CS-GY 6233 FINAL PROJECT REPORT

-Kaushik Mellacheruvu

-Jeremy Freeman

**Part 1**: The code has been provided to read/write from disk.

**Part 2**: The code has been provided to create a big file which takes reasonable time to read for a particular block size.

**Extra credit(dd)**: The dd command takes input and output files and a block size as parameters and copies the data from source to destination. When comparing our program (reader.c – we use this file to read file from disk) and the dd program with the same input file and block size, we see that the dd command appears to take significantly longer than our program. We have graphed our results. The dd command reads data from disk for a given block size and pastes it in a new file. Our reader file reads data from disk. We have compared the time taken for different block sizes and plotted a graph.

Note: We used a reader.c file since run2.c generates a new file on trial and error and wouldn’t be a logical comparison with the dd program. The dd program also writes to the new file so that also takes time.

A graph with a line and a red line

Description automatically generated

**Extra Credit(Google Benchmark):** We used the google benchmark to benchmark the read snippet of reader.c where the system call occurs. On running the analysis on this snippet using 5 combinations of block size and block count to read a 4 GB file, we got the following results. The benchmark code can be found in the zipped file. It’s named **benchmark.cpp**

A screenshot of a computer

Description automatically generated

As you can see as we increase the block size(Benchmark column format: BM\_ReadTime/Block\_size/Block\_Count), the throughput increases.

**Part 3**:

In this part, we tested the performance of a reader.c program that reads file from the disk. This file takes input of a file, block size and block count. This program outputs the performance for different block sizes and block count to read one particular file. Below is the graph that shows the performance of a reader.c to read a 2 gb file with different block sizes.

As you can see, the performance improves as the block size increases.

A graph of a graph

Description automatically generated

**Part 4**: In this part, we used the same reader.c file to check the performance with and without clearing the cache for different block sizes. Below is the graph that shows the behaviour of the program for different block sizes with and without clearing cache. We ran it multiple times and took the average values. As you can see the performance is slower when the cache is cleared as the program needs to pull the data from the disk rather than the cache.

A graph of a graph showing a line

Description automatically generated with medium confidence

**Extra Credit**: We use the command sudo sh -c "/usr/bin/echo 3 > /proc/sys/vm/drop\_caches" to clear the cache. The “3” argument tells the system to drop page cache, dentries and inodes. Dentries are structures that contain information about directory entries and inodes also contain information about files and directories. The command essentially clears the cached information about directory entries and inodes from memory (in addition to clearing the page cache), which forces the system to re-read the information from disk to recover it. Using “3" will result in a starker contrast between cleared and non-cleared cache results, as more information will need to be recovered.

**Part 5**: When we use reader.c to read a 1 gb file, we get the following results for the MiB/s and B/s when the block size is 1.

Performance(MiB/s) -> 1.997 MiB/s

Performance(B/s) -> 2094203.982509 B/s

We observe that the performance decreases a lot, and takes a lot of time to execute. Now we have also compared making various system calls on a 1 gb file for various block sizes. We have executed a program syscalltest.c to give the number of system calls per second for write, read and lseek and plotted their values against various block sizes. As you can see lseek does the least real work and hence has most number of system calls per second, read does lesser work compared to write and hence has higher system calls than write.

A graph of a chart

Description automatically generated with medium confidence

**Part 6**:

In this part we have tried to optimize the read operation when reading large files by adjusting the block size and adding multi-threading. After tuning the code, we set the number of threads to 128 and the block size as 16,384. We executed the iso file given in the project document to test our code. These are the performance numbers.

Filename: ubuntu-21.04-desktop-amd64.iso

File size: 2818738176 bytes

Block Size: 16384

Performance(Cached): 8.75 GiB/s

Performance(Non Cached): 2 GiB/s

These numbers are the best numbers we were able to achieve on our system with the most optimized code.

**Extra Credit(AWS)**: To test our fast.c on high end hardware, we created an instance of m5d.8xlarge. We could not create m5d.metal instance due to max spot instance limit and permission issues. The m5d.metal has a memory of 96 GB, while the m5d.8xlarge has a memory of 32 GB. We chose that as it was the biggest one we could launch. The m5d.8xlarge instance has NVMe drive attached to it. We attached it as per the instructions in [Configure-AWS-ECe2-Automation](https://www.notion.so/kyall/Configure-AWS-EC2-Automation-efe1659edb874f44a69f5884cad5e481). We connected to the instance through SSH and moved fast.c and the iso file given in the project document into it and ran the code. The following image shows the performance of the program on the AWS Instance:

A screenshot of a computer program

Description automatically generated

We can observe that this instance outperforms our local speed. I was able to get a maximum speed of 8.75 GiB/s on local while the AWS instance is almost running at 17 GiB/s.