Department of Systems and Computer Engineering, Carleton University
SYSC 4001 Operating Systems Winter 2018-19: Lab Exercise

Lab Exercise 2 - Linux Basics and Processes

Lab Exercise 2 is mainly based on the materials of the Linux reference book, *Beginning Linux Programming*, 4th ed. If not mentioned otherwise the pages referenced are pages from this book.

- More Linux basics – Shell programming (Chap. 2)
- The Standard I/O Library (Chap. 3)
- Command line arguments – passing arguments to programs, and getting system time (Chap. 4)
- Process creation

I. More Linux Basics and Standard I/O

1. What is a Shell?

Unix/Linux Shell is well known. Go over the brief description of Unix Shell at the beginning of Chapter 2 (p. 17 – p. 19) to understand what it is.

2. Pipes and Redirection

By default, the standard input (*stdin*) is read from the keyboard and output is displayed on the screen or a particular terminal, which is the standard output (*stdout*). The default settings can be changed using the "<" for input redirection and ">" for output redirection.

Redirecting Input and Output (pg. 21 – pg. 22)

- Compare the two commands:
  - "*ls –l*"
  - "*ls –l > lsoutput*".

Check the contents of *lsoutput* using command *cat* or *more*, e.g., "*cat lsouput*" and "*more lsoutput*".

- Repeat by comparing "*ls –al*" and "*ls –al > lsoutput"*.
- Compare the difference between ">" and ">>". Type "l*s > lsoutput*" and "*ls >> lsoutput*"

Pipes (pg. 22)

Pipes can be used to connect processes: the output of the first process (command) will become the input of the second process (command). The pipe operator is |.

- Type "*ps*" which shows the **process status information**, including process ID and other attributes.

  Try some options for the command:
  - $ ps –l (Note: $ is the system prompt. You don't need to type it.)
  - $ ps –ax
  - $ ps –af
  - $ ps –aux

  (The STAT column can have different letters in it: R means Running, S means Sleeping, T means Stopped, Z means Zombie). The output of the commands listed earlier may depend on Linux systems/versions. You may run "*man ps"* to find out more about the meaning of the output of some of those commands.

- Compare **redirection and pipes:**
  Using *redirection*:
  ```
  $ ps > psout.txt
  $ sort psout.txt > pssort.out
  ```

  Using *pipes*:
  ```
  $ ps | sort > pssort.out
  ```

  Another example:
  ```
  $ ps | sort | more
  ```

3. The Standard I/O Library (page 109)

You may need to know the basics of I/O for your assignments. Go over the basic functions: fopen, fclose, fgetc, fputc, scanf, and printf. Try a basic program by walking through the File Copy Program on page 118.

Note: a file must exist before it can be opened for reading.

4. Command Line Arguments (page 137)

These are very useful. You will be using it for your assignments. For these programs, *main* is declared as:

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

Try it out. Follow the example – Program Arguments args.c – on page 139. Walk through the code and see how it works. Compile and save the executable as *args*. Test the program again by typing:

- ./args arg1 arg2 arg3
- ./args test again

A simple usage: Rename *args* to *router*. Type

- ./router ottawa
- ./router toronto
- …

II. Processes and Signals (Chapter 11)

1. What is a Process? The Process Table

- Read the description of a Unix process on pg. 461. What are the key elements? Compare it to the definition described in the lecture notes.
- What is the Linux process table? (pg. 463). What is it used for?
- View processes using the *ps* command. Try different options again and identify as many attributes as possible from the output of a *ps* command. Do a *man* command to learn the output of the *ps* command if needed.

  - $ ps –af (Note: $ is the system prompt. You don't need to type it.)
  - $ ps –ax

    Go over the table on pg. 465 to see some process status that Linux supports.

2. Starting a New Process or Running a Command in a C program (page 468 –)

- Try it out –system on pg 469. Test both *system1* and *system2*.

- Try the example, *system1.c* on pg. 469, to run a command inside of a C program. Try a few of system commands in the C program, e.g., *ls, pwd*, etc.

3. Duplicating a Process Image using *fork()* system call (page 472 –)

Understand: How does a user process create a new process using *fork()*?

- In Unix/Linux, *fork*() is used to duplicate a process image or spawn a new process. Review the operation of *fork*() as shown in Figure 11-2 (pg. 473) of the reference book.

- Compile *fork1.c* using *gcc –o* option with a specific executable file name.
- Create a new *xterm* by typing *xterm&*.
- Run the program by typing the file name you specified and type the "*ps –l"* command on the new *xterm* and compare it with the "*ps –af*" or "*ps –afl"* commands.

  Note: you may need to increase the value of variable *n* in *fork1.c* to extend its running time.

- Identify as much information as you can with the *ps* (and its variants) command. What information is common and what information is different between the parent and the child?

- Determine the process id (*pid*) of the parent process and the child process from the *ps* command. Go over the source code and identify the *pid* for the child segment. What is the difference? Why?

- Modify fork1.c by adding a printf() statement for the parent to print the value of variable *pid*. Compile and run the code, and enter the "*ps –l"* command on a separate *xterm*. What is the value of *pid* obtained from the newly added printf() statement?

- Add the following statement in the child code segment right below "*This is a child*" message:
  printf ("child process: the value returned by fork is %d\n", getppid());

  Note: getpid() is used to get the current process's PID.

  Compile and run the code, and enter the *ps –l* command on a separate *xterm*. What is the return code from the above printf() statement? Which process has the same value as obtained from the *ps* command?

- Does the fork() function return two values or just one? If not, how does it work?

- Can a child process create another child process? Modify *fork1.c*. In the code segment for child, add another fork() in the child process to create another child process.

- Check the process structure or process tree by typing *ps –axuf*

- Test the computational time of fork(). The following code segment uses *gettimeofday()* to get two different timestamps and calculate the difference to find out the time it takes for a certain part of a program to execute. It is very useful for time measurements. Find the description for *gettimeofday()* using the *man* command or simply google it. What is the structure of *timeval*?

  Add the related code to the fork program and find out the time it takes to create a child process. Run multiple times and see the differences. You can also try to create more child processes with a loop and test the computational time.

- ***Submit your solution on cuLearn. Submit a single source file, called fork_timing.c***

Code to get time difference using gettimeofday():

```
#include <time.h>
#include <sys/time.h>
#include <stdlib.h>
#include <stdio.h>

#define MICRO_SEC_IN_SEC 1000000

int main(int argc, char **argv)
{

	int i;
	struct timeval start, end;


	gettimeofday(&start, NULL); // second parameter is for time zone; usually is NULL

	for (i = 0; i < 100000; i++)
	{
	}

	gettimeofday(&end, NULL);

    printf("Start Time: %lf sec from Epoch (1970-01-01 00:00:00 +0000 (UTC))\n", start.tv_sec +
(double)start.tv_usec/MICRO_SEC_IN_SEC);

    printf("End Time: %lf sec from Epoch (1970-01-01 00:00:00 +0000 (UTC))\n", end.tv_sec +
(double)end.tv_usec/MICRO_SEC_IN_SEC);


    printf("\nElapsed Time: %ld micro sec\n", ((end.tv_sec * MICRO_SEC_IN_SEC + end.tv_usec)
                                          - (start.tv_sec * MICRO_SEC_IN_SEC +
start.tv_usec)));

	return 0;
}
```