Homework 04

Allie Duncan, Zach Homans, and Kyle McConnaughay

Experiment 1:

Summary:

By using alarm and cpuloop, we set out to discover how scheduling works for two CPU-bound threads. This was done by running alarm and cpuloop concurrently in a bash shell. Time reports back how long the whole process took, the amount of time the CPU was busy and the amount of time the system took to perform function calls. We expected each thread to receive all of the CPU time that it requests because our computer has four cores. So we can theoretically run four CPU-intensive threads and have each receive most of the requested CPU time. The actual results closely matched our expectations; the threads received, on average, 99.5% of the CPU time they requested. This 0.5% difference can be explained by the fact that the processors also run background processes.

When using the command:

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

As stated above, the tread received approximately 99.5% of the CPU time with no statistically significant variation.

When using the command:

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

When sleep was used to increase delay, the percentage of the CPU that cpuloop gets remains the same.   This implies that the scheduler still runs these threads on separate processors. The running of alarm on one processor and running sleep with cpuloop on another means that they do not affect each other.

Finally, we use nice to adjust the thread's priority. For one process to get 90% of the CPU, the priority has to differ by a niceness of 7 (this is while running 12 background cpuloops).

Experiment 2:

Summary:

By using ioloop and cpuloop, we can examine how each use the CPU. Cpuloop, of course, uses almost all of the CPU time. However, ioloop barely uses CPU time. This process is more varied than the CPU-bound process, presumably because I/O depends more on the demand on the system than cpuloop. We expected them to barely affect each other because they don't utilize much of the same hardware. This expectation is confirmed by our experimental data. Ioloop uses less than 0 ms of CPU time while cpuloop monopolizes almost all of the CPU. This data was rather regular with no statistically significant variations.

Experiment-specific reflections:

Ioloop is much more variable than a CPU-bound process. Ioloop running in the foreground has little to no effect on CPU-heavy processes, such as alarm. This is because ioloop rarely requires attention from the CPU. This is evidence that the scheduler is succeeding at interweaving the two processes.

Experiment 3:

Summary:

We wished to study how two concurrent ioloop threads affected each other's performance. In order to make sure that the threads were both competing for time on the same processor, we ran twelve ioloop threads in the background for 100 seconds each. While those threads were running we ran two more concurrent threads, setting one for ten seconds of run time and the second for two seconds run time. The run time for both threads almost tripled, however. This is because all the threads were attempting to write data to the same structure, thus impeding the progress of all threads. The scheduler had much more difficulty interleaving these processes as opposed to the cpuloops because it lacked anything akin to multiple processors to use while running ioloop.