Our final solution to this project is a culmination of the tests ran at each step of the process. The process we followed to find an efficient solution at each step is outlined in this final report. This report includes graphs, tables, and figures showing runtimes of our implementation of an efficient program using C to open, read, write, and close a file from the disk. This process took into account the use of varying file and block size, block count, caches, system calls, and threads.

For this project we have a build file which creates 3 files for part 1 part 5 and part 6. Our part 1 file is used to run our test scripts which were written in python for parts 1-4. The second run file is for part 5 which has a addition of lseek in the disk read function. Lastly we have fact.c which based on our results from the part6 test script implements a single threaded read program with a block size of 4096.

For part 1 our main function has error checking for the number of arguments. We have two diskread/write functions which execute when called through the python scripts. For part 2 we found through our testing that a block size:1024 block count:3906250 gave us a run time within the threshold of 5-15 seconds.

For part3 based on this we run our scripts with varying block sizes of 512,1024,2048 which gave us the results seen in this report. For part 4 We ran this same script after rebooting and clearing the cache and got better performance when cached.

For part 5 we created a separate program in in which lseek was called in our read function and those results were compared to using the original program from part 1. We saw that using lseek gave us better performance as seen in the charts below.

Lastly for part 6 we ran a number of tests from our python test script and came to the conclusion that a block size of 4096 is ideal. Our final program fast.c has this set by default when it is reading a filename that is passed as a parameter.

First call our ./build.sh file this will compile all 3 of the necessary programs we have written in C. After which you may call our python scripts individually using the following syntax: (python3 partX.py) where “X” represents the script number you would like to call. Each one of our programs have a specific build file name which is called by the scripts.

1.Basics

The goal of this step was to implement an initial program to open, read, write, and close a file from the disk. The main additions to our programmed aimed to:

- Add parameter for the file name;
- Add parameter for how big the file should be (for writing);
- Add a parameter to specify how much to read with a single call (block size);

This program can be executed using the run command with the following format:

```
./run <filename> [-r|-w] <block_size> <block_count>
```

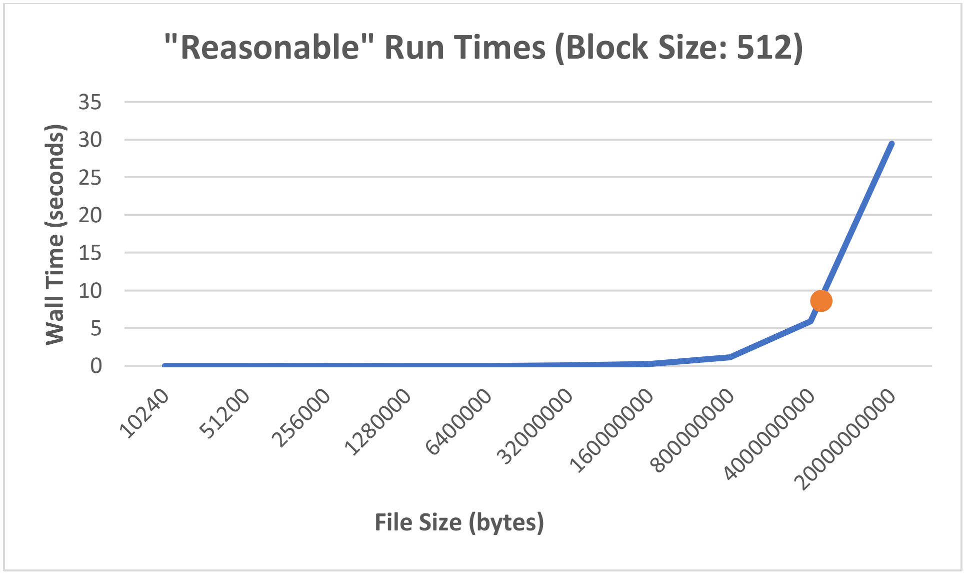
2.Measurement

The objective of this step is to find a file size, given a block size, that causes our program to run in "reasonable time". We measured time in this case using the time() function in python. By calling our program and measuring the time before and after our program ran we were able to deduce the wall time of running our program.

