DUE DATE:	By deadline posted on Canvas

CSE-381: Systems 2
Exercise #7
Max Points: 20

Objective: The objective of this exercise is to:
1. Understand programs that use C++ threads
2. Observe and control threads externally via Linux tools.

Submission: Save this MS-Word document using the naming convention `MUid_Ex7.docx` (example `raodm_Ex7.docx`) before proceeding with this exercise.

Fill in answers to all of the questions. For some of the questions you can simply copy-paste appropriate text from the terminal/output window into this document. You may discuss the questions with your instructor.

Name: Raobot

Part #1: Understand a simple multi-threaded C++ program
Estimated time to complete: 30 minutes

Background: Using threads in C++ is very simple – any function/method can be called/run as a thread. This makes running threads in C++ very straightforward using the `pthreads` library. However, the `pthreads` library should be explicitly linked into the program. Luckily, the `NetBean's` Miami University C++ project already links in `pthreads` library for us.

Exercise: The objective of this exercise is to understand the operation of a straightforward multithreaded C++ program.

1. Create a `NetBeans` project called `Timer` (name main file also `Timer`) -- ensure you use Miami University C++ Project. Download the supplied `Timer.cpp` program and copy it to your `NetBeans` project. Compile the program and ensure it compiles successfully.
2. Study the program – the program creates a few threads and all they do is sleep -- that is they don't do any computations. One of the threads counts down a timer and the program ends when the timer expires.

3. From the program answer the following questions (**in general, these are the level of detail you should be thinking about in every starter code**):
    a. Which method in the program is used to start threads? How did you figure it out (describe in English)?

The line of code that creates the threads is:

```
thrList.push_back(std::thread(timer, maxTime, i));
```

The above line of code calls the thread's constructor that creates the thread. The thread object (not the actual thread) is stored in the vector.

b. Where/how are arguments passed to the thread method (copy-paste line(s) of code below)? Is it pass-by-value or pass-by-reference?

The arguments are passed by value in the line of code:

```
thrList.push_back(std::thread(timer, maxTime, i));
```

This example uses pass-by-value approach.

c. If the timer method was modified to accept references as in, `void timer(int& maxTime, int& threadID)`, illustrate the line of code to be used to pass parameters (Hint: `std::ref`):

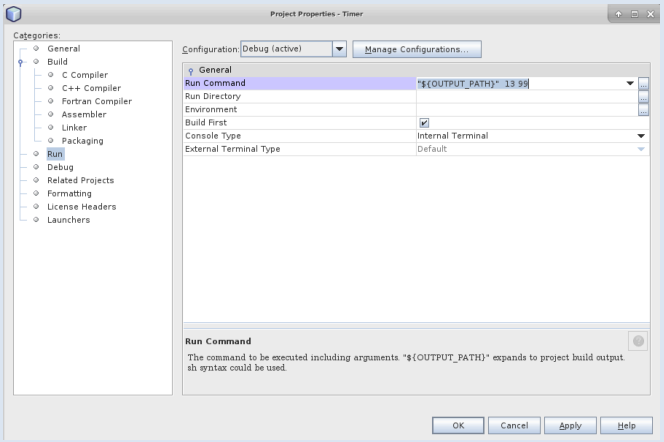
```
thrList.push_back(std::thread(timer, std::ref(maxTime),
                                     std::ref(i)));
```

d. Show the GNU/Linux command (that should be typed at the `bash` shell) to run the program with 13 threads for 99 seconds. What is the generic term used to refer to these argument values (*i.e.*, 13 and 99) to be passed to the program called?

```
$ ./Thread 13 99
```

The values "13 99" passed to the thread are called command-line arguments.

e. How should the above command-line arguments (*i.e.*, 13 and 99) be set in NetBeans in order to pass it onto this program? Show a screenshot below illustrating how you have set command-line arguments in NetBeans – if you are unsure how to do it see video on `Canvas→Video Demonstrations` page.



f. From `NetBean's` compile output, copy-paste the command printed by `NetBeans` that links in `pthread` library. Note this line of command has `-l` (lower-case ell) towards the end, where `-l` stands for "link-in the library".

```
g++  -fsanitize=address  -DGNUCXX_DEBUG        -o  Timer
build/Debug/GNU-Linux/Timer.o  -lboost_system -lpthread -
lmysqlpp
```

g. From the above command what are the other two libraries that are being linked into the program (Hint: words after the `-l` option to `g++`)?

The other two libraries being linked are: `boost_system` (for BOOST socket) and `mysqlpp` (MySQL++ C++ API)

Part #2: Observe behaviors of multi-threaded programs

Estimated time to complete: 30 minutes

Background: In most modern operating systems, including Linux, threads are implemented natively in the operating system. 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`). Similar to processes, each thread also gets an entry in the virtual `/proc/` file system. The `/proc` file system in Linux is a direct memory map to the kernel's internal 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. This part of the exercise is to be conducted using the C++ `Timer.cpp` program.

