Tyler Kobil - tak7229
Maximilian Christ - mmc639
OS Final Project

Introduction:

This document outlines the steps taken to complete the CS-GY 6233 Final Project, as well as relevant data analysis corresponding to those steps. The document will be organized by the parts outlined in the final project description. The github repository used for this project can be found at the following link: https://github.com/tkobil/NYU-Operating-Systems. To run all of these tests, we used a Macbook Air with an M1 chip.

Part 1: Basics

We wrote a program to handle disk reads and disk writes initially. This program is called part_one_run.c, and has been uploaded to our github repository. This is NOT the same program that should be used during the raw program performance competition, as this program makes no use of threading. There is a shell script, build.sh, that can be run to compile both this program, as well as the optimal run.c that incorporates multithreading. To run this program, use the following commands:

```
./build.sh
./part_one <filename> [-r|-w] <block_size> <block_count>
```

This program WILL NOT output the xor of all 4-byte integers in a file. The program that will be described in part 6, run.c, will accomplish this, and is the program that should be used in competition.

Part 2: Measurement

Using our basic program from part 1, we attempted to find optimal file sizes that took between 5 and 15 seconds to read. Rather than run a manual analysis, however, we developed a python script, find_file_size.py, to run various test cases defined in a .json file, and output a .csv file with the results. This script can also be found in our github repository.

The python script was continuously used throughout the length of this project to run various test cases and gather data. At this stage in the project, however, we just used it to call the C program developed in Part 1 to first write a file of a particular size, then read it in with a default block size. The results are shown in Figure 2a.

Tyler Kobil - tak7229
Maximilian Christ - mmc639
OS Final Project

File Size (bytes)	File Name	Block Size (bytes)	Read Time (seconds)
100,000	one_hundred_k_input.txt	512	0.004314899445
1,000,000	one_mil_input.txt	512	0.004384040833
10,000,000	ten_mil_input.txt	512	0.0223107338
100,000,000	one_hundred_mil_input.txt	512	0.06274271011
1,000,000,000	one_bil_input.txt	512	0.6708388329
10,000,000,000	ten_bil_input.txt	512	7.678215027
20,000,000,000	twenty_bil_input.txt	512	16.85146308
30,000,000,000	thirty_bil_input.txt	512	20.76283407
40,000,000,000	fourty_bil_input.txt	512	30.22432828
50,000,000,000	fifty_bil_input.txt	512	54.0186739

Figure 2a: Optimal File Size Results - Data

Using the above data, the graph in Figure 2b was created.

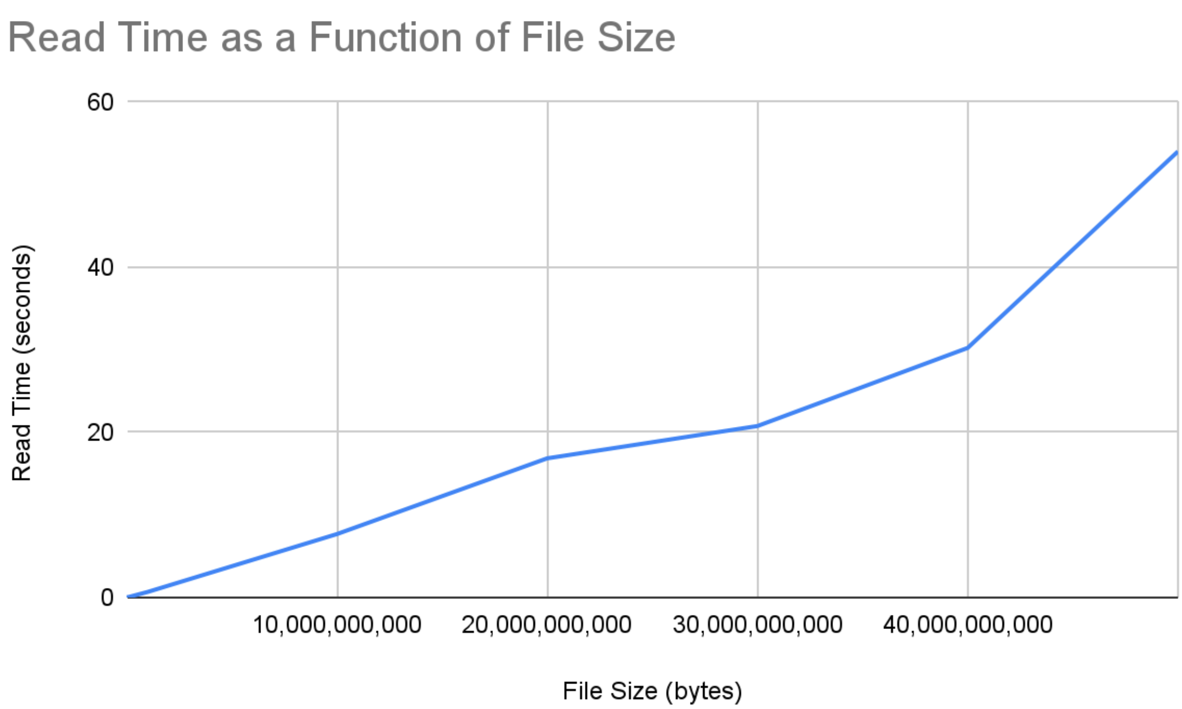


Figure 2b: Optimal File Size Results - Graph

Tyler Kobil - tak7229
Maximilian Christ - mmc639
OS Final Project

Using the data from Figure 2a and 2b, we are able to conclude that optimal file sizes would be somewhere between 10 and 20Gb.

