CS 350: Homework #7

Due 3PM, Tuesday, March 29

Kristel Tan (ktan@bu.edu)

March 22, 2016

Question 1

Part a

Yes, the tweak provided in this problem works because it achieves mutual exclusion. For instance, let P0 be the first process to enter the system and let P1 be the process to follow it. In order for the proposed algorithm to fail, P1 must enter the critical section before P0 although it entered the system later. This event can only happen if P1 is assigned a ticket lower than P0 upon entrance.

However, we know that according to the pseudocode this will never occur because of the `max` function implemented when assigning tickets. The `max` function ensures that a process will always be given a ticket one higher than the current maximum ticket value assigned to the previous process. Therefore, this version of the Bakery algorithm should work.

Part b

Please find the source code for this problem in the Java file named 'Q1B.java'. The code has been commented for clarity of what each method is doing. The following is a sample output of the program in which four processes are interleaved:

```
Thread 0 is starting iteration 0
We hold these truths to be self-evident, that all men are created equal,
that they are endowed by their Creator with certain unalienable Rights,
that among these are Life, Liberty and the pursuit of Happiness.
Thread 0 is done with iteration 0

Thread 1 is starting iteration 0
We hold these truths to be self-evident, that all men are created equal,
that they are endowed by their Creator with certain unalienable Rights,
that among these are Life, Liberty and the pursuit of Happiness.
Thread 1 is done with iteration 0

Thread 2 is starting iteration 0
We hold these truths to be self-evident, that all men are created equal,
that they are endowed by their Creator with certain unalienable Rights,
that among these are Life, Liberty and the pursuit of Happiness.
Thread 2 is done with iteration 0

Thread 3 is starting iteration 0
We hold these truths to be self-evident, that all men are created equal,
that they are endowed by their Creator with certain unalienable Rights,
that among these are Life, Liberty and the pursuit of Happiness.
Thread 3 is done with iteration 0

Thread 0 is starting iteration 1
```

```
We hold these truths to be self-evident, that all men are created equal,
that they are endowed by their Creator with certain unalienable Rights,
that among these are Life, Liberty and the pursuit of Happiness.
Thread 0 is done with iteration 1

...
```

This pattern is then printed out in like fashion for the remaining iterations and for each thread. As you can see, the output is printed cohesively as expected from part (a) of this problem.

Question 2

Part a

The code provided in this problem will work because it extends the same principles of rendezvous synchronization for 2 parties to N parties. As seen in the main for loop of the code, process i would first be signaled, initialized to 0. It then waits for the next j process and so on until all N processes have been signaled. Because no rendezvous code is executed until this for loop finishes and all processes arrive, we can be certain that no process proceeds until all of the processes have reached this point. In other words, all processes will enter the rendezvous instructions together, proving that it is a valid solution.

Part b

Yes, a process could be blocked as a result of executing the last instruction in the code provided. Tracing the first few iterations of the code, we can see that each process is dependent upon those after it to become unblocked. By the end of all the processes' for loops, `semaphore[i]` should have a value of 1. Calling the `wait(semaphore[i])` at the end will reduce it by 1, making its value 0. This will not cause any processes to be blocked because the value of the semaphore is not negative.

P0 Execution `signal(semaphore[i]);`	`// semaphore[0] = 1`

1st iteration
`for(j = 0; j < N; j++) {`
`  wait(semaphore[0]);`
`  signal(semaphore[0]);`
`}`
	`// semaphore[0] = 0`
	`// semaphore[0] = 1`

2nd iteration
`for(j = 0; j < N; j++) {`
`  wait(semaphore[1]);`
`  signal(semaphore[1]);`
`}`
	`// semaphore[1] = -1`
	`// P0 is blocked`

P1 Execution `signal(semaphore[1]);`	`// semaphore[1] = 0` `// P0 is unblocked`

1st iteration
`for(j = 0; j < N; j++) {`
`  wait(semaphore[0]);`
`}`
	`// semaphore[0] = 0`
	`// semaphore[0] = 1`

Part c

I think the last instruction in the above code is included so that `semaphore[i]` does not block any processes the next time it is signaled and so that the semaphores are reset to their original values, which is 0.

