**CSE-381: Operating Systems**

**Exercise #12**

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 effectiveness of using monitors to coordinate threads
* Observe and control threads externally via OS tools

**Submission**: Save this MS-Word document using the naming convention *MUid*\_Exercise12.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*\_Exercise12.docx.
2. Program developed in his exercise named with the convention MUid\_Ex12.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 #2: Using Monitors instead of Semaphores [10 points]

*Estimated time to complete: 45 minutes*

**Background**: The producer-consumer model for sharing data between multiple threads is widely used. There are several different approaches to implementing the producer-consumer model depending on multi-threaded or multi-process scenarios. Even in multi-threaded programs, several different approaches can be used to implementing producer-consumer type applications. Two commonly used approaches are:

* A semaphore-based approach that involves busy waiting (this implementation is given to you) which is useful in certain cases where the application can monopolize the resources and real-time interactions are highly desired.
* A monitor-based approach that avoid busy waiting (you need to implement this version of the program) and is the most commonly used approach as provides an efficient use of CPU cycles.

In this part of the exercise you are expected to:

1. Convert a given semaphore-based implementation of a producer-consumer type application into a version that uses monitors.
2. Compare the performance of the two versions of the same program (that should generate exactly the same output)

**Exercise:**

In data parallel applications the synchronization overheads are very minimal, if not totally absent. Consequently, at least in theory, linear performance improvements are to be expected – *i.e.*, if the program takes *t* seconds to run with 1 thread, then ideally it would take only *t*/*n* seconds to run when run using *n* threads. This part of the exercise explores the effectiveness of data parallel implementation by comparing the time taken to run the program with varying number of threads.

1. Download the supplied exercise12.cpp and save program to the Linux server and briefly review the operation of the program. Compile the program using the command:

|  |
| --- |
| $ g++ -std=c++0x -g -Wall exercise12.cpp -o exercise12 -lpthread |

1. Linux provides a time utility to measuring the time taken to run a program that can be used as shown in the sample output below (the actual timings you observe will be different and that is to be expected):

|  |
| --- |
| exercise12$ /usr/bin/time ./exercise12 > ex11.txt13.14user 0.03system 0:06.62elapsed 199%CPU (0avgtext+0avgdata 5520maxresident)k0inputs+264outputs (0major+394minor)pagefaults 0swaps |

Each time a program is run, the actual time taken to run the program will vary depending on the load and other activities occurring on the system. Consequently, on multi-user, multi-tasking systems timing measurements have to be repeated in order to ensure that consistent timings are obtained and the consistent timings are averaged to obtain a suitable runtime value.

Using the time command shown above, run the given program without any modifications three times (such that the timings are consistent) and note the timings in the table below:

|  |
| --- |
| **Timings from given semaphore-based application** |
| Observation #1 | Observation #2 | Observation #3 |
| Elapsed Time (sec) | %CPU | Elapsed Time (sec) | %CPU | Elapsed Time (sec) | %CPU |
|  |  |  |  |  |  |

1. Copy the supplied program to a file with the naming convention MUid\_exercise12.cpp (example: raodm\_exercise12.cpp) and modify MUid\_exercise12.cpp to use monitors instead of semaphores to avoid busy waiting in the given producer-consumer application. Refer to Slide #37 and Slide #38 in Synchronization.pptx off Niihka on how to use std::condition\_variable and std::unique\_lock to implement a monitor (using wait() and notify\_one() methods).

NOTE: The output from the revised version of the program should be exactly the same as that of the original program. You may verify that the outputs are identical by comparing output from your program against (assuming your revised program compiles and runs successfully) against the output from the given version of the program (that was redirected to a file named ex11.txt in prior step) in the following manner and ensuring that there is absolutely no output (or differences) reported by diff command:

|  |
| --- |
| $ /usr/bin/time ./MUid\_exercise12 > MUid\_ex11.txt$ diff ex11.txt MUid\_ex11.txt$ |

1. Now that you have successfully re-implemented the application using monitors (and you have verified it is generating the same output and consequently we are comparing mangoes-to-mangoes) it is important to understand the performance implications associated with it by comparing its performance against the semaphore version of the program. Using the time command shown above, run the given program without any modifications three times (such that the timings are consistent) and note the timings in the table below:

|  |
| --- |
| **Timings from revised monitor-based implementation** |
| Observation #1 | Observation #2 | Observation #3 |
| Elapsed Time (sec) | %CPU | Elapsed Time (sec) | %CPU | Elapsed Time (sec) | %CPU |
|  |  |  |  |  |  |

1. Based on the timings from the two version of the same program answer the following questions:
	1. Which version of the program has a lower average elapsed time? Show average elapsed times for the two versions and the difference in elapsed times as well.

|  |
| --- |
|  |

* 1. Which version of the program has a lower average %CPU? Show average %CPU for the two versions and the difference in %CPU as well.

|  |
| --- |
|  |

* 1. Using the difference in average elapsed time and average %CPU which version of the program seems to be performing better? Record your inferences in the space below and provide suitable explanation in support of your interference.

|  |
| --- |
|  |

# Part #2: Observe behaviors of multi-threaded programs [7 points]

*Estimated time to complete: 45 minutes*

**Background**: In most modern operating systems, including Linux, threads are implemented using kernel threads (rather than User threads). With kernel threads, the operating system is aware of multiple threads being used by a process. Furthermore, the OS manages multiple threads by suitably scheduling them and handling resources used by various threads. Most operating systems also provide the ability to run foreground or joinable threads and background or detached threads.

