CS-GY 6233 Final Project

Due December 16, 2021

By Xiao Lin Zhong xz3343 and Mohammed Sujon ms7327

https://www.notion.so/CS-GY-6233-Final-Project-372c172cfda24739a35ff8c352dba099

1. Basics

To execute this part of our project:

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

Our read function allocates `block_size` to the buffer then the file is read to that buffer and XOR is calculated. This is repeated `block_count` times, resulting in `block_size*block_count` being read.

2. Measurement

For this part, we wrote a shell script which takes:

```
./run2.sh <filename> <block_size>
```

that uses a for loop to determine the `block_count` that will give us a read time of 5 - 15 seconds.

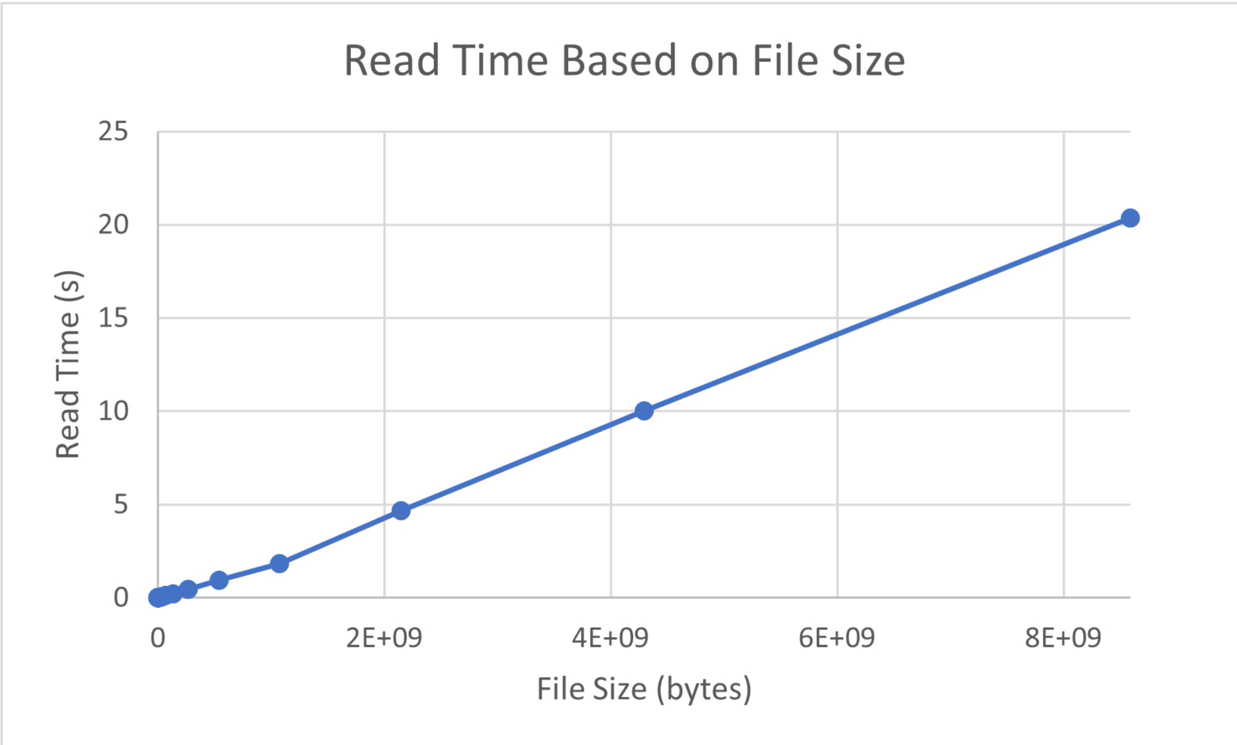
The file we will be using for the rest of this report is named `testfile.txt` of the size `4294967296 bytes = 4GB` with a read time of `10.036775 secs.`

From this experiment, we will keep the `block_count` at `4194304`.

