SoftSys Homework 4

Orion Taylor, Joe Gibson
9/17/2012
Olin College of Engineering

Experiment 1A:

The purpose of this experiment is to determine how forcing a thread to sleep affects the amount of CPU resources that are allocated to it. To do this we ran two threads concurrently. The first thread runs a function called alarm. Alarm runs a loop that eats up computation power for a given number of seconds. The second thread runs a function called cpuloop. cpuloop runs a loop that eats up computation power for a given number of iterations. A thousand iterations of this loop generally eats up about one second of processor time.

In this experiment, the alarm thread would be set to run in the background for 10 seconds. The cpuloop thread would be forced to sleep for a given amount of time, then run cpuloop such that cpuloop iterate enough to eat up about 3 seconds of processor time.

In our model, when two threads are running simultaneously, each receives half of the processing power. To test this, we ran cpuloop and alarm simultaneously without any wait. The ratio of cpu use time to run time for cpuloop was 0.4954, about ½. This is consistent with the model.

If the second thread sleeps such that all of cpuloop occurs simultaneously with the alarm thread, the cpuloop thread ought to be allocated only 50% of computer resources the whole time.

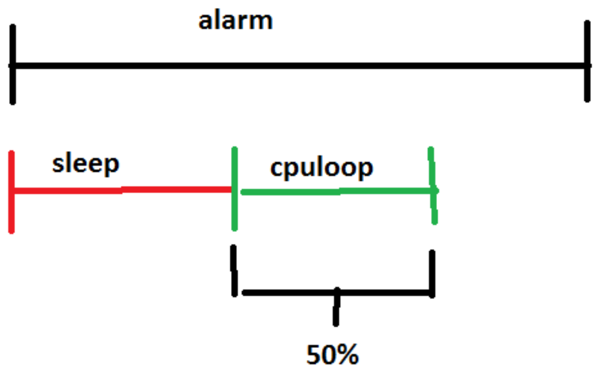

Figure 1: cpuloop only get 50% of cpu resources since alarm is in background.

If the second thread sleeps such that the first part of cpuloop occurs simultaneously with the alarm thread, but the second part does not, then the cpuloop thread ought to be allocated 50% of computer resources during the simultaneous portion and 100% of computer resources when it is running alone. The average amount of computer resources ought to be determined by how much time cpuloop runs during each section.

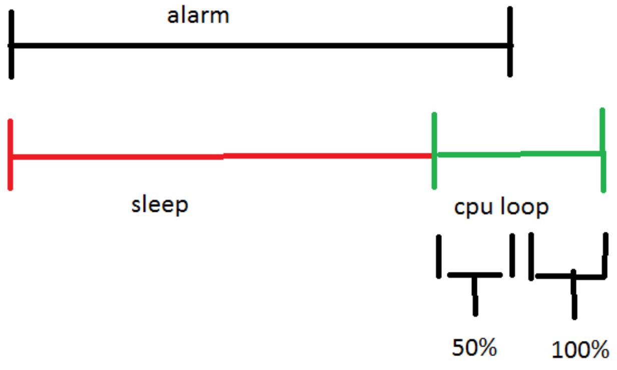

Figure 2: cpuloop get 50% of cpu resources when alarm is in background, but 100% when alarm is not running in background.

If the second thread sleeps such alarm has finished running by the time it wakes up, then it ought to be allocated 100% of cpu resources the whole time.

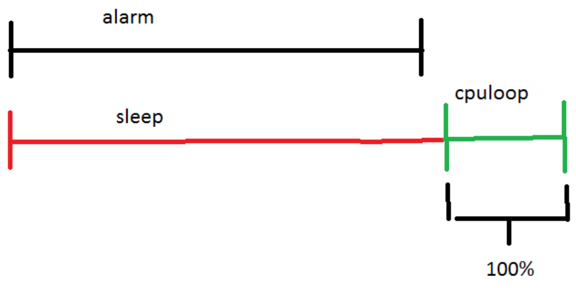

Figure 3: cpuloop recieves 100% of cpu resources when alarm isn't in background

We collected data (fraction of cpu time used for second thread) for sleep times ranging from 0 seconds to 15 seconds,usually at intervals of 2 seconds. Each data point represents the average of 3 samples at a given sleep time.

