Lab2Answers

Jain117 | Nitin Jain

Q3.

SPEACIAL INSRUCTIONS: The priority of the processes to run completely or the one which sleeps should be more the main's priority i.e. INITPRIO = 20. And the process that starves should be less or equal to main's priority so that it starves.

For this question, the output shows the total number of cpu cycles consumed by 6 different process. The first 3 process are created from the app_lim() and has equal priority of 21. The last process is also created using the app_lim() but has a lower priority of 19. As can be seen from the output that this process starves for cpu cycles because it has a lower priority than the main process and since the scheduler always runs a higher priority process before a lower one, this process starves. The 4th process app_sleep3 calls a function app_sleep() which has a sleep call and a print statement. This process is also created with a priority of 21 and as can be seen, it needs very few CPU cycles as it sleeps.

The output for test case is:

```
Going to sleep

app_lim0 cpu cycles: 10

app_lim1 cpu cycles: 10

app_lim2 cpu cycles: 10

app_sleep3 cpu cycles: 1

app_lim4 cpu cycles: 0
```

Q4.

IMPORTANT NOTES: *The priority from here on is inverted, so higher the priority value lower is the process priority in scheduling.*

SPEACIAL INSTRUCTIONS FOR Q4: The main process has not been hardcoded to run completely before any other process but this is required as otherwise the testing becomes difficult and the test cases explanation can be difficult. The default priority of main is retained as INITPRIO = 20. Now, because of the above reason we would like to run the test process once the main completes executing. ***For testing, set the lab2flag = 4 and initial priority of the cpubound process at more than or equal to 200 so that main gets enough slices to complete running fully. Eg:***

```
create(cpubound, 1024, 200, "cpubound process", 2, 10, 3000));
```

Here, the initial priority is 200 and this will then be updated by the dynamic process scheduling by adding the cpu cycles to this initial priority.

4.2

Changes made to code to make dynamic scheduling:

Firstly, every process priority is updated when context switching out. The old priority of the process is added with the number of cpu cycles it has consumed in the current slice. This new priority is set to the process and inserted into the ready queue based on its new priority. This implies, if a process consumes more cycles then it will end up with a higher priority value. From given design, we then want to order the ready queue in increasing order of priority, so that the highest priority process will be near the head of queue.

Thus, in the implementation of Dynamic Scheduling, the priority order is reversed by reversing the sign of check in resched.c and insert.c. Other changes which follows from this is that the null processor priority is set to the maximum so that all other processes can run and no process starve because of the null process.

4.3

Test Case output and explanation:

Case 1: 6 similar CPU bound process *part*-output and explanation:

```
PID: 3,

Outer loop: 10,

Process Priority: 200,

 Remaining time slice: 28


PID: 3,

Outer loop: 9,

Process Priority: 200,

 Remaining time slice: 18


PID: 3,

Outer loop: 8,

Process Priority: 200,

 Remaining time slice: 9
```

```
PID: 4,
Outer loop: 10,
Process Priority: 200,
 Remaining time slice: 28

PID: 4,
Outer loop: 9,
Process Priority: 200,
 Remaining time slice: 18

PID: 4,
Outer loop: 8,
Process Priority: 200,
 Remaining time slice: 9

PID: 5,
Outer loop: 10,
Process Priority: 200,
 Remaining time slice: 28

PID: 5,
Outer loop: 9,
Process Priority: 200,
 Remaining time slice: 18

PID: 5,
Outer loop: 8,
Process Priority: 200,
 Remaining time slice: 9

PID: 6,
Outer loop: 10,
Process Priority: 200,
 Remaining time slice: 28
```

```
PID: 6,

Outer loop: 9,

Process Priority: 200,

 Remaining time slice: 18


PID: 6,

Outer loop: 8,

Process Priority: 200,

 Remaining time slice: 9


PID: 7,

Outer loop: 10,

Process Priority: 200,

 Remaining time slice: 28


.

.

.

.


PID: 3,

Outer loop: 7,

Process Priority: 230,

 Remaining time slice: 29



PID: 3,

Outer loop: 6,

Process Priority: 230,

 Remaining time slice: 20



PID: 3,
Outer loop: 5,

Process Priority: 230,

 Remaining time slice: 10



PI


PID: 4,

Outer loop: 7,

Process Priority: 230,

 Remaining time slice: 29



PID: 4,

Outer loop: 6,

Process Priority: 230,

 Remaining time slice: 20



PID: 4,

Outer loop: 5,

Process Priority: 230,

 Remaining time slice: 10



PI


PID: 5,

Outer loop: 7,

Process Priority: 230,

 Remaining time slice: 29



PID: 5,
Outer loop: 6,

Process Priority: 230,

 Remaining time slice: 20



PID: 5,

Outer loop: 5,

Process Priority: 230,

 Remaining time slice: 10



PI


PID: 6,

Outer loop: 7,

Process Priority: 230,

 Remaining time slice: 29



PID: 6,

Outer loop: 6,

Process Priority: 230,

 Remaining time slice: 20



PID: 6,

Outer loop: 5,

Process Priority: 230,

 Remaining time slice: 10



PI


PID: 7,
Outer loop: 7,

Process Priority: 230,

 Remaining time slice: 29



PID: 7,

Outer loop: 6,

Process Priority: 230,

 Remaining time slice: 20



PID: 7,

Outer loop: 5,

Process Priority: 230,

 Remaining time slice: 10



PI


PID: 8,

Outer loop: 7,

Process Priority: 230,

 Remaining time slice: 29



PID: 8,

Outer loop: 6,

Process Priority: 230,

 Remaining time slice: 20



PID: 8,

Outer loop: 5,

Process Priority: 230,
 Remaining time slice: 10
.
.
.
.
.

CPU TIME consumed by process: 3 is 901,
Process Priority: 260,
 Remaining time slice: 3

CPU TIME consumed by process: 4 is 90 1,
Process Priority: 260,
 Remaining time slice: 2

CPU TIME consumed by process: 5 is 901,
Process Priority: 260,
 Remaining time slice: 3

CPU TIME consumed by process: 6 is 90 1,
Process Priority: 260,
 Remaining time slice: 3

CPU TIME consumed by process: 7 is 901,
Process Priority: 260,
 Remaining time slice: 3

CPU TIME consumed by process: 8 is 90
```

The output clearly shows the dynamic scheduling between 6 processes.

Experimenting with LOOP1 and LOOP2 variables:

- When loop2 value is high ~ 50000, each process takes more than 1 time slice to compute. Hence, the print after each inner loop doesn’t take place within one time slice and the output prints are overlapping between the different processes. Although the time sharing is much more clearly seen.

When the loop2 value is very small ~50, all the inner loop might get executed too soon to see the dynamic scheduling stabilizing.

An optimum value is for the given cpubound process is: 3000.

- When loop1 value is high, there are a lot of prints which can make it difficult to read the process priority.

  When the loop1 value is very small ~ 3-5, we are allowing lesser cycles for processes and thus giving the dynamic scheduling to stabilize.

  An optimum value used here is 10.

Once all the processes are running and dynamically prioritized (this can be seen in the change in the value of the priority value of the processes.). Note that the initial priority, when the process is created is always 200 as this is the initial priority given in the main process test case functions:

```
resume(pr[i] = create(cpubound, 1024, 200, "cpubound process", 2, 10,
3000));
```

Case 2: 6 IO Bound process output and explanation:

```
PID: 3,

Outer loop: 3,

Process Priority: 200,

 Remaining time slice: 0


PID: 4,

Outer loop: 3,

Process Priority: 200,

 Remaining time slice: 0


PID: 5,

Outer loop: 3,

Process Priority: 200,

 Remaining time slice: 0


PID: 6,

Outer loop: 3,
Process Priority: 200,
 Remaining time slice: 0

PID: 7,
Outer loop: 3,
Process Priority: 200,
 Remaining time slice: 0

PID: 8,
Outer loop: 3,
Process Priority: 200,
 Remaining time slice: 0

PID: 3,
Outer loop: 2,
Process Priority: 200,
 Remaining time slice: 6

PID: 4,
Outer loop: 2,
Process Priority: 200,
 Remaining time slice: 7

PID: 5,
Outer loop: 2,
Process Priority: 200,
 Remaining time slice: 7

PID: 6,
Outer loop: 2,
Process Priority: 200,
 Remaining time slice: 7

PID: 7,
Outer loop: 2,
Process Priority: 200,
 Remaining time slice: 7

PID: 8,
Outer loop: 2,
Process Priority: 202,
 Remaining time slice: 7

PID: 3,
Outer loop:

PID: 4,
Outer loop: 1,
Process Priority: 200,
 Remaining time slice: 13

PID: 5,
Outer loop: 1,
Process Priority: 200,
 Remaining time slice: 13

PID: 6,
Outer loop: 1,
Process Priority: 200,
 Remaining time slice: 13

PID: 7,
Outer loop: 1,
Process Priority: 200,
 Remaining time slice: 13

PID: 8,
Outer loop: 1,
Process Priority: 208,
 Remaining time slice: 14

CPU TIME consumed by process: 8 is 14 1,
Process Priority: 211,
 Remaining time slice: 12

CPU TIME consumed by process: 4 is 13
CPU TIME consumed by process: 6 is 13
CPU TIME consumed by process: 7 is 13
```

