**Score: \_\_\_\_\_**

**OSA7 – Race Conditions & Mutual Exclusion**

**Activities**

COMP256 – Computing Abstractions

Dickinson College

Spring 2023

Prof. Grant Braught

**Name:**

**Introduction:**

Threads are a high-level language abstraction and if we are not fully aware of some of the underlying details, we will find ourselves unable to fully explain their behavior.

This is very similar to earlier in the course when we were unable to explain why the following program did not contain an infinite loop until we understood that computers use two’s complement representations for int values:

int x = 1;

while(x > 0) {

x++;

}

Similarly, threads will exhibit behavior that is surprising, until we peel away the high-level language layer of abstraction and look at the details that it hides from us.

In today’s activities you will observe some of these surprising effects, come to understand the reasons behind them and one of the techniques for controlling them. More specifically, you will understand these effects as *race conditions* and see how *atomic execution* via *locking* with the Java synchronized mechanism can be used to protect *critical sections* avoid *race conditions* in multithreaded programs.

**The High-Level Language Abstraction:**

We’ll first examine, from a high-level language perspective, a multithreaded program that exhibits some surprising behavior.

🔑 1. Consider the ThreadRaceCondition2 example:

* <https://replit.com/@braughtg/ThreadRaceCondition2>

a. What are the names of the classes that define the Thread objects referred to by the variables ta and tb?

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | **Variable** | **Thread Subclass** |  |
|  | ta |  |  |
|  | tb |  |  |
|  |  |  |  |

b. What variable is shared by the two threads that creates the race condition?

c. If just the Thread referred to by ta were run, what value would the program produce?

d. If just the Thread referred to by tb were run, what value would the program produce?

e. If the full program with both threads did not contain a race condition, what value should it produce?

2. Now run the ThreadRaceCondition2 example 5 times and list the values generated in the table below:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | **Run** | **z** |  |
|  | 1 |  |  |
|  | 2 |  |  |
|  | 3 |  |  |
|  | 4 |  |  |
|  | 5 |  |  |
|  |  |  |  |

By running this program, you should see that the result will not always be as expected. Clearly the output might be as expected. However, it may also be some value either larger or smaller than expected as well. If you did not observe results that were as expected, and larger than expected, and smaller than expected, you should run the program a few more times until you convince yourself that each is possible.

**The View from the Assembly Language Level:**

The behavior of the above program may be surprising and seem a bit odd. The Add thread adds 3 to z 500,000 times and the Subtract thread subtracts 2 from z 500,000 times. Thus, if both Add and Subtract execute their net effect should be the same as adding 1 to z 500,000 times. However, as you saw that isn’t what typically happens. To understand the behavior of this program we’ll have to peel away a few layers of abstraction. We’ll need to consider both how this program might translate to assembly language and how the resulting threads are handled by the operating system.

🔑 3. The table below shows the high-level language lines of code from the Add and Subtract threads that do the addition and subtraction operations.

The Assembly row in the Add column shows how the addition statement might be translated into assembly language. The value at main memory address Z would be loaded from the main memory into R0, 3 would be added to R0, and then that result copied from R0 back into main memory at address Z.

Fill in the translation for the Subtract thread in the Subtract column on the right.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  | **Add** | **Subtract** |  |
|  | **HLL** | z = z + 3; | z = z - 2; |  |
|  |  | LOAD R0 Z |  |  |
|  | **Assembly** | ADD R0 R0 #3 |  |  |
|  |  | STORE R0 Z |  |  |
|  |  |  |  |  |