Extra Credit: Learn about the dd program in Linux and see how your program's performance compares to it!

The dd program in linux is a way to copy, and reformat files. It can read and write from and to files, and the block size can be set via operands. For these reasons, it is commonly used as a benchmarking tool for determining the read and write performance of a given drive. Two slightly modified python scripts were written in order to compare the performance of our read and write script, to the performance of dd. Since our script is set up to read and/or write an entire file start to finish, and not alternate between the two functions as in the case of copying a file, we decided to mimic the behavior of dd in our own program by simply writing an entire file, and then reading the entire file. We then timed the combined time of these actions in our modified python script ("find_file_size_read_write_time.py"). In order to test the performance of dd, we created the same starting files using our custom write function, but instead of using our read function, we would call
`dd if={input_filename} of={output_filename} bs={block_size} count={block_count}` using python's os.system method. This reads the input file we created previously (and is the same file used for testing our own functions), and writes it to an output file. We timed the duration of this function for various file sizes and the data can be seen below. The file we used to do this was called ("find_file_size_dd.py").

Consecutive Read and Write Times					
size	**name**	**block_size**	**block_count**	**mode**	**time_no_cache**
100,000	one_hundred_k_input.txt	512	1	r	0.03222203255
1,000,000	one_mil_input.txt	512	1	r	0.04275512695
10,000,000	ten_mil_input.txt	512	1	r	0.08453297615
100,000,000	one_hundred_mil_input.txt	512	1	r	0.3967530727
1,000,000,000	one_bil_input.txt	512	1	r	3.827342987
10,000,000,000	ten_bil_input.txt	512	1	r	38.82843804
15,000,000,000	ten_bil_input.txt	512	1	r	57.25338078
20,000,000,000	twenty_bil_input.txt	512	1	r	76.19207978
30,000,000,000	thirty_bil_input.txt	512	1	r	113.9930279
40,000,000,000	fourty_bil_input.txt	512	1	r	152.5701349
50,000,000,000	fifty_bil_input.txt	512	1	r	199.6583028

Figure 2c: Extra Credit: Consecutive Write+Read Performance

Tyler Kobil - tak7229
Maximilian Christ - mmc639
OS Final Project

dd Run Times					
size	**name**	**block_size**	**block_count**	**mode**	**time_no_cache**
100,000	one_hundred_k_input.txt	512	1	r	0.01590108871
1,000,000	one_mil_input.txt	512	1	r	0.0448012352
10,000,000	ten_mil_input.txt	512	1	r	0.12910676
100,000,000	one_hundred_mil_input.txt	512	1	r	0.9526777267
1,000,000,000	one_bil_input.txt	512	1	r	9.017789125
10,000,000,000	ten_bil_input.txt	512	1	r	87.94570303
15,000,000,000	ten_bil_input.txt	512	1	r	129.0240531
20,000,000,000	twenty_bil_input.txt	512	1	r	168.7609239
30,000,000,000	thirty_bil_input.txt	512	1	r	289.5005939
40,000,000,000	fourty_bil_input.txt	512	1	r	369.8240561
50,000,000,000	fifty_bil_input.txt	512	1	r	464.4587269