In Linux, there is very little distinction between processes and threads. In fact, the OS uses a common clone (read manual page on clone for details) system call to create processes and threads using slightly different parameters to the clone system call. Consequently, in Linux threads are called Light Weight Processes or LWP and similar to a regular process, each LWP is assigned a tid (thread ID, similar to pid) gets an entry in the virtual /proc/ file system (recollect that /proc/ is a direct memory map of kernel data structures). The entries /proc/ file system can be used to obtain additional information about the operation of various threads.

**Exercise**: The objective of this exercise is to explore the support provided by Linux for monitoring and controlling threads.

1. Download the supplied timer.cpp and save program to the Linux server and briefly review the operation of the program. The program creates a few threads and all they do is sleep. One of the threads counts down a 30 minute timer and the program ends when the timer expires.
2. Open two separate PuTTY windows and login to the Linux server. You will run the timer.cpp in one PuTTY window and observe behaviors in another PuTTY window.
3. Compile and run the program using the standard set of commands shown below:

|  |
| --- |
| **$** g++ -std=c++0x -Wall timer.cpp -o timer –lpthread**$** ./timer |

1. Using the process ID (pid) from the program, view the status information about the overall process by inspecting the /proc/*<pid>*/status file (if you still haven’t figured out how to view files in Linux, use the less command: less /proc/<pid>/status). From the output copy-paste the following information about the main process:

|  |  |
| --- | --- |
| What is the process ID (pid) |  |
| What is the parent process ID (ppid): |  |
| Number of threads in process (Threads): |  |

1. Now, use the ps command to list the pid and tid of the process and its LWP (aka threads) using the following command (where <pid> is the number reported by the timer program) and copy-paste it into the space further below:

|  |
| --- |
| $ ps –feaL | grep <*pid*> |

|  |
| --- |
| Output from above command goes here instead of this text |

1. Choose one LWP ID (fourth column in the above output) and view the status information about the LWP by inspecting the /proc/*<LWP-pid>*/status file. From the output copy-paste the following information about the LWP-process:

|  |  |
| --- | --- |
| What is the process ID (pid) |  |
| What is the parent process ID (ppid): |  |
| Number of threads in process (Threads): |  |

1. **Priority experiment:** Each process in Linux has a priority in the range 0 to 99, with zero being highest priority and 99 being lowest priority. By default processes and their threads are assigned priority depending on user settings. This priority value can be changed at a process level (and consequently for each thread) using a “niceness” value for each process. The niceness value generally ranges from -20 to +19, with -20 being the most favorable or highest priority for scheduling and 19 being the least favorable or lowest priority. The current scheduling priority for processes and associated niceness values can be viewed either using ps command or via the top command.

In this step we will record the priority and niceness values for the process and try to change it.

* 1. First record the current priority and priority value for the process and its threads using the following command (arguments include uppercase and lowercase ‘L’ letter):

|  |
| --- |
| **$** ps –feaLl |

|  |
| --- |
| Copy-paste the entries corresponding to the 6-threads of your process here instead of this line |

* 1. Indicate the current priority and niceness value for all the threads in the space below.

|  |  |
| --- | --- |
| The priority values for each of the 6 threads in the process: |  |
| (priority value is the 9th column in the output and is typically 60 to 80) |
|  |  |
| The niceness value for the threads is : |  |
| (It is the 10th column of output and typically tends to be zero by default) |

* 1. Now decrease (to increase priority you need to supply negative values) the priority for two of threads by increasing their niceness values to +5 and +10 respectively, using the command below (two times, once for each thread):

|  |
| --- |
| $ renice <niceValue> <pid/tid> |

* 1. First record the current priority and priority value for the process and its threads using the ps command (arguments include uppercase and lowercase ‘L’ letter):

|  |
| --- |
| **$** ps –feaLl |

|  |
| --- |
| Copy-paste 6-thread entries with modified priority and niceness of your process here instead of this line. |

1. Finally, observe the life-cycle action of threads by trying to stop just one thread in the program. From the output above, identify the pid/tid for a thread. **Do not use the pid of the main process as reported by the timer program!** Kill the thread using the kill command (kill <tid>) and note your observation below:

|  |
| --- |
| Indicate how many threads were killed and if the program continued to run in this space and instead of this line |

# Part 3: Submit files to Niihka

Upload just the following files to Nihhka:

1. This MS-Word document saved with the convention *MUid*\_Exercise12.docx.
2. Program developed in Part #1 named with the convention MUid\_Ex12.cpp.