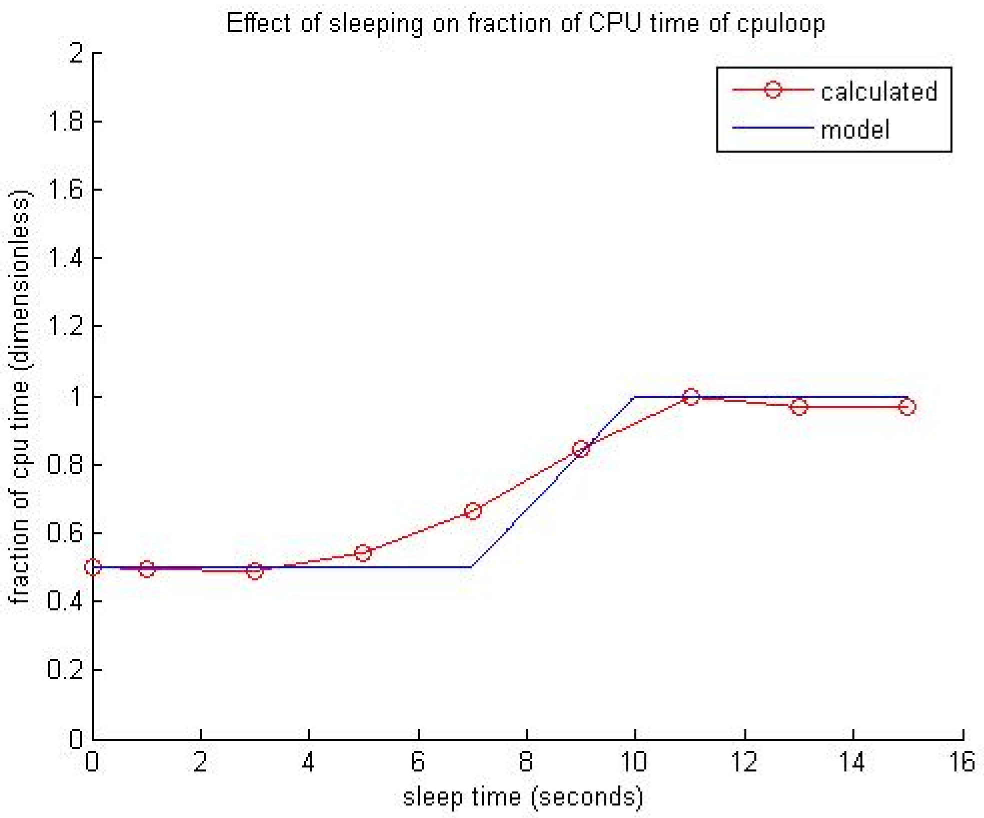

Figure 4: Results of first experiment

As it can be seen, for low values of sleep times, increasing the amount of time the second thread slept had no effect. This is because alarm would still be running in the background by the time the second thread had finished running, meaning that the second thread would be given 50% of the cpu's resources the whole time. At sleep values between 6 to 10 seconds, increasing the amount of time the second thread slept would increase the fraction of cpu resources allocated to it. This is because alarm would finish somewhere between when the second thread had woken up and when it had finished running. Increasing the amount of time the second thread slept would make alarm finish earlier during this interval, meaning that there would be more time for the second thread to run alone and receive 100% of cpu resources. For high values of sleep, increasing the amount of time the second thread slept would have no effect on the fraction of cpu resources allocated to it since alarm would have finished long before the second thread had woken up, meaning that it would receive 100% of cpu resources anyway. All of this is consistent with our model. The data we collected follows the same trend as the curve that our model predicts, but it doesn't match it. We suspect the difference may be due to the fact that switching between threads costs time, which our model didn't account for.

Experiment 1B:

The purpose of this experiment is to determine how the niceness of a thread determines what fraction of cpu resources are allocated to it. Niceness is one of the ways an operating system, like linux, can determine how threads are prioritized over one another. In linux, threads can have a niceness between -20 and 19, with lower priority threads having higher niceness values.