Block Count	Block Size	File Size (bytes)	Read Time (s)
1	1024	1024	0.000046
2	1024	2048	0.000073
4	1024	4096	0.000068
8	1024	8192	0.00006
16	1024	16384	0.000097

32	1024	32768	0.000135
64	1024	65536	0.000183
128	1024	131072	0.000303
256	1024	262144	0.000497
512	1024	524288	0.000978
1024	1024	1048576	0.001754
2048	1024	2097152	0.003822
4096	1024	4194304	0.007343
8192	1024	8388608	0.014939
16384	1024	16777216	0.029871
32768	1024	33554432	0.058561
65536	1024	67108864	0.119701
131072	1024	134217728	0.227195
262144	1024	268435456	0.454915
524288	1024	536870912	0.920756
1048576	1024	1073741824	1.81732
2097152	1024	2147483648	4.674928
4194304	1024	4294967296	10.036775
8388608	1024	8589934592	20.376236

Below is the graph representation of the results above:



Extra Credit:

To compare, we used the `dd` program in Linux:

```
sudo dd if=/home/os/Desktop/CSGY6233_Final_Assignment/testfile.txt of=/dev/null
```

The output of the `dd` program to read the same file:

```
8388608+0 records in
8388608+0 records out
4294967296 bytes (4.3 GB, 4.0 GiB) copied, 5.43401 s, 790 MB/s
```

We observed that the `dd` program is faster than our implementation by about `4.441095s`. This is expected because the `dd` program is optimized for performance than our implementation.

3. Raw Performance

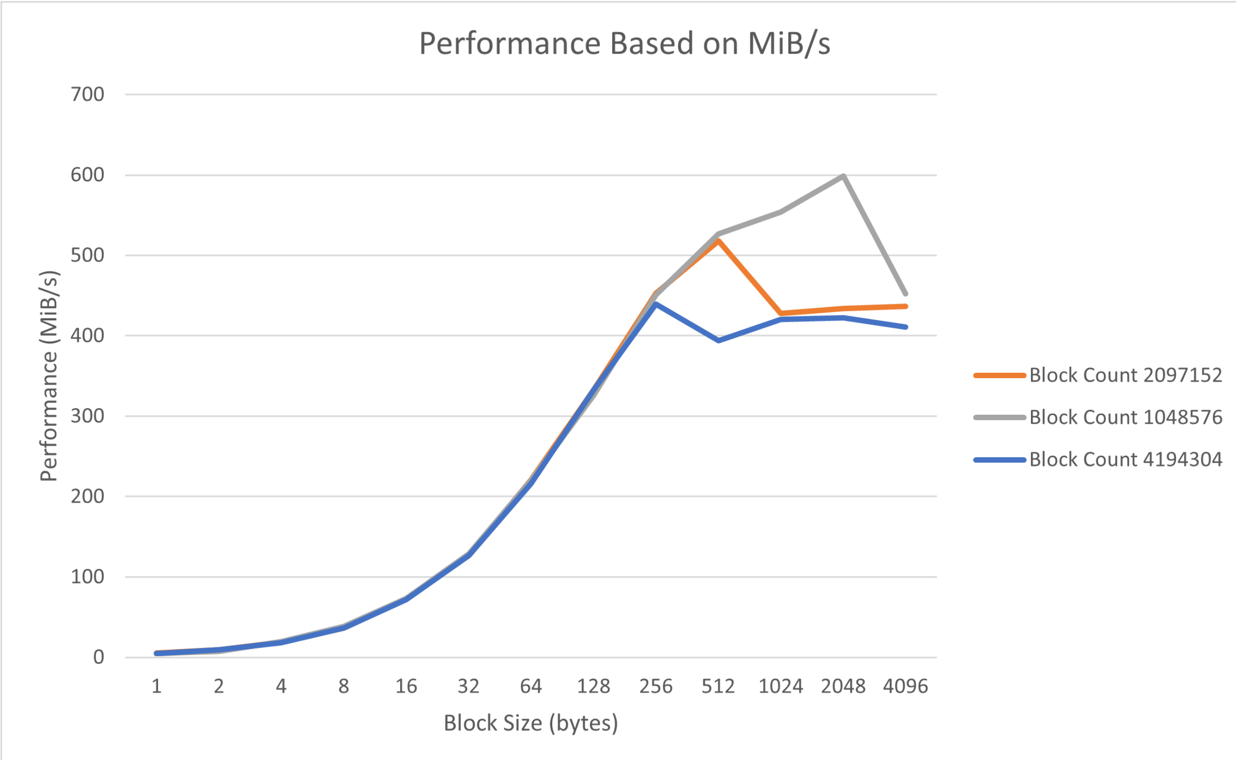
For this part, we copied over the shell script from part 2 and made a bit of change. To execute:

```
./run3.sh <filename> <block_count>
```