In doing this we wrote a python script that was able to run our "./run" command from inside the file. This file first used the write parameter (-w) to create a file of a specified size (block_size * block_count) which we could then test the runtime of our read function on. With a fixed block size, the python program then looped over and tested the read function on a number of different file sizes, increasing until we found a runtime that was close to reasonable. In this case we tested until we found a time above 5 seconds (we had to increase by a large number since doubling the file size caused our computers to crash before we could get to a reasonable time).

Below is the result of running this program on the block size 512.

File Size (B)	Runtime (s)
10240	0.005507708
51200	0.005400896
256000	0.004961729
1280000	0.006858826
6400000	0.013581038
32000000	0.046358109
160000000	0.221630335
800000000	1.156485796
4000000000	5.907911777
20000000000	29.46912408



As is clear from the graph and table above, the runtime of the read function increases with the increase of our file size. As far as a file size for our read function to run in "reasonable time", our program found an answer after some trial at 5.9 seconds. The "reasonable" file size for our program to run with block size of 512 is 4,000,000,000.

While this may be optimal size, it is also a size that caused some major issues for our hardware

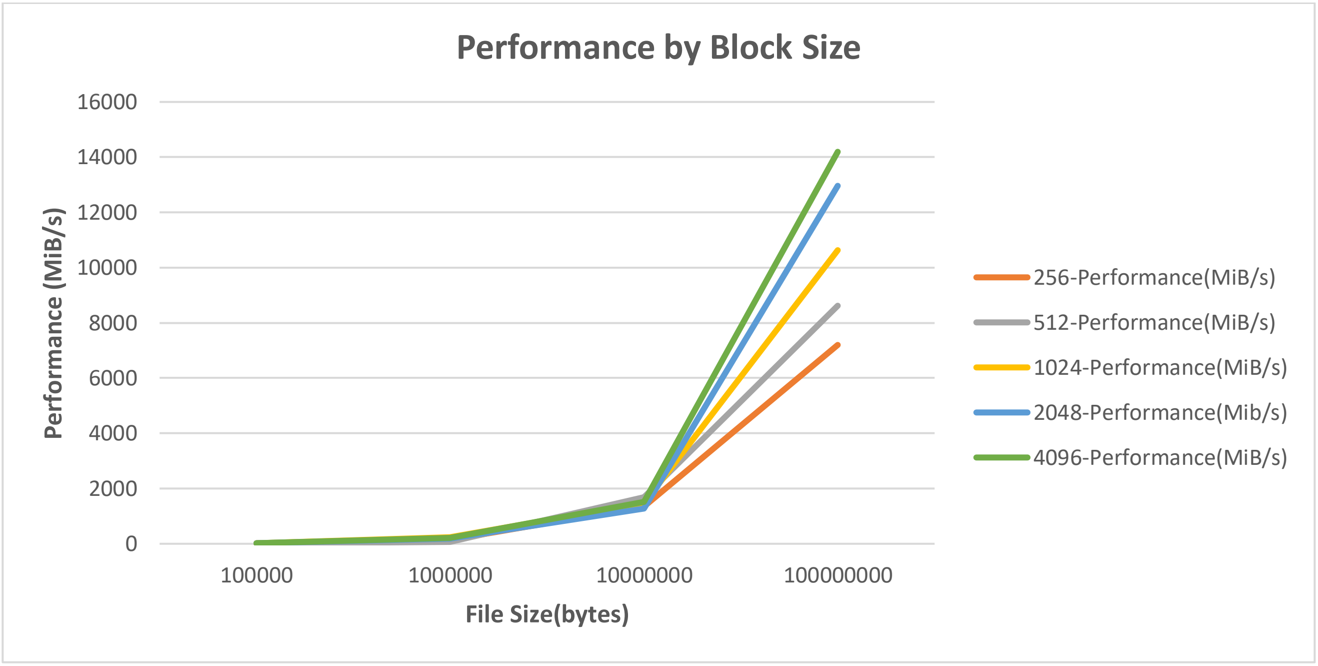
3.Raw Performance

The goal of this step was to find an efficient block size, because our program would undoubtably be using different file sizes. For our measurements in this step we used a metric of performance independent of size, as to be able to properly compare the effect that block size had on the performance of our program. The metric we used was MiB/s (megabytes/second), this translates to a performance of 1,000,000 bytes per second.

