1.0 | Introduction

The purpose of this report is to discuss a range of possible outcomes when code in Figure 1 is executed with atomic reads and writes and assuming a sequentially consistent model with shared memory. Firstly, the definition of a sequentially consistent model will be presented. Secondly, a discussion of possible outcomes of the code in Figure 1 will be discussed. The discussion will be broken down into scenarios under which a) the program terminates and final values of variables are stable, b) a race condition between a read and write occurs in the program but the program terminates and c) the program does not terminate. For the purposes of reference clarity, let us name the first thread starting with `while` *thread-A* and the second thread starting with `<await ...>` *thread-B.*

```
1     int x = 10, y = 0;
2     co
3         {
4             while (x != y) {
5                 x = x - 1;
6             }
              y = y + 1;
8         }
9         //
10        {
11            <await (x == y); >
12            x = 8;
13            y = 2;
14        }
15    oc
```

Figure 1 - Code to execute under a sequentially consistent model. (red) thread-A, (blue) thread-B

1.1 | Sequential Consistency

The model of sequential consistency can be defined with the following rules:

1. “ordering of atomic actions (particularly reads and writes to memory) from any one thread have to occur in normal program order
2. atomic actions from different threads are interleaved arbitrarily (ie in an unpredictable sequential order, subject only to rule 1)” [1]

Therefore, we can effectively apply the model by interleaving actions of both threads in a sequential nature. This approach allows us to better reason about the execution timeline as well as reason about the state of data in the program.

1.2 | Shared Memory & Atomic read/write

The model of shared memory encompasses the concept in which all concurrent execution threads have access to the same memory locations. Taking Figure 1 as an example, both concurrent threads have access to variables `x` and `y`. Memory access are atomic, therefore if the memory is snapshotted before a read or write, the value retrieved by an individual process will be the same as in the snapshot taken right before the read or write. However, operations such as `a = b + 1` or `x == y`, are not considered atomic. For example, `a = b + 1` will use the following instructions:

1. `load b into register_1`
2. `add register_1, 1 and put result into register_2`
3. `store register_2 into a`

From the above, we can see that this operation in fact requires multiple instructions to be performed and therefore is not considered an atomic operation.

2.0 | Initial Analysis

Working backwards from the condition under which each thread is capable to begin execution, we can see that *thread-A* is able to execute immediately as there is no blocking condition. On the other hand, *thread-B* cannot execute until `x == y`, which are initially set to 10 and 2 respectively. Therefore, the only thread that can begin execution initially is *thread-A*.

As a result, *thread-A* will execute a sequence of actions outlined in Figure 2 before the ‘interesting’ interleaving of the two threads can happen. In future sections, the sequence of actions will be starting at time 20 in order to simplify diagrams. Intermediate iterations of the while loop have been skipped for clarity.

Time	Instruction	Memory
1	`x != y  ## while condition check, true`	`x = 10, y = 0`
2	`x = x - 1`	`x = 9, y = 0`
3	`x != y  ## while condition check, true`	`x = 9, y = 0`
⋮	⋮	⋮
19	`x != y  ## while condition check, true`	`x = 1, y = 0`
20	`x = x - 1`	`x = 0, y = 0`

Figure 2 - Thread-A executes initially as thread-B is blocked.

2.1 | Termination Scenarios

In this section, possible interleaving of the two threads which result in a termination scenario will be considered. For clarity, only scenarios with 'interesting' or possibly unintended interleavings will be considered and therefore a full enumeration of all possible interleavings will not be presented as some interleavings do not have any impact on the final state of the program nor the memory state.

2.1.1 | Scenario 1

Let us consider a scenario in which *thread-B* is interleaved in right after the value of x is decreased to match that of y. It is therefore able to atomically check the result of `x == y` and unblock. Figure 3 shows a sequentially consistent timeline of execution.

Time	Thread-A	Thread-B	Memory
20	`x = x - 1`		`x = 0, y = 0`
21	`x != y ## false`		`x = 0, y = 0`
22		`<await (x == y); >`	`x = 0, y = 0`
23	`y = y + 1 ## 0 + 1`		`x = 0, y = 1`
24	`[terminated]`	`x = 8`	`x = 8, y = 1`
25		`y = 2`	`x = 8, y = 2`
26		`[terminated]`	`x = 8, y = 2`

Figure 3

In Figure 3, we can observe that only instruction executed by *thread-B* before *thread-A* terminates is unblocking of the execution order. *Thread-A* sets the value of `y`, however, *thread-B* overwrites it in the subsequent instructions. The resulting state of the variables is `x = 8, y = 2.`

2.1.2 | Scenario 2

Let us now consider a scenario in which overwriting of the final `y` value happens inside *thread-A* rather than *thread-B*. Figure 4 shows a sequentially consistent timeline for this scenario.