As in the previous experiment, one thread was set to run alarm for 10 seconds. The other thread was set to run cpuloop such that it would eat 3 seconds of cpu time. Alarm was given a niceness of 19, while cpuloop was given varying levels of niceness. The fraction of cpu time (cpu time used divided by run time) was collected, as was % cpu usage. According to our model, the lower the niceness that cpuloop is given, the greater fraction of cpu resources it ought to be allocated. Thus if cpu resources and niceness of cpuloop were plotted against one another, the curve ought to have a negative slope.

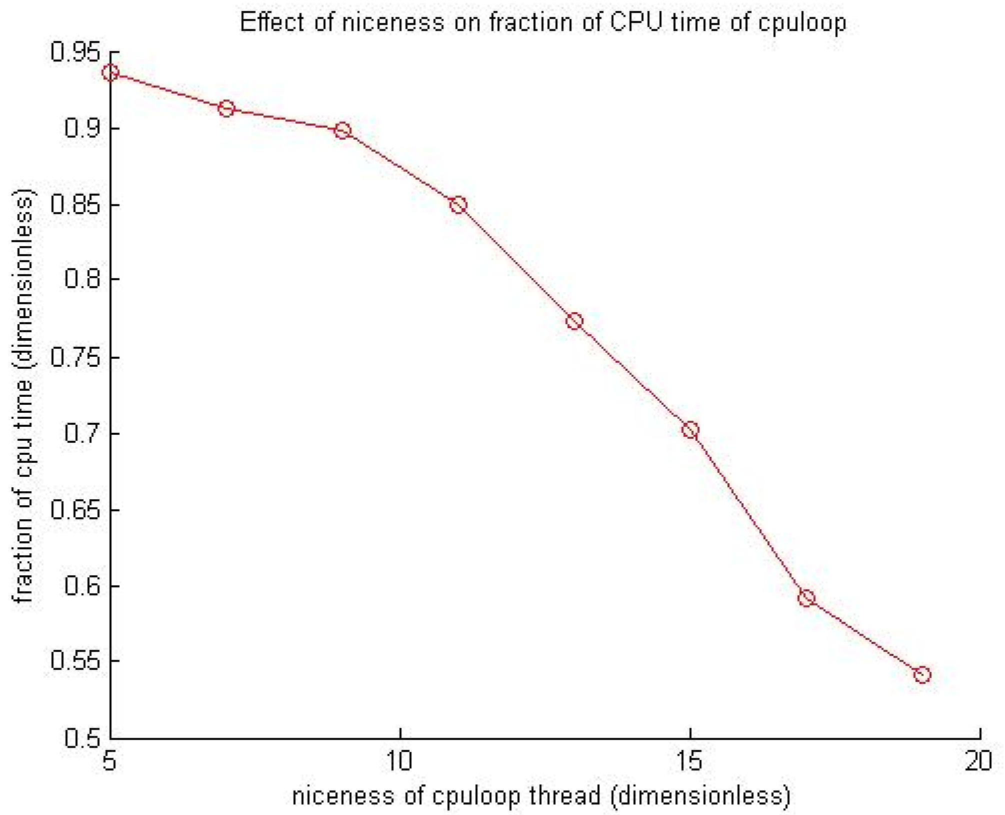

Figure 5: results from second experiment (calculated fraction of cpu time used)

As it can be seen, the data that was collected is consistent with our hypothesis. Threads with lower values of niceness will be given a greater fraction of cpu resources.

The percent cpu usage of the cpuloop thread that was spit out onto the command line for each output was also plotted against the niceness of the cpuloop thread. Unsurprisingly, the curve is very close to the one that we get when plotting the ratio of cpu time to run time against the niceness of the cpuloop thread. From this we can conclude that the computer probably calculates the percent cpu usage of a thread in the same way that we do: dividing the amount of time the thread runs on the cpu by the amount of actual time it took for the thread to run.