Here, we utilized a python file to run our program at varying block sizes, changing the file size after each run through to compare across performance spaces.

Below is a table and graph outlining the results of these tests:

File Size(B)	256-Performance(MiB/s)	512-Performance(MiB/s)	1024-Performance(MiB/s)	2048-Performance(Mib/s)	4096-Performance(MiB/s)
100000	18.22817905	11.58487502	18.75471293	20.23496719	21.05680004
1000000	129.8184407	65.77340087	225.5365919	195.1656042	217.5807439
10000000	1355.230864	1680.881658	1465.668658	1275.523523	1516.982169
100000000	7201.016379	8620.676614	10631.67981	12963.38742	14194.8829



Our results showed that across various file sizes, the performance of our disk read increased with the growth of our block size. This is consistent with our theory coming into this experiment. With the gradual decline in the growth rate of our performance, it is also safe to assume that 4096 is a very efficient block size to run this program, especially if we were to extrapolate further with the file sizes.

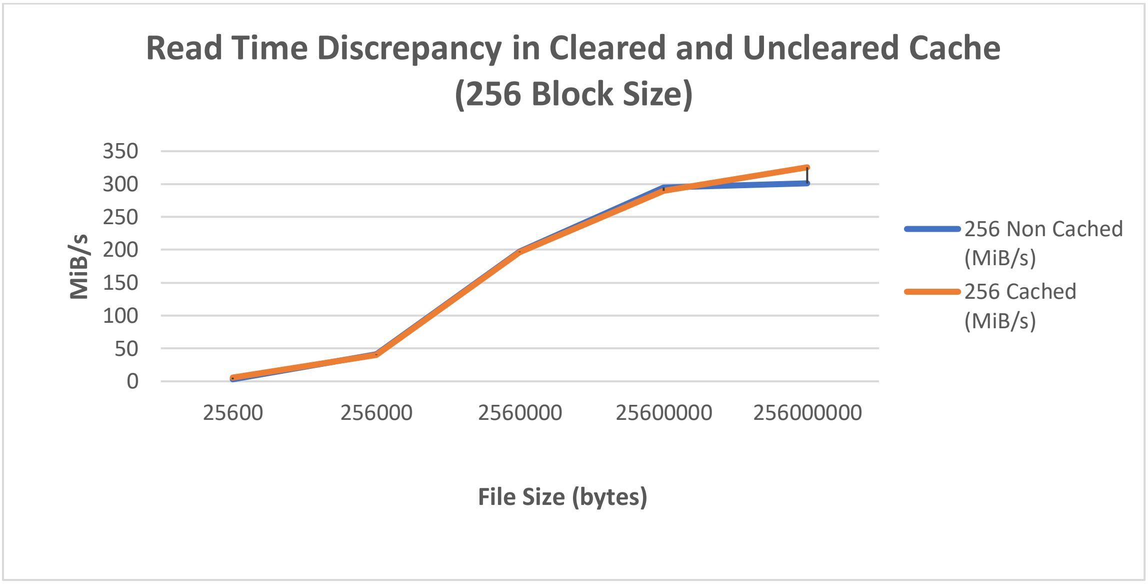
4.Caching

This step involved studying the use of caches on the performance of our program. In order to measure this, we were required to reboot our machines prior to running our initial performance test at each block size. We then ran another test immediately after, this time cached, and we compared our performances at each level.

