CS - GY 6233 Operating System Final Project

Team Member 1: Shuhao Du (sd3765)
Team Member 2: Yinchu Zhao (yz6615)

Github repo:
https://github.com/yz6615/OSFinalProject

Part 1.

We wrote a program (run.c) that can read and write a file from the disk. This program requires the name of the file, read mode or write mode, block size, and block count as inputs. In read mode, the program reads the file as an array of integers with the given block size and block count, and return the xor of all 4-byte integers in the file. In write mode, the program writes letter '2' multiple times to the file with the given block size and block count.
Way to execute:

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

For example, if we run the commands:

```
./build.sh
./run ubuntu-21.04-desktop-amd64.iso -r 4 50000000
```

The output in the terminal will be:

```
a7eeb2d9
```

Which is the xor of the whole iso file.

Part 2.

run2.c is a program that can find a file size which can be read in "reasonable" time. The input of the program is the name of the file and the block size. The output is the block count that will make the program read for 10 seconds. The program will use the block size to read the file from the offset 0 over and over again. In the meanwhile, a timer is used to record the time spent from the first iteration. When the timer reaches 10 seconds, the loop stops and outputs the number of the iteration, which is the same as the block count.
Way to execute:

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

For example, if we run the commands:

```
./build.sh
./run2 ubuntu-21.04-desktop-amd64.iso 4096
```

The output in the terminal will be:

```
7986697
```

Which is the block count when we use the block size 4096 to make the reading time to be exactly 10 seconds. The file size will be 4096 * 7986697

Extra Credit: learn about the dd program in Linux and see how your program's performance compares to it
By using the following command:

```
sudo dd if=test.txt of=test1.txt bs=4196 count=7986697
```

We have copied and written the same blocksize, blockcount, and file size to a new file. According to the dd command, the performance it reaches is as follows:

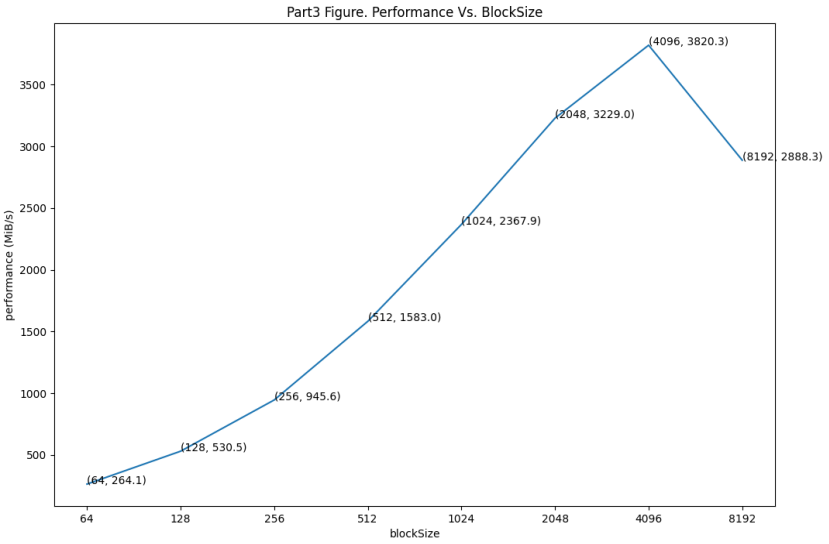
```
723304448 bytes (723 MB, 690 MiB) copied, 8.38061 s, 86.3 MB/s
```

Compared to our program, we could find out that the dd command outperformed ours. The reading time it takes for dd command is 8.38 seconds, which is a little faster than our program.

Part 3.

We have written a program that can output the performance it achieved in MiB/s. The program tries different block sizes and it will calculate the performance in megabytes per second. The program will output the statistics into a .txt file. Then, a python script is written to graph the data from the .txt file.

According to the program, the output graph is as follows:

