CSE-381: Systems 2

Due: Wednesday October 30 2019 before 11:59 PM

Email-based help Cutoff: 5:00 PM on Tue, Oct 29 2019

Homework #7

Maximum Points: 17

Submission Instructions

This homework assignment must be turned-in electronically via Canvas. First save this document using your MU ID (example: `raodm_hw7.docx`). Next, type in your responses to each question (right after the question in the space provided) in this MS-Word document. You may use as much space as you need to respond to a given question. Once you have completed the assignment upload the document back to Canvas.

Note that copy-pasting from electronic resources is plagiarism. Consequently, you must suitably paraphrase the material in your own words when answering the following questions.

Name: Noah Dunn

Objective

The objective of this part of the homework is to review and recapitulate (in preparation for Exam #2) various concepts related:

- Threads
- Race conditions
- Critical Sections
- Semaphores and Monitors

Required Reading

First review the following contents related to this homework exercise prior to proceeding with the homework

1. Review slides on Threads and Synchronization off Canvas.
2. Review Chapter 4 and Chapter 6 from the reference e-book "Operating System Concepts" (Link available off Syllabus page in Canvas) prior to proceeding with this exercise.

1. What is a race condition? Explain in sufficient detail on how to race conditions typically manifest (or how race conditions are observed by an user) themselves in programs? [**1 point**]

A race condition is a programmatic error in logic that occurs when a certain operation is dependent on being able to access certain memory at will. Race conditions typically manifest in the form of address sanitization errors when a user engages in operations that can modify memory(usually in mulit-threaded processes)

2. Complete the body of `main` method (without modifying any other part of the program) such that the C++ program will suffer from a potential race condition [**2 points**]

```
#include <iostream>
#include <thread>

void threadMain(std::string& str) {
   // Convert all characters
   for (char& c : str) {
      c++;
   }
}

int main() {

Using std::vector<std::thread> ThreadList;

ThreadList thrList
For(int  I = 0; (I < 10); i++){
    Std::string result = "";
   thrList.push_back(std::thread(threadMain, std::ref(result));
}
For(auto &x : thrList){
   x.join();
}



             return 0;
}
```

Briefly (1-to-2 sentences max) explain how the above code results in a race condition

At each pass of the for loop, a new instance of result is created, destroying the old version. If a thread is not done with using result before it gets destroyed, the method will continue trying to work, throwing a memory error.

3. Describe a critical section using a suitable C++ code fragment (don't write a whole program; but no code, no points. Most likely exam question) [**2 points**]

A critical section is a portion of code where only a single thread is permitted to avoid create substantial race conditions. All other threads have to block For example, take the following method and code block.

```
using ValDict = std::unordered_map<std::string, bool>;

void loadDictionary(std::string filename, ValDict& dict) {
    //  Get our input stream and a variable to store
individual words
    std::ifstream fileRead(fileName); std::string word;
    //  The unordered map we are going to use to our
dictionary in
    while (fileRead >> word) {
        //  Add them to the dictionary with true values
        dict[word] = true;
    }
}
```

```
If we wanted to send multiple items over a network that were
read into a dictionary from a file, we would not want to
begin sending until the dictionary was fully loaded.

If we created a thread,
ValDict dict;
Std::thread x = std::thread(loadDictionary, "inputFile.txt",
dict)

we would want to insure that the thread was finished(using
x.join();) before calling our network sending threads like

std::thread networkThread  = std::thread(sendStuff, dict) to
avoid us using our dictionary before it's done being loaded
```

4. Four conditions must be met to establish an effective critical section in a multi-threaded program. Assume the adjacent `threadMain` method is called from many threads. State each condition. Then briefly (1-or-2 sentences) if-and-how the

```
std::mutex mutex;
int shared = 0;
void threadMain() {
  std::lock_guard guard(mutex);
  shared++;
  // Sleep for few seconds
  sleep(10000 + rand());
}
```

adjacent method meets or violates each condition. [**4 points**]

i. No 2 Threads in the same CS simultaneously: Multiple threads are manipulating the method that is supposed to be engaging in behavior in the critical section, this is a violation.

ii. No assumptions about speed or # of cores/CPUs: The method meets this condition because nothing that occurs within the method is speed or core dependent.

iii. No thread outside a CS may block a thread in the CS: Any thread that calls the lock_guard method will cause an indefinite block in the CS. This is a violation.

iv. No thread should wait too long(max ~1ms wait time) to enter its CS: The only delay here is on exit due to the sleep command, not on entry. This should be valid .

5. What is a semaphore? What is the difference between a semaphore and a mutex? [**2 points**]

   A semaphore is a shared value that is taken care of by the Operating System on our behalf. In any way that we modify that counter value, it will be safe from the other threads while we are changing that. A mutex is the exact same thing, but it is represented in binary.

6. What is a monitor (or a condition variable)? What is the difference between a monitor and a mutex? [**1 points**]

   Monitors are abstracted to a higher degree than our mutexes, as they do not need any mutex to function. They act as variable to get us out of situations where threads may be blocking and we don’t want them to be, given that we have some additional information.

7. What is the producer-consumer model? Give one example scenario where this model is applicable [**1 points**]

   A producer consumer model exists when we need a bunch of tasks added to a line, and we need those tasks processed, taking according to whoever is first in the line. If we have 100 data sets we need to process in batch, our producers would begin adding them on to our line or queue, and our producers would begin processing the data, taking them off the line as soon as possible.

8. What is the dining philosopher's model? Give one example scenario where this model is applicable. [**1 points**]

   The dining philosoper's model occurs when we have multiple processes that need to share data, but we want to insure that a given process has all the pieces it needs before it begins.
   If we are playing a game in which I need a program to evaluate poker odds based on several other players, my program needs to avoid touching anyone else's "hands" until all their decisions have been made and they have placed their cards on the table for me to read.

9. The adjacent method is counting number of "TATA" motifs (or substrings) in a full human `dna` (a 6 GB long string, note that 6 GB is well outside the range of `int`). This method is rather slow. Multithread this method to run faster using '*k*' threads. You may add any number of helper methods and variables as needed. [**3 points**]

```
int countMotifs(const std::string& dna) {
  int count = 0;
  for (size_t i = 0; (i < dna.size()); i++) {
    if (dna.substr(i, 4) == "TATA") {
      count++;
    }
  }
  return count;
}
```

```
 std::mute countMutex;
  int count = 0;
  int k = 20;


int countMotifs(const std::string& dna) {
Using std::vector<std::thread> ThreadList;

ThreadList thrList;

int partition = dna.size() / k;
for(int i = 0 ; I < k ; I++){
   if(  I = k – 1){
  std::string dnaCut = dna.substr(partition * I, dna.size() - partition * I)
    thrList.push_back(std::thread(keepCount, std::ref(dnaCut));

  }
 Else{
 std::string dnaCut = dna.substr(partition * I, partition)
    thrList.push_back(std::thread(keepCount, std::ref(dnaCut));

}
}
For(auto &x : thrList){
    x.join();
}
Return sum


}
void keepCount(const std::string piece){
  for (size_t i = 0; (i < piece.size()); i++) {

if (piece.substr(i, 4) == "TATA") {
     Lock countLock(countMutex);
      count++;
    }

  }

}
```