Disksort Project
Final Report

Trent A. Fisher
Gregg A. Harris

CS432 – Dan Scott
Portland State University
Computer Science Department

1. Introduction

The purpose of this project is to learn how to write, implement, and test a disk scheduling algorithms for a bare bones Xenix configuration. We had to learn the system booting process using the supplied disk scheduler, this is so we are ensured of a working algorithm while booting the system and to gain confidence. After gaining experience we had to implement a supposed better and worse algorithm. All the algorithms are to be tested with a benchmark program supplied by the sysadmin that should give the algorithms a good workout. To implement an algorithm it first must be written and test compiled for bugs. When the algorithm is ready, it is transferred to a floppy and moved to the test machine, also called a crash box because of the frequency of times it crashes from the kernel hacks. On the test machine the algorithm is compiled into the file *xenix.dsa*. The test machine is then brought down and rebooted with the new disk scheduler. Usually it becomes evident during the booting process if the algorithm works because the kernel uses the scheduler to boot itself. Once the machine is up the algorithm is tested with the benchmark.

2. Benchmarking the Schedulers

The two algorithms we chose to implement and test were First-Come First-Served, and Shortest Seek Time First. Figure 1 shows the benchmark results for the standard kernel. Figure 2 shows the benchmark results for FCFS. Figure 3 shows the benchmark results for SSTF.

From these results the average iterations, execution time, and iterations per second were calculated, these are shown in figure 4. From these, the ratios comparing the averages between the three algorithms were calculated (figure 5).

3. Benchmarking the Benchmark

The benchmark that was used for this project left much to be desired. The results had a very wide variance in comparison to the differences between schedulers.

By hooking a tip line onto a crash-machine's console, we were able to capture the output from my scheduler. There were three pieces of data we gained from this: request distribution, the percent of requests on empty queues, and queue lengths.

3.1. Request Distribution

Figure 6 shows the distribution of requests on the disk during the benchmark. The distribution appears fairly even, the few high points can be explained away by the fact that the find's run *fgrep*'s on every file in the system, so the peak around cylinder 70 might be where it's binary is located.

Figure 1.

Standard Xenix Disksort Benchmarks
17 runs

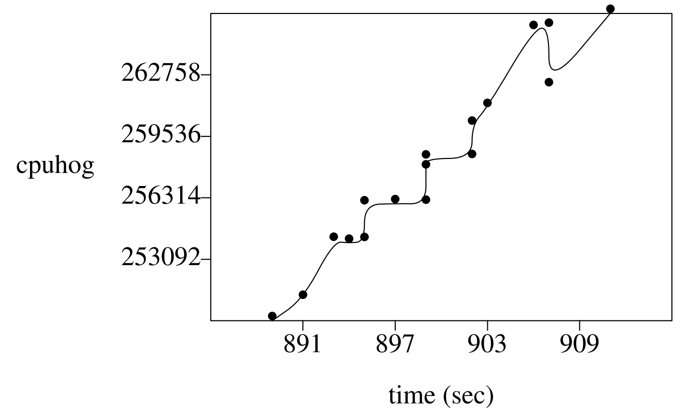


Figure 2.

First-Come First-Served Disksort Benchmarks
12 runs

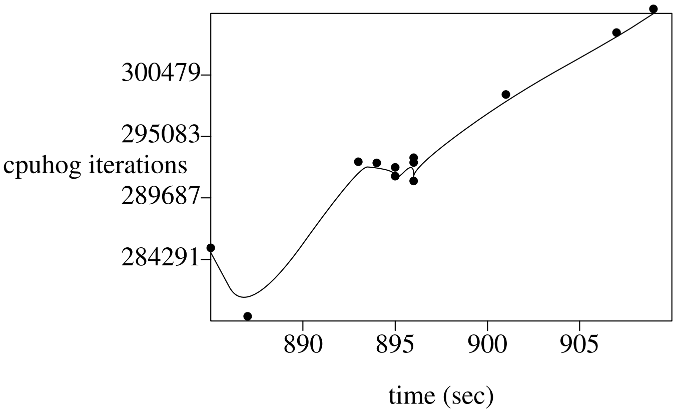


Figure 3.