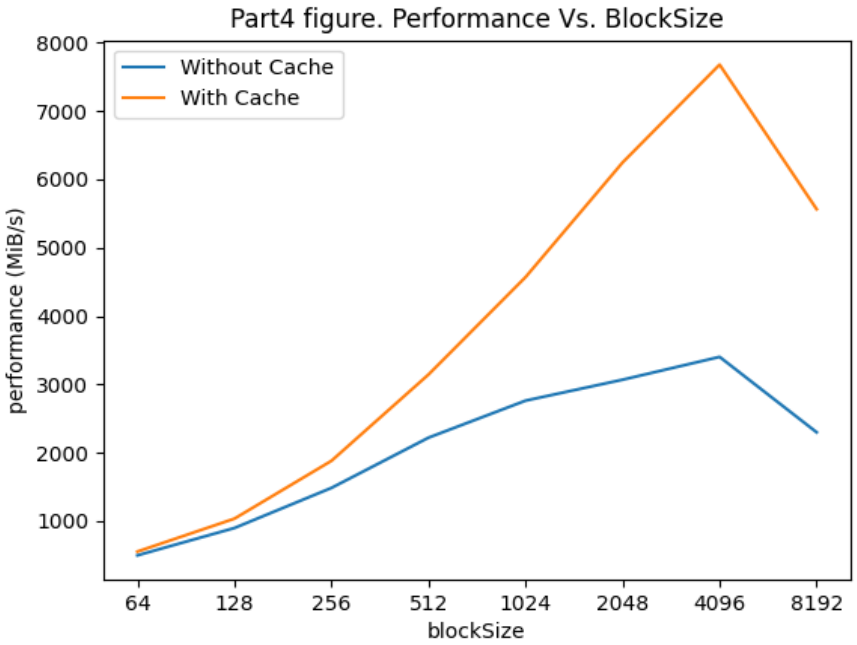


As we can see, the performance increases as the size of the block increases. At the block size of 8192, we can see a drop at performance. That is because the buffer size has exceeded its limit and cause the speed to slow down.

Part 4.

We have tried to clear the caches every time for the same blocksize. This means we clear the caches and run the program for the same block size twice. Thus, the first result is the performance without the cache and the second result is the performance with the cache. Just like part3, our program outputs the statistics into a .txt. file. Then, a python script is written to convert the data into a graph using the matplotlib module.

According to the program, the output graph is as follows:



As we can see, the performance increases as the size of the block increases for both results with and without the caches. For both performance with and without caches, we can see a drop of performance at the end of the graph. That is because the buffer has exceeded the limit so that the speed has slowed down for block size of 8192. At the same time, comparing the results for the same blocksize, we can see that the performance with the cache is faster than the performance without the cache, which is expected. We used the command: `sudo sh -c "/usr/bin/echo 3 > /proc/sys/vm/drop_caches"` to clear the caches before we run the program for each block size.

Extra Credit: Why "3"?

Echo 3 means to clear pagecache, dentries, and inodes. Compare the echo1, which only clears the pagecache and echo2, which clears dentries and inodes. In this program, we need to calculate and compare the performance based on the sizes of the block. Thus, for sure we have to clear the pagecache in order for comparison because for Linux systems, the page is always kept in the cache when there is enough free memory. At the same time we clear the dentries and inodes because inodes is a data structure which represents the file and dentries is another structure that represents a directory or a folder. Thus, by clearing those, the program has to go to the disk and read it into memory, which is clearly what we want in order to compare the performance with and without the cache.

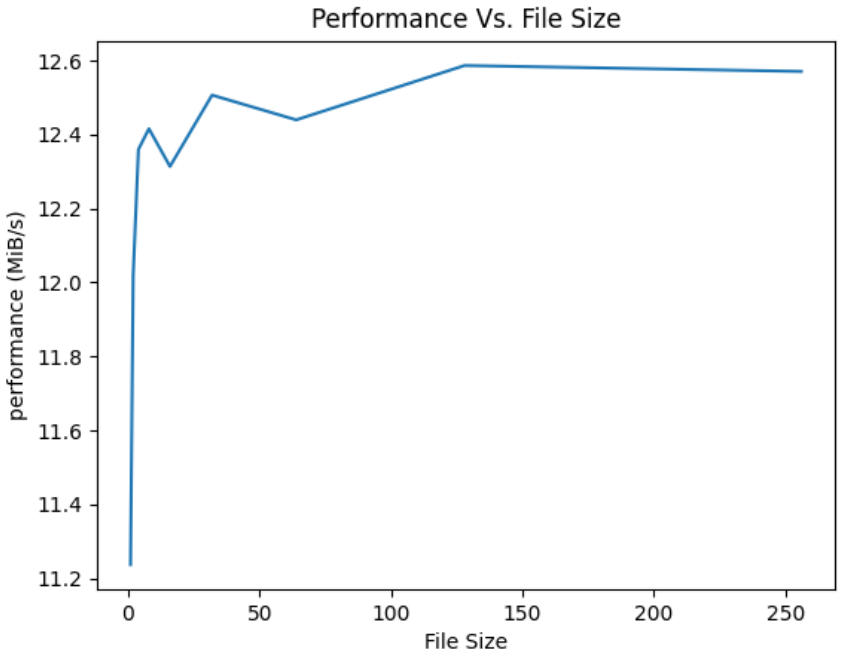
Part 5.

To measure the system calls, we set the block size to 1 byte, and change the file sizes to measure the performance in MiB/s in different block counts, then we can calculate the performance in B/s. We use lseek() instead of pread() or read() to make less work. The results are shown below:

File size in Bytes	Performance in MiB/s	Performance in B/s

1000000	11.236830	11236830
2000000	12.017564	12017564
4000000	12.359717	12359717
8000000	12.416498	12416498
16000000	12.313519	12313519
32000000	12.507290	12507290
64000000	12.440130	12440130
128000000	12.587468	12587468
256000000	12.570552	12570552

The graph of the performance (in MiB/s) vs. file sizes (in MB)



The program to record the data is part5.c, which writes the results to part5_data.txt. The part5.py is used to draw the graph by reading part5_data.txt and save the graph in part5.png. All the codes and data of this section can be found in the part5 folder.

