National University of Computer and Emerging Sciences

Laboratory Manual

for

Operating Systems Lab

Course Instructor	Namra Absar
Lab Instructor(s)	Ms. Haiqa Saman Mr. Zaeem Yousaf
Section	BCS-5C
Semester	Fall-2023

Department of Computer Science

FAST-NU, Lahore, Pakistan

Objectives

In this lab, students will practice:

1. Understand Process control system calls
2. Why we need system call
3. Difference between windows and Linux system calls
4. Working with Linux system calls
5. Creating process using Fork system call
6. Exec system call
7. 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**

Command Line Arguments:

Command-line arguments are a way to pass data to the program. Command-line arguments are passed to the main function. Suppose we want to pass two integer numbers to the main function of an executable program called a.out. On the terminal write the following line:

./a.out 1 22

./a.out is the usual method of running an executable via the terminal. Here 1 and 22 are the numbers we passed as command-line arguments to the program. These arguments are passed to the primary function. In order for the main function to be able to accept the arguments, we have to change the signature of the primary function as follows:

int main(int argc, char *argv[]);

argc is the counter. It tells how many arguments have been passed.

argv is the character pointer to our arguments.

argc in this case will not be equal to 2, but it will be equal to 3. This is because the name ./a.out

is also passed as command line argument. At index 0 of argv, we have ./a.out; at index 1, we have 1; and at index 2, we have 22. Here 1 and 22 are in the form of character string, we have to convert them to integers by using a function atoi. Suppose we want to add the passed numbers and print the sum on the screen:

```
cout<< atoi(argv[1]) + atoi(argv[2]);
```

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

a. What is a Process?

An instance of a program is called a Process. In simple terms, any command that you give to your Linux machine starts a new process.

Types of Processes:

- Foreground Processes: They run on the screen and need input from the user. For example Office Programs
- Background Processes: They run in the background and usually do not need user input. For example Antivirus.

  Init is the parent of all Linux processes. It is the first process to start when a computer boots up, and it runs until the system shuts down. It is the ancestor of all other processes.

Examples of Windows and Unix System Calls

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

1. Process Creation:

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination.

I. fork()

- Has a return value

- Parent process => invokes fork() system call
- Continue execution from the next line after fork()
- Has its own copy of any data

- Return value is > 0 //it's the process id of the child process. This value is different from the Parents own process id.
- Child process => process created by fork() system call
- Duplicate/Copy of the parent process //LINUX
- Separate address space
- Same code segments as parent process
- Execute independently of parent process
- Continue execution from the next line right after fork()
- Has its own copy of any data
- Return value is 0

II. wait ()

- Used by the parent process
- Parent's execution is suspended
- Child remains its execution
- On termination of child, returns an exit status to the OS
- Exit status is then returned to the waiting parent process //retrieved by wait ()
- Parent process resumes execution
- #include <sys/wait.h>
- #include <sys/types.h>

III. exit()

- Process terminates its execution by calling the exit() system call
- It returns exit status, which is retrieved by the parent process using wait() command
- EXIT_SUCCESS // integer value = 0
- EXIT_FAILURE // integer value = 1
-
- OS reclaims resources allocated by the terminated process (dead process) Typically performs clean-up operations within the process space before returning control back to the OS
- _exit()
- Terminates the current process without any extra program clean-up

- Usually used by the child process to prevent from erroneously release of resources belonging to the parent process

IV. execlp() is a version of exec()

- **exec()**
- The exec family of functions replaces the current running process with a new process. It can be used to run a C program by using another C program.an executable file
- Called by an already existing process //child process
- Replaces the previous executable //overlay
- Has an exist status but cannot return anything (if exec() is successful) to the program that made the call //parent process
- Return value is -1 if not successful
- Overlay => replacement of a block of stored instructions or data with another int

execlp(char const *file_path, char const *arg0,);

- Arguments beginning at arg0 are pointers to arguments to be passed to the new process.
- The first argument arg0 should be the name of the executable file.
- Example
- **execlp(/bin/ls , ls ,NULL) //lists contents of the directory**

a. **but exec or execlp is a system call which overwrites an already existing process (calling process), so if you want to execute some code after execlp system call, then write this system call in a child process of an existing process, so it only**

overwrite child process.

- Header file used -> unistd.h

2. Information Maintenance

i. sleep()

- Process goes into an inactive state for a time period
- Resume execution if
- Time interval has expired
- Signal/Interrupt is received
- Takes a time value as parameter (in seconds on Unix-like OS and in milliseconds on Windows OS)
- sleep(2) // sleep for 2 seconds in Unix
- Sleep(2*1000) // sleep for 2 seconds in Windows

ii. getpid() // returns the PID of the current process

-
- getppid() // returns the PID of the parent of the current process

-
- Header files to use
-
- #include <sys/types.h>

-
    - #include <unistd.h>
    -
    - getppid() returns 0 if the current process has no parent

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);


  return0;

}
```

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);

    return0;
}
```

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:

```
#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:

```
#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:

```
#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:

```
#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.

In Lab Tasks:

Question 1: Write a C or C++ program that accepts a file name and a substring as command line argument and prints the no of occurrences (Sequence also included) of substring in the given file on the console.If the file does not exist, print some error on the screen.

The file name is question1.txt with the following content:

“Hello, this is the command line argument practice and my second day in the lab of operating systems. I am enjoying it the alot. Hello, this is the command line argument practice and my second day in lab of operating systems. I am enjoying it the alot.Hello, this is the command line argument practice and my second day in the lab of operating systems. I am enjoying it the alot.”

Question 2: Write a program that uses two processes. One is called a parent and the other is called a child. The child read the same file (mentioned above) and finds the special symbols like (“, . ; :) in the file

While child is processing, Parent wait for child to finish with code and prints ‘Program completed’ message.

Note: Use Fork and File handling.