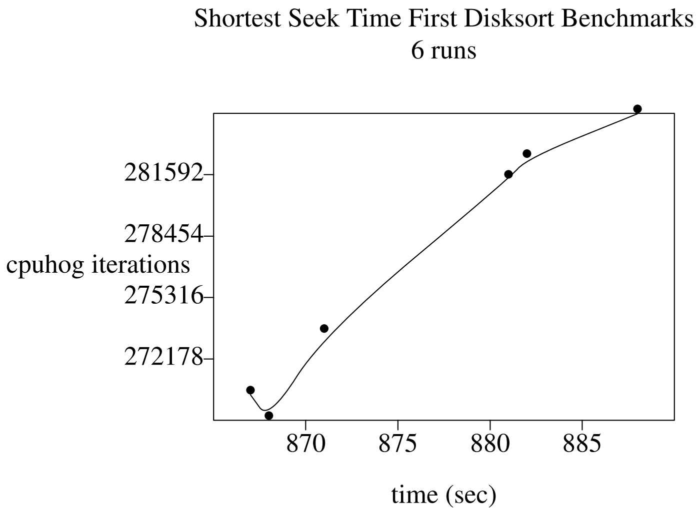


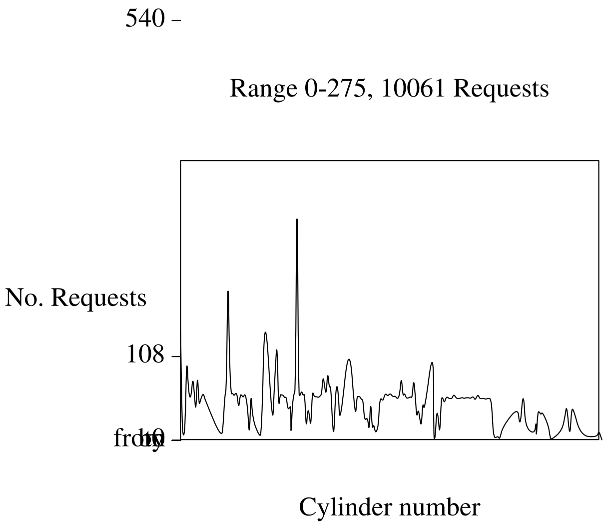
Figure 4. Average Results

	iterations	seconds	iter./sec.
SSTF	276904	876	315
FCFS	292980	896	326
CLOOK	257941	899	286

Figure 5. Ratios

seconds			
	SSTF	FCFS	CLOOK
SSTF	1.0000	0.9776	0.9744
FCFS	1.0228	1.0000	0.9967
CLOOK	1.0263	1.0033	1.0000
iterations			
SSTF	1.0000	0.9451	1.0735
FCFS	1.0581	1.0000	1.1358
CLOOK	0.9315	0.8804	1.0000
iterations/second			
SSTF	1.0000	0.9663	1.1014
FCFS	1.0349	1.0000	1.1399
CLOOK	0.9079	0.8773	1.0000

Figure 6. Disk Request Frequency



3.2. Empty Queues

The benchmark does not appear to create enough processes to keep the disk scheduler busy. To check this we modified *disksort* so that each time it was called, it would print a '-' if the queue was empty on entry, a '+' if not. Of the 9882 requests, 5348 of them were on empty queues (55%).

3.3. Queue Lengths

Then, to investigate the how much of a backlog of requests there were we modified *disksort* to print the queue length upon each entry. Figure 7 is a histogram showing the frequency of requests on various queue lengths. Keep in mind that calculating the queue length could, in the worst case, double the time *disksort* takes; even so there is still a noticable majority of requests on an empty queue. Again this appears to confirm the fact that there are not enough processes being run by the benchmark.

4. Conclusion

According to our calculations SSTF performed 3% better than the others in real time, but performed about 5% worse in iterations per second signifying that it uses more cpu time. FCFS gave the best performance in respect to iterations per second; however, the time it took was much longer, offsetting the better performance of the cpu time. The perfomance of SSTF ignores the possibility of starvation.

However, these comparisons of the performance of the algorithms don't mean a thing, since the benchmark is unsatifactory in determining the actual performance of the algorithms.

Figure 7. Frequency of Various Queue lengths