**CSE-381: Operating Systems**

**Exercise #10**

Max Points: 20

**Note: If you are using your personal machine then prior to commencing work on this exercise, you may need to install XMing, Putty, and WinScp as illustrated in LinuxEnvironment.pdf (and shown in the videos in the Handouts folder).**

|  |
| --- |
| **Objective**: The objective of this exercise is to explore the use of:   * Anonymous pipes to send and receive data. * Basic introduction to threads * Named pipes aka FIFOs (First-In-First-Out).   **Submission**: Save this MS-Word document using the naming convention *MUid*\_Exercise10.docx prior to proceeding with this exercise. Upload the following at the end of the lab exercise:   1. This MS-Word document saved with the convention *MUid*\_Exercise10.docx. 2. Program developed in Part #1 named with the convention MUid\_Ex10\_Part1.cpp. 3. Program developed in Part #2 named with the convention MUid\_Ex10\_Part2.cpp.   You may discuss the questions with your instructor. |

# Preliminaries

1. Log onto the Linux server for this course via the following steps (that were covered in the previous exercises and as illustrated in the [LinuxEnvironment.pdf](https://niihka.muohio.edu/access/content/group/b05b7d30-d8c6-4312-9559-4d980565cbaf/Handouts%20_%20Video%20Tutorials/LinuxEvironment.pdf)):
   1. Run the X-Server Xming.
   2. Use PuTTY to log into the Linux server cse381-f12.csi.muohio.edu.
   3. When you log onto the server, you will be presented with a shell (**$**) prompt. You need to perform various tasks by typing commands at the shell prompt and pressing the enter (↵) key.
   4. Start emacs and ensure you see the graphical screen for emacs.

# Part #1: Develop program to read/write messages [7 points]

*Estimated time to complete: 25 minutes*

**Background**: Anonymous pipes provide a convenience mechanism for IPC between processes running on the same computer. Anonymous pipes are also convenient mechanisms to interconnect processes on a Linux machine to perform tasks. Most of the command-line programs and utilities in Linux are designed to facilitate the use of anonymous pipes as they read and write data to standard input (std::cin) and standard output (std::cout) respectively.

**Exercise**: Using the sample program discussed in class (see Slide 32 and Slide 33 in IPC.ppt off Niihka) as a template, implement a program that displays the output of the following sequence of commands:

|  |
| --- |
| **$** ls -lR /usr/include | grep "z.\*\.h$" |

You may verify correct operation of your program by running the above command-sequence at the shell prompt. Using the output from your program (as well as output from the shell prompt) briefly describe what you think the sequence of commands is accomplishing in the space below:

|  |
| --- |
|  |

# Part #2: Simple threading [7 points]

*Estimated time to complete: 25 minutes*

**Background**: Threads are sub-processes or lightweight processes (LWP) that can be used to run parts of a program asynchronously and simultaneously. On Linux, threads are typically created and run using the pthreads POSIX C API. However, C++ provides more convenient wrappers and methods that closely reflect the API supported in Java and C# making it a bit easier to map the concepts covered in C++ to other programming languages.

**Exercise**: Using the example on Slide 30 of Threads.pptx (available off Niihka) as a guide, implement the runThreads() method in the following C++ program to run the threadMain() method using 50 threads (use the THREAD\_COUNT constant in your code rather than magic numbers) (see sample output further below and the command-line involving –l pthread in the sample output to compile your program):

|  |
| --- |
| **#include** <vector>  **#include** <algorithm>  **#include** <iostream>  **#include** <thread>  **#include** <mutex>  **#define** THREAD\_COUNT 50  **void** **threadMain**();  **int** num = 0;  **void** **runThreads**() {  // Create 50 threads that run threadMain()  // Ensure you wait for all threads to join  }  //------ DO NOT MODIFY CODE BELOW THIS LINE -------  **void** **threadMain**() {  **for**(**int** i = 0; (i < 1000); i++) {  num++;  }  }  **int** **main**() {  runThreads();  std::cout << "Value of num = " << num << std::**endl**;  **return** 0;  } |

