Homework #6 (100 Points)

Due 03:00pm on Tuesday 03/22/2016 through gsubmit.

Feel free to make assumptions, if you feel that such assumptions are justified or necessary. Please state your assumptions clearly. Unreasonable assumptions lead to unreasonable grades!

---

1. In class we discussed a non-preemptive Highest-Slowdown-Next (HSN) scheduler, which tries to be "fair" by normalizing the queueing delay by the service time.
    a. The non-preemptive nature of that scheduler was a major disadvantage, especially for processes with short CPU burst times (i.e., I/O-bound jobs). Use an example to explain/illustrate the "unfairness" of the scheduler for I/O bound jobs.

    To address the problem with non-preemptive HSN scheduling, a preemptive version is to be devised.

    b. Explain why preemption at job arrival times is not a good solution. Use an example to explain/illustrate your argument.
    c. Explain why preemption using a fixed timeout is a better solution, but that it could still be problematic for I/O-bound jobs. Use an example to explain/illustrate your argument.
    d. Can you think of a preemptive version of HSN that addresses the issues with the versions described in parts (b) and (c). Explain how your approach would work. How would your version "fix" these issues?

2. The following protocol was suggested for mutual exclusion between two processes P0 and P1.

```
Process P0:
repeat
  flag[0]:=true;
  if flag[1]{
    if (turn==1) {
      flag[0]:=false;
      while (turn==1) {};
      flag[0]:=true;
    }
  }
  Critical Section
  turn:=1;
  flag[0]:=false;
     Remainder Section
forever

Process P1:
repeat
```

```
    flag[1]:=true;
    if flag[0]{
      if (turn==0) {
        flag[1]:=false;
        while (turn==0) {};
        flag[1]:=true;
      }
    }
    Critical Section
    turn:=0;
    flag[1]:=false;
       Remainder Section
  forever
```

a. Show that the above protocol is buggy. In particular, write down a sequence of instruction executions leading to both P0 and P1 being in the critical section at the same time.

b. Fix the above code

3. Read this short primer about multithreading in Java. Using multithreading you are able to have multiple threads (for our purposes, we can also think of them as processes as well).

   a. Write a Java program that will spawn and run concurrently two threads of the same code. The code that each thread will follow should consist of a loop that executes five times. Each iteration of the loop should start by having the thread print the first of the five lines below (using a "print" statement), then sleep for some random amount of time (say between 0 and 20 msec), then print the second line, then sleep again, etc. until all five lines below are printed out.

      "Thread i is starting iteration k"
      "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 i is done with iteration k"

      Run your code and show that the output of the two processes will be interleaved (making the output incomprehensible).

   b. To make the above work, you will need to use a critical section so that the five print statements in a single iteration are all inside that critical section. Use the following version of the Dekker's algorithm and show that the statements are printed cohesively.

      CS Entry Protocol for Procedss i

```
     flag[i]:=true;
     while flag[j]{
        if (turn==j) {
          flag[i]:=false;
          while (flag[j]==true) {};
          flag[i]:=true;
        }
     }
```

CS Exit Protocol for Process i

```
      turn:=j;
      flag[i]:=false;
```

c. Can you illustrate (using your code) that using the above Dekker algorithm, it is possible for (say) thread 0 to be executing its entry protocol while thread 1 manages to get into the critical section more than once. Hint: You may want to use artificial delay (e.g., sleep in the body of the loop) instructions that will make the threads proceed at different speeds.

d. Replace the above Dekker algorithm with the following Peterson algorithm and show that the scenario you created in (c) does not occur.

CS Entry Protocol for Process i

```
      turn:=j;
      flag[i]:=true;
      while (flag[j] && turn==j) {};
```

CS Exit Protocol for Process i

```
      flag[i]:=false;
```

4. As discussed in class, all software approaches for mutual exclusion rely on "busy waiting". In this problem, we will "measure" the overhead from this busy waiting. We will do so simply by counting the number of times a busy waiting loop is executed in a Java implementation of the critical section solution using Peterson's algorithm (in part 3.d, above).

   You can do so by having each thread keep a count of how many times it executed the "while" loop. What results do you obtain? e.g., on average, how many times is the busy-waiting loop executed per request for the critical section? You may want to repeat your experiment a number of times and report a 95th-percent confidence interval.

   Note: By incrementing a counter in the busy-waiting loop, you are effectively undercounting the number of iterations -- i.e., without that counter in the picture, the loop condition would have been checked more times.