# Thread1 Demonstration

## Files

The files for this demonstration can be found in the rlogin cluster in the directory

`/web/courses/cs3214/spring2014/butta/examples/thread-demo/thread1`

The files are `Makefile proc.c thread1.c thread2.c thread3.c`

The `“make” command` by default will create four executables named `proc, thread1, thread2, and thread3`. The programs for these executables have different ways of performing the same simple task – incrementing a counter that is initialized to zero by 20,000.

## Purpose

The purposes of this demonstration are

- to see how to create threads and identify what code they are to execute
- to see the differences between processes and threads
- to see the differences between threads that execute sequentially and threads that execute concurrently
- to see how threads can interfere with each other through race conditions
- to see how to prevent race conditions among threads using locks

## Step 1

1. Use the `Makefile` to create the executable programs with the command `“make”`.
2. At the shell prompt execute `proc`. Observe the output that is produced. Examine the code in `proc.c` and answer question 1.
3. Examine the code in `thread1.c`. Note that the procedures `thread1` and `thread2` each increment the global variable `count`. Examine the code in `main`() which has calls to `pthread_create` and `pthread_join`. Use `“man 3 pthread_create”` to show the description of this function. Briefly read this description. Do not worry about the details of the arguments; just get a sense of what the function does. Use `“man 3 pthread_join”` to show the description of this function. Briefly read this description. Answer question 2.
4. At the shell prompt execute `thread1`. Observe the output that is produced. Execute `thread1` several times observing the output in each case. Examine the code in `thread1.c` and answer question 3.
5. At the shell prompt execute `proc` and then execute `thread1`. Observe the output produced by each and answer question 4.

## Step 2

1. At the shell prompt execute `thead1` several times and observe the output. Execute `thread2` several times and observe the output. Answer question 5.
2. Examine the code in `thread2.c` and answer question 6.
3. At the shell prompt execute `thead2` several times and observe the output. Compare the cases where the expected value for count is reported with those cases where a value other than expected value for count is reported. Observe carefully the output indicating the starting and termination of the threads. Answer question 7.
4. The execution of `thread2` illustrates a *race condition*. For a description see: http://en.wikipedia.org/wiki/Race_condition. Summarize your observations by answering question 8.

**Step 3**

1. Read the first part of the explanation of a *critical section* on Wikipedia at: http://en.wikipedia.org/wiki/Critical_section. Examine the code in `thread2.c`. Focus on the code in the `thread1()` and `thread2()` procedures. Answer question 9.
2. At the shell prompt execute `thead3` several times and observe the output. Observe carefully the output indicating the starting and termination of the threads. Answer question 10.
3. Examine the code in `thread3.c` that you identified as being part of the critical section. Read the first two paragraphs in the DESCRIPTION section at: http://www.linuxmanpages.com/man3/pthread_mutex_lock.3thr.php and also the paragraphs that describe the two procedures `pthread_mutex_lock` and `pthread_mutex_unlock.` Answer question 11.

**Questions**

Based on your observations above, answer these questions:

1. How many times has the global variable count been incremented by each child process? What is the final value of count reported by the parent process? How do you explain this outcome?
2. In `thread1.c` how many threads are created in `main()`? Based on the information in the `man` pages, what does the sequence `pthread_create(…)` followed by `pthread_join(…)` do? What procedure is performed by the first thread created? What procedure is performed by the second thread created? Does the second thread created begin execution before or after the first thread has terminated? Would you describe the threads as executing sequentially or concurrently? For a definition see: http://en.wikipedia.org/wiki/Concurrency_(computer_science) .
3. In each case when `thread1` is executed is the output always the same? Does the output from the two threads appear to overlap or is the output from one entirely before the output from the other? Does this confirm or contradict your answers in question 2?
4. How many child processes are created by `proc`? How many threads are created by `thread1`? How many total times has an increment of count been performed by `proc`?

How many total times has an increment of count been performed by `thread1`? What is the final value of `count` reported by `proc` and the final value of `count` reported by `thread1`? How do you account for this?

5. Is the output produced by `thread1` always the same? Is the output produced by `thread2` always the same? Which of these would you describe as *deterministic* and which would you describe as *non-deterministic*? For a definition see: http://en.wikipedia.org/wiki/Nondeterministic_algorithm.
6. In `main()` of `thread2.c` is the pattern of the calls to `pthread_create(…)` and `pthread_join(…)` the same as it is in `thread1.c`? Which of these would you describe as a pattern for sequential execution of threads and which would you describe as a pattern for concurrent execution of thread?
7. When the expected value of count is reported by `thread2` does the execution appear to be *sequential* or *concurrent*? When a value other than the expected value of count is reported by `thread2` does the execution appear to be *sequential* or *concurrent*?
8. Which choice of words in brackets in the following two sentences best summarizes your observations of the execution of `thread1` and `thread2`?
    a. The [sequential/concurrent] execution of threads [always/sometimes/never] leads to [deterministic/non-deterministic] behavior and results in [expected/unexpected] results.
    b. The [sequential/concurrent] execution of threads [always/sometimes/never] leads to [deterministic/non-deterministic] behavior and results in [expected/unexpected] results.
9. In `thread2`.c can you identify one or more lines of code in the `thread1()` and `thread2()` procedures that constitute a critical section?
10. Is the output from `thread3` always the same? Do the threads appear to execute concurrently or sequentially? Is the reported final value of `count` the expected value?
11. Are the two threads in the thread3 executing running sequentially or concurrently? Given your understanding of the `pthread_mutex_lock` and `pthread_mutex_unlock` procedures, how many threads can be executing concurrently in the critical section you identified? What prevents or allows this degree of concurrent execution in the critical section? Which choice of words in brackets in the following sentence best summarizes your observations of the execution of `thread3`?

    The use of `pthread_mutex_lock` and `pthread_mutex_unlock` [before/surrounding/in] a critical section [allows/prevents/causes] concurrent execution of threads in the critical section thus [triggering/eliminating] race conditions that lead to [deterministic/non-deterministic] program behavior.