For each of the block sizes tested, we have displayed a table and graph showing the performance of each test. Each test is measured in MiB/s (megabytes per second) and represents the number of system calls this program makes in a second.

Block Size: 256

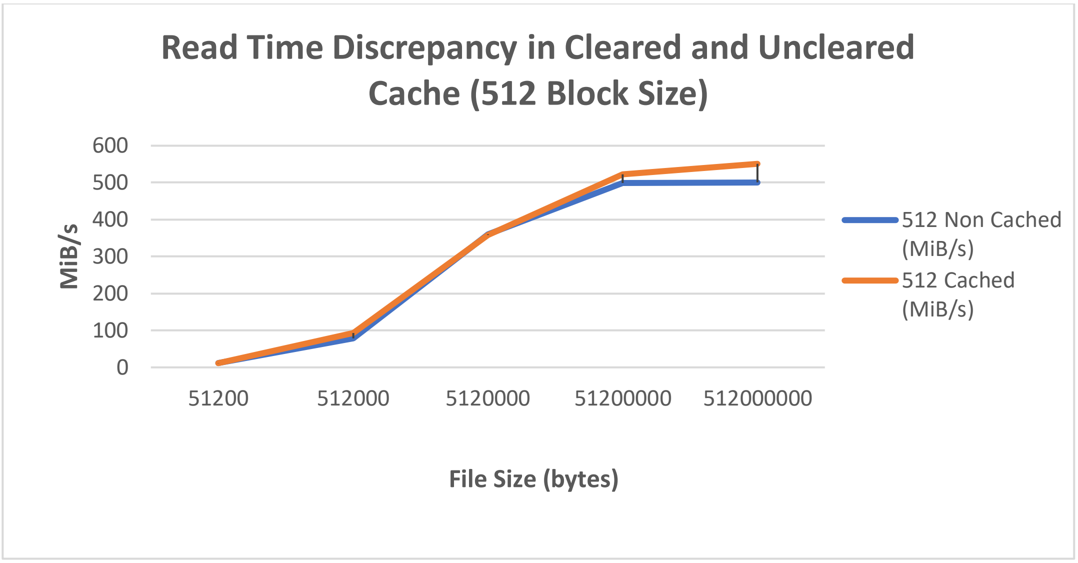
File Size (B)	256 Non Cached (MiB/s)	256 Cached (MiB/s)
25600	3.062234269	5.706536054
256000	40.84998379	40.45596715
2560000	197.789862	196.2283346
25600000	294.6424266	289.8158994
256000000	301.1841921	325.5456927



Block Size: 512

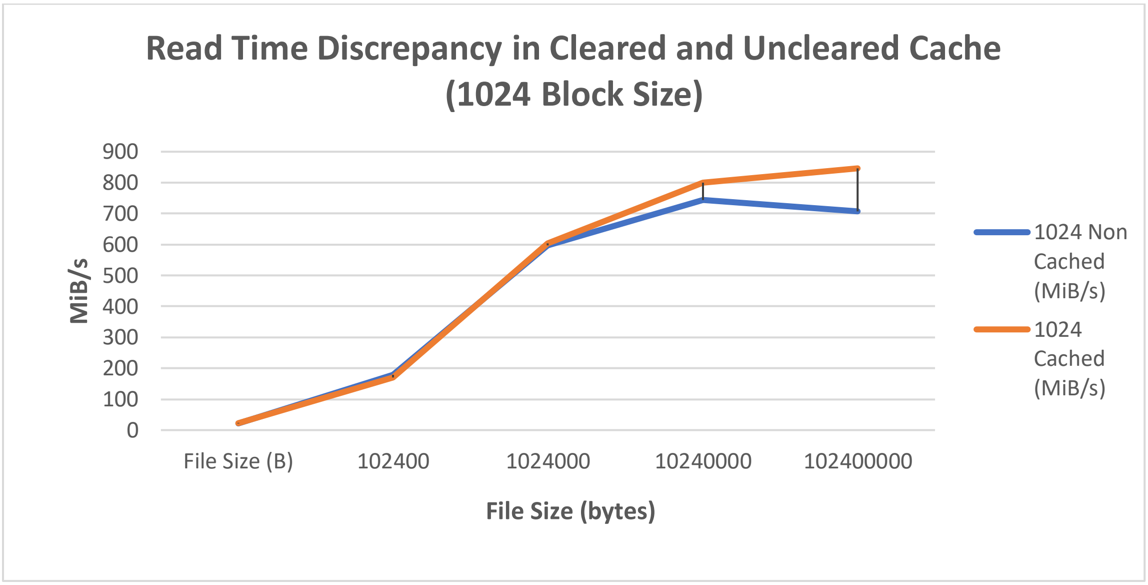
File Size (B)	512 Non Cached (MiB/s)	512 Cached (MiB/s)
51200	11.7278338	11.37016809
512000	78.0732803	93.70701436
5120000	360.8670366	358.2124517
51200000	498.3011567	522.7373996

