Name: Jude Onyia

Student ID: V00947095

Course: ECE 596C

Due Date: July 29, 2020

Assignment 6: Non – Programming Exercise

7.1 a)

This program creates two threads with shared variables x and y initially set to zero. Thread *t1* assigns the value 1 to x, then assigns the value 2 to y. While thread *t2* prints the value of y, then the value of x. Both threads are allowed to finish before the program ends. The program contains two possible data races: thread *t1* could writing to x at the same time that thread *t2* accesses x to print its value, and thread *t1* could be writing to y at the same time that thread *t2* accesses y to print its value. These two data races can be fixed by using mutexes to make the access of each variable mutually exclusive as shown below.

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

int x = 0;
int y = 0;
std::mutex m_x;
std::mutex m_y;

int main()
{
        std::thread t1([ ] () {
                std::unique_lock lock_x(m_x, std::defer_lock);
                std::unique_lock lock_y(m_y, std::defer_lock);
                lock_x.lock();     x = 1;       lock_x.unlock();
                lock_y.lock();     y = 2;       lock_y.unlock();
        });
        std::thread t2([ ] () {
                std::unique_lock lock_x(m_x, std::defer_lock);
                std::unique_lock lock_y(m_y, std::defer_lock);
                lock_y.lock();     std::cout << y << " ";     lock_y.unlock();
                lock_x.lock();     std::cout << x << std::endl;     lock_x.unlock();
        });
        t1.join();
        t2.join();
}
```

Once these data races have been resolved, considering the SC-DRF memory model, there are six possible sequentially consistent orders that the program could follow, these are listed below along with the possible printed values of the variables. There is freedom for instructions A1 and B1 to occur concurrently. Also, instructions A2 and B2 can occur concurrently.

A1: x = 1,   A2: y = 2,   B1: std::cout << y << " ",   B2: std::cout << x << std::endl

First instruction	Second instruction	Third instruction	Fourth instruction	Printed value of variable x at end of program	Printed value of variable y at end of program
A1	A2	B1	B2	1	2
A1	B1	A2	B2	1	0
A1	B1	B2	A2	1	0
B1	A1	A2	B2	1	0
B1	A1	B2	A2	1	0
B1	B2	A1	A2	0	0

7.1 b)

This program is similar to the previous, it creates two threads with shared variables x and y initially set to zero. Thread *t1* assigns the value 1 to x, then assigns the value 2 to y. While thread *t2* prints the value of y, then the value of x. Both threads are allowed to finish before the program ends. However, this program uses a single mutex to make the access to both variables mutually exclusive. This results in having no data races; however, this puts the entire code executed by each thread in a *happens-before* relationship. This causes only two possible sequentially consistent orders that the program could follow, as shown below. One case is that thread *t1* locks the mutex first, executes its code, and unlocks the mutex; thread *t2* can now lock the mutex, execute its code, and unlocks. The other case is vice-verse, thread *t2* locks and executes first, then thread *t1* follows. This forces the program to execute like it were a fully synchronous program.

A1: x = 1,   A2: y = 2,   B1: std::cout << y << " ",   B2: std::cout << x << std::endl

First instruction	Second instruction	Third instruction	Fourth instruction	Printed value of variable x at end of program	Printed value of variable y at end of program
A1	A2	B1	B2	1	2
B1	B2	A1	A2	0	0

7.1 g)

This program creates two threads with shared variables *x* initially set to zero and *done* initially set to *false*. Thread *t1* assigns 42 to *x*, then assigns *true* to *done*. Thread *t2* actively reads the *done* and does not stop until *done* is *true*, then *t2* uses an assertion to check if *x* is equals to 42. Both threads are allowed to finish before the program ends. With the way the program is written, there is only one possible data race that could occur: if *t1* is writing to *done* while *t2* is reading the state of *done*. This can be fixed by using a condition variable instead of actively checking *done* in *t2*. Due to the way the program is written, there are three possible sequentially consistent orders that the program could follow, as shown below. This is because B2 can not occur before A2, *t2* will continue to execute B1 until A2 has been executed. Therefore, the assert in B2 will always be true.

A1: x = 42,   A2: done = true,   B1: while (!done),   B2: assert(x == 42)

First instruction	Second instruction	Third instruction	Fourth instruction
A1	A2	B1	B2
A1	B1	A2	B2
B1	A1	A2	B2

7.1 i)

This program creates two threads with shared variables x and y initially set to zero. Thread *t1* checks if *x* is 1, and if it is, sets *y* to 1. Thread *t2* checks if y is 1, and if it is, sets *x* to 1. Both threads are allowed to finish before the program ends. Due to the way the program is written, there is no data race. There are only two possible sequentially consistent orders that the program could follow:

1. Thread *t1* checks the value of *x* in the if-statement which will result in *false*, thread *t2* checks the value of *y* in the if-statement which will result in *false* as well.
2. The vice-versa occurs where *t2* checks first, then *t1*.

7.1 l)