**Sample output**: The modified version of your program with 50 threads essentially increments the shared variable num 50,000 times. Ideally the final output from your program should be as shown below:

|  |
| --- |
| **$** g++ -std=c++0x -g -Wall raodm\_Ex10\_Part2.cpp -o raodm\_Ex10\_Part2 **-l pthread**  **$** ./raodm\_Ex10\_Part2  Value of num = 50000  **$** |

**Observations**: This exercise uses a simple multi-threading approach to share the num variable, which can cause inconsistent operation during runtime. Consequently the output from the program can vary. Run your program 10 times and copy-paste the outputs in the space below. It is expected that the output from the program is not the same in each run due to multi-threading issues called “race conditions”.

|  |
| --- |
|  |

# Part #3: Using named pipes (aka FIFOs) [6 points]

*Estimated time to complete: 25 minutes*

**Background**: A named pipe (also known as a FIFO because it performs First-In-First-Out type IPC) is a commonly used pipe concept on Linux and Unix-like systems. It is one of the different methods of inter-process communication. The concept is also found in Microsoft Windows, although the semantics differ substantially. Unlike a traditional pipe that is "unnamed" (as it exists anonymously) and persists only for as long as the interacting process is running, named pipes exist beyond the lifetime of processes. A named pipe is system-persistent and exists beyond the life of the process. It can be deleted once it is no longer being used. Processes generally attach to the named pipes as if they were a routine files to perform inter-process communication (IPC) by reading and writing data to it. Usually pipes are used to exchange text data. However, the pipes may also be used for interchange of binary data.

**Exercise**: This exercise requires you to create a FIFO and use it for exchanging data between two processes as directed below:

1. Create a FIFO using the following command at the shell prompt (where MUid is your Miami University unique ID):

|  |
| --- |
| **$** mkfifo -m 0744 *MUid* |

Note that the permissions are intentionally setup as 0744 (What do those permissions do?). To verify the named pipe has been created , run ls -l (ell) to list the file entries and copy-paste the entry corresponding to the FIFO you created in the previous command and paste it in the box below:

|  |
| --- |
|  |

Note the permissions associated with the entry at the beginning of the line. Now, using that information, describe the permissions provided for user, group, and other in the space below for the named pipe that you have created (note the first character in permissions):

|  |
| --- |
|  |

1. Now use the FIFO that was created earlier to perform some simple IPC between two different processes. For this we will create one process that sends data to the FIFO and another one that reads data from the FIFO as directed below:
   1. Open two different PuTTY windows and log into the Linux machine.
   2. In one of the windows print the contents of the FIFO using the following command (where *MUid* is the FIFO that was created earlier):

|  |
| --- |
| **$** cat *MUid* |

* 1. In the other PuTTY window write data to the FIFO using the following command:

|  |
| --- |
| **$** ps > *MUid* |

* 1. You should observe the output displayed by the PuTTY window in which the earlier cat command was run.

1. Observe how the FIFO enables communication between processes using following steps:
   1. Run “ls –l” and record the size of the FIFO in the space below:

|  |  |
| --- | --- |
| Size (in bytes) of the FIFO was: |  |

* 1. Repeatedly redirect output of ps to the named pipe to check if the size of the FIFO reported by “ls –l” ever changes from the value recorded in the previous step.
  2. Observe the behavior when data is written to the FIFO (say via ps > *MUid*) but no process is reading data from it. What happens with the program that is writing data to the FIFO (note its process state by listing all of your processes using the ps –fea | grep *MUid* command)? Briefly describe your inference in the space below:

|  |
| --- |
|  |

1. Now remove write permissions on your FIFO for yourself using the following command (and verify permission settings on the FIFO using ls -l command):

|  |
| --- |
| **$** chmod 544 *MUid* |

1. Try to write data to the FIFO and ensure that permission is denied to write data to the FIFO.
2. You may re-enable permissions on the FIFO (using the command chmod 744 *MUid*) and try using the named pipe.
3. Finally delete the FIFO that you created using the rm command as shown below:

|  |
| --- |
| **$** rm *MUid* |

# Part 4: Submit files to Niihka

Upload just the following files to Nihhka:

1. This MS-Word document saved with the convention *MUid*\_Exercise10.docx.
2. Program developed in Part #1 named with the convention MUid\_Ex10\_Part1.cpp.
3. Program developed in Part #2 named with the convention MUid\_Ex10\_Part2.cpp.