

OS FINAL PROJECT REPORT

INSTRUCTIONS

1. Unzip the folder and run the shell script (or you can compile and run then individually)
2. Run the script: `project.sh`.
3. To Run: `./project.sh` (please check the permissions as I have already given 777 permissions)
4. If it says permission denied. Please use the command:
`sudo chmod 777 project.sh`

NOTE:

For all parts:

We have created a small file of around 400MB in the script for just running the script, Feel free to replace it with a bigger file size.

So script automatically creates the file (`sample.txt`) and writes using part1 program(`run1.c`)

for part 3: we have started block size values from 64 as it would take significant amount of time for 1,4,8,16 bytes..etc to read bigger file size (for eg: 3GB)

for part4: We have excluded this part in the shell script as we cannot get the accurate results for cached as it might be the first time you might run. So, we have included values, graphs, and our observations. You can test it by running multiple times and then clear the cache.

Program names:

`Run1.c` = Part1

`Run2.c` = Part2

`Run3.c` = Part3

`Run3.c` = Part4

`Run5.c` = Part5

`Fast.c` = Part6

Part-1

Header Files and Libraries:

- The program includes several standard C libraries such as <stdio.h>, <unistd.h>, <fcntl.h>, <stdlib.h>, <assert.h>, <errno.h>, <string.h>, <sys/stat.h>, and <sys/time.h>. These libraries provide functions for file I/O, memory allocation, error handling, and time measurement.