This program creates two threads with a shared variable *counter* initialized to zero. Each thread tries to increment *counter* 100,000 times. This will cause a data race when both threads try to write to *counter* at the same time. This can be fixed by making the access to *counter* mutually exclusive using a mutex. There is a wide number of ways in which the 100,000 incrementations of *counter* from each thread can be interleaved.

7.1 m)

This program creates two threads with a shared object *w* of class *Widget*. Objects of class *Widget* have two variables *x* and *y*, and two mutexes *xMutex* and *yMutex*. Thread *t1* locks *xMutex* and sets *x* of object w to 1. Thread *t2* locks *yMutex* and sets *y* of object w to 1. There are no data races in this program. Since the threads are locking different mutexes, line 18 and line 24 can execute concurrently. In a sequentially consistent order, either line 18 could be executed first, then line 24, or vice-versa.

7.10)

A1: $x = 1;$

A2: $a = y;$

B1: $y = 1;$

B2: $b = x;$

All possible sequentially consistent executions of this program

First instruction	Second instruction	Third instruction	Fourth instruction	Value of variable a	Value of variable b
A1	A2	B1	B2	0	1
A1	B1	A2	B2	1	1
A1	B1	B2	A2	1	1
B1	A1	A2	B2	1	1
B1	A1	B2	A2	1	1
B1	B2	A1	A2	1	0

There is one combination of that cannot be obtained for a and b: where $a = 0$ and $b = 0$.

7.12 a)

In this scenario, the **assertion in instruction B2** will be **true sometimes**. The only two possible sequentially consistent executions that reaches the line of the assertion and makes it true are {A1,A2,B1,B2} and {A1,B1,A2,B2}, shown in the table below. Where A1 executing before B1 allows the program to reach the assertion line and A2 executing before B2 makes the assertion true. The execution order that reaches the assertion line and makes it false is {A1,B1,B2,A2}, where A1 executing before B1 allows the program to reach the assertion line and B2 executing before A2 makes the assertion false. Other sequentially consistent execution orders do not reach the assertion line because B1 is executed before A1.

First instruction	Second instruction	Third instruction	Fourth instruction	State of assertion in B2
A1	A2	B1	B2	True
A1	B1	A2	B2	True
A1	B1	B2	A2	False
B1	A1	A2	B2	Not reached
B1	A1	B2	A2	Not reached
B1	B2	A1	A2	Not reached

7.12 b)

In this scenario, if the program can reach the **assertion in instruction B2**, the assertion will **always be true**. This is because there is only one possible sequentially consistent execution order that can reach the assertion: {A1,A2,B1,B2}, this is because A2 is executed before B1. In this case, since A1 must be executed before A2 and B1 must be executed before B2, this results in A1 executing before B2, causing the assert to be true. Other sequentially consistent execution orders do not reach the assertion line because B1 is executed before A2. Also, it is not possible for B2 to execute before A2 because thread 2 actively executes B1 until thread 1 executes A2, therefore, B2 cannot be executed until thread 2 stops executing B1 which happens after thread 1 executes A2. This case is marked in red.

First instruction	Second instruction	Third instruction	Fourth instruction	State of assertion in B2
A1	A2	B1	B2	True
A1	B1	A2	B2	Not reached
A1	B1	B2	A2	Not reached
B1	A1	A2	B2	Not reached
B1	A1	B2	A2	Not reached
B1	B2	A1	A2	Not reached

7.12 c)

In this scenario, the program reaches both assertions only in four possible sequentially consistent execution orders as shown in below, where A2 is executed before B1. The **assertion in instruction B2 will always be true** if the program reaches that assertion. This is because in those four possibilities, A1 is executed before B2. The **assertion in instruction B3 will be true sometimes**. Only if A3 is executed before B3, then the assertion is true, this is the case in 3 out of the 4 possibilities. In the fourth, the assertion is false because B3 is executed before A3. Also, it is not possible for B2 or B3 to execute before A2 because thread 2 actively executes B1 until thread 1 executes A2, therefore, B2 or B3 cannot be executed until thread 2 stops executing B1 which happens after thread 1 executes A2. This case is marked in red.

First	Second	Third	Fourth	Fifth	Sixth	State of assertion in B2	State of assertion in B3
A1	A2	A3	B1	B2	B3	True	True
A1	A2	B1	A3	B2	B3	True	True
A1	A2	B1	B2	A3	B3	True	True
A1	A2	B1	B2	B3	A3	True	False
A1	B1	A2	A3	B2	B3	Not reached	Not reached
A1	B1	A2	B2	A3	B3		
A1	B1	A2	B2	B3	A3		
A1	B1	B2	A2	A3	B3		
A1	B1	B2	A2	B3	A3		
A1	B1	B2	B3	A2	A3		
B1	A1	A2	A3	B2	B3		
B1	A1	A2	B2	A3	B3		
B1	A1	A2	B2	B3	A3		
B1	A1	B2	A2	A3	B3		
B1	A1	B2	A2	B3	A3		
B1	A1	B2	B3	A2	A3		
B1	B2	A1	A2	A3	B3		
B1	B2	A1	A2	B3	A3		
B1	B2	A1	B3	A2	A3		
B1	B2	B3	A1	A2	A3		

Part C Report and Comment on the Julia set program

The average amount of time required for the execution of the *compute_julia_set* function is recorded below. The results obtained are to be expected because the system that the program ran on utilizes an intel core i7-8565u processor which has 4 cores. Therefore, it makes sense that the fastest computation will typically occur with 4 threads running, where the system can fully utilize the hardware by delegating each core to run one thread if the system is able to. Also, as expected, running the program with multiple threads improves performance in terms of speed compared to running the program with a single thread.

Threads	Time (seconds)
1	0.973576
2	0.502833
4	0.305151
8	0.378237