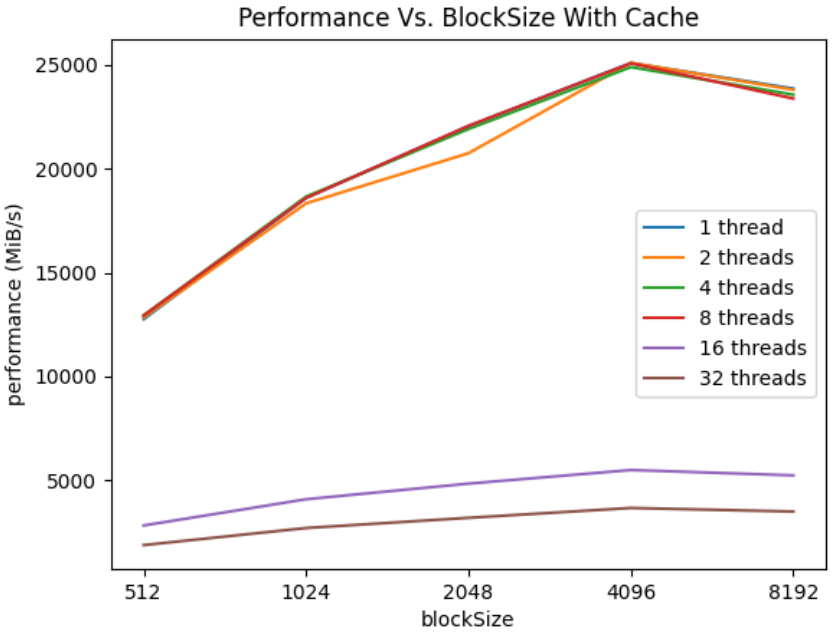
Part 6.

To optimize the performance, we need to apply multiple threads and see how many threads will maximize our performance.

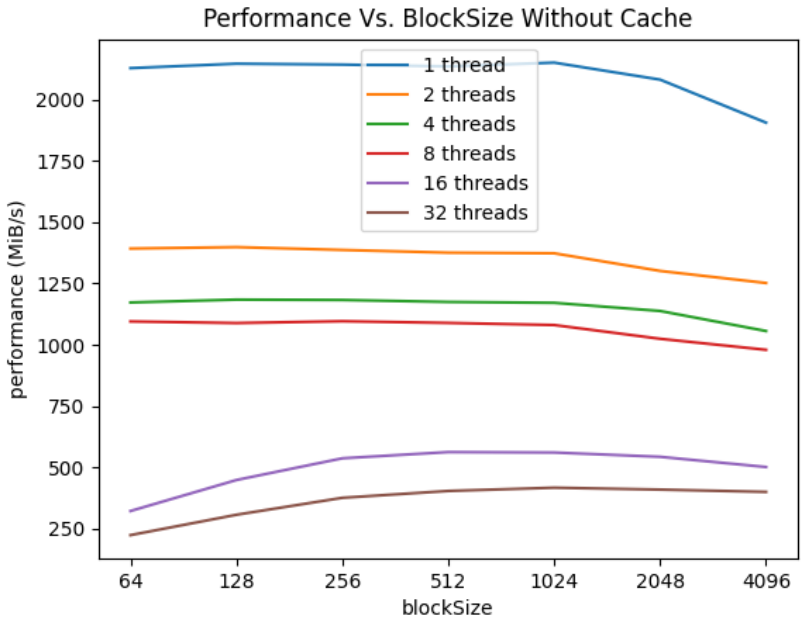
part6_size.c measures the cached performance by changing the block size and number of threads. This program records the block sizes, cached performance in MB, and the number of threads into thread_data_cache.txt. part6_with_cache.py is used to generate the graph of the thread_data_cache.txt and store it in part6_with_cache.png.

part6_size_no_cache.c measures the non-cached performance by changing the block size and number of threads. The program records the block sizes, non-cached performance in MB, and the number of threads into thread_data_no_cache.txt. part6_without_cache.py is used to generate the graph of the thread_data_no_cache.txt and store it in part6_without_cache.png.

All the codes and data of this section can be found in the part6 folder.
The graph of cached performance vs. block size on different thread numbers is shown below.



The graph of non-cached performance vs. block size on different thread numbers is shown below.



We can see that the 1 thread performance in the non-cached program is better than the multiple threads, and there is no difference in the cached program when the thread number is smaller than 16. The overall performance of the multithreading is better than what we have in part3 and part4. In conclusion, we decided to use 1 thread and block size 4096 as the parameter for our fast program. The block count doesn't matter since we need to read the whole file. We noticed the program compiled with -O3 is faster than the regular program.
Way to execute:

```
./build.sh
./fast <filename>
```

For example, if we run the commands:

```
./build.sh
./fast ubuntu-21.04-desktop-amd64.iso
```

The output in the terminal will be:

```
a7eeb2d9
```

Which is the xor of the whole iso file.