.	Task No	Description	Marks
1	1	Problem Modeling	20
2	2	Procedures and Tools	10
3	3	Practice tasks and Testing	35
4	4	Evaluation Tasks (Unseen)	20
5		Comments	5
6		Good Programming Practices	10

Further Reading

This section provides the references to further polish your skills.

Slides

The slides and reading material already shared in WhatsApp Group.