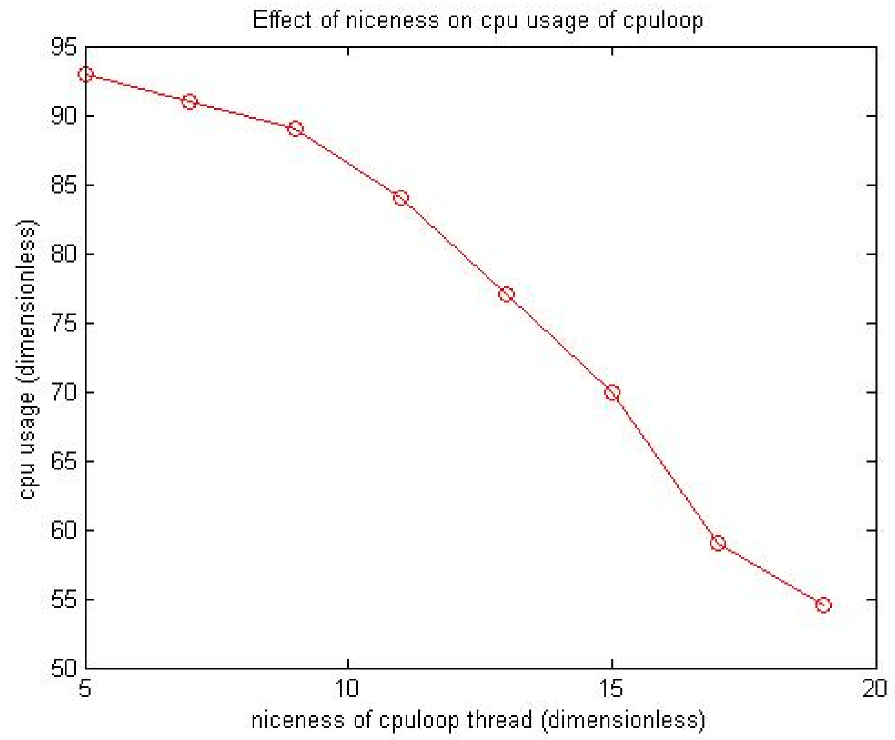

Figure 6: Evidence that the computer computes % cpu usage the way we do

Experiment 2:

The purpose of this experiment is to determine how the performance of writing to a disk is affected by the running of high cpu usage threads in the background. To do this we used a thread that ran a program called ioloop. ioloop runs a loop for a given number of iterations. Each iteration, the loop writes a single character to a file on the hard drive.

First, the performance of ioloop, when running alone, was tested. To do this, we ran ioloop 5 times. Each time we measured how much cpu time was used, how much system time was used, and how much actual time was used. An average cpu utilization ([average cpu use + average system use] / [average run time]) was calculated. The calculated cpu utilization was 3.9432e-04. This is a very very small number. From this, we can conclude that the limiting step in running ioloop was the input/output component. How do we know this? The cpu and system were barely used at all during the running of ioloop, despite the fact that it ran for a significant amount of time. Since there were no other major threads running on the computer, the issue was not that the computer was allocating resources to other threads. The only possibility is that the computer was waiting for the I/O device, the hard drive. Thus, ioloop is a variable process rather than a cpu bound process.

Second, the performance of ioloop when running while alarm was running in the background was measured. To do this, we ran ioloop with alarm (for ten seconds) in the background 5 times. Once again, we measured the cpu time used, the system time used, and the actual time the process took. Once again, cpu utilization was calculated. This time, the calculated cpu utilization was 0. Yes, the calculated cpu utilization was 0. The fact that the cpu utilization is low is consistent with our model of ioloop being a variable process. On the other hand, the fact that the cpu utilization decreased when alarm was run in the background is not explainable by our model. This seems to suggest that a better indicator for performance is not cpu utilization, but rather actual run time. Looking at actual runtimes, the runtime of ioloop (when run alone) took, on average, 2.028 seconds. When alarm was running in the background, ioloop took, on average, 1.896 seconds. This is still not explainable by our model. It may just be that our sample size was not large enough. Given enough samples, it may be possible for the runtime with alarm in the background to actually be larger than the runtime alone.