Figure 2d: Extra Credit: dd Performance

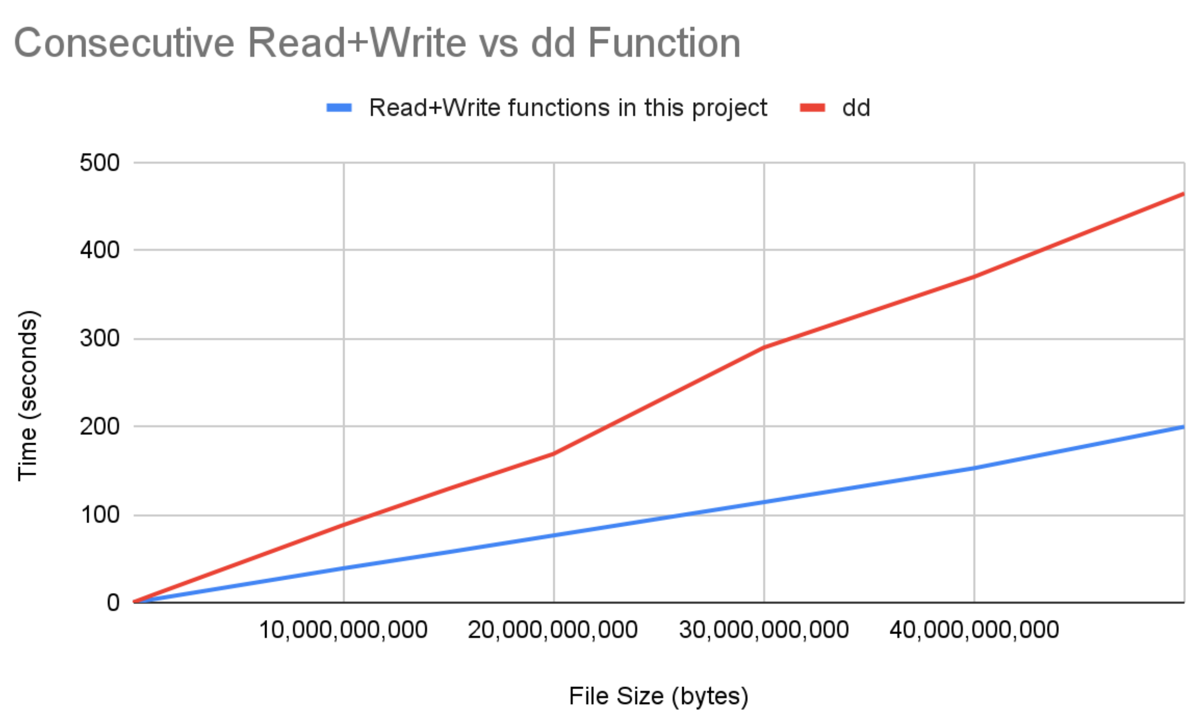


Figure 2e: Extra Credit: Consecutive Read+Write vs dd Performance

Tyler Kobil - tak7229
Maximilian Christ - mmc639
OS Final Project

Therefore, figure 2e shows that our script consistently performs better than the linux dd program, for most file sizes. The dd program has the ability to do much more than is possible than our program. For example, it can perform many different transformations on the data while it copies it. This most likely adds extra bulk, and slows down the code. Also, if our code was written to read and then write block by block to exactly mimic the "copy" operation, there is potential for a small performance hit. Approximating the "copy" operation to a basic write file and read file action-pair is close enough for our purposes without modifying our code too greatly. Please note, this extra credit assignment was run on a 2018 Intel Macbook Pro with an APPLE SSD AP0512M, which is different from many of the other data in this project.

References used for the Extra Credit Relating to dd:
https://man7.org/linux/man-pages/man1/dd.1.html
https://en.wikipedia.org/wiki/Dd_(Unix)

Extra Credit: Learn about Google Benchmark — See if you can use it.

Before beginning to implement Google Benchmark, we researched the product to see if it could be useful. Essentially, Benchmark is a library in C++ that provides "benchmarks," for programs that a user writes. In other words, it provides timing data, CPU iterations, and other performance statistics of a given program. It can be included like any other normal library, and is rather simple to implement. Before running the software, Cmake needs to be installed on the local machine (https://cmake.org/download/). All of the installation requirements are noted in the README file in the github repository for Benchmark. One change that was needed to our original code, is we needed to change the file extension of part_one_run.c to .cpp, so that it can be compiled as c++. Thankfully, no errors were found when switching to the c++ compiler. Additionally, we removed the main() function, and added the benchmark code for testing the read and write functions. As a proof of concept, we tested the read and write functions on a file with a size of 1 billion bytes using a block size of 512 bytes. The edited file is included in the github repository and is named "benchmark_extra_cred.cpp". To compile the benchmark code, the user must run the following command:

```
g++ benchmark_extra_cred.cpp -std=c++11 -isystem benchmark/include
-Lbenchmark/build/src -lbenchmark -lpthread -o mybenchmark
```

Then to run the benchmark and get additional information in an output file called results.json, the user must run the following command:

```
./mybenchmark --benchmark_out=results.json
--benchmark_out_format=json
```

Now the console should have given benchmark output for both the read and write commands, and additional information is contained in the json file. For example, the console output can be seen below (Sidenote: the filename "part_one_run.cpp" in the screenshot is not up to date):

```
[Maximilians-MBP:GitHub maximilianchrist$ g++ part_one_run.cpp -std=c++11 -isyste
 m benchmark/include -Lbenchmark/build/src -lbenchmark -lpthread -o mybenchmark
[Maximilians-MBP:GitHub maximilianchrist$ ./mybenchmark --benchmark_out=results.j
 son --benchmark_out_format=json
 2021-11-30T22:38:54-05:00
 Running ./mybenchmark
 Run on (8 X 2300 MHz CPU s)
 CPU Caches:
   L1 Data 32 KiB (x4)
   L1 Instruction 32 KiB (x4)
   L2 Unified 256 KiB (x4)
   L3 Unified 6144 KiB (x1)
 Load Average: 1.98, 1.97, 1.90
 -----------------------------------------------------
 Benchmark            Time             CPU   Iterations
 -----------------------------------------------------
 BM_Write   11211793604 ns   11095062000 ns            1
 BM_Read     1474417618 ns    1439783000 ns            1
 Maximilians-MBP:GitHub maximilianchrist$
```

Figure 2f: Extra Credit: Google Benchmark Console Results

As can be seen in these results, the read time for a file size of one billion bytes using a block size of 512 bytes is ~1.47 seconds, which is very close to the result that was generated on the same machine using our custom python script: ~1.52 seconds. Note: this section was done on a 2018 Intel Macbook Pro, and that is why the value is 1.52 seconds (and not the value found in figure 2a). Python's runtime is always slower than C, and this slightly higher result in our custom python script is probably due to that overhead associated with calling the time functions in python. Unfortunately, this extra credit was done after the rest of the assignment, so we have no need to use Google Benchmark for all of our analysis. However, this experiment shows that Google Benchmark can be used to write automated testing scripts to test performance for our purposes. Additionally, it appears that the values it provides are very similar to the values generated in the rest of the project. In fact, Google Benchmark can even pass in varied inputs to the functions that it tests, so it could be used in place of the python scripts that were written for timing many different test cases in the rest of this project.

Part 3: Raw Performance

To measure raw performance of our program, we decided to see how performance changed with respect to the block size for the read system call. We added more test cases to the python script mentioned in Part 2, and gathered more data. The data is shown below in Figure 3a.

