UNIVERSITY OF EDINBURGH

Parallel Programming Languages & Systems

Exercise Sheet 1

Matric: S1045049

Georgi Tsatsev

2/12/2014

Task 1:

```
int x = 10; y = 0;

Thread 1                 //              Thread 2

1.  while (x!=y) {       1.               <await (x==y);>

2.  x = x-1;             2.              x = 8;

3. }                     3.              y = 2;

4.  y = y+1;
```

In this task we are asked to discuss the possible outcomes of the above program assuming atomic reads and writes and a sequentially consistent model of shared memory. There is also no issue about "wrapping around" when arithmetic would otherwise overflow.

Since we have a sequential consistency the first line of Thread 2 would await for the condition x == y to become true before continuing to line 2 and 3. The operation x == y itself is not atomic but with the await construct it is treated as atomic. So in Thread 2 we can observe that there are only atomic actions but in Thread 1 this is not the case. Taking all that into account we can assume several possible interleavings of the two threads.

Outcome 1:   x = 8 and y = 2
This happens when Thread2 exits the await block when in Thread 1 the while loop has completed and x=0 and y = 0. After that Thread 2 can execute its second line (x = 8) so the final value of x becomes 8. And Thread1 iterates the y (y = y + 1, hence y =1) before Thread2 gets to its third line. After Thread 1 has executed all of its lines Thread2 assigns the value 2 for y (y = 2).

Outcome2:   x = 8 and y = 3
This case is similar to outcome 1 after x=0 and y=0 in the end of thread1's while loop and thread2's second line (x = 8) instead of Thread1 executing its fourth line Thread2 executes y=2 first. After that Thread1 does y = y+1 (y = 2 +1) so y becomes 3.

Outcome3:   x = 2 and y = 3
In this scenario Thread 1 is in its while loop and reaches the point when x=0 and y=0 which triggers the await in Thread2 so it can continue to its other lines. While Thread1 still being in the while loop Thread2 assigns x=8 and y=2 so the condition for the while is not met. Thread2 has finished its code and Thread1 executes the while loop until x!=y which is when x = 2 and iterates y in the end so y becomes 2+1 = 3.

Outcome4:   x = 0 and y = 2
Thread1 reaches the point when x = 0 and y =0 (the end of the while loop but before the final check) and Thread2's executes its first line since x==y and its second line x=8. So Thread1 is still in the while loop and decrements x to the point when x = 0 again. After Thread1 finishes the while loop it executes the increment of y (its fourth line so y=0+1) but Thread2 executes its third line (y = 2) afterwards so the final value of y is 2.

Outcome5:   x = 0 and y = 3
Almost the same as outcome4 but after Thread1 decrements x after Thread2 has assigned it the value of 8 instead of Thread1 executing its fourth line first it executes it after Thread2's third line (y=2). So in the end y=y+1 in Thread1 is executed when y is already 2 so its final value would be 3.

Outcome6:   x is decrementing and y = 2 (fails to terminate)
This is the case when the program fails to terminate (assuming that no overflowing would occur). It happens when Thread1 has reached the final decrementation of x (x==0==y) and Thread2 executes its first and second line (x = 8). So the check for "x!= y" in the next loop would not be satisfied since x=8 and y=0. Thread1 decrements the loop until x = 1 and at that point Thread2 does its third line where y=2.In this conditions we would never exit the while loop since x would be decrementing and y would be 2. The exit condition (x!=y) would never be met and the program would fail to terminate.

Outcome7:   x = 0 and y = 1 (fails to terminate)
This happens when Thread1 has finished its while loop and has done y=y+1 where their values are now x=0 and y=1 and Thread2 has not yet started executing. So Thread2 would have not done the await condition so it would result in a deadlock. So the program would fail to terminate.

Outcome8:   x = 0 and y = 1
This is the case when the while loop in Thread1 has reached and x=0 and y=0 but has not finished its last check and in Thread2 the await and x=8 is executed. The while loop decrements the x until x=0 and after that in Thread1 y is being read as 0 in Thread2 y=2 is written so Thread2 has finished its work. In Thread1 since it has read y as 0 it increments it by 1 and writes y=0+1 so finally y=1.

Outcome9:   x = 8 and y = 1
The while loop in Thread1 has executed and x=0 and y=0 so in Thread2 it executes await and x=8. In Thread1 y is being read as 0 and in Thread2 y is written as 2 at the same time. So in the end Thread1 saw y as 0 and writes it as 0+1 which is 1 and its final value is 1.

Task 2:

In this task we are given a coloring algorithm for a distributed (message passing) model with one processor per node of the graph. Our task is to discuss this algorithm in context of shared variable programming and replace the message passing by usage and synchronization of shared variables.

One approach to this problem would be to implement some kind of a bag of tasks pattern where the bag of tasks would represent the messages that each process (node) are passing in the original version. So the bag would consist of a shared address space that would have the degree random value and the first legal color of each vertex. Basically we might have a farmer node that maintains the bag. For example we can use the node with the highest degree or an external node to synchronize the coloring of the whole graph. The work of each node would be the coloring and it would depend on the degree of its neighbors and the possible colors and its random variable. So any worker(node) would only color itself when it has been assigned by the main (farmer) node. That would result in more nodes than processors and the issue that might arise would be that more computations and resources would be needed for the farmer node thus resulting in poor performance but it would imitate the message passing that was in the original algorithm. The assignment of tasks would be as in the original algorithm the node with the highest degree would have priority over its neighbors and that would mean that it would be the first to be given a task and until its task has been completed none of its neighbors would be given the coloring task unless their colors have been predetermined as different than it. If it were the case that there was no farmer node we might implement the bag of words pattern by using the shared bag of tasks consisting of the different neighbor dependencies (degree , random variable and possible color) and an additional locking mechanism to prevent neighboring nodes to color at the same time. Such locking mechanism might be as in the Bakery Algorithm’s turn mechanism where the turns of coloring for neighbors would depend on each node’s degree and random variable.

Another approach would be to implement it as the Interacting Peers Pattern. It represents the case of each processor communicating only with its neighboring processors perfectly. In this case the messages that are passed between adjacent nodes would be replaced by the barriers thus preventing the coloring of neighboring nodes. The node with the highest degree (or in a case of equal the node with the highest random variable) is being colored or has not yet been colored. The synchronization between the nodes is based on barriers that prevent the initialization of the critical section (the coloring) in each node until all the nodes have reached to that point. These barriers can be used when the nodes with the highest degree are being colored after that all their neighbors would wait for the synchronization before attempting their critical section (coloring). The condition of the coloring can again be based on a shared variable that contains each nodes turn to color as in the Bakery algorithm. The turn can again be determined by each nodes degree and random variable that they would calculate before entering their critical section.

There are other possible patterns that might be used for the shared variable version such as The Pipeline Pattern where the initial stage would consist of the producer (highest degree node) being colored first and each of its neighbors acting as consumers and each consumer can be a producer when looking at its neighbors in the second stage thus resulting in some kind of a dependency graph starting at the nodes with the highest degree.