The shell script `run3.sh` loops calls the `run.c` in a for loop where `i` is multiplied by 2 until `4096`. For this experiment, we ran `./run3.sh testfile.txt 1048576`, `./run3.sh testfile.txt 2097152`, and `./run3.sh testfile.txt 4194304`. The results are recorded below:

Block Size	**MiB/s (BC = 1,048,576)**	MiB/s (BC = 2,097,152)	MiB/s (BC = 4,194,304)	Average MiB/s
1	5.0517	5.1612	4.784	4.999
2	7.5723	9.2353	9.527	8.778
4	19.6601	18.6073	18.521	18.930
8	38.8842	36.8034	36.772	37.487
16	73.1877	73.0674	71.777	72.677
32	129.1031	126.6619	126.833	127.533
64	220.5353	221.2141	216.177	219.309
128	326.0100	332.5371	332.848	330.465
256	450.9686	452.7892	439.298	447.685
512	526.6650	517.8536	393.603	479.374
1024	554.0898	427.9132	420.532	467.512
2048	599.0283	434.0838	422.523	485.212
4096	451.9211	436.6723	410.802	433.132

Using the data in the chart above, the following graph displays the performance (`MiB/s`) based on block sizes:



For experiments where the block count is larger, the peak performance happens earlier. For block count `1048576`, it maximum read speed happens when the block size is `1024` at `554 MiB/s`. For block count `2097152`, its maximum read speed happens when the block size is `512` at `517 MiB/s`. Lastly, for the block count of `1048576`, its maximum read speed happens when the block size is `256` at `439 MiB/s`. It is expected that the smaller block sizes result in a slower read time since the smaller the block size, the more system calls are made.