512000000	500.3836843	551.0194381



Block Size: 1024

File Size (B)	1024 Non Cached (MiB/s)	1024 Cached (MiB/s)
102400	21.643657	22.4221733
1024000	178.7707511	170.0977147
10240000	597.7186729	604.4141987
102400000	744.0303532	801.0189143
1024000000	707.149741	846.3234888



The result of our tests at each blocks size showed that there was a clear time advantage to running this program with an uncleared cache at each level.

We ran section 4 after a full reboot and after using the command “sudo purge” which gave us similar results as this is the command to clear the cache on a Mac.

Extra credit
In the command:

sudo sh -c "/usr/bin/echo 3 > /proc/sys/vm/drop_caches”

The use of the number three in echo is used to indicate that it wants to run all three options that the echo has for clearing caches. Option 1 will clear the page cache only and 2 will clear the dentries and inodes. Option 3 clears all three of these. The difference here is that the page cache contains any memory mappings to blocks on disk, dentries is a data structure that represents a directory, and inode is a data structure that represents a file.

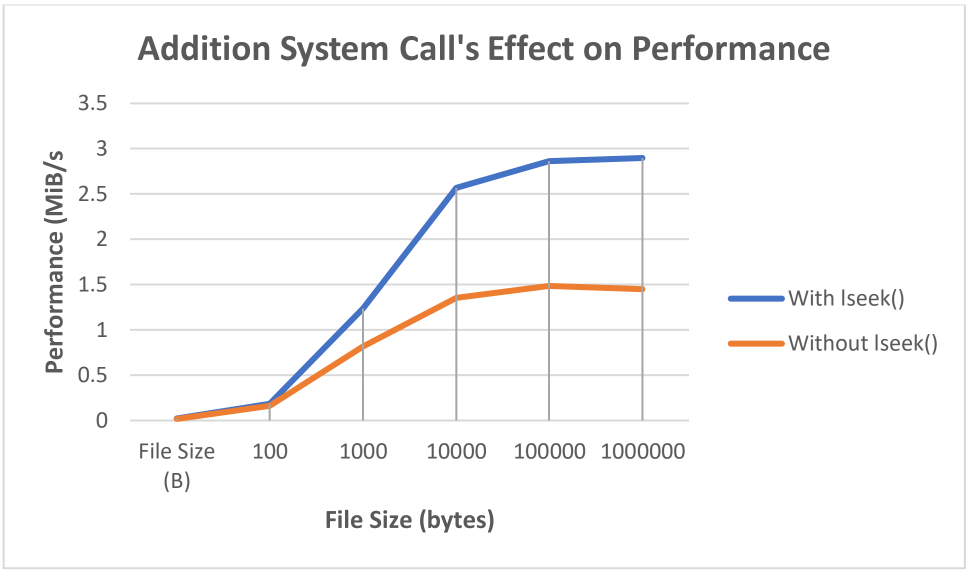
5.System Calls

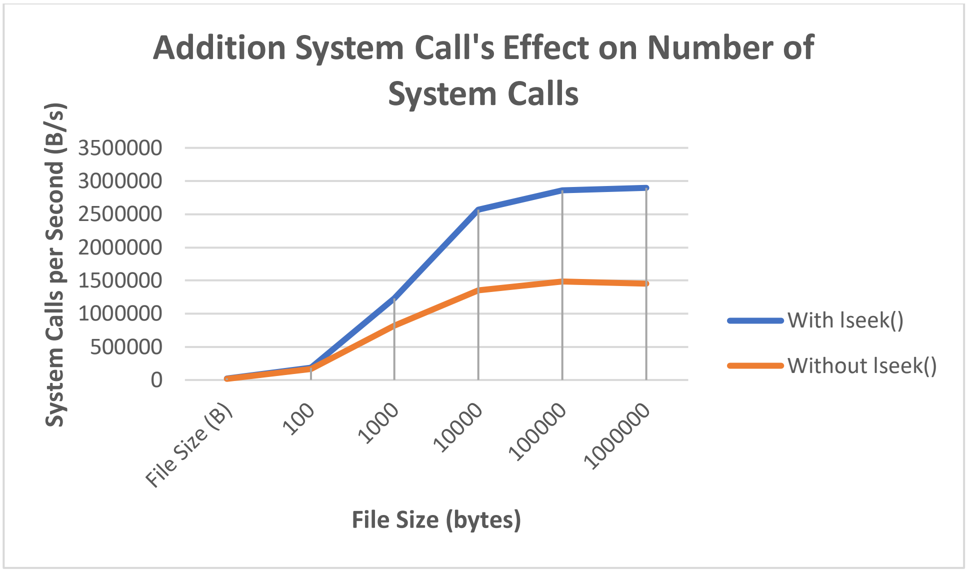
In this step we looked to find what effect system calls had on the performance of our program. In doing this we used the system call lseek to arbitrarily seek in the file to show how it changes our runtimes in both MiB/s (megabytes/second) and B/s (bytes/second), which is the number of system calls made by the program.

The theory here is, since system calls are expensive, the program that does less work and spends most of its’ time trapping in the kernel should have the better performance. To show this we created a similar python script and disk read to what we have been using in the past, but with several additions. Our C file was changed so that instead of using read in our disk read, we are using lseek. In our Python script, we edited our code to output the runtime and performance at six different files sizes (incremented by the degree 10).

