UMass Lowell **Assignment #3**
CS 308 - Intro to Operating Systems

IMPLEMENTING A CPU SCHEDULER – DUE: FEBRUARY 16, 2005

Assignment Summary

You will implement a CPU scheduler in the OSP framework.

Practical Notes

For the implementation project, there are two things to turn in:

- The paper assignment cover sheet (a copy is attached) describing your results.
- Electronic files using our `submit` system.

Part I: Textbook Problems.

Prepare written answers to the following questions. *Turn these in with your cover sheet submission on the assignment due date.*

1. Tanenbaum, Problem 1, pp. 153 (process state transitions).
2. Tanenbaum, Problem 15, pp. 154 (threads without select).
3. Tanenbaum, Problem 18, pp. 154 (races).
4. Tanenbaum, Problem 21, pp. 154 (Peterson's solution).

Part II: Implementation.

In this part of the assignment, you will implement one module of a simplified operating system, and will test it using the operating system simulator *OSP*. You will implement `CPU`, the CPU scheduler.

You should write the code to satisfy the descriptions given in the OSP Programmer's Manual and the requirements below, making use of what you have learned about operating systems to fill in the missing details. The modules you write will be linked with the rest of the system, and tested with various simulation parameters.

Before attempting to do this assignment, make sure that you have skimmed through all of *OSP: An Environment for Operating Systems Projects*, and have studied in detail Sections 1.1, 1.2, 1.3, 1.4, 1.7, and 2.

The "class account" referred to in the OSP pamphlet is

```
/usr/cs/fac4/fredm/308/files/osp/
```

The subdirectory for this assignment is `cpu`. To begin the assignment, copy the files:

```
cpu/Makefile
cpu/cpu.c
cpu/dialog.c
cpu/osp.o
cpu/OSP.demo
```

into your own work directory. *You should work on* `mercury` *or another department Linux machine. You will link to object code that has been compiled for Linux.*

You will have to edit the file `cpu.c` to incorporate your own code. After edits, run `make` to compile and link with the rest of the system, resulting in an executable file `OSP`.
You will submit your program and three simulation runs electronically using the `submit` program. *Further information about what files to submit will be posted on the ikonboard.*

Evaluate your scheduler under three different performance runs. Simulation parameters for the three runs are in:

```
cpu/sim/par.trace
cpu/sim/par.high
cpu/sim/par.low
```

The first parameter file will produce a short simulation run with the trace switch on. The `par.low` file will test your program with low frequency of new process arrival, whereas in `par.high` this frequency is high.
To compare your results with the statistics generated by the standard CPU scheduler, see:

```
cpu/sim/run.low
cpu/sim/run.high
```

Make sure that your *name* and *login name* are included in your source file. Also, include (as comments) a one-half to one page explanation of the statistics obtained during your test runs, and how they compare with those obtained by the standard scheduler.

Module `CPU`
The module `CPU` includes the function `insert_ready`, which is used to insert a PCB into the ready queue at the proper position, and the dispatcher (function `dispatch`), which performs a context switch to the next process to be run. The dispatcher must perform the following tasks:

- If there is currently a process running, and it has not finished its CPU burst, then return that process to the ready queue, changing its state from “running” to “ready.”
- Select the next process to be run (if any) from the ready queue, changing its state from “ready” to “running.”
- Initialize the `last_dispatch` field in the PCB to the current system time, and make any initializations required in other scheduling fields in the PCB.
- Set the timer to interrupt at the end of the next quantum.
- Perform a context switch to the new process by making the page table base register `PTBR` point to its page table.

Scheduler Details

You should implement a scheduler that has a two-level feedback queue, consisting of a “high priority queue” and a “low priority queue.”

You should maintain information about the recent CPU usage of a process, and attempt to put I/O-bound processes in the high-priority queue and CPU-bound processes in the low-priority queue, so that on the average, the high-priority queue contains processes with short CPU bursts and the low-priority queue contains processes with longer CPU bursts.

The dispatcher should not always prefer the high-priority queue over the low-priority queue, but instead should attempt to allocate CPU time between the two queues to get the best system performance. One scheme would be to allocate CPU time equally between the two queues, so that the high-priority queue will tend to cycle faster than the low-priority queue (assuming you have managed to get mostly processes with short CPU bursts in the high-priority queue and mostly processes with long CPU bursts in the low-priority queue).

You should try to tune your scheduler to get the best performance under a variety of conditions. The main parameters you can adjust are the ratio of CPU time allocated to high-priority queue as opposed to the low-priority queue, and the method by which you decide into which queue a process should be placed.

HINT: First, build your understanding by implementing a simple round-robin scheduling algorithm. Getting this running will be most of the battle. Once you have this working, then implement the more sophisticated scheduling algorithm described above.