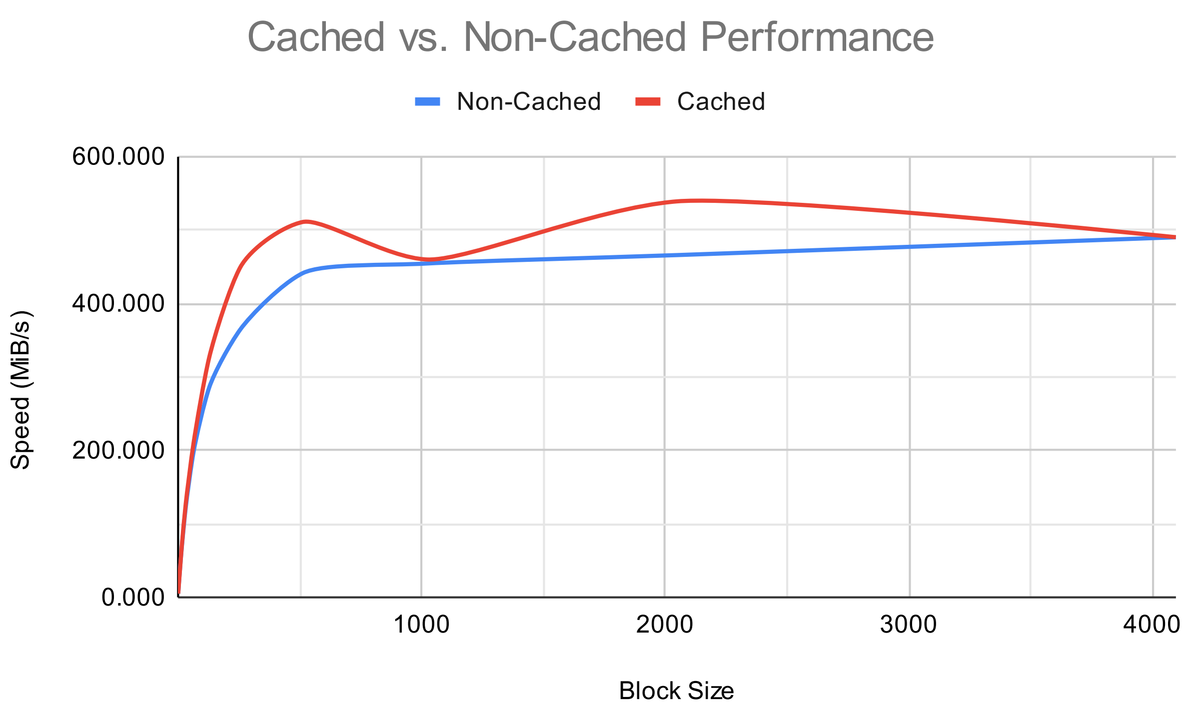
4. Caching

We observed that the performance after the caching were better in terms of `MiB/s` compared to when our program was reading `testfile.txt` after clearing the cache. This is expected because the first time when the file is read, we read from the hard drive. But after the first read, the file is in the cached and can be read again from the RAM at a higher speed.

Shown below are performance metrics without caching and with caching.

Block Size	MiB/s (Non-Cached)	MiB/s (Cached)
1	4.913	4.941
2	9.814	9.723

Block Size	MiB/s (Non-Cached)	MiB/s (Cached)
4	19.539	18.785
8	37.022	36.249
16	69.152	70.850
32	122.603	127.348
64	197.908	209.700
128	285.203	325.949
256	364.438	449.123
512	440.981	510.651
1024	454.194	459.590
2048	465.602	538.701
4096	489.846	489.901



Extra Credit

The reason why 3 is used for the `sudo sh -c "/usr/bin/echo 3 > /proc/sys/vm/drop_caches"` command is because we want to clear the `pagecache`, `dentries` (directory entries), and `inodes`. If we used `1` it would only clear the `pagecache`, and if we used `2` it would only clear up the `dentries` and `inodes`. Since caching consists of `pagecache`, `dentries`, and `inodes` for our purposes we want to clear all three, so using `3` in the command is the best option in our case.

5. System Calls

For measuring system calls with `1 byte` block size we wrote a shell script, with the following usage:

```
./run5.sh <filename>
```

The script will run a loop reading a file executed in block_size of 1 with various file sizes. Our performance in `MiB/s` and `B/s` when reading an `8 GB` file with a block size of `1 Byte` is shown below.

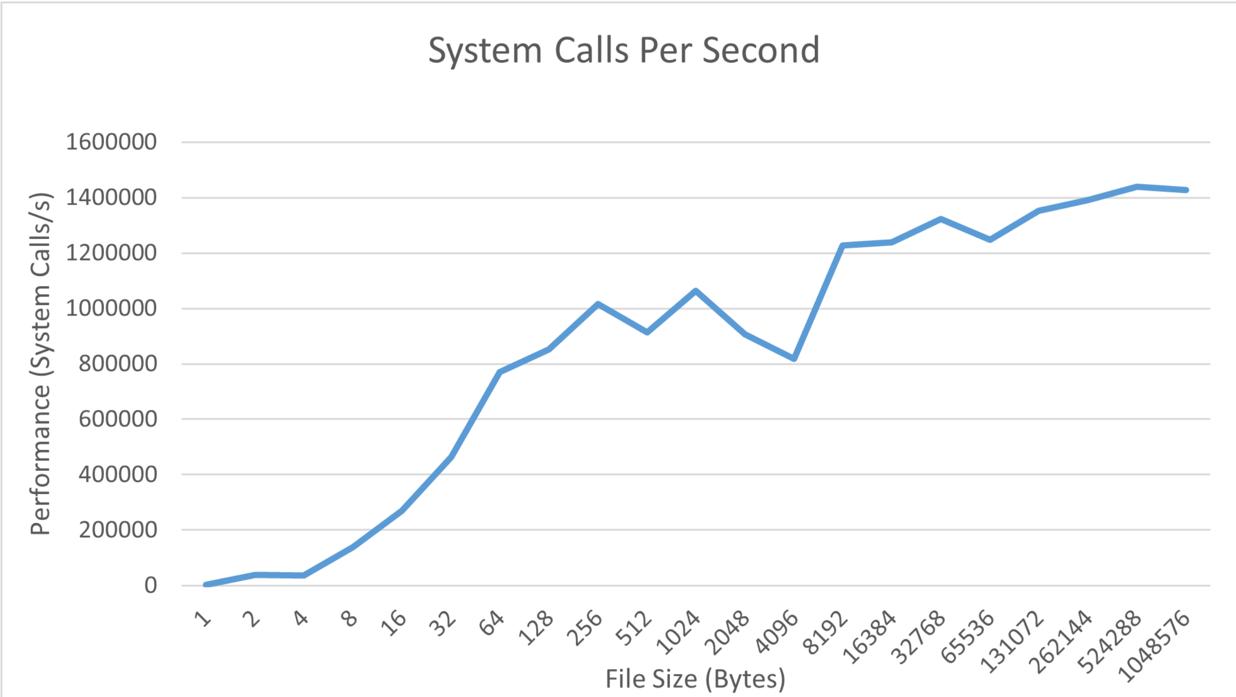
Measure performance MiB/s when using block size of 1 byte