2. Open two separate terminal windows to the Linux server used for this course – you will run the `Timer` (C++ executable) in one terminal and observe its behaviors in another terminal.

3. Compile the program via NetBeans. Run the program from a terminal window. The program will print the process ID (`pid`) for the process and then start counting down a simple timer.

4. Using the process ID (**`pid`**) printed by the program, view the status information about the overall process by inspecting the `/proc/`***`pid`***`/status` file – *e.g.* `/proc/`**`2018`**`/status` (if you still haven't figured out how to view files in Linux, remember to use the `less` command: `less /proc/pid/status`, use cursor keys to scroll and `q` to quit). From the output copy-paste the following information about the main process:

What is the process ID (`pid`)	29516
What is the parent process ID (`ppid`):	29515
Number of threads in process (`Threads`):	5

Notice the discrepancy in the number of threads reported by the operating system versus the number of threads the program claims to have created. Why is there a discrepancy?

The number of threads reported by the OS is 1 more than the number of threads created by the program. This is because every process has 1 thread that is running the `main` method, which then starts 5 more threads resulting in 6 threads.

5. Now, use the `ps` command to list the `pid` and `tid` (thread ID) 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 (the `grep` command searches & prints lines with word `LWP` or `<pid>`):

```
$ ps -feaL | grep -E 'LWP|<pid>'
```

```
raodm@os1:~/ $ ps -feaL | grep -E 'LWP|29516'
UID        PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
raodm        29516  29515  29516    0    6  14:25  pts/17       00:00:00
/home/raodm/NetBeansProjects/Timer/Timer
raodm        29516  29515  29517    0    6  14:25  pts/17       00:00:00
/home/raodm/NetBeansProjects/Timer/Timer
raodm        29516  29515  29518    0    6  14:25  pts/17       00:00:00
/home/raodm/NetBeansProjects/Timer/Timer
raodm        29516  29515  29519    0    6  14:25  pts/17       00:00:00
/home/raodm/NetBeansProjects/Timer/Timer
raodm        29516  29515  29520    0    6  14:25  pts/17       00:00:00
/home/raodm/NetBeansProjects/Timer/Timer
raodm        29516  29515  29521    0    6  14:25  pts/17       00:00:00
/home/raodm/NetBeansProjects/Timer/Timer
raodm    29726  1303 29726  0    1 14:27 pts/10   00:00:00 grep -E LWP|29516
```

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

   What is the process ID (`pid`) 29516

   What is the parent process ID (`ppid`): 29515

7. **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 indicated by the `PRI` column. 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.

   a. 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 | grep -E 'LWP|<pid>'
   ```

   Example: `ps -feaLl | grep -E 'LWP|29516'`

```
$ ps -feaLl | grep -E 'LWP|29516'
F S UID          PID    PPID     LWP  C NLWP PRI  NI ADDR SZ WCHAN  STIME TTY
TIME CMD
0 S raodm      29516   29515   29516  0     6  80   0 - 5368737472 futex_ 14:25
pts/17 00:00:00 /home/raodm/NetBeansProjects/Timer/Timer
1 S raodm      29516   29515   29517  0     6  80   0 - 5368737472 hrtime 14:25
pts/17 00:00:00 /home/raodm/NetBeansProjects/Timer/Timer
1 S raodm      29516   29515   29518  0     6  80   0 - 5368737472 hrtime 14:25
pts/17 00:00:00 /home/raodm/NetBeansProjects/Timer/Timer
1 S raodm      29516   29515   29519  0     6  80   0 - 5368737472 hrtime 14:25
pts/17 00:00:00 /home/raodm/NetBeansProjects/Timer/Timer
1 S raodm      29516   29515   29520  0     6  80   0 - 5368737472 hrtime 14:25
pts/17 00:00:00 /home/raodm/NetBeansProjects/Timer/Timer
1 S raodm      29516   29515   29521  0     6  80   0 - 5368737472 hrtime 14:25
pts/17 00:00:00 /home/raodm/NetBeansProjects/Timer/Timer
0 S raodm     29728    1303   29728  0     1  80   0 -  3236 pipe_w 14:28 pts/10
00:00:00 grep -E LWP|29516
```

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

The priority (**PRI**) values for each of the threads in the process: 80
(priority value is the 9th column in the output and is typically 60 to 80)

The niceness (**NI**) value for the threads is : 0
(It is the 10th column of output and typically tends to be zero by default)

c. 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 separate times, once for each thread**):

```
$ renice <niceValue> <tid>
Example:
$ renice +5 6483
```

d. Next, record the new priority and niceness value for the process and its threads using the `ps` command (arguments include uppercase and lowercase 'L' letter):