Global Variables:

- static unsigned int fxor: A static variable to store the XOR checksum.

Utility Functions:

- double now(): Retrieves the current time for performance measurement.
- long filesize(const char *filename): Returns the size of the specified file.
- Main Function: The main function is the entry point of the program. It parses command line arguments to determine the file name, operation mode (read or write), block size, and block count. Performs error checks on the input parameters and exits the program if invalid inputs are detected.

Read Mode:

- If the mode is set to read (-r), the program opens the specified file in read-only mode. Checks if the requested file size (block size * block count) is less than or equal to the actual file size.
- Reads the file in blocks, calculates the XOR checksum, and measures the time taken. Prints the XOR checksum and the performance (MB/s).

Write Mode:

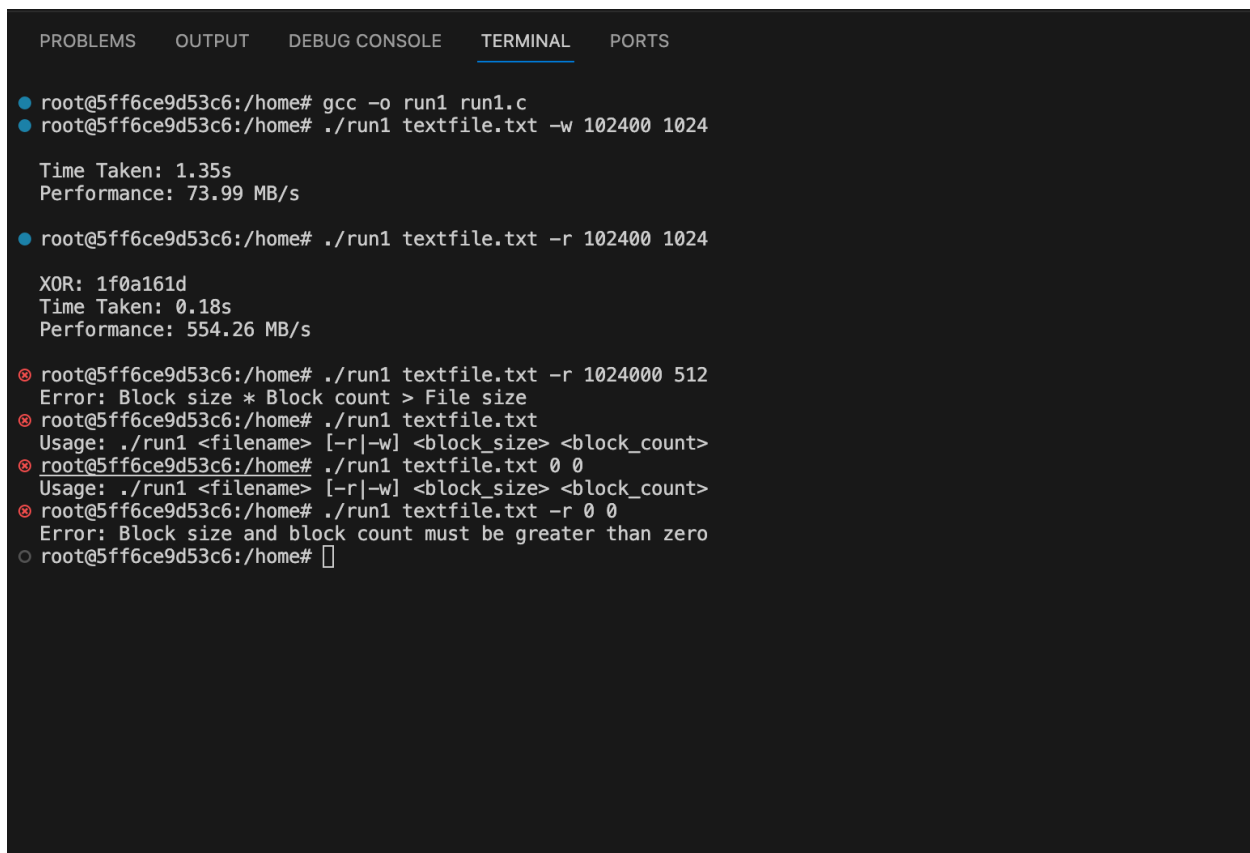
- If the mode is set to write (-w), the program opens the specified file in write-only mode (creates a new file if it does not exist).
- Generates random data(alphabets) for each block and writes it to the file.
- Measures the time taken and prints the performance (MB/s).

