Assignment 4

Scheduling

Final Patch Due: 11:00 pm, Friday, April 8, 2005

In this assignment you will be implementing two alternative scheduling algorithms for OS/161: *nonpreemptive random scheduling* and the *multilevel feedback scheduling* algorithm described in class.

Preliminaries

Recall that you *must* have your path correctly set to undertake all CS 182 assignments. If you haven't done so already, put the appropriate line in your `.login` or `.bashrc`.

You no longer need your code from Assignment 3. You may delete your entire source tree (although you may wish to save your patch and any other changes for posterity).

For this assignment, check out and `setup` your `assign4` directory in the usual way, then configure and build the kernel using the `HW4` configuration files, `HW4-RR`, `HW4-MLF`, and `HW4-RAND` (see previous assignments for details).

Note that a single instance of the OS/161 sources can support multiple configurations and builds. For this homework, you will have build areas named `HW4-RR`, `HW4-MLF`, and `HW4-RAND` in `src/kern/compile`. Each build directory is independent.

Code Reading

You should have already read enough of OS/161 to understand how the thread system and scheduler works at a basic level. You should review `scheduler.c` and `thread.c` to ensure you understand how the existing round-robin scheduler works. Note that the scheduler has changed very slightly since the last time you saw it—the `hardclock` function in `hardclock.c` used to call `thread_yield`, but now it calls `thread_timeryield`, so that the involuntary context switches initiated by `hardclock` can be differentiated from other, voluntary, context switches.

In addition, you should make sure you understand the "`brie`" patch that provides file and process system calls. This patch is already applied to the code, and on display on the CS 182 Wiki site. In addition, a patch has been applied to add a simple shell to OS/161. The shell is available by using the `s` command from the OS/161 menu.

Design and Implementation Requirements

The most fundamental requirement is that your code be simple and easy to follow. You *may not* use any clever data structures (such as a priority queue) to represent the queue of ready processes. You must use an array (as in the provided code) and use a simple linear search—the array is usually going to be short enough that any inefficiency shouldn't matter too much.

Multilevel Feedback Scheduler

Your multilevel feedback (MLF) scheduler should behave according to the description given in the *Example "Real" Scheduler* class handout (also available on the course website). You may add code that should only be compiled in when the MLF scheduler is used by bracketing this MLF-specific code as follows:

```
#include "opt-mlfsched.h"
    ⋮
#ifdef OPT_MLFSCHED
    ⋮     (MLF-specific code)
#endif /* OPT_MLFSCHED */
```

The same technique can also be applied to make conditional code for the RR and RAND schedulers.

Nonpreemptive Random Scheduler

Your nonpreemptive random scheduler (RAND) scheduler should pick a random ready task and run it until the task either completes or blocks.

You should, however, determine a way to handle processes that are compute-bound and do not complete in a reasonable amount of time. It is up to you to decide on a reasonable strategy to handle such tasks—you will need to preempt such unresponsive tasks, but it is up to you to decide on (and justify) the correct way to handle these processes once the kernel has made the determination that they are taking too long.

Patch

As usual, once you have final working code, you should create a patch for OS/161. The patch should be named `sched.patch` and placed in the `handin` directory, as usual. Probably you should not create the patch until after you have completed the performance analysis step described in the next section.

Performance Analysis

In this part of the assignment, you will perform a qualitative and quantitative analysis of the performance of all three scheduling algorithms. In order to compare performance quantitatively, you will need some numerical measures.

You should have noticed System/161 provides some hardware performance counters, such as the number of cycles in supervisor mode, the number of cycles in user mode, and so forth, which it prints when the kernel exits. These measures, along with the fact that kernel menu arguments can be given on the `sys161` command line should allow you to obtain some basic measurements of execution time.

You may find it useful to implement some software counters as well. As a start, we suggest:

- The number of voluntary context switches (that is, when a process gives up the processor voluntarily, typically by requesting I/O).
- The number of involuntary context switches (that is, when a process is forced to give up the processor, typically because a quantum expired).
- The response time (or time to completion) of a process.

You should add the necessary infrastructure to maintain these statistics as well as any other statistics that you think you will find useful in tuning your schedulers.

Once you have added any instrumentation, it is time to tune your operating system.

Comparing Schedulers and Tuning Scheduler Parameters

Maintain all your data and observations from this section in a file named `performance.txt` in the `handin` directory.

P1. Choose several of the test programs from `testbin` to test your scheduler (e.g., `add.c`, `hog.c`, `farm.c`, `sink.c`, `kitchen.c`, `ps.c`), or write your own.

P2. Run each of the programs using the default time slice and the round-robin scheduling algorithm. Record the average performance of a few runs of each in `performance.txt`. (Your goal should be to improve on these numbers.)

P3. Vary your time slice. Before running the program, predict what should happen to performance (and discuss your prediction with your partner). Now run the programs with the new time slice and discuss the results relative to your predictions. Make sure you explain these results thoroughly.

P4. Run the programs under the MLF and RAND schedulers. Once again, compare what actually happened to what you predicted and explain the difference.

P5. Experiment by varying some of the parameters of the MLF algorithm, and record the results.

P6. Based on your results, choose a "default" scheduling policy (both algorithm and time slice) that you think will be the best for a variety of workloads.

Although the steps above form a linear sequence, you can use your judgment in developing a testing strategy. In particular, it may help to analyze just one program first, and then choose another that seems like it might have quite different behavior.