File Size (Bytes)	Time (s)	MiB/s
1	0.000527	0.001809628684
2	0.000052	0.0366797814
4	0.00011	0.03467906605
8	0.000058	0.131541285
16	0.000059	0.2586235434
32	0.000069	0.4422837409
64	0.000083	0.7353633283
128	0.00015	0.8138020833
256	0.000252	0.968812004
512	0.00056	0.8719308036
1024	0.000963	1.014083593
2048	0.002256	0.8657468972
4096	0.005006	0.7803136237
8192	0.006678	1.169886193
16384	0.013225	1.18147448
32768	0.024751	1.262575249

File Size (Bytes)	Time (s)	MiB/s
65536	0.052503	1.190408167
131072	0.096888	1.290149451
262144	0.188475	1.326435867
524288	0.364261	1.372642144
1048576	0.735185	1.360201854

Measure performance in B/s. This is how many system calls you can do per second.

File Size (Bytes)	Time (s)	B/s (System Calls)
1	0.000527	1897.533207
2	0.000052	38461.53846
4	0.00011	36363.63636
8	0.000058	137931.0345
16	0.000059	271186.4407
32	0.000069	463768.1159
64	0.000083	771084.3373
128	0.00015	853333.3333
256	0.000252	1015873.016
512	0.00056	914285.7143
1024	0.000963	1063343.718
2048	0.002256	907801.4184
4096	0.005006	818218.1382
8192	0.006678	1226714.585
16384	0.013225	1238865.784
32768	0.024751	1323906.105
65536	0.052503	1248233.434
131072	0.096888	1352819.751
262144	0.188475	1390868.815
524288	0.364261	1439319.609
1048576	0.735185	1426275.019



We've observed that for our dataset (shown about) we made an average of made 854,312 system calls per seconds.

Try with other system calls that arguably do even less real work (e.g. lseek)

