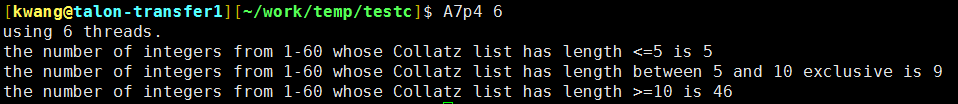
CSCI 3232 Systems Software Assignment 7

Upload all your files to the correct dropbox in Folio before the deadline --- **11:30PM, Apr 13, Saturday, 2019.**

Note: You need to be able to figure out whether there are race conditions in a program and be able to figure out the output without actually running the code. You should also be able to apply mutex and semaphore techniques to synchronize your programs should any race conditions arise.

1. (20 pts) Read given code RaceOrNot1.c and write all possible outputs of the program. Assume there will be no thread creation or joining failures or mutex failures. If you believe there is only one possible output, you just need to write that output.
2. (20 pts) Read given code RaceOrNot2.c and write all possible outputs of the program. Assume there will be no thread creation or joining failures or semaphore failures. If you believe there is only one possible output, you just need to write that output.
3. (20 pts) Read given code RaceOrNot3.c and write all possible outputs of the program. Assume there will be no thread creation or joining failures or semaphore failures. If you believe there is only one possible output, you just need to write that output.
4. (30 pts) From previous homework you are already familiar with the math function *f* defined on positive integers as f(x)=3x+1 if x is odd and f(x)=x/2 if x is even. Given any integer *var*, iteratively applying this function *f* allows you to produce a list of integers starting from *var* and ending with 1. For example, when *var* is 6, this list of integers is 6,3,10,5,16,8,4,2,1, which has a length of 9 because this list contains 9 integers (call this list the Collatz list for 6). Write a C or C++ program **A7p4.c** (or **A7p4.cpp**) that accepts one command line argument which is an integer *n* between 2 and 6 inclusive. Use pthread to create *n* threads to count how many integers between 1 and 60 inclusive have their Collatz list length (a) less than or equal to 5; (b) greater than 5 and less than 10; (c) greater than or equal to 10. You should divide this list generation and length calculating task among the *n* threads as evenly as possible. For example, if *n* is 3, then each thread is supposed to process 20 Collatz lists. Print out the three numbers representing the counts for (a)(b)(c) mentioned above. Use mutex or semaphore to avoid race conditions if necessary. Note: if you do not use pthread to divide the task among the threads, you may get zero points. A sample run of the program is shown below. You do NOT need to submit screen shots. Instead submit your source file together with a working makefile to compile this program.



1. (10 pts) (1) Summarize in your own sentences what are the classical synchronization problems (producer & consumer problem, reader & writer problem, dining philosophers problem). (2) Understand why the four conditions for a deadlock are sufficient and necessary for it to occur. Submit your descriptions of the three synchronization problems and the four conditions for a deadlock.

1.

There is a race condition due to the fact that, even though there are mutex locks around the critical code that modifies the global variable, one of the update methods uses a different mutex variable than the rest. Because of this, it and one of the other update method can modify it at the same time.

Possible Outputs:

a a

b b

a b

b a

2.

Due to this initialization of the semaphore using 2 for the 3rd parameter, this allows 2 threads to edit the critical sections of code that are flanked by semaphore. Due to this, there is a race condition.

Possible Outputs:

a a

b b

a b

b a

3.

Since the value used as the 3rd parameter is 1, only one thread will be allowed to access the critical sections of code that modify the global variables. Due to this, there is no race conditions.

Possible Outputs:

b a

5.

The producer & consumer problem is cause when one thread uses the output of another thread, so the threads have to synced on when to give information or when to grab information. The readers-writers problems is an issue of data sync. When one is editing information, you don't want another to edit it or others to read it while it is being edited. The dining philosophers problem is an issue of limited resources available to threads.

The following is the four conditions needed

1. Mutual Exclusion is when a resource can only be accessed by one thread at a time.
2. Hold and wait is when a thread is waiting for a resource that another thread has.
3. No preemption is when a thread only releases the resource voluntarily after finishing the task
4. Circular wait is where a group of threads are waiting for a resource held by another in a way that each thread is waiting on the next and the last thread is waiting on the first thread.

All of these conditions combined lead to a situation where resources can only be used by one thread and threads will only release them after finishing a task, but all the threads are waiting on another thread to release a resource and none will be able to get resource because of the circular wait.