Tyler Kobil - tak7229
Maximilian Christ - mmc639
OS Final Project

File Size (bytes)	File Name	MiB/s - Block Size (256 bytes)	MiB/s - Block Size (512 bytes)	MiB/s - Block Size (1024 Bytes)	MiB/s - Block Size (2048 Bytes)
100,000	one_hundred_k_input.txt	4.34386321	16.13690366	5.341021266	9.572102789
1,000,000	one_mil_input.txt	141.6850995	321.8465316	143.5962888	224.522456
10,000,000	ten_mil_input.txt	498.7222506	1127.288951	948.7658342	2340.310233
100,000,000	one_hundred_mil_input.txt	826.5566773	1549.234304	2891.426996	4355.591555
1,000,000,000	one_bil_input.txt	854.6109377	1334.071674	2700.987457	4741.646478
10,000,000,000	ten_bil_input.txt	801.0211994	1435.339887	2211.981256	2630.410357
15,000,000,000	fiftenn_bil_input.txt	808.1577035	1442.399936	2368.265242	2866.210952
20,000,000,000	twenty_bil_input.txt	812.036663	1493.23379	2423.041162	2876.319576
30,000,000,000	thirty_bil_input.txt	800.2306166	1455.472502	2518.741364	3054.206933
40,000,000,000	fourty_bil_input.txt	819.1099859	1494.985088	2464.980776	3118.813285
50,000,000,000	fifty_bil_input.txt	806.0584915	1389.314595	2456.227938	3134.547242

Figure 3a: Performance vs Block Size - Data

Using the data from Figure 3a, we created a graph representing Performance (MiB/s) as a function of block size. This is shown in Figure 3b.

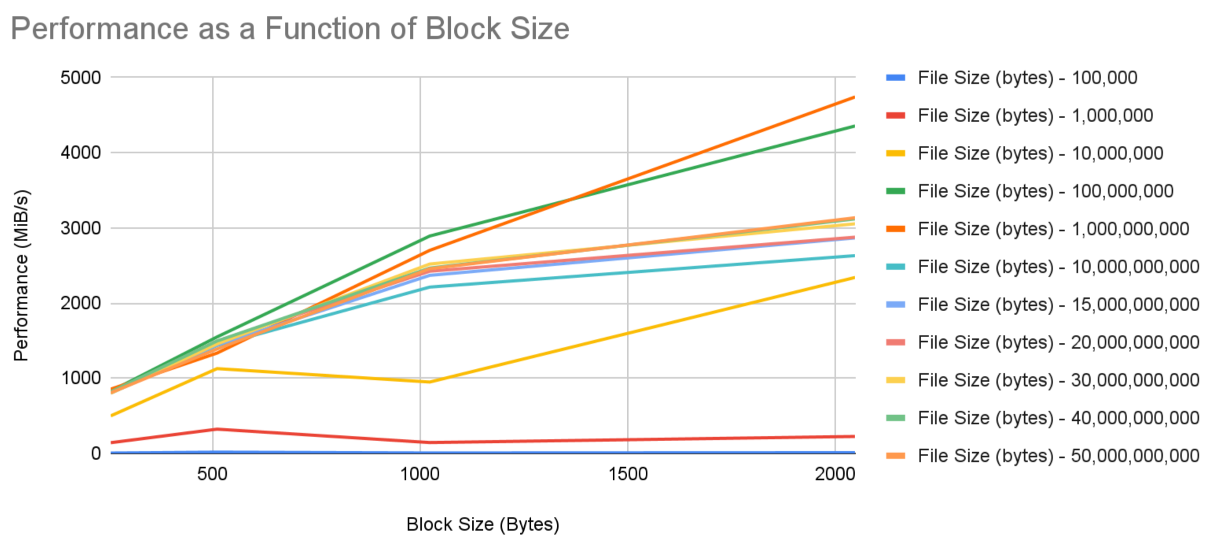


Figure 3b: Performance vs Block Size - Graph

Tyler Kobil - tak7229
Maximilian Christ - mmc639
OS Final Project

As we can see in Figure 3b, regardless of the file size, performance seems to increase with block size. This is to be expected, considering the smaller the block size, the more system calls that need to be made.

Part 4: Caching

Now, to measure the effect of disk caching, we re-ran the analysis from Part 3, however this time we repeated each test case twice, clearing the disk cache in between reads. The data gathered is shown below in Figure 4a.

Block Size	Performance (MiB/s) No Cache - File Size (bytes) 10,000,000,000	Performance (MiB/s) w/ Cache - File Size (bytes) 10,000,000,000	Performance (MiB/s) No Cache - File Size (bytes) 30,000,000,000	Performance (MiB/s) w/ Cache - File Size (bytes) 30,000,000,000	Performance (MiB/s) No Cache - File Size (bytes) 50,000,000,000	Performance (MiB/s) w/ Cache - File Size (bytes) 50,000,000,000
256	801.0211994	818.3594074	800.2306166	810.5014149	806.0584915	821.4748242
512	1435.339887	1505.901655	1455.472502	1491.949464	1389.314595	1464.060601
1024	2211.981256	2493.354657	2518.741364	2579.194831	2456.227938	2538.139745
2048	2630.410357	3102.80144	3054.206933	3223.454101	3134.547242	3069.123536

Figure 4a: Effects of Disk Caching - Data

Using the data in Figure 4a, we graphed the relationship between Performance and Block Size for cache and no cache scenarios in Figures 4b-4d.

