Computer System Fundamentals HW #6

Quan Zhou

Mar 21st, 2016

Problem 1

(a) Since the Highest Slowdown Next (HSN) scheduler tries to select the job with the maximum slowdown, therefore preventing longer jobs from keeping waiting, jobs with short CPU burst times will be postponed. For a system where I/O bound jobs are requested frequently, HSN scheduler would be unfair to these short jobs as the scheduler ranks jobs by slowdown, which is the ratio of $\frac{T_q}{T_s}$.

(b) Preemption at job arrival times would not be a good solution because the current job whether a long or short job will be interrupted by the highest slowdown job in the queue. As the scheduler would give priority to the newly arrived and the highest slowdown job, short jobs (I/O bound jobs) won't be executed efficiently: any short job that is currently running will be interrupted by higher-slowdown jobs, resulting in a long time to be executed.

(c) With a fixed timeout, it would be fairer to both long and short jobs: in the above example: short job would be forced to execute should it exceed preemption timeout. This allows the scheduler to control response times by preventing certain long jobs running for too long in order to let another process run.Though depending on the settings for the "timeout", in situations when short jobs are arriving frequently, they would still pay the cost (of waiting) for longer jobs.

(d) To improve the preemptive scheduler based of HSN,

Problem 2

(a) By examining the above code, one fatal problem is the critical sections is accessible to both P_0 and P_1 whenever it is their turn: the critical section is accessed after the loop where the flag is set up and the turn is checked. Notice the end of the loop basically only sets flag[i] :=true . I tried to run P_0 first and then P_1 arrives right before the line if flag[1]{. We have line by line code until the following and realize both P_0 and P_1 can access the critical section.

Step	Process P0	Process P1
1	while (turn==1){};	
2		while (flag==0){};
3	flag[0] := true;	
4		flag[1] := true;
5	[P0 enters critical section]	
6		[P1 enters critical section]

(b) The following code should fix the bug and mutual exclusion is achieved (based of Dekker's Algorithm): And like wise for Process P1, we have:

Algorithm 1 Dekker's Algorithm

```
Process P0:
repeat
    flag[0]:=true;
    while flag[1] do
        if turn==1 then
            flag[0]:=false;
            while turn==1 do
                waiting
            end while
            flag[0]:=true;
        end if
    end while
    Critical section
    turn[1]:=true;
    flag[0]:=false;
    Remainder section
until forever
```

Algorithm 2 Dekker's Algorithm

```
Process P1:
repeat
    flag[1]:=true;
    while flag[0] do
        if turn==0 then
            flag[1]:=false;
            while turn==0 do
                waiting
            end while
            flag[1]:=true;
        end if
    end while
    Critical section
    turn[0]:=true;
    flag[1]:=false;
    Remainder section
until forever
```

Problem 3

(a) Codes see src folder and output for each part is in log folder.

(b) Codes see src folder and output for each part is in log folder.

(c) Yes. I did two different runs:
1. k ¡ 6 and sleep for 20 msec for each printed message
The output shows the threads being served in the following order: 0 0 1 0 1 0 0 1 1 1.
During thread 0's entries, thread 1 was able to access through the critical section twice.
2. k ¡ 30 and sleep for 20-40 msec for each printed message
The output shows the threads being served in the following order:
0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1
Again during thread 0's entries, thread 1 was able to access through the critical section twice.

(d) Codes see src folder and output for each part is in log folder. we see that the scenario in (c) does not occur in Peterson's Algorithm.

Problem 4

`See src folder for codes`

I did 100 samples/runs of the code where thread 0 and 1 execute the while loop defined by Peterson's algorithm. I took the average and computed the s (sample standard deviation) and hence the 95% confidence interval for each thread:

The estimated times the "while" loop is accessed during the multi-thread processing:
Thread 0: (115.79, 113.57)
Thread 1: (111.55, 133.03)
`in millions times`