```
$ ps -feaLl | grep -E 'LWP|<pid>'
```

```
$ ps -feaLl | grep -E 'LWP|29516'
F S UID          PID    PPID     LWP  C NLWP PRI  NI ADDR SZ WCHAN    STIME TTY
TIME CMD
0 S raodm      29516  29515  29516  0    6  80   0 - 5368737472 futex_ 14:25 pts/17
00:00:00 /home/raodm/NetBeansProjects/Timer/Timer
1 S raodm      29516  29515  29517  0    6  80   0 - 5368737472 hrtime 14:25 pts/17
00:00:00 /home/raodm/NetBeansProjects/Timer/Timer
1 S raodm      29516  29515  29518  0    6  85   5 - 5368737472 hrtime 14:25 pts/17
00:00:00 /home/raodm/NetBeansProjects/Timer/Timer
1 S raodm      29516  29515  29519  0    6  85   5 - 5368737472 hrtime 14:25 pts/17
00:00:00 /home/raodm/NetBeansProjects/Timer/Timer
1 S raodm      29516  29515  29520  0    6  80   0 - 5368737472 hrtime 14:25 pts/17
00:00:00 /home/raodm/NetBeansProjects/Timer/Timer
1 S raodm      29516  29515  29521  0    6  80   0 - 5368737472 hrtime 14:25 pts/17
00:00:00 /home/raodm/NetBeansProjects/Timer/Timer
0 S raodm      29853   1303  29853  0    1  80   0 -  3236 pipe_w 14:29 pts/10
00:00:00 grep -E LWP|29516
```

8. 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:

When I killed thread with LWP 29853, the whole process with all 6 threads were killed! This is because it is not possible to kill just 1 thread; instead all the threads are killed.

Part #3: Demonstrating and understanding a race condition

Estimated time: 20 minutes

DUE DATE:	By deadline posted on Canvas

Watch the following video introducing the issue of race condition https://youtu.be/HmThxUGg4Kc. This part of the exercise essentially has you developing and running the example discussed in this video.

Background: Race conditions occur when shared variables are incorrectly updated by multiple threads. Specifically, multiple threads must be updating (directly or indirectly) a shared primitive data type.

Exercise: Complete this part of the exercise in the following manner:

1. Create a `NetBeans` project called `exercise7_part3` without any starter code but using the Miami University C++ project template. Ensure your project is created `os1.csi.miamioh.edu`. For off-campus access to the server, you will need to use VPN.
2. Download and `scp` the `exercise7_part3.cpp` starter code to your NetBeans project.
3. Briefly study the `threadMain` function in the starter code.
4. Now, we are going to implement the `main` function to start 10 threads and each thread should call the `threadMain` method. The main function will be intentionally implemented (*in practice, programmers create race conditions unintentionally*) such that it gives rise to a race condition.
5. Briefly study the following implementation. Next, copy-paste the following implementation of the `main` function to your program:

```
int main() {
    int x = 0;  // A shared variable to demo race conditions.
    // The list of threads.
    std::vector<std::thread> threads;
    for (int i = 0; (i < 10); i++) {
        threads.push_back(std::thread(threadMain, std::ref(x)));
    }
    // Wait for the threads to finish
    for (auto& t : threads) {
        t.join();
    }
    // Print the value of x (to observe race condition)
    std::cout << "x = " << x << std::endl;
    return 0;
}
```

6. Briefly describe how the implementation in the `main` function causes a race condition to rise.

   The `main` function passes a reference to the same variable `x` to each of the threads. Since a reference is passed, the threads end-up incrementing the same variable causing a race condition.

7. Illustrate output from 3 separate runs of your program in the space below:
   a. Output from 1st run:

```
x = 111380
```

b. Output from 2nd run:

```
x = 79474
```

c. Output from 3rd run:

```
x = 79474
```

8. Briefly describe how the above outputs illustrate the side effects of a race condition

Ideally, the expected output of the program should always be 500,000 (*i.e.*, 10 threads incrementing a variable 50,000 times). However, the program prints out completely different and wrong results each time it is run. This is due to race conditions in the program.

9. Briefly describe how you could fix the above race condition.

One simple way would be to create an `std::vector` of 10 integers and give separate values to different threads so that they don't share variables – as in:

```
std::vector<int> vals(10);
```

and when creating threads, do:

```
threads.push_back(std::thread(threadMain, std::ref(vals[i])));
```

Now each thread will get its own separate integer to operate on. Hence, a race condition will not arise because the threads are not sharing the same variable.

Part #4: Submit files

Once you have successfully completed the lab exercise, upload the following at the end of the lab exercise:

- This MS-Word document saved as a PDF file with the convention *`MUid_Ex7`*`.pdf`. (example: `raodm_Ex7.pdf`).

Upload each file individually to Canvas. Do not upload archive files such as zip/7zip/rar/tar/gz etc. Ensure you click the `Submit` button on Canvas once you have uploaded all the necessary files.