Tyler Kobil - tak7229
Maximilian Christ - mmc639
OS Final Project

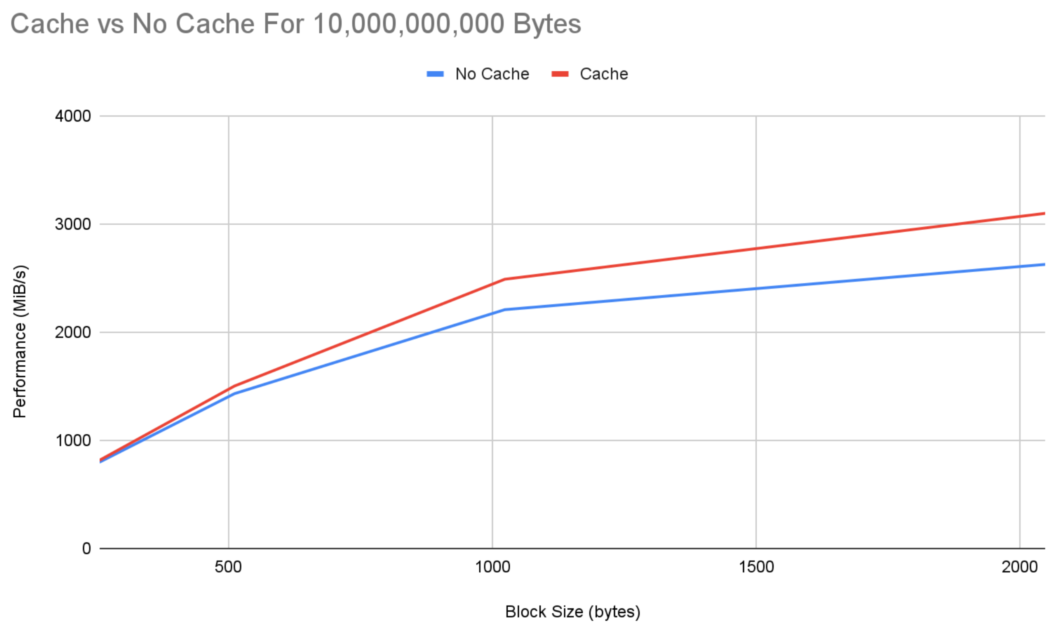


Figure 4b: Effects of Disk Caching - 10Gb

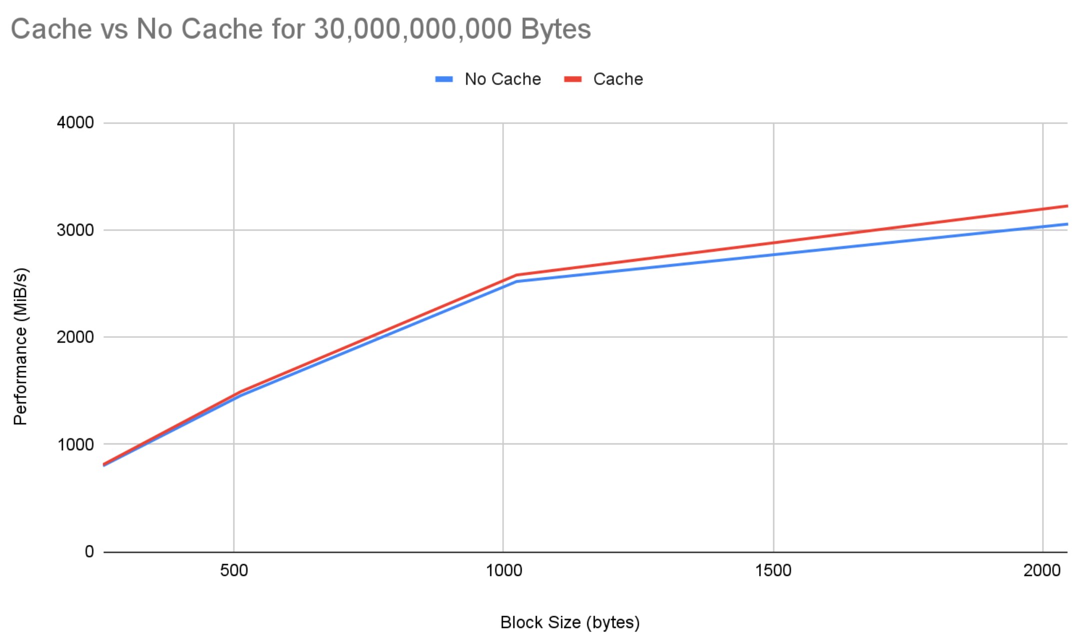


Figure 4c: Effects of Disk Caching - 30Gb

Tyler Kobil - tak7229
Maximilian Christ - mmc639
OS Final Project

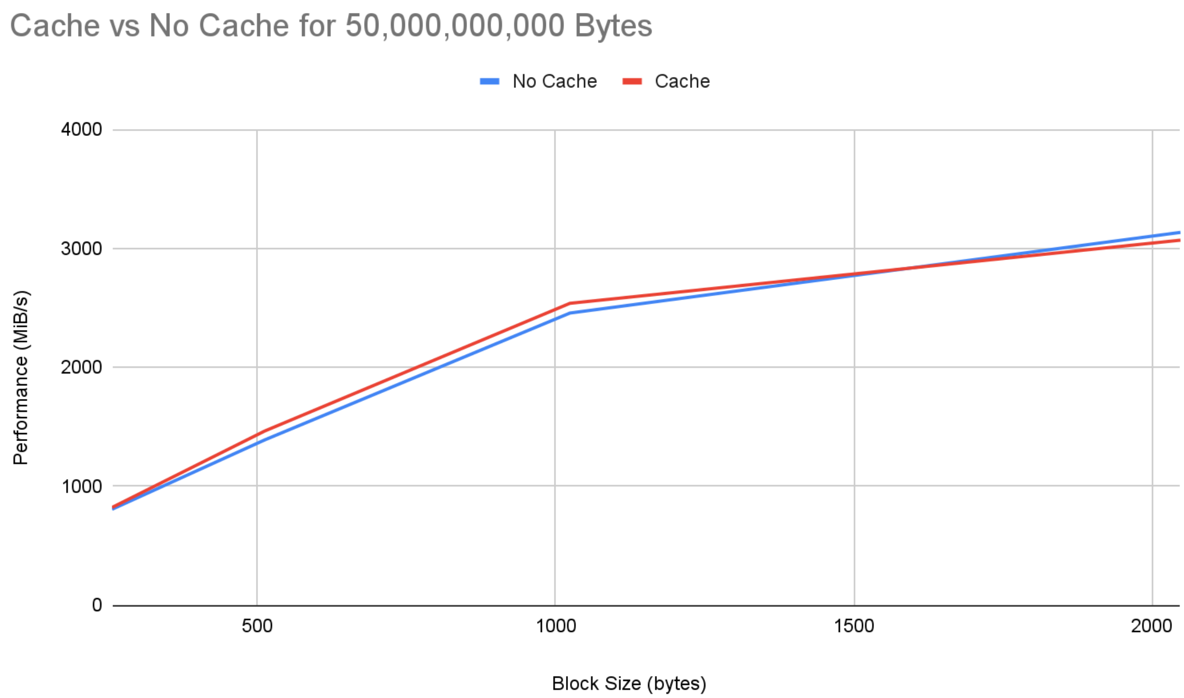


Figure 4d: Effects of Disk Caching - 50Gb

As seen in Figures 4b-4d, Disk caching increases performance slightly, regardless of the block size or file size.

Extra credit: On Linux there is a way to clear the disk caches without rebooting your machine. E.g. `sudo sh -c "/usr/bin/echo 3 > /proc/sys/vm/drop_caches"` Why "3"? Read up on it and explain.

The system file /proc/sys/vm/drop_caches in Linux is a kernel file that can be written to, in order to clear caches. There are 3 values that can be written to it in order to accomplish different things (1, 2, or 3). Writing 1 to the file clears pagecache. Writing 2 to the file clears what are called reclaimable slab objects. These include objects such as dentries and inodes. Writing 3 to the file clears both pagecache, and reclaimable slab objects. All of this information can be found in the Linux Kernel documentation https://www.kernel.org/doc/html/latest/admin-guide/sysctl/vm.html?highlight=drop_caches.

Tyler Kobil - tak7229
Maximilian Christ - mmc639
OS Final Project

Part 5: System Calls

To measure system calls more thoroughly, we set block size to 1 byte for various file sizes, and re-ran our analysis. The results are shown in Figure 5a. We generated data to measure the number of system calls per second (B/s), as well as the performance in MiB/s.

File Size (bytes)	File Name	Block Size	Time (sec)	System Calls Per Second	Performance (MiB/s)
100000	one_hundred_k_input.txt	1	0.0460562706	2171257.001	2.171257001
1000000	one_mil_input.txt	1	0.2858371735	3498495.272	3.498495272
10000000	ten_mil_input.txt	1	2.83823204	3523320.101	3.523320101
100000000	one_hundred_mil_input.txt	1	28.43798518	3516423.522	3.516423522
1000000000	one_bil_input.txt	1	285.4529781	3503203.948	3.503203948

Figure 5a: Measuring Performance of System Calls with Read

