

CSE-381: Systems 2

Exercise #7

Max Points: 20

Name: ce zhang

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.docx* (example *raodm.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.

Part #0: Introduction to threads

Estimated time to complete: 25 minutes



Now wait for your instructor to introduce threads by covering the first 15 slides in the lecture slides titled *09_Threads.pdf* on Canvas.

The concepts and code are involved. So, ensure you pay attention to it so that you can complete this exercise successfully.

Part #1: Understand a simple multi-threaded C++ program [10 points]

Estimated time to complete: 15 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 `pthread` library. However, the `pthread` library should be explicitly linked into the program. Luckily, the NetBeans' s Miami University C++ project already links in `pthread` 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 (describe in English) did you figure it out?

The line of code that creates the threads is:

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

this code calls the thread's constructor which can creates the thread and it is stored in a vector

- b. Where/how are arguments passed to the thread method (copy-paste line(s) of code below)?

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

- 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:

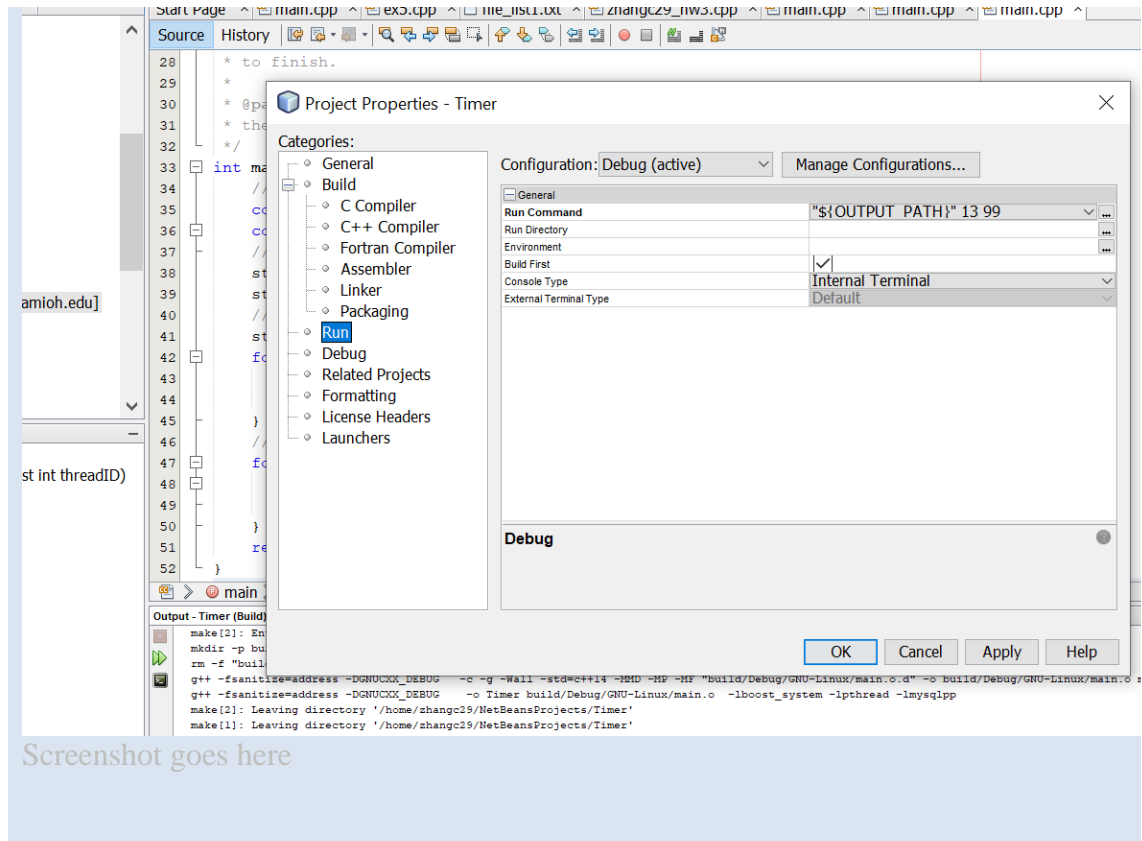
```
thrList.push_back(std::thread(timer, std::ref(maxTime),std::ref(i)));
```

- d. Show the command to use to run the program with 13 threads for 99 seconds. What are these values 13 and 99 to be passed to the program called?

```
./ thread 13 99
```

The values "13 99" are called command-line arguments.

