Ryland Nezil
V00157326
SENG 475

For the julia set computation it was found that, on average, computation with:

- 1 thread took 2.64 seconds;
- 2 threads took 1.34 seconds;
- 4 threads took 0.69 seconds; and
- 8 threads took 0.36 seconds.

Therefore, doubling the threads used for computation generally corresponded to halving the computation time. These results were expected, as 2 threads executing in parallel should yield double the performance of a single thread executing sequentially. Furthermore, the machine this test was carried out on possessed an 8-core 8-thread processor, thus allowing it to take full advantage of all the threads used in this example.

**7.1**

*a)*

This assertion is true **sometimes**. Justification: the following two orders of execution are valid:

```
x = 1;
if( x == 1 )
     assert( y == 1 ); // FAIL
y = 1;
```

```
x = 1;
y = 1;
if( x == 1 )
     assert( y == 1 ); // PASS
```

*b)*

This assertion is true **always**. Justification: thread 2 cannot ever reach the assertion checking that x == 1 until y becomes a non-zero value, and x is always equal to 1 by the time y == 0 becomes false.

*c)*

This assertion is true **sometimes**. Justification: the following two order of execution are valid:

```
x = 1;
y = 1;
z = 1;
while( !y ) {}
assert( x == 1 ); // PASS
assert( z == 1 ); // PASS
```

```
x = 1;
y = 1;
while( !y ) {};
assert( x == 1 ); // PASS
assert( z == 1 ): // FAIL
z = 1;
```

**7.3**

```
x = 1;
a = y;
y = 1;
b = x;
// a == 0, b == 1
```

```
x = 1;
y = 1;
a = y;
b = x;
// a == 1, b == 1
```

```
x = 1;
y = 1;
b = x;
a = y;
// a == 1, b == 1
```

```
y = 1;
x = 1;
a = y;
b = x;
// a == 1, b == 1

y = 1;
x = 1;
b = x;
a = y;
// a == 1, b == 1

y = 1;
b = x;
x = 1;
a = y;
// a == 1, b == 0
```

The combination a == 0, b == 0 is never possible.

**7.4**

Note to marker: whenever I talk about thread "grabbing a mutex", I mean grabbing the same mutex. I understand that each thread grabbing a different mutex does not accomplish the desired result.

*a)*

Program outputs "1 2" to stdout. Data races occur due to thread 1 trying to write x while thread 2 tries to read x, and due to thread 1 trying to write y while thread 2 tries to read y. These could be remedied by having both threads grab a mutex as soon as they begin executing, and having thread 2 test whether y == 2 after grabbing the mutex; if this test fails, thread 2 should wait to be notified by a condition variable. Thread 1 should notify the condition variable as its last act.

*b)*

Program outputs "1 2" to stdout. No data races occur, however program may not always produce the desired output. This could be fixed by implementing a condition variable identically to as described in part *a)*.

*g)*

Program sets variable x to value 42 then asserts x == 42. Data race occurs due to thread 1 trying to write done while thread 2 tries to read done. This could be solved by having thread 1 grab a mutex before trying to write done, and having thread 2 grab a mutex as soon as it begins executing. Then, inside the while loop in thread 2, it should wait on a condition variable. Thread 1 should notify this condition variable after writing to done.

*i)*

Program "measures" which thread begins executing fastest by having the last thread to execute set either x or y to 1. Data races occur due to thread 1 trying to write y while thread 2 tries to read y, and due to thread 1 trying to read x while thread 2 tries to write x. This could be fixed by having both threads grab a mutex immediately upon beginning their execution.

*l)*

Program increments counter up to 200,000. Data race occurs due to threads 1 and 2 trying to write to counter at the same time. This could be solved by having both threads grab a mutex as their first act during execution.

*m)*

Program sets data members x and y of Widget object w to 1. No data races occur since the threads are reading to and writing from different memory locations.