LAB3Answers

Problem 3.
For testing test cases, include the lab2.h file in testcases.

The TS scheduling table derived from the Solaris TS and reduced to 20 priority levels is constructed to be:

```
/* Initializing the TS scheduler priority data structure */
struct ts_disptb tsdtab[NOF_PRIOIRITIES] = {
    {0, 17, 200},/* Prioirity 0*/
    {0, 17, 200},/* Prioirity 1*/
    {0, 17, 200},/* Prioirity 2*/
    {0, 17, 160},/* Prioirity 3*/
    {1, 17, 160},/* Prioirity 4*/
    {2, 17, 160},/* Prioirity 5*/
    {3, 17, 160},/* Prioirity 6*/
    {4, 17, 120},/* Prioirity 7*/
    {5, 17, 120},/* Prioirity 8*/
    {6, 17, 120},/* Prioirity 9*/
    {7, 17, 80},/* Prioirity 10*/
    {8, 18, 80},/* Prioirity 11*/
    {9, 18, 80},/* Prioirity 12*/
    {10, 18, 40},/* Prioirity 13*/
    {11, 18, 40},/* Prioirity 14*/
    {12, 19, 40},/* Prioirity 15*/
    {13, 19, 40},/* Prioirity 16*/
    {14, 19, 40},/* Prioirity 17*/
    {15, 19, 40},/* Prioirity 18*/
    {16, 19, 40},/* Prioirity 19*/
};
```

The above is an array of structures. The structure itself is exactly the same as the one given in the lab question. The array index itself is the current priority and this is used to get the next priority from the table based on the process being classified as IO-Bound or CPU-Bound.
As can be noticed above, a constant NOF_ PRIOIRITIES (actually a macro constant) has been declared which represent the total number of priority levels available for processes. This value is equal to 20.

Firstly, for the classification of a process into IO-bound or CPU-bound, a variable called timesliceconsumed is used as a flag in clkhandler.c and resched.c. if the time slice allotted to a process is consumed fully, then this is set to 1. Based on this variable the new priority of the process is looked up in the tedaat table above.

Multi-level feedback queue has been implemented. This is basically an array of structures, each structure containing a head and a tail in the readyqueue. By getting head and tail for each priority level and with implementation of the mltfbq_insert() and mltfbq_dequeue() functions, processes can be added based on priority in constant time.

To make sure that the null process runs only when no other process is ready, I have made sure that the priority of the null process remains 0. New process priority can be set to any level between 0 and 19 in the process creation, but for all the test cases a median priority of 10 is set. This is also because in the beginning a process is neither IO-bound nor CPU-bound.

Additionally, the priority of main has been set to 11, more than the testing processes. This is done so that the all test process can be setup and also so that main process gets enough slices to do this.

Testing

Only CPU-bound:
6 CPU-intensive processes are run at the same time with an initial priority of 10. I observed that the priorities continuously and sharply decrease for all the process, finally reaching 0, as each timeslice is consumed completely by the processes. Also, as the priorities are decreasing the timslice itself is increasing as we intended from the tsdat table. Additionally, the rate of decrease is the same and hence we see equal sharing of the CPU among the processes. Finally, all the processes end up using same number of CPU cycle, thus this scheduling is fair for CPU bound processes.

Only IO-bound:
6 IO-intensive processes are run at the same time with an initial priority of 10. I observed that the priorities continuously and increase for all the process as each process relinquish cpu, thus not consuming the timeslice assigned completely. Also, as the priorities are increasing the timeslice assigned to these processes on rescheduling itself is decreasing as we intended from the tsdat table. Finally, all the processes end up using almost same number of CPU cycle, thus this scheduling is fair for only IO bound processes.

Half and Half
3 CPU- bound and 3 IO- bound processes were run with the same initial priority of 10.

IO- bound:
To start of all the processes were at priority 10 but very soon the priority of all the IO-bound processes reached 19 quickly. Among themselves these process achieve aqlmost equal cpu cycle sharing which is fair. Also, these processes complete after all the CPU-bound processes have been completed.

CPU-bound:
These processes begin with a priority of 10 but soon stabilize to a priority between 7-9. This is because, there is a balancing act between these process being preempted and hence gaining priority and sometimes using up all of their cpu cycles (since these are CPU intensive) and losing some priority. Overall they finish before IO-bound and share exactly the same number of cycles on completion.