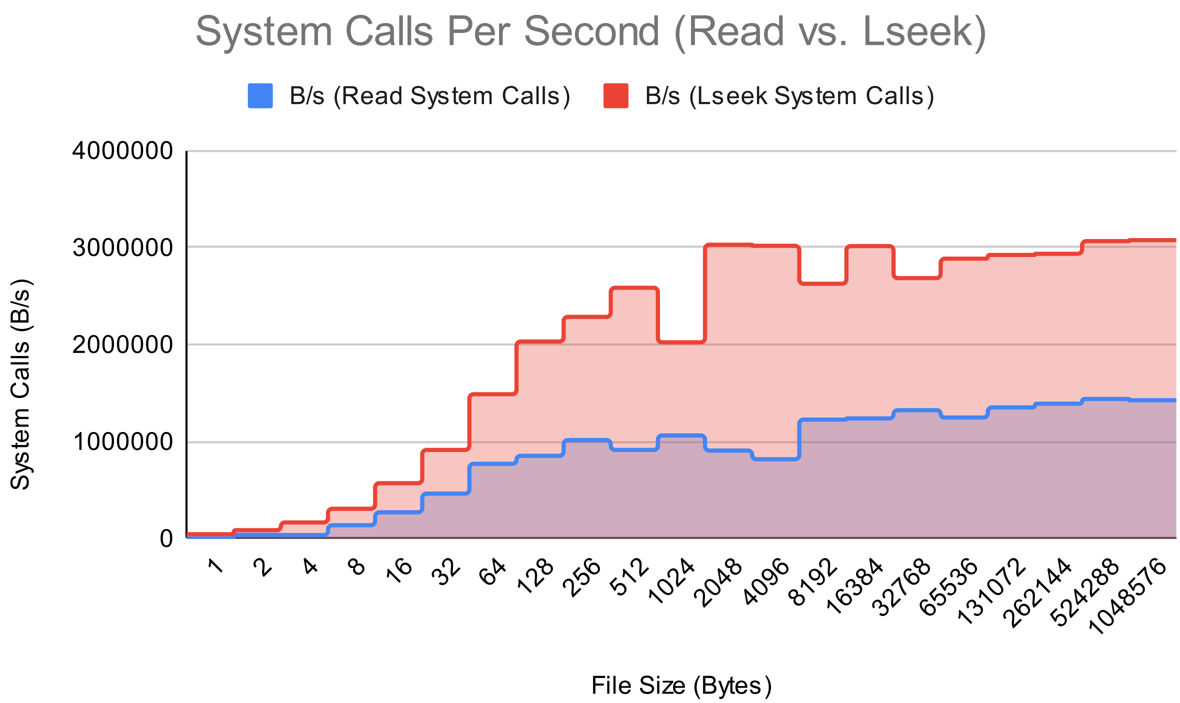
For measuring performance using other system call we made a copy of our `run.c` and modified our `readFile` function to use `lseek` system call instead of `read` system call—this program is in `run_lseek.c`. To read files with various size with `1 byte` block size using `lseek` we wrote a shell script, with the following usage:

