Assignment Due date: Sunday 10/21/2020 11:55PM

This practice assignment is intended to provide the way to use the most common system calls in order to make input-output operations on files, as well as operations to handle files and directories in Linux. To complete this practice assignment, you will need to write some code and files, then execute a series of experiments, and finally answer some questions about the results. This will re-quire that you to get onto a Linux machine and run various commands. It is expected that most students will use the machines in the CS Dept. Linux Lab. If you do not re-member your CS Dept. Linux login info, please contact the CS sysadmin Michael Barkdoll (mbarkdoll@cs.siu.edu). Moreover, students can use their own Linux machines, but must be able to install any required software. For example, on Debian-based systems like Ubuntu, the easiest way to install GCC compiler and other build tools is by installing the "build-essential" meta-package by doing:

sudo apt-get install build-essential

Recommended Systems/Software Requirements:

• Any flavour of Linux References:

1. "Advanced Programming in the UNIX ® Environment, 2nd Edition, W. R. Stevens, S. A. Rago, 2013
2. Unix concepts and applications, 4th Edition, Sumitabha Das, TMH. 2006.
3. Beginning Linux Programming, 4th Edition, Matthew, Neil and Richard Stones, 2007.
4. You can use online resources.

Tasks:

- **Task 1**: Learning how to use the system calls for opening, reading and writing to files - open, read, write and lseek system calls: The basic I/O-related system calls are:
  open() – opens file and returns file handle
  close() – closes file
  read() – reads specified number of bytes from open file
  write() – writes specified number of bytes to open file
  lseek() – re-position read/write file offset

Login into your Linux system, open the Terminal and use the man command to read the manual pages of *open*, *read*, *write* and *lseek* system calls.

- **Tasks 2**: Perform the following steps:
  1. Download the file "*io_system_calls.c*" provided in the helpful resources section in D2L.
  2. Compile using gcc.
  3. This C program intends to read 100 characters from a file and to print these 100 characters on the terminal. It uses open system call to open a file. The open call returns a file descriptor fd which is then used in the successive read and write system calls to access the

- **Task 3**: Simulating the ls command
  1. Download the file "simulating_ls.c" that simulates the "ls" command provided in the helpful resources section in D2L.
  2. Compile using gcc. This C program performs the simulation of the ls command in Linux which lists all the folders and sub-folders of its present working directory.
  3. Download the file "*ShellCall_ls.c*" that runs the ls command.
  4. Compile using gcc. This C program creates child process using fork and then call *execlp* to run the ls command

Assignments:

Using similar approaches as covered in Task 2 and Task 3, implement the following using system calls.

1. Update the ShellCall_ls.c . The updated program "ShellCall.c" should:

- Print a prompt as follows $<yourname, current time>
- Allows user to type a string command: should be one of the UNIX commands such as ls, ps, etc. Note that the path for Unix commands is */usr/bin*.
- Create a new child process.
- Allow the child process calls *execlp* to run that command. Read the man page of this library call to learn what arguments to pass to it. This replaces the binary executable on the child process, but the parent goes on with the code you wrote.
- Enable the parent process calls wait so that it blocks until the child terminates and passes back its termination status.
- If the child process terminates without error, the parent spawns another child and, again, calls wait so that it can block until the child terminates. Repeat this process until type a command "quit".

2. Using a similar approach as covered in Task 3, implement in C the following UNIX services using System calls to develop *catcount.c* and *mv.c*.

catcount.c: Write C program that receives one command line argument of type string: the name of a text file. Your program will work as follows:

- Start out by spawning a child process.
- The child process calls ***execlp*** to run **/bin/cat** with the command line argument that the parent received.
- The parent process calls wait so that it blocks until the child terminates and passes back its termination status.
- If the child process terminates without error, the parent spawns another child and, again, calls wait so that it can block until the child terminates.
- The new child calls ***execlp*** again, but this time it runs ***/usr/bin/wc*** on the same argument that the parent received from the command line (the file name passed to cat previously). The command ***wc*** command counts the number of bytes, lines, etc.
- Once the parent learns that the child has terminated, it goes on to terminate also. If the parent gets to this point, it's because all has gone well, so its termination status should be 0.

mv.c: Basically, this program you should implement a simple C version of the UNIX mv command. It renames a file if both the inputs are file names (no need to consider directories).

3. Write simple C program to determine the size of a file using the "*lseek*" system call. Once you found out the size, calculate the number of blocks assigned for the file. Compare these results with the similar results obtained when using the Unix command "*stat*".

To test the program and give consistent results, you must create 100M bytes file with random contents using the following Linux utility dd:

dd if=/dev/urandom of=/tmp/testfile bs=4096 count=25600

You can use "ls -lh /tmp/testfile" to make sure the file is created with size 100Mbytes.

4. Write a C program that deletes a directory with all its subfolders. The name of the directory should be read from the command line.