Homework 3

Software Systems **Allen B. Downey**
Spring 2005

Due Thursday 24 February.

The purpose of this assignment is to investigate the Linux scheduler and try to figure out what's happening in the kernel by running user-level experiments.

Preparation

1. Download the code for this week:

   ```
   wget wb/ss/code/sched.tgz
   tar -xzf sched.tgz
   ```

2. One of the programs you have is called `alarm`; it executes an infinite loop and computes cosines (for no good reason other than keeping the processor busy).

   Read `alarm.c` and make sure you understand what it does. Notice that the program takes a command-line argument. Please read pages 201–206 of the cow book for information about command-line arguments.

3. `Makefile` contains instructions for compiling this project. Please read pages 313–316 of the cow book for information about Makefiles. To compile `alarm`, just type `make alarm`.

4. After you compile alarm, run it with the `time` command:

   ```
   $ make alarm
   gcc -o alarm alarm.c -lm
   $ time ./alarm 3
   Alarm clock

   real    0m3.002s
   user    0m3.010s
   sys     0m0.000s
   ```

   The message "Alarm clock" is printed by `alarm` when the 3-second alarm clock goes off. The next three lines are produced by `time` to report the elapsed "real" time from the beginning to the end of the program, the amount of CPU time the program executed in user mode, and the amount of CPU time spent performing system calls on behalf of this process.

   Notice that in this case, we managed to use 3.01 seconds of CPU time in only 3.002 seconds, which is a pretty good indication that these reports are only approximate. Type `man time` to get the manual page for the time command.

5. You can run two commands on the same line if you separate them with a semi-colon.

```
$ time ./alarm 3 ; time ./alarm 3
```

In this case, the two instances of `alarm` run sequentially and produce the same output we saw before. Now try this:

```
$ (time ./alarm 3 &) ; time ./alarm 3
```

The ampersand causes the first command to run in the background, so the two processes run concurrently. The parentheses are necessary to keep the shell happy.

Look at the output of this command and make sure you understand it before you continue.

6. Write a program called `cpuloop` that runs a mathematically intensive loop for a fixed number of iterations (hint: start with `alarm.c`). Make it take a command line argument that controls the number of iterations. Add a line to the Makefile that specifies how to compile `cpuloop`. Compile the program by running `make cpuloop`.

   Calibrate the program so that the argument is approximately the run time in milliseconds. The program should not print anything.

   This program will be CPU-bound; that is, its run time will be primarily determined by how much CPU time it gets. Other programs might be I/O-bound, meaning that their performance is determined by the performance of one of the I/O systems, and relatively insensitive to the amount of CPU that's available.

Experiment 1

If you start two processes at the same time, you expect each of them to get about half the CPU cycles. Test this hypothesis by running the following command:

```
$ (./alarm 10 &) ; time ./cpuloop 3000
```

Make sure that the alarm time is long enough that `cpuloop` completes before alarm. What fraction of the CPU time did `cpuloop` get? Run your programs several times to get an idea of how consistent your results are. If you see variation in the results, can you explain it? Warning: make sure `alarm` completes before you start the next run.

Now instead of starting the programs at the same time, introduce a delay between when you start alarm and when you start `cpuloop`:

```
$ (./alarm 10 &) ; sleep 1; time ./cpuloop 3000
```

Again, make sure `cpuloop` has time to complete before alarm finishes. As the delay increases, what happens to the percentage of the CPU that `cpuloop` gets? What does this relationship tell you about the processor scheduling policy?

Print and read the documentation of `nice` and use it to adjust the priority of `alarm` and `cpuloop`. How much does the priority of the two processes have to differ before one of them gets 90% of the CPU?

As a possible extra exploration, plot the relationship between the difference in priority and the ratio of CPU allocated to the processes.

Experiment 2: CPU-bound vs. I/O bound

Make a copy of `cpuloop` called `ioloop`. In `ioloop`, replace the mathematical busy work with some I/O busy work. As one possibility, you could add a line that prints the value of `i`. Printing things involves a lot of I/O-intensive interaction with the video card. Alternatively, you could experiment with other I/O operations, like reading and writing files.

Again, calibrate the program so that the command-line argument you provide determines the run time of the program, approximately, in milliseconds.

Run the program for 10 seconds and compare the total CPU time (user + system) to the real time. This ratio is sometimes called "utilization." If it is significantly below 100%, it is probably because the process is spending lots of time waiting for I/O.

As usual, run the program several times to get an idea of the variability in performance. Is this process more or less variable than the CPU-bound process?

Run the program again with alarm running in the background. What effect do you expect the background process to have on the performance of `ioloop`? What do you see?

Now try running `cpuloop` in the background and `ioloop` in the foreground. Tune the two programs so that they finish at the same time. What effect does the foreground process have on the background process? Is the scheduling succeeding at interleaving the two processes?

Experiment 3: I/O-bound vs. I/O-bound

Run two instances of `ioloop` concurrently and study how they interact and affect each other's performance.