Parallel Programming Assignment 1

s0943941

30/01/13

1. Statement Analysis

```
S1: x = x + y;
S2: y = x - y;
S3: x = x - y;
```

   (a) Without the co notation around the statements and with each statement being executed atomically, each of these statements will be executed in sequence. S1 must come before S2, which comes before S3.

Formula	Result	X Value, Y Value
Initial		(x = 2, y = 5)
S1	x = 2 + 5 = 7	(x = 7, y = 5)
S2	y = 7 - 5 = 2	(x = 7, y =2)
S3	x = 7 - 2 = 5	(x = 5, y = 2)

   Assuming an initial value of $x = 2, y = 5$, the final result will have to be $x = 5, y = 2$, given a sequentially consistent model of shared memory.

   (b) Because all of the statements are surrounded with the co notation, they are executed concurrently with each other. Thus, each of the statements has the potential to occur before the other, and we must consider 3! cases to capture all possibilities with which the statements could occur. They are:

      i. $S1 \rightarrow S2 \rightarrow S3$
      ii. $S1 \rightarrow S3 \rightarrow S2$
      iii. $S2 \rightarrow S1 \rightarrow S3$
      iv. $S2 \rightarrow S3 \rightarrow S1$
      v. $S3 \rightarrow S1 \rightarrow S2$
      vi. $S3 \rightarrow S2 \rightarrow S1$

   Plugging in the initial value of $x = 2, y = 5$, we arrive at the following results.

	Statement	Statement	Statement	End Result
i.	S1: 2 + 5 = 7	S2: 7 - 5 = 2	S3: 7 - 2 = 5	(X = 5, Y = 2)
ii.	S1: 2 + 5 = 7	S3: 7 - 5 = 2	S2: 2 - 5 = -3	(X = 2, Y = -3)
iii.	S2: 2 - 5 = -3	S1: 2 +(-3) = -1	S3: -1 - (-3) = 2	(X = 2, Y = -3)
iv.	S2: 2 - 5 = -3	S3: 2 - (-3) = 5	S1: 5 +(-3) = 2	(X = 2, Y = -3)
v.	S3: 2 - 5 = -3	S2: (-3) - 5 = -8	S1: (-3)+(-8) = -11	(X = -11, Y = -8)
vi.	S3: 2 - 5 = -3	S1: (-3) + 5 = 2	S2: 2 - 5 = -3	(X = 2, Y = -3)

Because of the order in which the statements can occur, there are three distinctive results that can happen from these statements. Each of these are distinct possibilities of what might happen if we try and execute the statements of part b.

i. $x = 5, y = 2$, as we saw in the first case.
ii. $x = 2, y = -3$, as we see in four of the six cases.
iii. $x = -11, y = -8$, as we see in the fifth case.

(c) In this case, two parts of the code will attempt to execute concurrently. The first will wait for x to be greater than y - once it does, it will execute S1 and then S2 in that order. The second will execute S3 atomically. Once both statements have completed, the atomic statement S2 will execute.

```
co <await (x>y) S1; S2;> // <S3;> oc <S2;>
```

However, with initial values of $x = 2, y = 5$, the condition for the await statement is not valid. It must wait until the values change again. Therefore, the only statement that can be executed is S3, which results in the value of $x = -3, y = 5$. The away condition still is not satisfied. Thus, we enter a deadlock, because all processes must be completed before proceeding outside the co and oc lines of code. The values will remain at $x = -3, y = 5$, and the program will continue to run in this deadlocked state indefinitely.

2. Program Analysis

   The program is compromised of three different sections:

   (a) The initialization of the variables, which occur first due to their location outside the co - oc notation. Therefore, $x = 10, y = 0$ initially.

   (b) This code

```
1. while (x!=y) {
2.     x = x - 1;
3. }
4. y = y + 1;
```

   (c) which can occur in parallel with this segment of code

```
5. <await(x==y);>
6. x = 8;
7. y = 2;
```

   due to the surrounding notation of co, //, and oc.

   Since the await condition is that x equals y, we can ignore the second statement for the first few iterations while y $=$ 0 and x $>$ 0 (it will not proceed until $x = y$).

   We deal with a number of cases:

- First Case:

  Lines 1-3 decrement x to 0. While loop condition terminates (because $x = y = 0$), but before the other segment of code has a change to run, line 4 is called and y is increased. The first segment of code is finished, but the await condition has not been satisfied in the second segment. The await forces a deadlock to occur, and the program does not terminate. The final values are $x = 0, y = 1$ in a deadlocked state.

- Second Case: Once Lines 1-3 decrement x to 0, the await statement is executed atomically. A number of cases arise for depending on when the await statement, and subsequent statements are executed.