Time	Thread-A	Thread-B	Memory
20	`x = x - 1`		`x = 0, y = 0`
21	`x != y ## false`		`x = 0, y = 0`
22		`<await (x == y); >`	`x = 0, y = 0`
23		`x = 8`	`x = 8, y = 0`
24		`y = 2`	`x = 8, y = 2`

25	`y = y + 1  ## 2 + 1`	`[terminated]`	`x = 8, y = 3`
26	`[terminated]`		`x = 8, y = 3`

Figure 4

The value of `y` is first set by *thread-B* and subsequently incremented by *thread-A.* The final state of variables after both threads terminate are `x = 8, y = 3.`

2.1.3 | Scenario 3

Let us now consider a scenario where thread-B executes immediately when after it's blocking condition is satisfied.

Time	Thread-A	Thread-B	Memory
20	`x = x - 1`		`x = 0, y = 0`
21		`<await (x == y); >`	`x = 0, y = 0`
22		`x = 8`	`x = 8, y = 0`
23		`y = 2`	`x = 8, y = 2`
24	`x != y  ## true`	`[terminated]`	`x = 8, y = 2`
25	`x = x - 1`		`x = 7, y = 2`
26	`x != y  ## true`		`x = 7, y = 2`
27	`x = x - 1`		`x = 6, y = 2`
⋮	⋮		⋮
32	`x != y  ## true`		`x = 3, y = 2`
33	`x = x - 1`		`x = 2, y = 2`
34	`x != y  ## false`		`x = 2, y = 2`
35	`y = y + 1  ## 2 + 1`		`x = 2, y = 3`
36	`[terminated]`		`x = 2, y = 3`

Figure 5

Figure 5 shows the scenario where *thread-B* updates the value of `x` before the while loop terminates and looping continues until `x == y` again. The stable values of this scenario are `x = 2, y = 3.`

2.1.4 | Scenario 4

Let us consider a scenario similar to Scenario 3 with the difference that the value of `y` is only set from thread-B at as the last instruction of the program.

Time	Thread-A	Thread-B	Memory
20	x = x - 1		x = 0, y = 0
21		<await (x == y); >	x = 0, y = 0
22		x = 8	x = 8, y = 0
23	x != y ## true		x = 8, y = 0
24	x = x - 1		x = 7, y = 0
25	x != y ## true		x = 7, y = 0
26	x = x - 1		x = 6, y = 0
⋮	⋮		⋮
35	x != y ## true		x = 1, y = 0
36	x = x - 1		x = 0, y = 0
37	x != y ## false		x = 2, y = 0
38	y = y + 1 ## 1 + 1		x = 2, y = 1
39	[terminated]	y = 2	x = 2, y = 2
40		[terminated]	x = 2, y = 2

Figure 6

Figure 6 illustrates the scenario where the order of setting the value of `y` inside *thread-B* influences the number of iterations *thread-A* makes inside the while loop.The final values are `x = 2, y = 2.`

2.2 | Termination with read/write race condition

In this section, we consider scenarios under which race conditions occur in statements where values are first read into registers, operated on and then written back to memory.

Time	Thread-A	Thread-B	Memory
20	`x = x - 1`		`x = 0, y = 0`
21	`x != y ## false`		`x = 0, y = 0`
22		`<await (x == y); >`	`x = 0, y = 0`
23		`x = 8`	`x = 8, y = 0`
24	`k = y ## 0, instruction level`		`x = 8, y = 0`
25		`y = 2`	`x = 8, y = 2`
26		`[terminated]`	`x = 8, y = 2`
27	`m = k+1 ## m = 0 + 1, instruction level`		`x = 8, y = 2`
28	`y = m ## instruction level`		`x = 8, y = 1`

Figure 7

Thread-A begins the execution of `y = y + 1`, however, the instruction takes longer to execute than setting the value of `y` to `2` inside *thread-B* atomically. *Thread-A* sees an incorrect value of `y` as it has began executing a longer statement. The final value of `y` is independent of *thread-B* despite clearly writing to the same memory location. The final observed values are `x = 8, y = 1.` This is not a different final result from previous scenarios, however, the method of reaching the value is different.

A similar case of non-atomicity of binary operations can be observed in other areas of the program, specifically then in the following statements:

1. `x != y`
2. `x = x - 1`
3. `y = y + 1`
4. `x == y`

Given the particular program, the race conditions in the statements listed above do not influence the final outcome of the program nor do they cause the program to not terminate. However, a more serious piece of code could easily affect the final values of variables in the program as well as the program flow.

2.3 | Non-terminating scenario

Let us now consider a scenario which prevents the program from terminating. This scenario occurs when *thread-A* executes completely before *thread-B* becomes unblocked.

Time	Thread-A	Thread-B	Memory
20	`x = x - 1`		`x = 0, y = 0`
21	`x != y  ## false`		`x = 0, y = 0`
22	`y = y + 1  ## 0 + 1`		`x = 0, y = 1`
23	`[terminated]`	`[blocked]`	`x = 8, y = 1`

Figure 8

In this scenario, *thread-B* never completes as *thread-A* modifies the value of `y` required to unblock *thread-B*.

References

[1] University of Edinburgh, Parallel Programming Languages and Systems - Lecture Slides http://www.inf.ed.ac.uk/teaching/courses/ppls/pplsslides.pdf