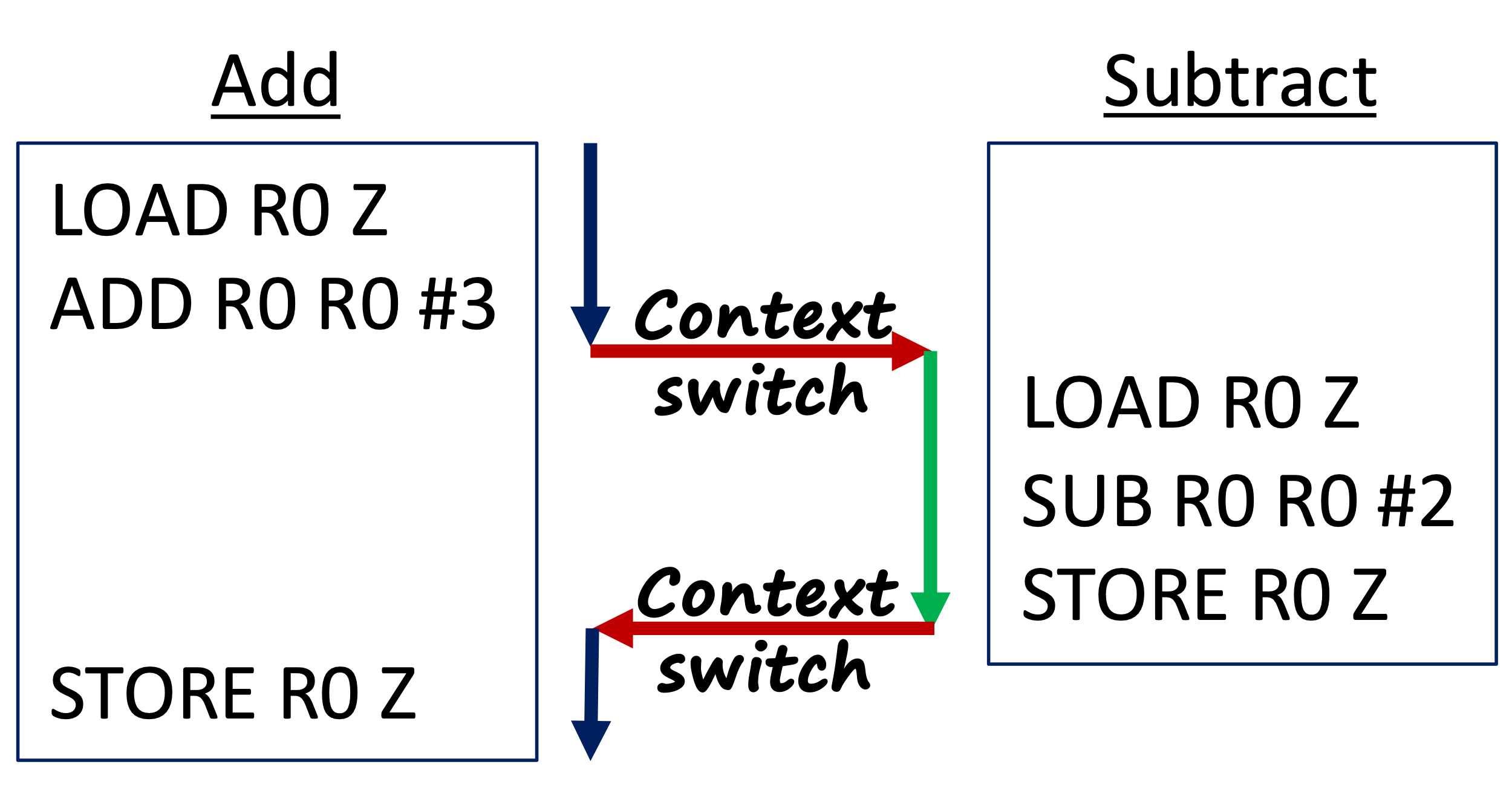
Now recall that each thread has its own execution context, and that the operating system will use time sharing to run them. The scheduler will switch the threads between the ready and running states using timer interrupts. For example, if the OS schedules the Add thread to run, its execution context will be loaded onto the CPU and the CPU will begin executing its instructions. When a timer interrupt occurs, the OS will save the context of the Add thread and will eventually schedule the Subtract thread to run. The context of the Subtract thread will be loaded onto the CPU and its instructions will begin to be executed. The diagram below shows one pattern in which this might happen:



In the above diagram, the blue arrow (on the left) represents the CPU executing instructions from the Add thread. The red arrow (in the middle) represents a timer interrupt and the OS scheduler switching contexts from the Add thread to the Subtract thread. And finally, the green arrow (on the right) represents the CPU executing instructions from the Subtract thread.

🔑 4. If you assume that the value of Z is 0 before the sequence of instructions shown in the diagram above is executed, what will the value of Z be when both threads are complete (i.e. when the end of the green arrow is reached)?

🔑 5. The order in which the statements are executed in the figure above is just one possibility. The figure below shows another possibility, this time with two context switches.



This question investigates how this change in order affects the outcome of the computation. Assume again that the value of Z is 0 before the instructions above are executed. Also, keep in mind that Add and Subtract each have their own execution context (i.e. their own copy of R0), but that they share the same Z.

a. Assuming the above sequence of instructions, what value what value will be placed into R0 when the Add thread executes the instruction: ADD R0 R0 #3?

b. What value will be loaded into R0 when the Subtract thread executes the instruction: LOAD R0 Z?

c. What value will be placed into R0 when the Subtract thread executes the instruction: SUB R0 R0 #2?

d. What value will be stored into Z when Subtract executes the instruction:   
STORE R0 Z?

e. What value will be stored into Z when Add finally executes the instruction:   
STORE R0 Z?

6. Give a figure similar to those shown above that illustrates a sequence of execution that results in the value of Z being -2 at the end.

7. Give a figure similar to those shown above that illustrates a different sequence from those you’ve seen already, but that will also result in the correct value of Z=1 at the conclusion of both threads.

🏆 8. Now consider the two threads shown below. Assuming that the variable M has the initial value 10 and is shared by Thread 1 and Thread 2, what are all of the possible values that M might have after both threads complete?



**Race Conditions:**

The examples above in questions 5, 6 and 8, where the sequence in which the instructions in the threads are executed affects the result are called race conditions. A *race condition* exists when the exact ordering of the execution of the (assembly/machine language) instructions in two or more threads can affect the result. This name makes some sense because in such situations it is a like a “race” between the instructions in the two threads (i.e. their order of execution) that determines the final result.