The table in Figure 5a shows that we were able to consistently make around 3.5 million system calls per second. It is no surprise that performance takes a large hit when we reduce block size down to 1 byte. This is due to the number of system calls that need to be made. When we use a system call that does even less real work, like lseek, we can isolate the cost of just the system call closer to reality. This is shown in Figure 5b. Using lseek, we can see that we can consistently make somewhere around 7 million system calls per second. Again, it is no surprise that read takes longer than lseek, because lseek does very little actual work and spends most of the time trapping into the kernel.

File Size (bytes)	File Name	Block Size (bytes)	Time (seconds)	System Calls Per Second	Performance (MiB/s)
100,000	one_hundred_k_input.txt	1	0.03680896759	2716729.279	2.716729279
1,000,000	one_mil_input.txt	1	0.1461839676	6840695.437	6.840695437
10,000,000	ten_mil_input.txt	1	1.428714991	6999296.617	6.999296617
100,000,000	one_hundred_mil_input.txt	1	14.14877176	7067751.299	7.067751299
1,000,000,000	one_bil_input.txt	1	140.9443247	7095000.114	7.095000114

Figure 5b: Measuring Performance of System Calls with lseek

Tyler Kobil - tak7229
Maximilian Christ - mmc639
OS Final Project

Part 6: Raw Performance

To optimize for raw performance, we tested our program while varying block size and thread count for various file sizes. Initially, we expected to see performance increase with both block size and number of threads. However, this did not prove to be true.

Naively, we assumed multithreading disk IO would improve performance. However, this is not the case in reality. Due to disk geometry, a disk can only search for one piece of data at a time. Now, when we use multithreading, we are basically creating competition over one shared resource - the disk. So, in reality, we introduce more blocking, as well as the overhead of creating and managing the threads. What we initially thought was a feature turned out to slow performance down.

In the below graphs, we can clearly see that performance increases as we increase thread count, no matter the block size or the file size. For optimal performance, we would recommend a large block size, because the more block sizes we use, the less system calls we will have to make. The optimal parameters would be 4096 byte block size single-threaded. Both cached and non-cached numbers are shown in the table below for various block sizes and thread counts.

