Processes

Objectives

1. To write C programs in Linux passing their input arguments via argc, argv
2. To write C programs in Linux that get the ID of a process as well as the ID of its parent process
3. To write C programs in Linux that clone themselves using fork routine
4. To write C programs in Linux that create a new process

The tasks of this lab session will be run on Linux.

Aim

The aim of this session is to familiarize yourselves with multiple processes and threads.

How to Compile and run a C program in Linux.

In this lab session we will be using gedit text editor to write our programs and the Linux Terminal to compile and run our programs. Please note that there are more efficient programming environments to develop software such as Eclipse (it is free), but for small programs, this is not necessary.

You can compile a .c file using the following command:

```
gcc source.c -o exec -other_options
```

gcc is a well-known and used compiler. The executable name is specified just after the '-o' ('o' stands for output). *'-other_options'* refers to other compilation options, e.g., '-lm' will include the 'math.h' library.

Then, you can run the executable by using the following command:

```
./exec
```

Passing Input arguments – Argc, Argv

So far, the programs we have written run with a single command, i.e., *'./exec'*. However, what if we want to pass information from the command line to the program we are running?

Up until now, the skeletons we have used for our C programs have looked something like this:

```
int main() { }
```

From now on, our examples may look a bit more like this:

```
int main (int argc, char *argv[]) { }
```

As you can see, main now has arguments. 'argc' stands for argument count while 'argv' stands for argument vector. 'argc' is the number of command-line arguments passed by the user including the name of the program, e.g., if we run program using the following command './exec hi', then argc equals to 2.

‘argv’ is an array of character pointers listing all the arguments. Thus, ‘argv’ is an array of pointers, which is an array where its elements are memory addresses; the first pointer (argv[0]) points to the first string, the second to the second etc.

Compile the input_arguments.c file using the following command gcc input_arguments.c -o exec. Then run the program normally, but add some parameters too :

```
./exec arg1 arg2
```

The output will be

```
argc = 3
arg[0] = "./p"
arg[1] = "arg1"
arg[2] = "arg2"
```

Task1. Study input_arguments.c program

Make sure you understand what this program does

Getting a process' process ID (PID)

Every process on the system has a unique process ID number, known as the pid. This is simply an integer. You can get the pid for a process via the getpid() system call.

Task2. Compile and run ex1.c.

This is a small program that simply prints its process ID number and exits.

Getting a parent process' process ID (PID)

Processes in Unix have a hierarchical relationship. When a process creates another process, that new process becomes a child of the process, which is known as its parent. You can get the parent's pid (process ID) for a process via the getppid system call.

Task3. Compile and run ex2.c.

This is a small program that simply prints its process ID number and its parent's process ID number and then exits.

Task4. Print the running processes in Linux terminal

Type the following command:

```
top
```

this command shows all the processes running; each process has a PID number, e.g., 293. If you want to kill process 293 type ‘*kill 293’.* Type q to exit

Type the following command:

ps –ef

it will display all the processes running, including the user's ID, the PID, the PPID (parent process id), its start time and full command line. 'top' command allows us to display the process statistics continuously until stopped while 'ps' gives us a single statistics snapshot.

Cloning processes (fork)

The fork system call allows us to clone a process. **When fork() is called, the operating system creates a new clone process, i.e., a process creates a copy of itself**. As soon as the fork call returns, both the parent and child are now running at the same point in the program. The only difference is that the child gets a different return value from fork. **The parent gets the process ID of the child that was just created. The child gets a return of 0. If fork returns -1 then the operating system was unable to create the process**.

It is important to note that nothing is shared after we call fork. Even though the child is running the same code and has the same files open, it maintains its own copy of all memory. If a child changes a memory location, the parent won't see the change (and vice versa).

Task4. Compile, run and study fork.c.