Finally, the performance of ioloop while cpuloop was running in the background was measured. To do this, we ran ioloop with cpuloop (for 3 seconds) in the background 5 times. Once again, we measured the cpu time used, the system time used, and the actual time the process took. As in the previous test, the calculated cpu utilization was 0. The amount of time it took to run ioloop was actually lower than in the previous two tests, at a whopping low of 1.87 seconds. As before, our model can account for the fact that the cpu utilization is low. It still cannot for the fact that the cpu utilization and runtime is lower than what was measured in the previous two tests.

Statistics, on the other hand, can account for these anomalies. The calculated standard deviation of the run times collected during tests one two and three were .2345 seconds,.1137 seconds, .1126 seconds respectively. These standard deviations are large enough to suggest that the anomaly was the result of having a small sample size.

Experiment 3:

The purpose of this experiment is to determine what effect, if any, two threads running ioloop simultaneously will have upon one another. We tested the interaction between two threads running ioloop in two different ways.

In the first set of experiments, one thread ran an ioloop that wrote characters to a file on the hard drive for approximately 2000 milliseconds, while the second thread ran an ioloop that wrote characters to the hard drive for a variable amount of time. For each amount of time, 3 samples were taken and then averaged. The fraction of a threads run time over the sum of the run times of the threads was calculated and plotted below. Why was this the quantity that was calculated? Relative run time seems like a decent metric for comparing the performance between two threads. If the hard drive devotes equal amounts of time to the read/write functions of each thread when they are running simultaneously, then the thread that has to write more characters should have a longer run time.

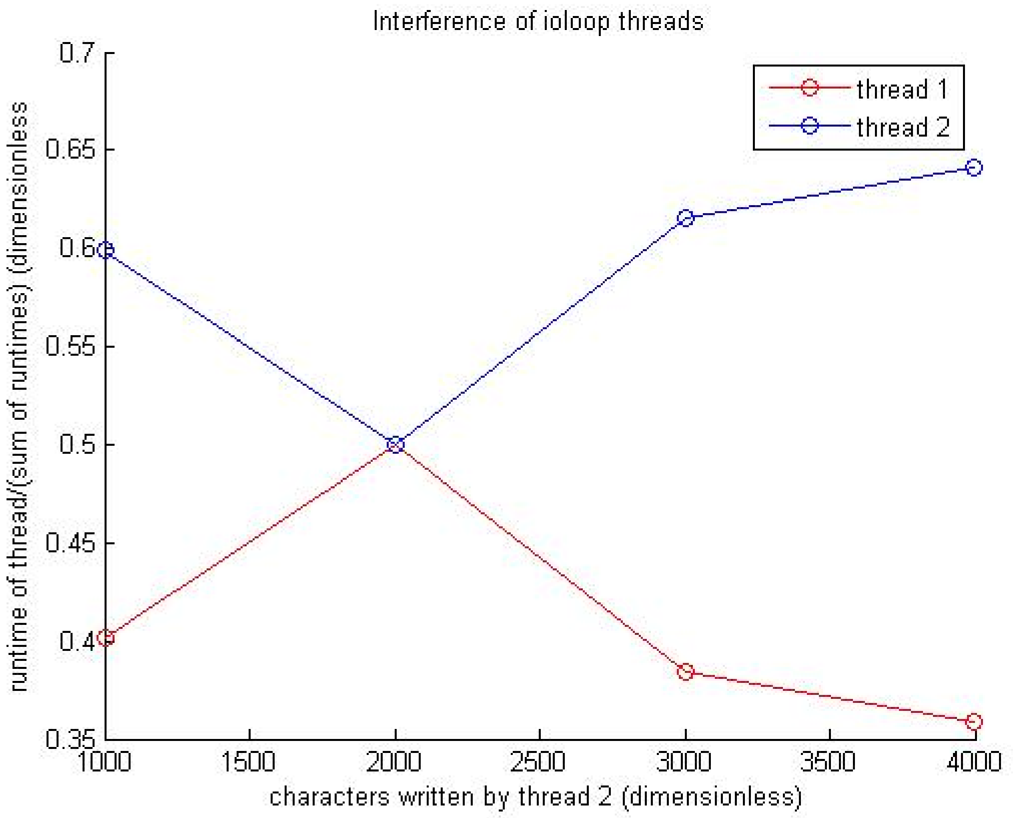

Figure 7: What weird data! This curve doesn't look nice at all! It looks as if some data was accidentally inverted but it wasn't (we checked!)