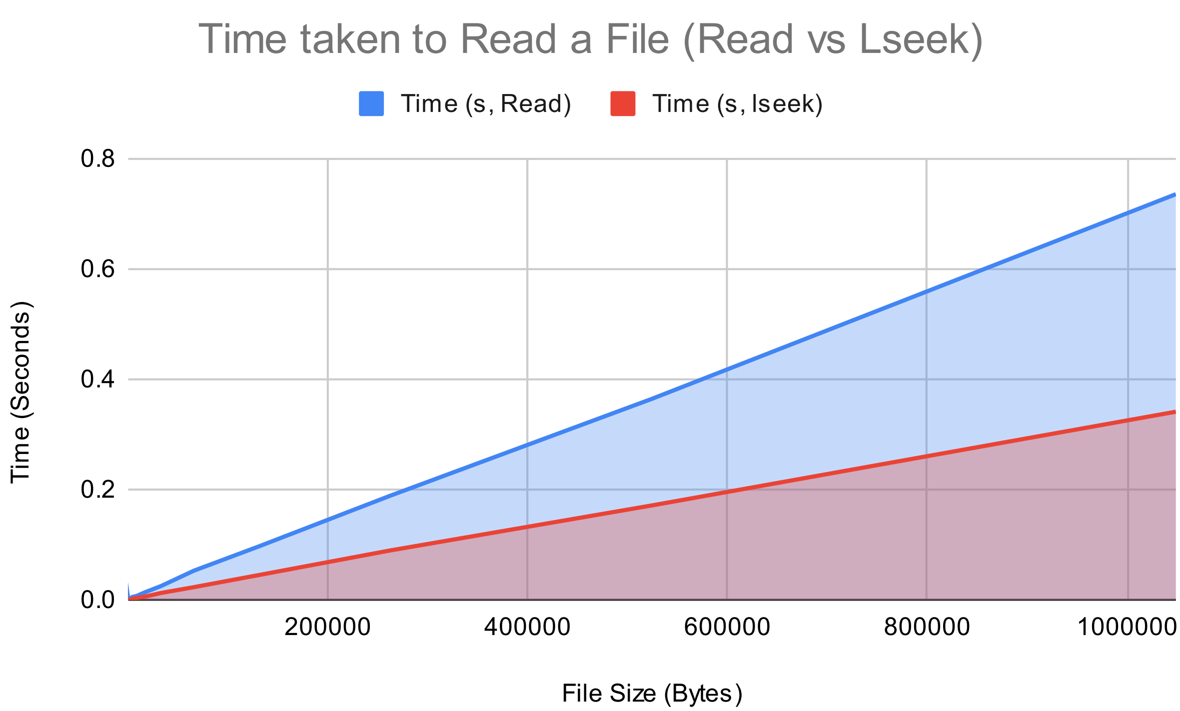
```
./run5_lseek.sh <filename>
```

Shown below are our performance metrics in `MiB/s` and `B/s` when reading an `8 GB` file with a block size of `1 Byte` for `read` and `lseek` system calls:

File Size (Bytes)	Time (s, Read)	MiB/s (Read)	B/s (Read System Calls)	Time (s, lseek)	MiB/s (lseek)	B/s (Lseek System Calls)

File Size (Bytes)	Time (s, Read)	MiB/s (Read)	B/s (Read System Calls)	Time (s, lseek)	MiB/s (lseek)	B/s (Lseek System Calls)
1	0.000527	0.001809628684	1897.533207	0.000022	0.04334883256	45454.54545
2	0.000052	0.0366797814	38461.53846	0.000023	0.08292820143	86956.52174
4	0.00011	0.03467906605	36363.63636	0.000024	0.1589457194	166666.6667
8	0.000058	0.131541285	137931.0345	0.000026	0.2934382512	307692.3077
16	0.000059	0.2586235434	271186.4407	0.000028	0.5449567522	571428.5714
32	0.000069	0.4422837409	463768.1159	0.000035	0.8719308036	914285.7143
64	0.000083	0.7353633283	771084.3373	0.000043	1.419422238	1488372.093
128	0.00015	0.8138020833	853333.3333	0.000063	1.937624008	2031746.032
256	0.000252	0.968812004	1015873.016	0.000112	2.179827009	2285714.286
512	0.00056	0.8719308036	914285.7143	0.000198	2.466066919	2585858.586
1024	0.000963	1.014083593	1063343.718	0.000506	1.929965415	2023715.415
2048	0.002256	0.8657468972	907801.4184	0.000676	2.889238166	3029585.799
4096	0.005006	0.7803136237	818218.1382	0.001357	2.878592483	3018422.992
8192	0.006678	1.169886193	1226714.585	0.003118	2.505612572	2627325.208
16384	0.013225	1.18147448	1238865.784	0.005433	2.875943309	3015645.132
32768	0.024751	1.262575249	1323906.105	0.012203	2.560845694	2685241.334
65536	0.052503	1.190408167	1248233.434	0.022709	2.752212779	2885904.267
131072	0.096888	1.290149451	1352819.751	0.044823	2.788746849	2924213.016
262144	0.188475	1.326435867	1390868.815	0.089278	2.800241941	2936266.493
524288	0.364261	1.372642144	1439319.609	0.17097	2.924489677	3066549.687
1048576	0.735185	1.360201854	1426275.019	0.340798	2.93428952	3076825.568


System Calls Per Second (Read vs. Lseek)
B/s (Read System Calls)
B/s (Lseek System Calls)
System Calls (B/s)
4000000
3000000
2000000
1000000
0
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
File Size (Bytes)


Time taken to Read a File (Read vs Lseek)
Time (s, Read)
Time (s, lseek)
Time (Seconds)
0.8
0.6
0.4
0.2
0.0
200000
400000
600000
800000
1000000
File Size (Bytes)

We've observed that when using `lseek` in our particular implementation it look more system calls than the read system call. On average, `lseek` had `1,989,232`system calls vs `854,312`system calls using read system call. However from test runs we also observed that using `lseek` yielded faster read times compared to `read` system calls—almost twice the speed. The aforementioned results are expected because `lseek`does less work than `read`. Unlike `read`, which takes file size, `lseek` increments file pointers with a time complexity of O(1).