  – Once x is decremented to 0, the await statement is triggered and x is assigned to 8 (lines 5 and 6 execute before the first segment reaches line 3). The while loop will continue (because $x = 8$ does not equal $y = 0$). Four possibilities can occur:

    * Another eight iterations of Lines 1-3 occur, decrementing x back to 0. While condition is set to false, and it terminates. Line 7 occurs, setting $y = 2$. Line 4 occurs, setting $y = 3$. The final values are $x = 0, y = 3$ in a terminated, stable state.

    * Another eight iterations of Lines 1-3 occur, decrementing x back to 0. While condition is set to false, and it terminates. Line 4 occurs, setting $y = 1$. Line 7 occurs, setting $y = 2$. The final values are $x = 0, y = 2$ in a terminated, stable state.

    * Another eight iterations of Lines 1-3 occur, decrementing x back to 0. While condition is set to false, and it terminates. Line 4 increases y to $y = 1$, but before it can set the value, Line 7 sets $y = 2$. Then line 4 finishes and sets $y = 1$. The final values are $x = 0, y = 1$ in a terminated, stable state.
    * At any point between when x is set to 8, and before it is decremented to (and including) 2, Line 7 occurs, setting $y = 2$. While condition is set to false when x reaches 2 and it terminates ($x = y = 2$). Line 4 occurs as the last line, setting $y = 3$. The final values are $x = 2, y = 3$ in a terminated, stable state.
    * At any point between when x is decremented to 1 (or immediately after Line 1) and before x is decremented to 0 (and before the condition is checked), Line 7 occurs, setting $y = 2$. Since y is now greater than x, the while condition still holds and x continues to decrement. But because x can only decrease, the while condition can never be satisfied (y will always be greater than x), x will continue to decrement, and the program cannot finish. The final result is a non-terminating, non-stable program.
- Once x is decremented to 0, the while loop can terminate ($x = y = 0$). The await condition is satisfied and the second segment proceeds (this could proceed in a reverse order as well, with the await condition proceeding and then the while loop terminating). Three possibilities can then occur:
    * Line 6 sets $x = 8$, Line 7 sets $y = 2$, and then Line 4 increments y to $y = 3$. The final values are $x = 8, y = 3$ in a terminated, stable state.
    * Line 6 sets $x = 8$, Line 4 increments y to $y = 1$, and then Line 7 sets $y = 2$. The final values are $x = 8, y = 2$ in a terminated, stable state.
    * Line 6 sets $x = 8$, Line 4 increases y to $y = 1$, but before it can set the value, Line 7 sets $y = 2$. Then Line 4 finishes and sets $y = 1$. The final values are $x = 8, y = 1$ in a terminated, stable state.

  Note, in this section, Line 6 can occur in a different positions (before Line 4 and after Line 4) without changing the final value, because it's the only line able to manipulate x.

It is also important to note that the sequence that results in $y = 1$ (reading the value, but having another line execute before it can finish setting) cannot affect x variable in this situation. The only possibile location where this can occur is on Line 2, within the while loop. The await statement, which prevents Line 6 from executing, needs x to equal 0 before it finishes. Once the x value is 0, either x is immediately set to 8 or the loop is exited - there is no point where x can be read and another statement occurs in between.

3. DEC Pseudocode

```
var = 1
co [i = 1 to n] {
    while(something) {
        while(var < 1 || DEC(var));
        critical section;
        var = 1;
        non-critical section;
    }
}
```

In the pseudocode, we first initialize a variable *var* to 1. From the lecture slides, we have n processes that we set to operate in parallel. Then, given some condition (*something*, again from the lecture slides), proceed into the area dealing with the critical sections. Each process will enter the while loop. The first process will check if var < 1 ($var = 1$) and returns false. Thus, it executes the DEC atomic instruction on var. var is decremented ($var = 0$) and returns false. Both boolean values are false for this first process, and so it exits the while loop and enters the critical section.

A second thread checks if var < 1 ($var = 0$) and returns true. Taking advantage of the lazy ||, the process continues in the while loop without executing the DEC atomic instruction. This ensures that the cache is not constantly being written, thus making it as cache efficient as possible. All other processes will remain in this loop until the first process has finished the critical section and resets var ($var = 1$). The same initial condition holds, and another process is able to enter into the critical section. After all the threads have entered the critical section, the var is also reset ($var = 1$) if it needs to be used again.

Suppose two or more processes check var < 1 before proceeding to the DEC atomic instruction. Could multiple entries enter the critical section? No. If two processes simultaneously check var < 1 ($var = 1$), they both return false. Thus, both processes will check DEC to see if it is okay to proceed. The first process will execute DEC, decrementing var ($var = 0$) and returning false. Both values returned false, so it proceeds to the critical section. The second process will also execute DEC, decrementing var ($var = -1$) and returning true (since it is less than 0). The second value returns true, so it remains in the while loop (looping on the initial condition of var < 1 which remains true). This is only possible because DEC is executed atomically. Multiple threads attempting to enter the critical section is the reason that var is reset to 1, rather than simply re-incremented.

Collaboration: Work was discussed with s0821562.