As it can be seen, thread 2 always got shafted, taking longer than thread 1 to run in every case. The exception was when thread 2 also ran for approximately 2000 milliseconds, in which case both took the same amount of time to run. I am unsure why the results are what they are for when thread 1 runs for approximately 2000 milliseconds and thread 2 1000 milliseconds. The large variance of the ioloop program (as explained in experiment 2) would account for this behavior though.

In the second set of experiments, both threads ran ioloop. The each ioloop was set to iterate enough times to take 2 seconds. The primary difference was that thread 1 was set to a niceness of 0, while thread 2 was set to varying levels of niceness. For each niceness, 3 samples were collected and averaged. The relative runtime of each thread (run time over sum of run times) was calculated and plotted.

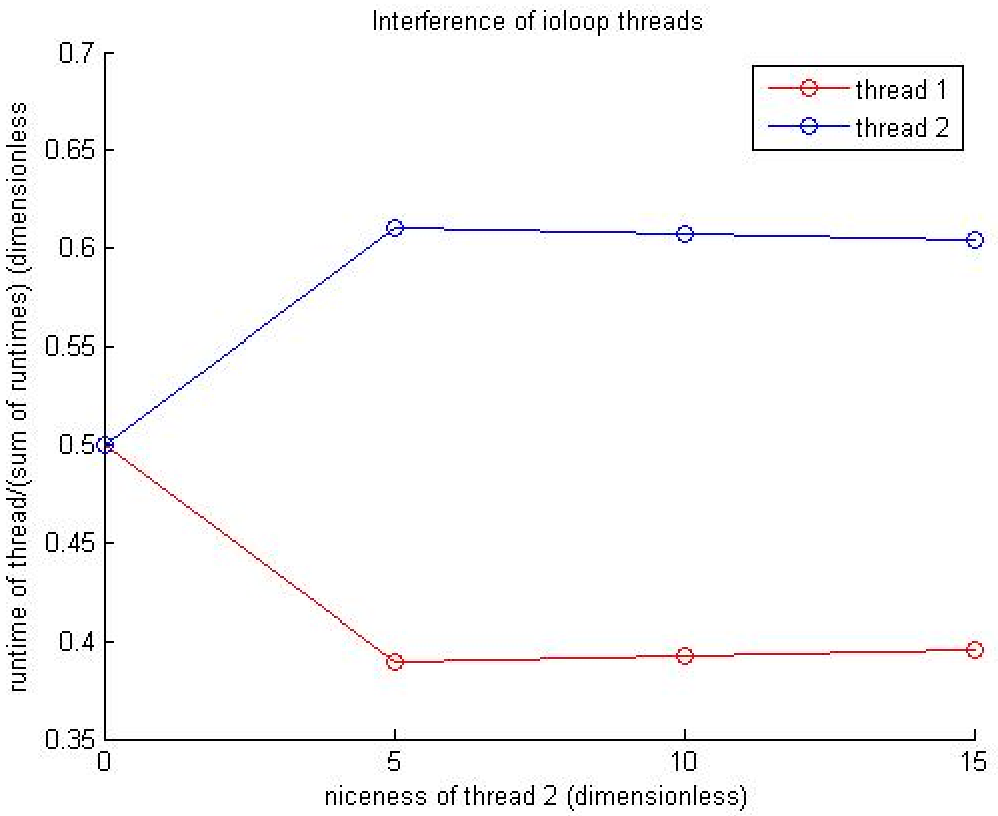

Figure 8: The thread with a lower niceness (thread 1) received a higher priority and thus took a shorter amount of time to run. Finally, a result that makes sense!

As it can be seen, when both threads had the same niceness (when thread 2 had a niceness of 0), each thread took the same amount of time to run. When thread 2 was set to higher values of niceness, and thus prioritized less, it required more time to run, since the hardware focused more power focusing on thread 1. This result is consistent with our model of niceness. Of course, increasing the niceness of thread 2 could only improve the speed of thread 1 up to a point, after which the time it takes to read/write to the disk limited how much more efficient it could be.