Below is a table and two graphs outlining the tests we ran on our function. The performances of our disk read using lseek and and the standard read are shown.

File Size (B)	Sys Calls per Second (B/s)	Performance (MiB/s)	Sys Calls per Second- lseek (B/s)	Performance-lseek (MiB/s)
100	17405.91775	0.017405918	22775.3258	0.022775326
1000	163081.9239	0.163081924	187087.0244	0.187087024
10000	815584.0317	0.815584032	1235799.646	1.235799646
100000	1354810.618	1.354810618	2570700.793	2.570700793
1000000	1485435.535	1.485435535	2862058.636	2.862058636
10000000	1450445.243	1.450445243	2897305.094	2.897305094





As you can tell from the graphs, there begins to be a very noticeable discrepancy in the performance and number of calls respectively as the files get bigger. It is clear that utilizing lseek will give us a better performance, as is consistent with our theory, because it does less work than read.

6.Raw Performance

This step was the culmination of all of the previous steps as well as an attempt to maximize the performance of our disk program. This step included trying variations of our code utilizing the efficient outcomes of our previous tests. We tested our code both on our local machines and in Ubuntu on Anubis to make sure that our results were consistent. In this, we chose to exclude the use of multithreading in our final implementation. This is because our OS uses a scheduler to make sure IO processes are fairly distributed among the threads. This would cause the read head to go back forth between different parts of the disk, slowing down our process with every addition of a thread.

The idea behind our final implementation was simply to maximize the performance of our disk read in terms of megabytes per second. The final implementation of our disk read can be used by running the command:

```
./fast <file_to_read>
```