Question 3

Part a

Please find the source code for this problem in the Java file named 'Q3A.java'. The code has been commented for clarity of what each method is doing. The initialized data structures are defined at the beginning of the class as follows:

```
 private int id;
 private static int N;
 private final Random rand = new Random();
 static volatile Semaphore [] B = new Semaphore [N];
 static {
   for(int i = 0; i < N; i++) {
     B[i] = new Semaphore(0, false);
   }
 }
 static volatile int[] R = new int[N];
 static volatile int counter = 0;
```

- `private int id` specifies the ID unique to each process
- `private static int N` specifies the number of processes with 0 representing the process of lowest priority and N being the process of highest priority
- `private final Random rand = new Random();` declares a new Java random object to be used for implementing random times spent in the critical section
- `static volatile Semaphore [] B = new Semaphore [N];` declares an array of N semaphores — the for loop after this declaration initializes all the semaphores to have 0 permits and false fairness
- `static volatile int[] R = new int[N]` declares an array of size N to keep track of the priority for each semaphore
- `static volatile int counter = 0;` initializes the counter variable to 0, which is used in the `newWait()` and `newSignal()` functions

Part b

Please find the source code for this problem in the Java file named 'Q3B.java'. The code has been commented for clarity of what each method is doing. The following is a sample output of the program in which five processes are interleaved for 3 iterations. Please note that the source code allows the processes to request the critical section for 10 iterations, but the pattern of the output seen here remains similar.

```
P0 is requesting CS
P0 is entering CS on iteration 1
P4 is requesting CS
P3 is requesting CS
P2 is requesting CS
P1 is requesting CS
P0 is exiting the CS
P0 is requesting CS
P4 is entering CS on iteration 1
P4 is exiting the CS
P4 is requesting CS
P3 is entering CS on iteration 1
P3 is exiting the CS
P3 is requesting CS
P4 is entering CS on iteration 2
P4 is exiting the CS
P4 is requesting CS
P3 is entering CS on iteration 2
P3 is exiting the CS
P3 is requesting CS
P4 is entering CS on iteration 3
P4 is exiting the CS
P3 is entering CS on iteration 3
P3 is exiting the CS
P2 is entering CS on iteration 1
P2 is exiting the CS
P2 is requesting CS
P1 is entering CS on iteration 1
P1 is exiting the CS
P1 is requesting CS
P2 is entering CS on iteration 2
P2 is exiting the CS
P2 is requesting CS
P1 is entering CS on iteration 2
P1 is exiting the CS
P1 is requesting CS
P2 is entering CS on iteration 3
P2 is exiting the CS
P1 is entering CS on iteration 3
P1 is exiting the CS
P0 is entering CS on iteration 2
P0 is requesting CS
P0 is entering CS on iteration 3
```

Part c

Please find the source code for this problem in the Java file named 'Q3C.java'. The code has been commented for clarity of what each method is doing. The following is a sample output of the program in which five processes are interleaved.

```
P0 is requesting CS
P0 is entering CS on iteration 1
P4 is requesting CS
P3 is requesting CS
P2 is requesting CS
P1 is requesting CS
P0 is exiting the CS
P0 is requesting CS
P4 is entering CS on iteration 1
P4 is exiting the CS
P4 is requesting CS
P4 is entering CS on iteration 2
P4 is exiting the CS
P4 is requesting CS
P4 is entering CS on iteration 3
P4 is exiting the CS
P4 is requesting CS
P3 is entering CS on iteration 1
P3 is exiting the CS
P3 is requesting CS
P3 is entering CS on iteration 2
P3 is exiting the CS
P3 is requesting CS
P2 is entering CS on iteration 1
P2 is exiting the CS
P2 is requesting CS
P1 is entering CS on iteration 1
P1 is exiting the CS
P1 is requesting CS
P1 is entering CS on iteration 2
P1 is exiting the CS
P1 is requesting CS
P1 is entering CS on iteration 3
P1 is exiting the CS
P1 is requesting CS
P1 is entering CS on iteration 4
P1 is exiting the CS
P1 is requesting CS
P1 is entering CS on iteration 5
P1 is exiting the CS
P1 is requesting CS
P1 is entering CS on iteration 6
P1 is exiting the CS
P1 is requesting CS
P1 is entering CS on iteration 7
P1 is exiting the CS
P1 is requesting CS
P1 is entering CS on iteration 8
P1 is exiting the CS
P1 is requesting CS
P1 is entering CS on iteration 9
P1 is exiting the CS
P1 is requesting CS
```

```
P1 is entering CS on iteration 10
P1 is exiting the CS
```

I am unsure if this output is correct, but it appears that the processes begin to stop requesting the critical section. If you analyze the source code, you will see that I modified the `max()` function from part (b) return the process that has requested the critical section the least number of times, becoming a `min()` function instead. This is being kept track of by the `numAccess` array. Every time another process accesses the critical section, the `numAccess` array of that process's index is incremented by 1.

Part e

- Starvation is a problem for the semaphore implemented in part (a) because in a system where a process can request the critical section an infinite amount of times, it becomes possible that a process of higher N can continue preempting processes of lower priority

- Starvation is not a problem for the semaphore implemented in part (c) because each process will eventually get served. Processes that don't enter the critical section that often will get higher priority and vice versa. This rule will balance out the system, serving every request at some point.

- Starvation is not a problem for the semaphore implemented in part (d) because each process that enter the system is served right away regardless of its priority. In other words, very new process entry has the highest priority.