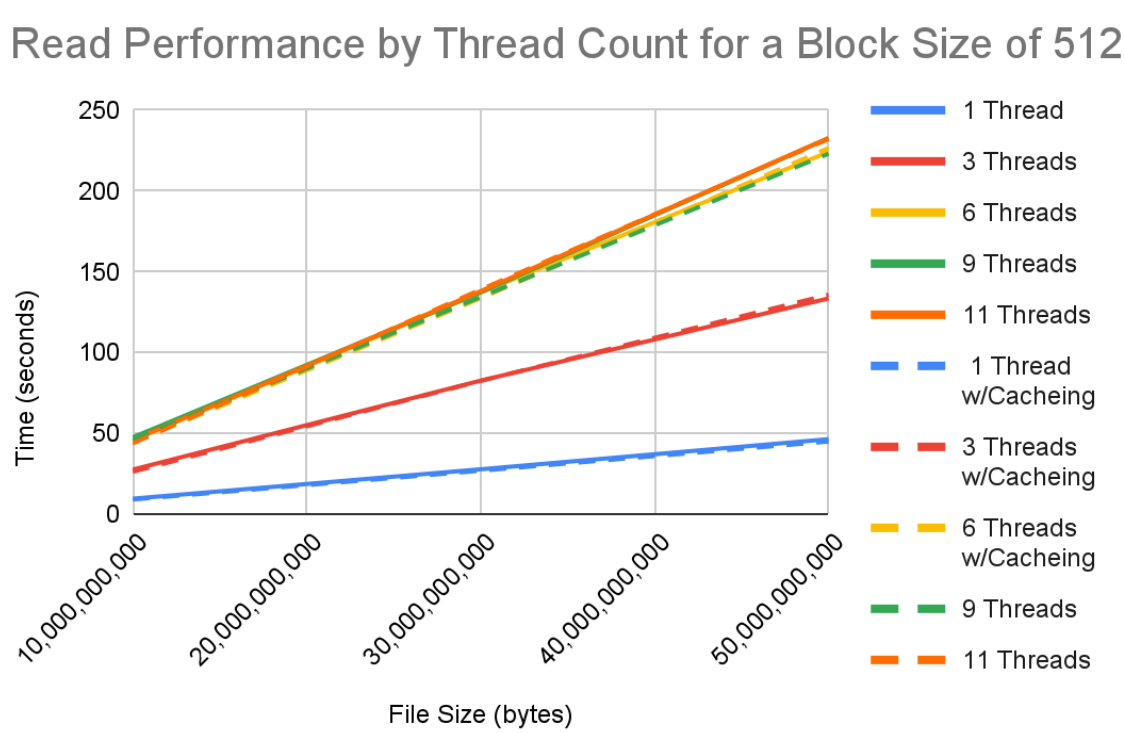
To run the code used in this section, reference run.c. The below commands can be used:

./build.sh
./run <filename> [-r|-w] <block_size> <block_count> <num_threads>

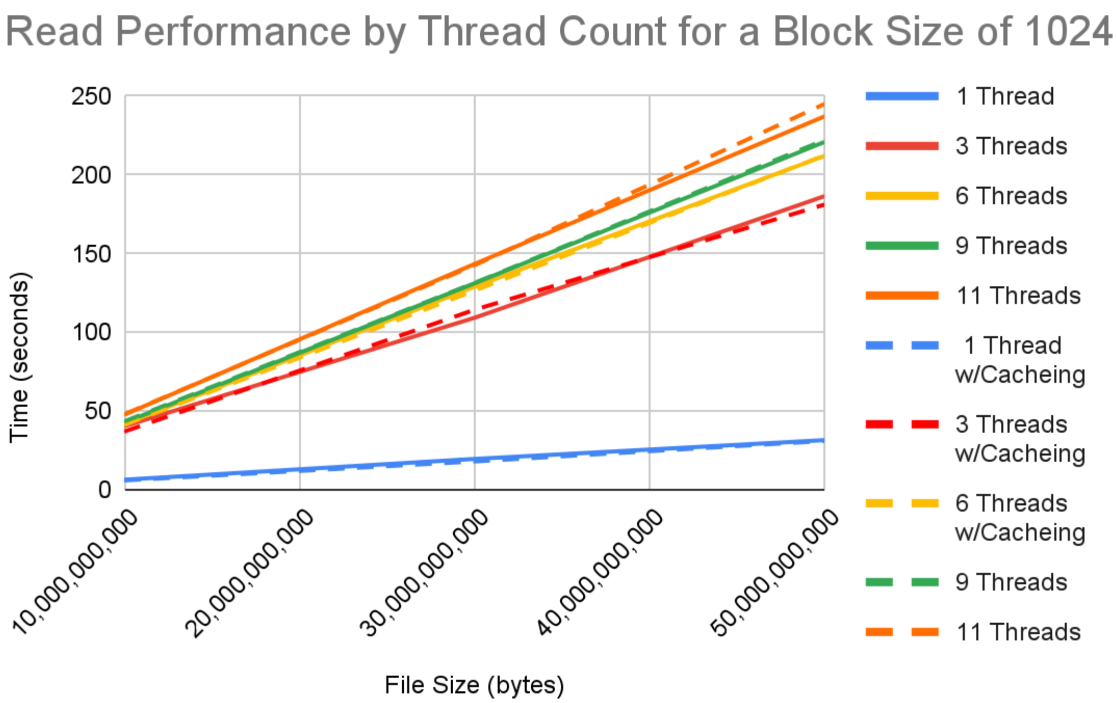
block_size	num_threads	10Gb_no_cache	10Gb_cache	30Gb_no_cache	30Gb_cache	50Gb_no_cache	50Gb_cache
1024	1	1.508660078	1.201308727	1.373629093	1.366541862	1.384665966	1.242297888
1024	11	7.556052446	3.067155838	9.047149658	2.490560055	7.59067297	3.348144054
1024	3	6.016493797	1.458653927	7.273690224	4.834891796	6.171352148	1.602003098
1024	6	8.576308966	5.555611134	7.630893946	2.083016872	7.090167761	5.078955889
1024	9	8.664182186	4.092171907	7.274999142	4.366017103	8.746085167	6.268676996
2048	1	1.059423923	1.026261091	1.125800133	0.9594509602	1.100738049	1.130785942
2048	11	6.358307123	2.995685101	5.348116875	3.135405064	5.232490063	2.273192883
2048	3	6.472846985	3.766714096	7.56882	3.477576017	6.28110218	3.833611965
2048	6	8.856578827	5.777282	7.193297863	3.27462697	8.328381062	3.543490887
2048	9	4.992516041	4.457742929	6.232195139	5.725086927	5.082195759	4.635998964
3072	1	1.02342701	0.979170084	1.074448109	0.9465122223	0.9978330135	1.033562899
3072	11	5.564842939	1.558996916	7.336303949	3.222656012	5.966611862	1.956187963
3072	3	4.18090868	0.7142138481	5.291666031	2.339491844	4.517238855	1.469753265
3072	6	7.76450181	7.331388712	5.583913088	5.613327026	6.873503923	6.791048288
3072	9	5.843242884	1.359703064	7.304310083	0.6982369423	5.610928059	5.094222069
4096	1	0.9142131805	0.961663961	1.051272869	1.05339098	0.9204080105	0.9904887676