This is a small program that clones itself. The parent prints a message stating its process ID and the child's process ID. It gets its process ID via the getpid system call and it gets its child's process ID from the return of fork. The child prints its process ID. The parent and child then each exit. Note that exit takes a parameter. This becomes the exit code of the program. The convention for Unix systems is to exit with a code of zero on success and a non-zero on failure. This helps in scripting programs.

Task5. Compile, run and study fork2.c.

Make sure you understand how 'a' and 'b' variables change their values. Both the parent and the child process have their own 'a' and 'b' variables.

The following functions contain pointers and this why the source code will always be given to you. The aim of this session is to understand how processes work, not to develop such code. Before we study the next functions, let us revise the pointers and strings in C. A pointer is a memory address. A string in C is an array of characters and can be defined in two different ways.

- Char string1[] = "hi y"; In this case, an array is created large enough to hold the string; the array needs 5 characters here. Note that the last element of each array is the '\0' (zero) character. There is no need to explicitly define the number of characters, e.g., char string [5]="hi y";. However, this is correct too. Note that the string will be allocated into the stack.
- Char *string2 = "hi y"; In this case, string2 is a pointer (memory address). This is the memory address of the memory location that the string is stored into memory. The string will not be stored into the stack, but into the data segment. The pointer will be stored into the stack though. A string pointed by a pointer cannot change, because it is located in a read only memory segment.

String1: | h | i | | y | \0 |

String2: mem.address -> | h | i | | y | \0 |

EXECVE system call and its variations

The fork system call creates a new process, by copying itself. Thus, the child process executes exactly the same code as its parent. What if we need to create a process to run another program? **The execve() system call replaces the current process with a new program.** This system call returns only if the named program could not run. execve() replaces the current running process with a new process.

Task6: Run program1 via program2.

Program1.c contains a very simple program that prints 'Hi from program 1'. Compile program1.c and name the executable 'program1'. Now, compile and run program2; it will call the executable 'program1'. Program2 contains 'execve("program1",args,NULL);' command. Compile and run program2.c. You will realize that process2 is replaced by process1.

The 'execve' function is defined as follows

*int execve(const char *path, char *const argv[], char *const envp[]);*

Note that the 1st parameter is a constant string, which is the file name of the binary to execute. The 2nd parameter is an array of pointers to strings (as in task1) passed to the new program (the last item in the list must be zero). The 3rd argument is always zero (not studied in this module).

Task7: Run program3 via program4.

Program3.c contains a very simple program that prints the argv arguments. Compile program3.c and name the executable 'program3'. Run program3 as follows

```
./program3 word1 word2
```

We will realize that word1 and word2 are passed as input to program3, and they are printed. This is what program3 does.

Now, program4 will call the executable 'program3'. To do so, program4 contains '*execve("program3",temp,NULL);*' command. 'temp' contains the operands to pass as input to program3. Compile and run program4.c. You will realize that process4 is replaced by process3.

Variations of execve()

In addition to the execve system call, there are a number of library routines similar to execve such as execlp and execvp.

execlp, which allows you to specify all the arguments as parameters to the function. It is defined as follows

int execlp(const char **file*, const char **arg0*, ..., const char **argn*, (char *)0);

Note that the first parameter is the name of the binary file to execute. Then, a variable length list of arguments is passed. The first argument, arg0, is required and must contain the name of the executable file for the new process image. The last parameter must be a null pointer.

Task8: Compile and run execlp.c.

This program uses execlp to execute the command echo.

Task9. Study the fork_exec.c program.

The fork system call creates a new process. The execve system call overwrites a process with a new program. A process forks itself and the child process executes a new program, which overlays the one in the current process. Because the child process is initially running the same code that the parent was and has the same open files and the same data in memory, this scheme provides a way for the child to tweak the environment, if necessary.

This is a program that forks itself and the child executes the echo command as before. Five seconds later, the parent prints a message saying, I'm still here!. We use the sleep library function to put the process to sleep for five seconds.

Further Reading

1. The fork system call, available at https://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html
2. Fork in C, available at https://www.geeksforgeeks.org/fork-system-call/