- e. How should the above arguments (i.e., 13 and 99) be set in NetBeans in order to pass it onto this program? Show a screenshot below illustrating how to set it in NetBeans -
- if you are not sure how to do it see video on Canvas→Video Demonstrations page



- f. From NetBeans' 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 -DGNU_CXX_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?

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 [10 points]

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`) 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`). From the output copy-paste the following information about the main process:

What is the process ID (<code>pid</code>)	13922
What is the parent process ID (<code>ppid</code>):	7631
Number of threads in process (<code>Threads</code>):	6

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. Every process has 1 thread process which is running process which is 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>'
```

```
zhangc29@os1:~$ ps -feal | grep -E 'LWP|13922'
UID      PID    PPID    LWP    C  NLWP  STIME  TTY          TIME CMD
zhangc29 13922   7631  13922   0    6  14:59 pts/15      00:00:00 ./Timer
zhangc29 13922   7631  13923   0    6  14:59 pts/15      00:00:00 ./Timer
zhangc29 13922   7631  13924   0    6  14:59 pts/15      00:00:00 ./Timer
zhangc29 13922   7631  13925   0    6  14:59 pts/15      00:00:00 ./Timer
zhangc29 13922   7631  13926   0    6  14:59 pts/15      00:00:00 ./Timer
zhangc29 13922   7631  13927   0    6  14:59 pts/15      00:00:00 ./Timer
zhangc29 14019   8314  14019   0    1  15:00 pts/12      00:00:00 less
/proc/13922/status
zhangc29 17058   8314  17058   0    1  15:10 pts/12      00:00:00 grep --color=auto -E
LWP|13922
```

Output from above command goes here instead of this text

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) 19322

What is the parent process ID (ppid): 19321

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>'
```

```
zhangc29@osl:~$ ps -fealL | grep -E 'LWP|13922'
F S UID          PID  PPID  LWP  C NLWP PRI  NI ADDR SZ WCHAN  STIME TTY
TIME CMD
0 S zhangc29 13922  7631 13922  0   6  80   0 - 5368737831 futex_
14:59 pts/15 00:00:00 ./Timer
1 S zhangc29 13922  7631 13923  0   6  80   0 - 5368737831 hrttime
14:59 pts/15 00:00:00 ./Timer
1 S zhangc29 13922  7631 13924  0   6  80   0 - 5368737831 hrttime
14:59 pts/15 00:00:00 ./Timer
1 S zhangc29 13922  7631 13925  0   6  80   0 - 5368737831 hrttime
14:59 pts/15 00:00:00 ./Timer
1 S zhangc29 13922  7631 13926  0   6  80   0 - 5368737831 hrttime
14:59 pts/15 00:00:00 ./Timer
1 S zhangc29 13922  7631 13927  0   6  80   0 - 5368737831 hrttime
14:59 pts/15 00:00:00 ./Timer
0 T zhangc29 14019  8314 14019  0   1  80   0 - 2462 signal 15:00
pts/12 00:00:00 less /proc/13922/status
0 S zhangc29 23429  8314 23429  0   1  80   0 - 3607 pipe_w 15:37 pts/12
00:00:00 grep --color=auto -E LWP|13922
```

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

The priority 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 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>'
```

```
zhangc29@osl:~$ ps -feall | grep -E 'LWP|24375'
F S UID                PID  PPID    LWP   C NLWP PRI   NI ADDR SZ WCHAN   STIME TTY
TIME CMD
0 S zhangc29 24375    7631 24375   0    6  85    5 - 5368737831 futex_ 15:42
pts/15 00:00:00 ./Timer
1 S zhangc29 24375    7631 24376   0    6  80    0 - 5368737831 hrttime 15:42
pts/15 00:00:00 ./Timer
1 S zhangc29 24375    7631 24377   0    6  80    0 - 5368737831 hrttime 15:42
pts/15 00:00:00 ./Timer
1 S zhangc29 24375    7631 24378   0    6  80    0 - 5368737831 hrttime 15:42
pts/15 00:00:00 ./Timer
1 S zhangc29 24375    7631 24379   0    6  80    0 - 5368737831 hrttime 15:42
pts/15 00:00:00 ./Timer
1 S zhangc29 24375    7631 24380   0    6  80    0 - 5368737831 hrttime 15:42
pts/15 00:00:00 ./Timer
0 S zhangc29 24581    8314 24581   0    1  80    0 - 3607 pipe_w 15:44 pts/12
00:00:00 grep --color=auto -E LWP|24375
```

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 24375, 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: 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.docx`.

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.