Among the above classes:
Since, the CPU- bound processes are context switched out more and than in the case of only CPU- bound processes and also maintain a higher priority which implies a smaller time slice,; all this means that CPU-bound along with IO-bound processes take more time to complete than in the earlier case of just CPU-bound. Also, they maintain a higher average priority level. Finally, they still end up executing earlier than the IO- bound processes. The IO-bound on the other hand maintain a higher priority level when compared to CPU-cound processes and cause them to preempt often. Overall, there is fair sharing of cycles among the 2 groups and there is a stability in between these two groups.

Problem 4:

For testing include the sendtb() as extern in the testing file.

The blocking send has been implemented as sendbt() system call. This system call, allows to have a timed waiting send as well as a blocking send.
“wait” is a parameter of this system call; when this is positive, the sending process will be waiting a maximum of “wait” milliseconds. If this is 0, then sendbt() becomes a blocking send, meaning the process goes into a waiting state called PR_SLEEP which will become ready only when the receiver is ready to take it’s message.

Implementation is done by using a per-receiving process queue called the receiverq in receiverq.c. The enqueue and dequeue functions are implemented in the same file such that the above operations can be done in constant time O(1).

Test Cases:
Note that for all the below test cases the same receiver is used which sleeps for 100 ms after each receive() call.

Test case 1:
In this case, 4 processes are created and all are blocking sends to the same receiver. **As expected, all the messages are received as each send is blocking. (the first received pid is not printed as programming not done that way)**

Output:
Sending from PID: 4, to 3, value: 100
Sending from PID: 5, to 3, value: 125
Sending from PID: 6, to 3, value: 150
Sending from PID: 7, to 3, value: 175
Receiving from PID: 5 Receiving value: 100
Receiving from PID: 6 Receiving value: 125
Receiving from PID: 7 Receiving value: 150 Receiving value: 175

Test Case 2:
In this case, 4 processes of varying wait time, each wait time more that 100 ms is tested.

Sending from PID: 4, to 3, value: 200
Sending from PID: 5, to 3, value: 225
Sending from PID: 6, to 3, value: 250
Sending from PID: 7, to 3, value: 275
Receiving from PID: 5 Receiving value: 200
Sending not done
Sending not done Receiving value: 225

As it clear, even though the wait time were more than the receiver sleep, still only 2 messages are received. This is because the timer to all of these processes take place at the same time and in the sleep times only 2 processes could be read. Note that this is because sleeptime of receiver is 100 ms, thus within in 225 ms only 2 messages can be read.

Test Case 3:
In this case, 4 processes, each with wait time less than 100ms is tested.

Sending from PID: 4, to 3, value: 300
Sending from PID: 5, to 3, value: 325
Sending from PID: 6, to 3, value: 350
Sending from PID: 7, to 3, value: 375
Sending not done
Sending not done
Sending not done Receiving value: 300

As expected, not all the messages are received as the sleep time of the receiver is greater than the wait time of the sending processes. An important thing to notice here is that, the first message is still received. This is because, in the beginning, the 1-word buffer of the receiver will be empty and hence, the first sender can put it's message in this empty buffer. But after this, the receiver sleep and hence all other process can't send the messages.

Test Case 4:
In this case, 1 process with negative wait time is tested.
Sending from PID: 4, to 3, value: 350
Sending not done

This is as expected as wait <0 is illegal.

Bonus Question:

To calculate time taken by a reschedule operation, I have taken the clktime value before the resched() is called and subtracted it with the clktime just before ctxsw() is called. This is a good measure of the time taken per context switch as ctxsw() are fixed number of cycles always and thus the above computed value will be a fixed delta from the actual value.

Running this only over CPU-bound processes and varying the number of processes we get this table:

Number of processes	Total time for rescheduling	Number of resched() called	Avvg time for one resched()
50	964	81	11.9
100	1954	159	12.2
150	2895	222	12.96

The avg resched time is increasing slightly as the processes increase. To run the test case find lab3q6.c file. As can be seen above resched is almost constant time O(1) which is because of Multilevel feedback queue.