CS4410 - Fall 2008
Homework 2 Solution
Due September 23, 11:59PM

Q1. Explain what goes wrong in the following version of Dekker's Algorithm:

```
CSEnter(int i)
{
      inside[i] = true;
      while(inside[j])
      {
           inside[i] = false;
           while(turn == j) continue;
           inside[i] = true;
      }
}

CSExit(int i)
{
      turn = j;
      inside[i] = false;
}
```

Answer:
There is a possibility of starvation here. Suppose process j is in critical section and process i is busy waiting at the inner while loop. Now when process j exits the critical section it sets turn = i, but it immediately tried to access the critical section and so sets inside[j] = true. Right before process j executes "while (inside[i])", scheduler schedules process i. Now process i exits from the inner while loop as turn = = i now, but it finds inside[j] to be true so continue the outer while loop. Then after it executes "inside[i]=false", the scheduler schedules process j and it finds the condition at outer while loop to be false and enters the critical section. As a result, process i busy waits at outer while loop. These steps can repeat arbitrary number of times and that starves process i and it may never enter the critical section. In the original algorithm, there is an extra checking "if (turn==j)" which ensures that this starvation never happens.

Grading guide:
-5 : Failing to mention starvation / violation of bounded waiting; mentioning it violates mutual exclusion (two threads inside the critical section simultaneously) which actually does not happen here.
-1 : Mentioning starvation, but no explanation how it happens
-5 : Just mentioning it does not have "if (turn==j)", but no explanation on why it is wrong (i.e., starvation/violation of bounded waiting)

Q2. Round-robin schedulers normally maintain a list of all runnable processes, with each process occurring exactly once in the list. What would happen if a process occurred twice in the list? Can you think of any reason for allowing this?

Answer:

If a processor occurs more than once in the round-robin list, then it will get two turns for each pass through the list. One reason for allowing this would be to implement a primitive priority system since the more times it occurs on the list, the higher the percentage of time the CPU will spend on that process.

Grading guide:
Almost all answers were correct. Some of you lose 3 points for failing to explain the why part.

Q3. Five batch jobs A through E, arrive at a computer center at almost the same time. They have estimated running times of 11, 6, 2, 4, and 8 minutes. Their (externally determined) priorities are 3, 5, 2, 1, and 4, respectively, with 5 being the highest priority. For each of the following scheduling algorithms, determine the mean process turnaround time.

Ignore process switching overhead.

(a) Round-robin

(b) Priority scheduling

(c) First come, First served (run in order 11, 6, 2, 4, 8)

(d) Shortest job first

For (a), assume that the system is multiprogrammed, and that each job gets its fair share of the CPU. For (b) through (d) assume that only one job runs at a time, until it finishes. All jobs are completely CPU bound. Assume that the time quantum of the scheduler is 1 minute.

Answer:

Remember that the turnaround time is the amount of time that elapses between the job arriving and the job completing. Since we assume that all jobs arrive at time 0, the turnaround time will simply be the time that they complete.

(a) Round Robin: The table below gives a break down of which jobs will be processed during each time quantum. A * indicates that the job completes during that quantum.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	A	B	C*	D	E	A	B	D	E	A

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
B	D*	E	A	B	E	A	B*	E	A	E	A	E*	A	A	A*

Average turnaround = (8 + 17 + 23 + 28 + 31)/5 = 107/5 = 21.4 minutes

(b) Priority Scheduling:

1-6	7-14	15-25	26-27	28-31
B	E	A	C	D

Avg. turnaround = (6 + 14 + 25 + 27 + 31)/5 = 103/5 = 20.6 minutes

(c) FCFS

1-11	12-17	18-19	20-23	24-31
A	B	C	D	E

Avg. turnaround = (11 + 17 + 19 + 23 + 31)/5 = 101/5 = 20.2 minutes

(d) SJF

1-2	3-6	7-12	13-20	21-31
C	D	B	E	A

Avg. turnaround = (2 + 6 + 12 + 20 + 31)/5 = 71/5 = 14.2 minutes

Finding the average process turnaround time can also be done by multiplying the number of unfinished processes by the time they remain unfinished dividing by number of processes. This makes finding solutions for parts b, c, and d particularly efficient. For example, the calculation for part C can be represented by (5*11 + 4*6 + 3*2 + 2*4 + 1*8)/5.

Grading guide

Each part of the four parts of question 3 were worth a total of 2.5 points. More than 2.5 points could not be lost for any part. They were graded as follows:

-1 Intermediate mistake in work such as incorrect process turnaround time or diagram, but correct final answer.

-2 Intermediate mistake in work, but correct final answer given that mistake. For example using process turnaround time of 15 minutes when it should be 25 minutes.

-1 Mistake in arithmetic.

-1 Using wrong time quantum in part a.

-1 Reversing priority in part b.

-1.25 Calculating wait time instead of turnaround time.

-2 Not using given values. For example making A take 12 minutes to complete instead of 11. (Points only taken off once for all of question 3)

-1 Determining order of round-robin by priority. As described on page 194 [Silberschatz], Round-Robin scheduling is the same as first come first serve, except with preemption. The assumption is that the processes are delivered in A,B,C,D,E order, and if a different assumption is made then they should be round-robin scheduled in the same order as the FCFS method.

-2.5 Incorrect final answer and incorrect or missing work.

Q4. Can we use the test-and-set instruction to implement a lock on a multiprocessor environment? Explain why or why not.

Answer:

In the multiprocessor environment, the test-and-set instruction should still work. Because it operates atomically on a single memory location, even with multiple processors sharing the same memory, the test-and-set instruction will still operate as expected.

Grading guide:
-3 why part is wrong
-5 answer wrong, but relevant analysis is somewhat ok

Q5. Consider the environment where many user-level threads are mapped to a single kernel thread (i.e., the many-to-one model). Describe a situation the one-to-one model outperforms this model.

Answer:

The chief benefit of the one-to-one model over the many-to-one model is that it allows for more concurrency within a program. In the many-to-one model, if a thread makes a blocking system call, all the threads of that process have to wait to be scheduled again.

Clearly, any environment where threads make frequent system calls is one where the one-to-one model outperforms many-to-one.

Grading guide:
-5 answer wrong, but relevant analysis is somewhat ok