National University of Computer and Emerging Sciences

**Laboratory Manual**

*for*

**Operating Systems Lab**

**(CL-220)**

| | |
|---|---|
| Instructor | Dr. Rana Asif Rehman |
| Lab Instructor(s) | Mr. Sohaib Ahmed<br>Ms. Haiqa Saman |
| Section | 4B |
| Semester | Spring 2023 |

Department of Computer Science

FAST-NU, Lahore, Pakistan

## Objectives

In this lab, students will practice:

1. Exec system call
2. Reading and writing files using open, write, and read system calls

**Important Note:**

- **Comment your code intelligently.**
- **Indent your code properly.**
- **Use meaningful variable names.**
- **Use meaningful prompt lines/labels for input/output.**
- **Use meaningful project and C/C++ file name**

# 1 Types of System Calls:

There are 5 different categories of system calls

1. Process control

2. File management

3. Device management

4. Information maintenance

5. Communication

Unix and Windows have different system calls as can be seen from the table below

# 2 Process Control

- Fork()
- Exec()
- Wait()
- Exit()

## 2.1 Exec ()

The exec system call is used to execute a file which is residing in an active process. When exec is called the previous executable file is replaced and new file is executed.

More precisely, we can say that using exec system call will replace the old file or program from the process with a new file or program. The entire content of the process is replaced with a new program.

The user data segment which executes the exec() system call is replaced with the data file whose name is provided in the argument while calling exec().

The new program is loaded into the same process space. The current process is just turned into a new process and hence the process id PID is not changed, this is because we are not creating a new process we are just replacing a process with another process in exec.

PID of the process is not changed but the data, code, stack, heap, etc. of the process are changed and are replaced with those of newly loaded process. The new process is executed from the entry point.

Exec system call is a collection of functions and in C programming language, the standard names for these functions are as follows:

- execl
- execle
- execlp
- execv
- execve
- execvp

It should be noted here that these functions have the same base *exec* followed by one or more letters. These are explained below:

**l:** l is for the command line arguments passed a list to the function

**p:** p is the path environment variable which helps to find the file passed as an argument to be loaded into process.

**v:** v is for the command line arguments. These are passed as an array of pointers to the function

## 2.2 Syntaxes of exec family functions:

The following are the syntaxes for each function of exec:
int execlp(const char* file, const char* arg, …)
int execv(const char* path, const char* argv[])

## 2.3 Execv ()

In execv() ,system call doesn't search the PATH. Instead, the full path to the new executable must be specified.

**Path:**

The path to the new program executable.

**Argv:**

Argument vector. The argv argument is an array of character pointers to null-terminated strings. The last member of this array must be a null pointer. These strings constitute the argument list available to the new process image. The value in argv[0] should point to the filename of the executable for the new program.

```
#include <unistd.h>

int main(void) {
  char *binaryPath = "/bin/ls";
  char *args[] = {binaryPath, "-lh", "/home", NULL};

  execv(binaryPath, args);

  return 0;
}
```

## 2.4 Execlp ()

execlp() uses the PATH environment variable. So, if an executable file or command is available in the PATH, then the command or the filename is enough to run it, the full path is not needed.

```
#include <unistd.h>

int main(void) {
  char *programName = "ls";
  char *arg1 = "-lh";
  char *arg2 = "/home";

  execlp(programName, programName, arg1, arg2, NULL);

  return 0;
}
```

### 2.4.1 Example 1: Using exec system call in C program

Consider the following example in which we have used exec system call in C programming in Linux, Ubuntu: We have two c files here example.c and hello.c:

#### *2.4.1.1 example.c*

**CODE:**

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
   printf("PID of example.c = %d\n", getpid());
   char *args[] = {"/hello.c", "C", "Programming", NULL};
   execv(args[0], args);
   printf("Back to example.c");
   return 0;
}
```

#### *2.4.1.2 hello.c*

**CODE:**

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
   printf("We are in Hello.c\n");
   printf("PID of hello.c = %d\n", getpid());
   return 0;
}
```

**OUTPUT:**

```
PID of example.c = 4733
We are in Hello.c
PID of hello.c = 4733
```

### 2.4.2 Example 2: Combining fork() and exec() system calls

Consider the following example in which we have used both fork() and exec() system calls in the same program:

#### *2.4.2.1 example.c*

**CODE:**

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
   printf("PID of example.c = %d\n", getpid());
   pid_t p;
   p = fork();
   if(p==-1)
   {
      printf("There is an error while calling fork()");
   }
   if(p==0)
   {
   printf("We are in the child process\n");
   printf("Calling hello.c from child process\n");
   char *args[] = {"Hello", "C", "Programming", NULL};
   execv("./hello", args);
   }
   else
   {
      printf("We are in the parent process");
   }
   return 0;
}
```

### *2.4.2.2 hello.c:*

**CODE:**

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
   printf("We are in Hello.c\n");
   printf("PID of hello.c = %d\n", getpid());
   return 0;
}
```

OUTPUT:

```
PID of example.c = 4790
We are in Parent Process
We are in Child Process
Calling hello.c from child process
We are in hello.c
PID of hello.c = 4791
```

These are some errors you can encounter with:

**E2BIG**

The argument list and the environment is larger than the system limit of ARG_MAX bytes.

**EACCES**

The calling process doesn't have permission to search a directory listed in *file*, or it doesn't have permission to execute *file*, or *file*'s filesystem was mounted with the ST_NOEXEC flag.

**EINVAL**

The *arg0* argument is NULL.

**ELOOP**

Too many levels of symbolic links or prefixes.

**EMFILE**

Insufficient resources available to load the new executable image or to remap file descriptors.

**ENAMETOOLONG**

The length of *file* or an element of the *PATH* environment variable exceeds PATH_MAX.

**ENOENT**

One or more components of the pathname don't exist, or the *file* argument points to an empty string.

**ENOMEM**

The new process image requires more memory than is allowed by the hardware or system-imposed memory management constraints.

**ENOTDIR**

A component of *file* isn't a directory.

**EPERM**

The calling process doesn't have the required permission, or an underlying call to *mmap()* failed because it attempted to set PROT_EXEC for a region of memory covered by an untrusted memory-mapped file.

**ETXTBSY**

The text file that you're trying to execute is busy (e.g., it might be open for writing).

## 3 Open System call for Filing

Open system call is used for opening a file.
**int open(const char *****pathname***, int** *flags***, mode_t** *mode***);**

1. pathname is a file name
2. The argument *flags* must include one of the following *access modes* **O_RDONLY**, **O_WRONLY**, or **O_RDWR**. These request opening the file in read-only, write-only, or read/write modes, respectively. Apart from above, flags can also have any of the following:

**(A) O_APPEND (file is opened in append mode)**
**(B) O_CREAT** (If *pathname* does not exist, create it as a regular file.)
**(C) O_EXCL** Ensure that this call creates the file: if this flag is specified in conjunction with **O_CREAT**, and *pathname* already exists, then **open**() fails.

**Note: to use two flags at once use bitwise OR operator, i.e., O_WRONLY | O_CREAT**

3. **Mode is only required when a new file is created and is used to set permissions on the new file**

## 4 Read/Write System Call

The read () system call receives the user's input from the file that is the file descriptor and puts it in the buffer "buff", which is just a character array. It can only read up to specified number of bytes at a time.

read (int fd, void* buf, bytes);

write (int fd, void* buf, bytes);

## 5 Seek() System Call

Read system call reads from start. What if we want to read or write from a specific portion of file? Lseek() is used to read or write at specific portion of file.

Lseek(fd,bytes, SEEK_CURR)

Instead of SEEK_CURR, which is used to move pointer from current position upto specified bytes, you can also use SEEK_SET to set pointer from beginning of file and SEEK_END from end of file.

## 6 Inlab Question

**(Last Lab Question)**

- Create 2 processes using fork system call such that all the processes have same parent. In first process, use array of alphabets and special characters entered through command line. This process will print array and its size and then call exec system to pass the previous array and remove all special characters. So that, only alphabets remain in that array. This file further calls exec and pass updated array to this. In this program, you have to find number of elements in array and its size in bytes. In second fork you need to print PID of this process. PIDs of both exec and fork should be printed on your screen. Parent should wait until child has finished its execution.

**(New Question)**

- Design a program using lseek() to read alternative byte from a file. You have to create a txt file and write" OperatingSystem" in it and then add any alphabet after each letter in previously mentioned text. Using lseek() read alternative alphabets and write it on screen.

**Note**: Use only read, write and open system calls. Use of Cin, cout, prinf, ofstream, ifstream etc. will result in zero marks.

**Help:**

man 2 open

man 2 read

man 2 write