The output clearly shows the dynamic scheduling between 6 iobound processes. Overall the output is very similar to the above but since the processes mostly sleep, the number of cpu cycles consumed are much less compared to the above cpubound case. The total number of cpu cycles consumes by each process (because they are the same) is 13.

Experimenting with LOOP1 and LOOP2 variables:

- When loop2 value is high ~ 50000, each process sleeps a lot. Hence, many times the cpu runs the null process and this amount of sleep is unnecessary. It doesn't even show anything about scheduling as a there might be only 1 or 2 processes actually in the readylist while the rest are sleeping as this happens very fast.

  When the loop2 value is very small ~5, doesn't allow enough sleep and this might look closer to the cpubound processes.

  An optimum value is for the given cpubound process is: 100.
- When loop1 value is high, there are a lot of prints which can make it difficult to read the process priority.

  An optimum value used here is 3.

Once all the processes are running and dynamically prioritized (this can be seen in the change in the value of the priority value of the processes.). Note that the initial priority, when the process is created is always 200 as this is the initial priority given in the main process test case functions:

```
resume(pr[i] = create(iobound, 1024, 200, "cpubound process", 2, 10,
3000));
```

Half and Half output and explanation:

```
PID: 3,
Outer loop: 5,
Process Priority: 200,
 Remaining time slice: 28
```

```
PID: 3,

Outer loop: 4,

Process Priority: 200,

 Remaining time slice: 18


PID: 3,

Outer loop: 3,

Process Priority: 200,

 Remaining time slice: 9


PID: 5,

Outer loop: 5,

Process Priority: 200,

 Remaining time slice: 28


PID: 5,

Outer loop: 4,

Process Priority: 200,

 Remaining time slice: 18


PID: 5,

Outer loop: 3,

Process Priority: 200,

 Remaining time slice: 9


PID: 7,

Outer loop: 5,

Process Priority: 200,

 Remaining time slice: 28


PID: 7,

Outer loop: 4,

Process Priority: 200,

 Remaining time slice: 18


PID: 7,
Outer loop: 3,
Process Priority: 200,
 Remaining time slice: 9

PID: 3,
Outer loop: 2,
Process Priority: 230,
 Remaining time slice: 29

PID: 3,
Outer loop: 1,
Process Priority: 230,
 Remaining time slice: 20

CPU TIME consumed by process: 3 is 30

PID: 5,
Outer loop: 2,
Process Priority: 230,
 Remaining time slice: 28

PID: 7,
Outer loop: 2,
Process Priority: 230,
 Remaining time slice: 28

PID: 7,
Outer loop: 1,
Process Priority: 230,
 Remaining time slice: 19

CPU TIME consumed by process: 7 is 30

PID: 5,
Outer loop: 1,
Process Priority: 240,
 Remaining time slice: 27

CPU TIME co

PID: 4,
Outer loop: 3,
Process Priority: 200,
 Remaining time slice: 0
PID: 6,
Outer loop: 3,
Process Priority: 200,
 Remaining time slice: 0
nsumed by process: 5 is 40

PID: 8,
Outer loop: 3,
Process Priority: 200,
 Remaining time slice: 0

PID: 4,
Outer loop: 2,
Process Priority: 200,
 Remaining time slice: 6

PID: 6,
Outer loop: 2,
Process Priority: 200,
 Remaining time slice: 7

PID: 8,
Outer loop: 2,
Process Priority: 206,
 Remaining time slice: 6

PID: 4,
Outer loop: 1,
Process Priority: 200,
 Remaining time slice: 12


PID: 6,
Outer loop: 1,
Process Priority: 200,
 Remaining time slice: 13
CPU TIME consumed by process: 6 is 13CPU TIME consumed by process: 4 is 18

PID: 8,
Outer loop: 1,
Process Priority: 206,
 Remaining time slice: 12

CPU TIME consumed by process: 8 is 12
```

The output clearly shows the dynamic scheduling between 3 cpubound and 3 iobound processes.