OUTPUT/Conclusion:

The program provides a s reading or writing data to a file while calculating the XOR checksum. Performance is measured in terms of data throughput, and the XOR checksum is displayed for read operations.

The following screenshot includes:

1. Successful Execution of reading a file.
2. Successful Execution of writing content to a file.
3. Throw an error if you try to read more than the size of the file.
4. Throw an error if it's not correctly executed.
5. Throw an error as invalid block size if its 0 for both read and write.



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

● root@5ff6ce9d53c6:/home# gcc -o run1 run1.c
● root@5ff6ce9d53c6:/home# ./run1 textfile.txt -w 102400 1024

Time Taken: 1.35s
Performance: 73.99 MB/s

● root@5ff6ce9d53c6:/home# ./run1 textfile.txt -r 102400 1024

XOR: 1f0a161d
Time Taken: 0.18s
Performance: 554.26 MB/s

⊗ root@5ff6ce9d53c6:/home# ./run1 textfile.txt -r 1024000 512
Error: Block size * Block count > File size
⊗ root@5ff6ce9d53c6:/home# ./run1 textfile.txt
Usage: ./run1 <filename> [-r|-w] <block_size> <block_count>
⊗ root@5ff6ce9d53c6:/home# ./run1 textfile.txt 0 0
Usage: ./run1 <filename> [-r|-w] <block_size> <block_count>
⊗ root@5ff6ce9d53c6:/home# ./run1 textfile.txt -r 0 0
Error: Block size and block count must be greater than zero
○ root@5ff6ce9d53c6:/home#
```

Part-2

Measurement

Functions:

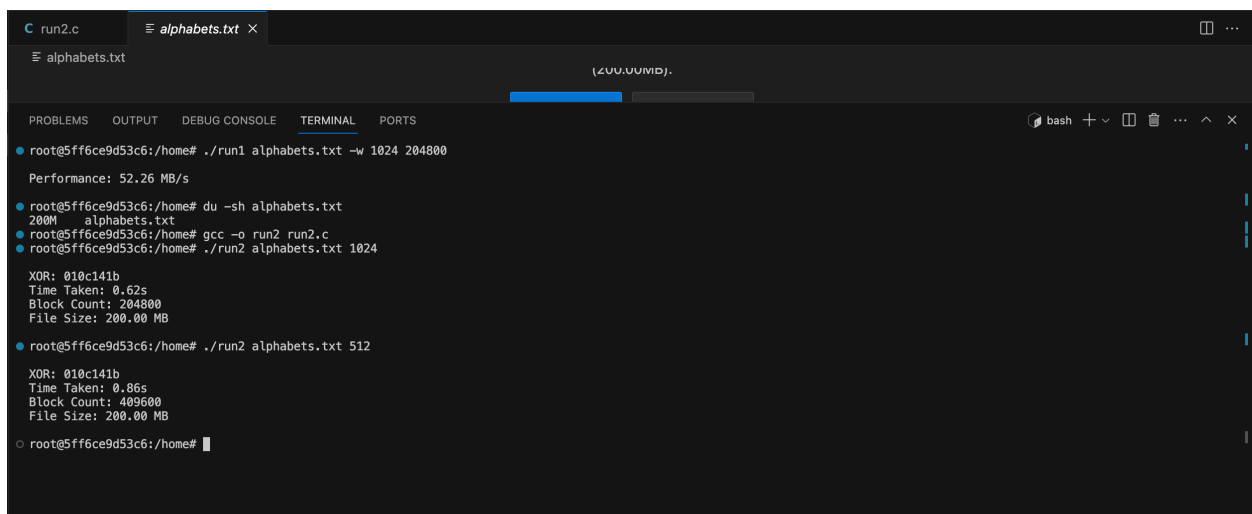
- `getCurrentTime()`: Retrieves the current time in seconds, facilitating time measurements.
- `xorBuffer()`: Performs XOR on an array of unsigned integers.
- `getFileSize()`: Retrieves the size of the specified file.
- Main Function (`main ()`): Parses command-line arguments for filename and block size. Opens the file in read-only mode and verifies its existence. Iteratively reads blocks of data from the file, computes the XOR, and tracks relevant metrics.

OUTPUT/Conclusion:

Outputs the XOR result, time taken, block count, file size.

The following screenshot includes:

1. Creating/writing data to a file with 200 MB using `run1`.
2. Checking the size of the file using `du -sh` command.
3. Compiling the `run2.c`
4. Testing the `run2` program and it correctly returns file size as 200MB. Returns XOR value, Time taken, Block Count, File Size.
5. Testing with different block size.



```
C run2.c  alphabets.txt x
alphabets.txt  (L000000MB).
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
● root@5ff6ce9d53c6:/home# ./run1 alphabets.txt -w 1024 204800
Performance: 52.26 MB/s
● root@5ff6ce9d53c6:/home# du -sh alphabets.txt
200M  alphabets.txt
● root@5ff6ce9d53c6:/home# gcc -o run2 run2.c
● root@5ff6ce9d53c6:/home# ./run2 alphabets.txt 1024

