Assignment Seven: Synchonization Primitives and Critical Section Protection

EECE 4029: Operating Systems and Systems Programming
ELTN 4022: Operating Systems

Department of Electrical Engineering and Computer Science
University of Cincinnati, Cincinnati OH
Revision Date: August 16, 2021

Student Name: **Brian Culberson**

1.1: The assembled version of the above sample code contains these lines:

```
ADD R3,R0,R1        ; R3=a+b
LEA R4, c           ; get address of global variable c
STR R3, R4, #0      ; put result into global variable c
LEA R0, LOCK_C
AND R1,R1,#0
STR R1,R0,#0
```

an optimizing compiler, for reasons that only it knows, decides that doing BLOCK stores and writes to memory are better for performance, so it collects all the memory loads and all of the memory stores into localized blocks as follows:

```
LEA R0, LOCK_C
LEA R4, c           ; get address of global variable c
AND R1,R1,#0
ADD R3,R0,R1        ; R3=a+b
STR R1,R0,#0
STR R3,R4, #0       ; put result into global variable c
```

What is *wrong* with this "optimized" version of the code IF you were actually running it in a multi-threaded environment even if it does all the same things as the original code in terms of modifying memory as needed when it ALL runs? Be specific in your answer. (10 points)

Since the lock check is after getting the variable and adding the variable, it is doing unneeded steps to get the problem completed. I would run the steps first to check if it should even be doing the operation. Reading values at the same time at the same register I would think is a bad idea to do.

1.2: Consider again this lock code:

```
inline void lock(lock_variable_type *lock_variable)
 { asm volatile ("":::  "memory");
   while (*lock_variable);
```

```
    *lock_variable = 1;
 }
```

Is there a possible race condition in here? Explain where it is. You should use the assembled version of the call to lock provided earlier in this writeup as part of your explanation. (10 points)

For this to work there needs to be an atomic operation on the lock variable. Since there is not atomic operation on the lock variable, it is a local variable, there can be a race condition here.

1.3: Assume you are multi-threading your code and multiple threads are making calls to the given `lock()` code and accessing the same lock. Would you consider this implementation wasteful of CPU? Why or why not? (10 points)

It is wasteful depending on how fast the lock Is opened. If it is going to take 5 minutes, then there should be a different way to complete the lock. If it is only going to take less than a second, then it should be checking whenever the lock is clear.

1.4: Assume you are multi-tasking your code and multiple threads were being multitasked on a SINGLE CPU. Is the given code for `lock()` dangerous in a way different than introducing a race condition? If so, how so. Explain your answer. (10 points)

if the code is multitasked through one cpu there still can be a race condition if it decides to go to another thread in the middle of the lock sequence.

IN ADDITION TO YOUR ANSWERS TO THE ABOVE QUESTIONS TURNED IN AS A SINGLE PDF. YOU SHOULD ALSO TURN IN A TWO C SOURCE CODE FILES AS A TEXT FILES THAT CONTAINS YOUR IMPLEMENTATION OF A READER-WRITER LOCK AS DETAILED IN THE HOMEWORK WRITEUP. YOU SHOULD ALSO ANSWER THIS QUESTION ABOUT YOUR IMPLEMENTATION HERE (60 POINTS)

What happens when there's MANY reader threads? (20 of 60 points): Use the Unix\Linux time command to determine how long (in wall clock, real time) it takes for the sample program to finish when setting the reader (checksum) thread count to 1, 10, and 20. You may or may not see a trend in how long it takes the process to complete for different numbers of reader threads. In your own words, explain WHY you are seeing what you are seeing. Note that what you see may very well depend on HOW you implement the reader-writer lock primitives. Provide your observations in written form, with any graphs or diagrams you consider appropriate

With the reader threads the biggest thing that I see is it gets longer and longer to complete. This makes sense since there are more threads that must check to see if the sum is correct. With 1 and 5 threads it does complete but with the 20 threads it didn't really like that. It was using my whole CPU to complete, and I shut it off after 2 min of run time, mostly because the fans started to get loud. This could be a deadlock but I wasn't sure how to test this since it worked for all other kinds of thread combination.