For the group of cpubound processes the dynamic scheduling is stable and as expected. Each process uses up their time slice and when their priority gets updated, the next cpu bound is context switched in. This is also the case for the iobound process group but in these processes, the processes sleep and hence don't use the time slice completely.

The output is exactly as expected. The cpubound processes, being cpu intensive utilize their time slice completely while the iobound sleep processes sleep in the background. Thus in the beginning we see prints only from the cpubound process. The cpubound process also finishes up execution before the iobound processes can use up their first time slices in the above test case. Once all the cpubound processes complete execution, the iobound processes start.

An important thing to note here is that the priority of the iobound processes is more than cpubound as printed in the output but since, the iobound processes sleep, this gives leaves only the cpubound processes in the ready queue and hence these get the cpu cycles.

If the loop values are adjusted, then we can see iobound processes causing the cpubound processes to context switch owning to their superior priority.

Q5.

SPEACIAL INSTRUCTIONS FOR Q5: The main process has not been hardcoded to run completely before any other process but this is required as otherwise the testing becomes difficult and the test cases explanation can be difficult. The default priority of main is retained as INITPRIO = 20. Now, because of the above reason we would like to run the test process once the main completes executing. ***For testing, set the lab2flag = 5 and initial priority of the cpubound process at more than or equal to 200 so that main gets a higher priority always which is required to create all the processes in time. Eg:***

```
create(cpubound, 1024, 200, "cpubound process", 2, 20, 50000));
```

Here, the initial priority is 200 and this will then be updated by the dynamic process scheduling by adding the cpu cycles to this initial priority.

I came up with 2 solutions to this problem.

Set lab2q5sol = 1 or 2 in main.c to get the test solution 1 or solution 2 respectively.

Solution 1:

The first is a simple and O (1) solution which works well but the priority can become too big after a very long time and can lead to overflow. Assuming this is taken care of (as this is the same issue with clktimemsec) the approach is to change the initial priority of a newly created process to be equal to the (= priority of the first process – 1) in the ready queue. This ensures a kind of normalization of priority and doesn't give a newly created process too much advantage. This method also ensures that each new process always has the highest priority which is fitting with our scheme of 1/prcpumsec priority. The output for both Sol 1 and Sol2 is similar and is explained below.

Solution 2:

This is a more complex solution and will take O (n) time but it normalizes all the priorities hence, priority values don't go to very large values.

The solution here is that at each rescheduling, the priority of each process in the ready list is updated with a linear function in the cycles consumed by the process and the cumulative waiting time of the process. This is kind of an adaptation of the aging solution on process i.e as a process waits more, it's priority should be increased.

The linear solution I have found which works well is this:

$$\begin{aligned} &New\ priority \\ &= initial\ priority \\ &+ (cpu\ cycles\ consumed\ by\ process) - \frac{(cumulative\ waiting\ time\ of\ process)}{(no\ ready\ processes * QUANTUM)} \end{aligned}$$

To show, how effectively this scheduling works, here is part of output:

PID: 4,

```
Outer loop: 4,

Process Priority: 1520,

 Remaining time slice: 19

ning time slice: 6


PID: 7,

Outer loop: 7,

Process Priority: 15


PID: 5,

Outer loop: 4,

Process Priority: 1539,

 Remaining time slice: 15


PID: 6,

Outer loop: 4,

Process Priority: 1550,

 Remaining time slice: 20


PID: 8,

Outer loop: 7,

Process Priority: 1550,

 Remaining time slice: 15


PID: 3,

Outer loop: 3,

Process Priority: 1547,

 Remaining time slice: 20

29,

 Remaining time slice: 5


PID: 4,
Outer loop: 3,
Process Priority: 1580,
 Remaining time slice: 28
```

As we can see, even though the priority of the first process with pid 3 is almost equal to the priority of the last process created after 2750ms which is pid 8. Also notice, that the pid 3 process has not been starved by process with pid 8 as pid 3 is in 7th Outer loop (i = 3 decrementing from 10) and pid 8 is in 3rd outer loop (i = 7 decrementing from 10). Thus, this shows that the above is a fair scheduling and works as requested to be designed.

Bonus Problem:

Every important code statement has been documented just above the statement. Hope you find the comments useful.