It is not required viewing, but if you would like to see a real-world example of how race conditions can cause problems check out the video *What is Race condition in Operating System: Real Life Example* from the HowTo channel.

* <https://www.youtube.com/watch?v=s8_ZxcG7Jco> (5:33)

**Mutually Exclusive Execution:**

From the above examples and from class it should be clear that having race conditions in a program is problematic.

9. Now, consider questions #4 and #7 from above, where in both cases the correct result was obtained. These two examples suggest a way that race conditions can be avoided. What was different about the order of execution in questions #4 and #7 as compared to the orders in the other questions (i.e. #5 and #6)? In other words, why do we get the correct result in #4 and #7 and the wrong answer in #5 and #6?

As it turns out, if we can in some way force all three of the statements in Add to execute before the corresponding three statements in Subtract, or vice versa, then the race condition would be eliminated. That is, we want to prevent the instructions from interleaving as they did in the examples in questions #5 and #6. The technical term for preventing these interleaved executions is *mutually exclusive execution*. This simply means that once one critical section starts, it will complete before the other will be allowed to start. That is, the critical sections execute mutually exclusively.

Each programming language that supports multi-threading will have its own unique syntax for enforcing mutual exclusion and most have several ways. Java is no exception and has numerous ways that it can be accomplished. We will look at just one, called *synchronized blocks*.

10. Consider the ThreadMutualExclusion2 example:

* <https://replit.com/@braughtg/ThreadMutualExclusion2>

Comparing this program to the ThreadRaceCondition2 program from questions #1 and #2 there are only a few small changes. But these changes will eliminate the race condition from the program.

a. What is the data type of the object that is created on line 3 of this program and what is the name of the variable that references that object?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | |  |  |  |
|  | Object Data Type | |  |  |
|  | Reference Name | |  |  |
|  | |  |  |  |

b. Compare the Add and Subtract classes in ThreadMutualExclusion2 to those in ThreadRaceCondition2. What statement was added to the run method in each of these classes?

11. Run the ThreadMutualExclusion2 example 5 times

a. List the values generated in the table below:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | **Run** | **z** |  |
|  | 1 |  |  |
|  | 2 |  |  |
|  | 3 |  |  |
|  | 4 |  |  |
|  | 5 |  |  |
|  |  |  |  |

b. Do your results suggest that the race condition has been removed? Briefly explain why you think it has or has not based on part i and the results you observed earlier in question #2.

c. Do you think that running the program 5 times is sufficient to prove that the program no longer contains a race condition? How many times would be sufficient?

12. Consider the program ThreadMutexExercise:

* <https://repl.it/@braughtg/ThreadMutexExercise>

a. Study the ThreadMutexExercise program. If it did not contain a race condition, what should the output be when it is run?

b. Run the ThreadMutexExercise program 5 times and list the values generated in the table below:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | **Run** | **output** |  |
|  | 1 |  |  |
|  | 2 |  |  |
|  | 3 |  |  |
|  | 4 |  |  |
|  | 5 |  |  |
|  |  |  |  |

c. As the results in part b show, the ThreadMutexExercise program does in fact contain a race condition. Fork the ThreadMutexExercise and modify it to use a lock to remove the race condition. Be sure to run your program enough times to convince yourself that you have successfully removed the race condition. Include the full program below as your answer to this question.

**Taking it further wth Synchronized Methods:**

**This section is optional.**

As mentioned earlier, Java provides a number of different ways to remove race conditions by ensuring mutual exclusion. One is the synchronized blocks that we have seen above. Another is through synchronized methods. Synchronized methods are a lot like synchronized blocks. When a method is declared synchronized, it is the functional equivalent of placing the entire body of the method inside a synchronized block and using the current object (i.e. this) as the lock variable. You can read about synchronized methods in the Java Tutorial:

* <https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html>

🏆 🏆 13. Fork the ThreadMutexExercise class from question #13 again and refactor it by adding a class that encapsulates the shared variable and provides synchronized methods for modifying it that ensure that race conditions cannot exist between the threads. Include the full code of your program as your solution to this question.

**Some Final Thoughts:**

Race conditions are just one of many challenges that exist in writing multithreaded programs. The Operating Systems course presents some of the other challenges and some of the mechanisms that have been developed to help programmers address them. There are also complete courses at the undergraduate and graduate levels on concurrent programming that deal solely with how to write multithreaded programs so that they work correctly. Getting these programs to work correctly is notoriously difficult, in part because they require programmers to constantly be thinking across levels of abstraction to identify the ways in which they might go wrong. For this reason, the development of new high-level language abstractions that help programmers write correct (e.g. race condition free) multithreaded programs is an active research area.

Optional: To help me improve and scope these activities for future semesters please consider providing the following feedback.

a. Approximately how much time did you spend on this activity outside of class time?

b. Please comment on any particular challenges you faced in completing this activity.