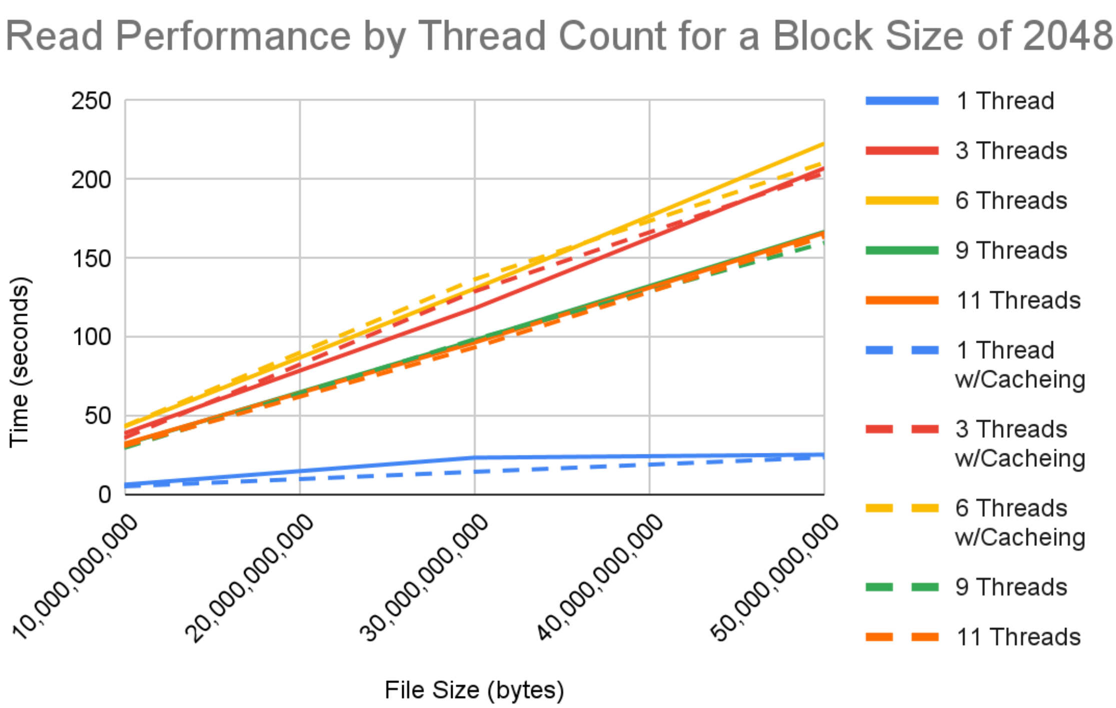
Tyler Kobil - tak7229
Maximilian Christ - mmc639
OS Final Project

			4				
4096	11	5.861382246	2.24910593	4.691130877	2.280465841	5.702045918	2.483864069
4096	3	6.432970047	4.167627096	8.174990654	5.336124897	7.447544098	4.008857727
4096	6	5.815917969	5.059044838	4.697979927	3.753090858	6.102114916	2.549948692
4096	9	4.538342953	3.893323898	5.681116104	5.208180189	4.668573856	2.465421677
512	1	2.306886196	1.921280861	2.197884083	1.990294933	2.044486046	2.032030344
512	11	7.923131943	4.733029842	9.663500071	3.969162941	8.003093004	6.628930092
512	3	6.177213907	2.741388083	5.185475826	2.867855787	5.267148972	3.874300718
512	6	9.287672043	4.342202902	7.977171183	4.94916296	9.250619173	5.379842758
512	9	7.775403738	2.746369123	8.042077065	3.974967957	9.450071096	4.101987839



Tyler Kobil - tak7229
Maximilian Christ - mmc639
OS Final Project





Tyler Kobil - tak7229
Maximilian Christ - mmc639
OS Final Project