Sam Fellers <sfellers>

Lab 3 Answers

3

After running my test cases located in system/main.c (function testcpuusage()), quite a few times, I conclude that when all priorities are the same the cases will get approximately the same cpu time. However if one of the 3 processes the function creates has a lower priority of any value, it will get ignored. This can be explained by saying process 3 has a priority of 9 and process 1 and 2 have priority values of 10. If process 1 is running and resched() is called, the highest item on the readylist should be process 2. Process 1 should then get added back to the top of the readylist as it has a higher priority than 3, there is never an instance where 2 ready processes have the same priority thus the round robin scheduling does not take effect. This should happen until all processes are killed.

4.3

My design choice was to add a condition in resched() when checking to see if the cpu should be handed over. In the if statement :

```
if( ptold->prprio < firstkey(readylist) && (currpid != NULLPROC))
      return;
```

This checks if the priority of the running process is less than the first item of the readylist. If it is then we should continue running the current process. I added (bolded) a check to see if the running process was the null process. This check ensures that even if the priority of the nullproc is lower we should still give up the cpu since the nullprocess should only run when there is nothing left for the cpu to do.

The CPU does not share its time fairly for the 4 cpubnd() processes, the results differed quite a bit but every trial I ran with different LOOP1 and LOOP2 values ended with one process taking up quite a bit of CPU time.

The CPU does share well when it is using the 4 iobnd() processes, this is because it gets a context switch every loop it runs which forces it to context switch out in favor of a new process each time rather than just ALU operations. Also note that loop values should be lower with iobnd() because it sleeps so often it takes a while to run.

When we mix them iobnd() end up with a much higher priority while cpubnd() don't take much time at all and get done quickly while iobnd() keep ctxswitching out.

5

The defense against an attack like this is isolation/protection. If you give processes access to the same stack you are putting a writable return address somewhere in the memory at any given point. Separating stacks is the way to do it.

Bonus

I would create a flag inside process.h that specifies an RT process and make a create function createRT() which does the same thing as create, but sets this flag to be 1 (create would init it to 0) . I would then create a separate priority queue, which these processes would be inserted in) that would be checked **before** readylist when calling resched(). If the queue was empty readylist would then be checked for ready processes.