XOR: 010c141b
Time Taken: 0.62s
Block Count: 204800
File Size: 200.00 MB

● root@5ff6ce9d53c6:/home# ./run2 alphabets.txt 512

XOR: 010c141b
Time Taken: 0.86s
Block Count: 409600
File Size: 200.00 MB

○ root@5ff6ce9d53c6:/home#
```

Extra Credit: “dd” program in Linux and see how your program's performance compares to it!

dd command: It is a versatile utility for copying and converting files. It stands for "data duplicator" and is commonly used for tasks such as creating disk images, copying data between devices, and manipulating data streams.

dd if=input_file of=output_file bs=block_size count=number_of_blocks

- **if=input_file:** Specifies the input file. It can be a regular file, a device, or a special file.
- **of=output_file:** Specifies the output file where the data will be written.
- **bs=block_size:** Sets the block size for data transfer. It can be specified with units like k for kilobytes, M for megabytes, or G for gigabytes.
- **count=number_of_blocks:** Specifies the number of blocks to copy.

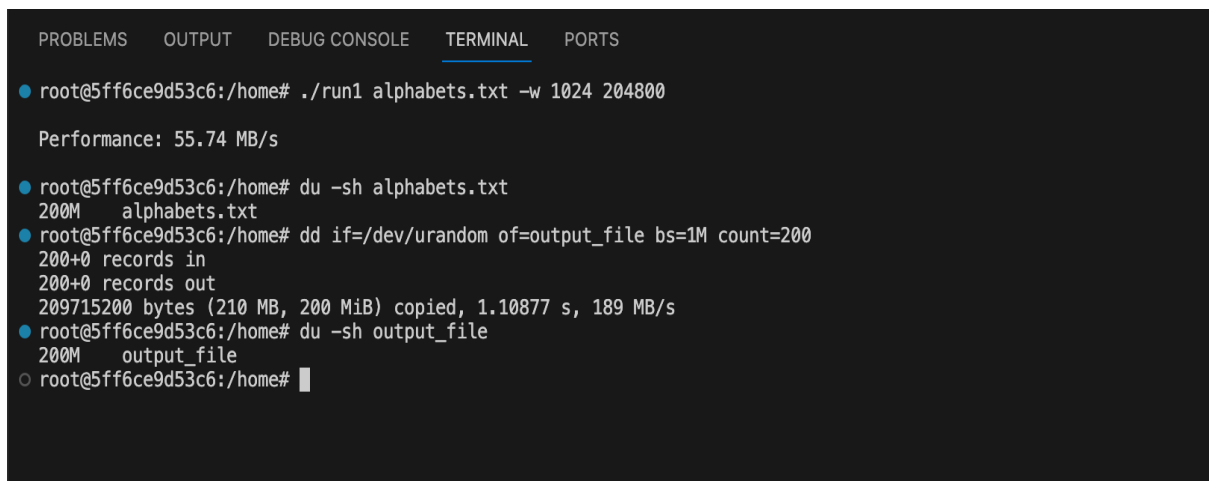
Example usage:

dd if=/dev/urandom of=output_file bs=1M count=1000

Comparison with my program:

The following screenshot includes:

1. create/write a file using write and see the performance.
2. Check the size of the file by write operation/program 1.
3. Use dd command to create a file of same size and see the performance.
4. Check the size of the file generated by dd command.



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

● root@5ff6ce9d53c6:/home# ./run1 alphabets.txt -w 1024 204800

Performance: 55.74 MB/s

● root@5ff6ce9d53c6:/home# du -sh alphabets.txt
200M    alphabets.txt
● root@5ff6ce9d53c6:/home# dd if=/dev/urandom of=output_file bs=1M count=200
200+0 records in
200+0 records out
209715200 bytes (210 MB, 200 MiB) copied, 1.10877 s, 189 MB/s
● root@5ff6ce9d53c6:/home# du -sh output_file
200M    output_file
○ root@5ff6ce9d53c6:/home#
```

My Observation:

- I have created the file of same size using two different ways.
- Using dd command took me very less time to write to a file when compared to using my run1 program.
- I have seen that the performance using dd command is almost **morethan 3 times** better/faster than using my program1(which I used for part1).
- **Performance: 55.74 MB/s** using my program
- **Performance: 189 MB/s** using dd command.

Extra Credit: Google Benchmark

NOTE: Though I have not used Google Benchmark in my program, but I just explored about it and how we can use it.

Google Benchmark is a C++ library, and it is used to simplify the process of benchmarking code, allowing developers to measure the performance of specific code snippets or functions in a controlled and repeatable manner.

Features Which I found:

1. **Simple Syntax:** straightforward syntax for writing benchmarks, making it easy for us to measure the performance of the code.
2. **Statistical Analysis:** It performs statistical analysis on benchmark results, providing information about the mean, standard deviation, and other statistical measures. This helps us understanding the variability of performance measurements.
3. **Cross-Platform:** It is designed to work on various platforms, including Linux, Windows, and macOS.

How to Use Google Benchmark:

To use Google Benchmark, you typically follow these steps:

1. Include the Library:

```
#include <benchmark/benchmark.h>
```

2. **Define Benchmarks:** Write benchmark functions using the `BENCHMARK` macro. The benchmark functions should contain the code you want to measure.

Part-3

3. Raw Performance

Logic I have used for Performance Measurement:

performance = FileSize/ runtime.