6. Raw Performance

For this part, we wrote a shell script, with the following usage:

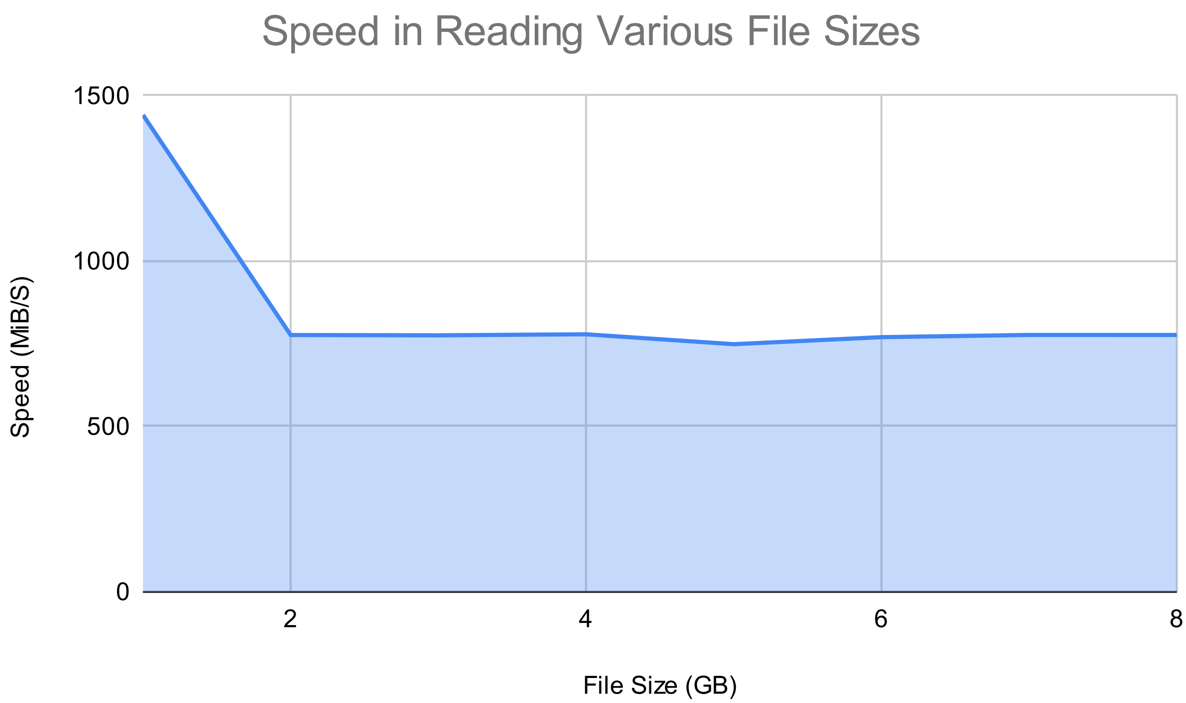
```
./fast <filename>
```

The script will run the entire file, and print out an XOR value. We ran the file using the `ubuntu-21.04-desktop-amd64.iso` test file that was provided in the project requirements page. Shown below is our output:

```
xor: a7eecd27
[run.c] reading completed in 3.430202 seconds
```

After running our app using block sizes ranging from 1 to 4096 on the `testfile.txt`, we've observed that our ideal block size 2048. Here are our performance metrics running `fast.c` with various files sizes:

File Size (GB)	File Size (Bytes)	Time (Seconds)	MiB/S
1	1073741824	0.710148	1441.952945
2	2147483648	2.637327	776.5438264
3	3221225472	3.960533	775.653176
4	4294967296	5.259469	778.7858432
5	5368709120	6.83736	748.8270327
6	6442450944	7.978094	770.1087503
7	7516192768	9.227511	776.807527
8	8589934592	10.547261	776.6945371



For future experiments, we’d like to dive deeper into why the computer reads 1GB files two times fast than larger files.