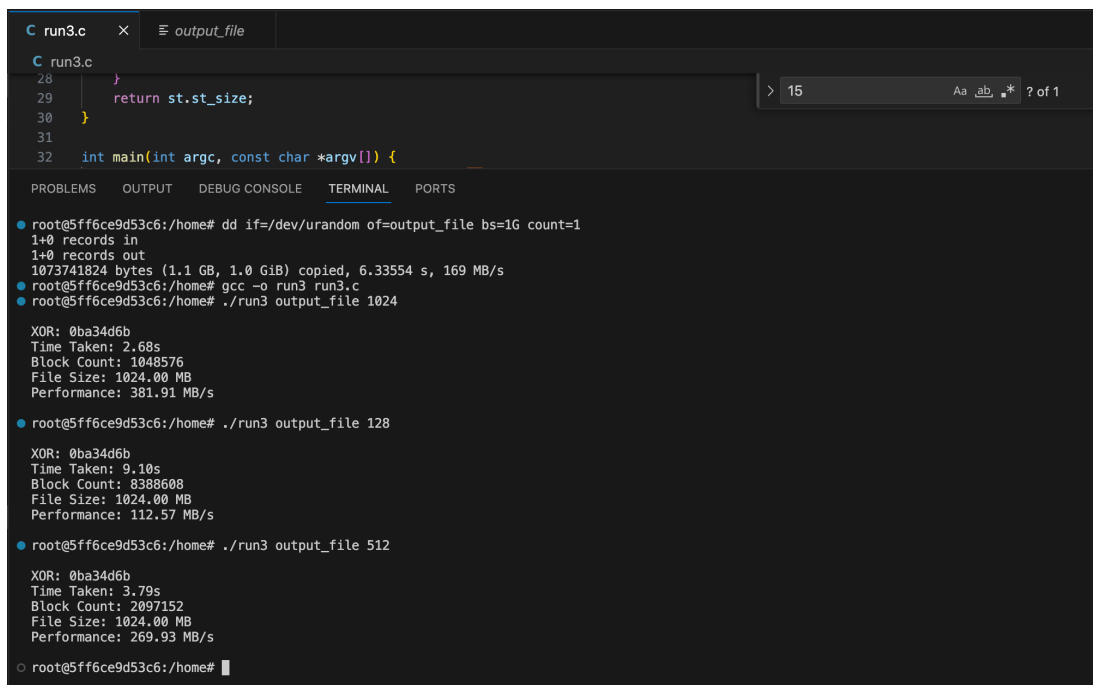
The code now measures and reports performance in MiB/s and the computation considers the total file size processed and the time taken to read the file.

OUTPUT:

We have included the performance metric in MiB/s, providing a clearer representation of the benchmark results.

The following screenshot includes:

1. Creating file of size 1GB.
2. Compiling the run3.c
3. Testing the output_file with different block sizes and observing its performance.



The screenshot shows a code editor with a file named `run3.c` and a terminal window. The code in `run3.c` is as follows:

```
28 }
29 return st.st_size;
30 }
31
32 int main(int argc, const char *argv[]) {
```

The terminal window shows the following commands and output:

```
root@5ff6ce9d53c6:/home# dd if=/dev/urandom of=output_file bs=1G count=1
1+0 records in
1+0 records out
1073741824 bytes (1.1 GB, 1.0 GiB) copied, 6.33554 s, 169 MB/s
root@5ff6ce9d53c6:/home# gcc -o run3 run3.c
root@5ff6ce9d53c6:/home# ./run3 output_file 1024

XOR: 0ba34d6b
Time Taken: 2.68s
Block Count: 1048576
File Size: 1024.00 MB
Performance: 381.91 MB/s

root@5ff6ce9d53c6:/home# ./run3 output_file 128

XOR: 0ba34d6b
Time Taken: 9.10s
Block Count: 8388608
File Size: 1024.00 MB
Performance: 112.57 MB/s

root@5ff6ce9d53c6:/home# ./run3 output_file 512

XOR: 0ba34d6b
Time Taken: 3.79s
Block Count: 2097152
File Size: 1024.00 MB
Performance: 269.93 MB/s

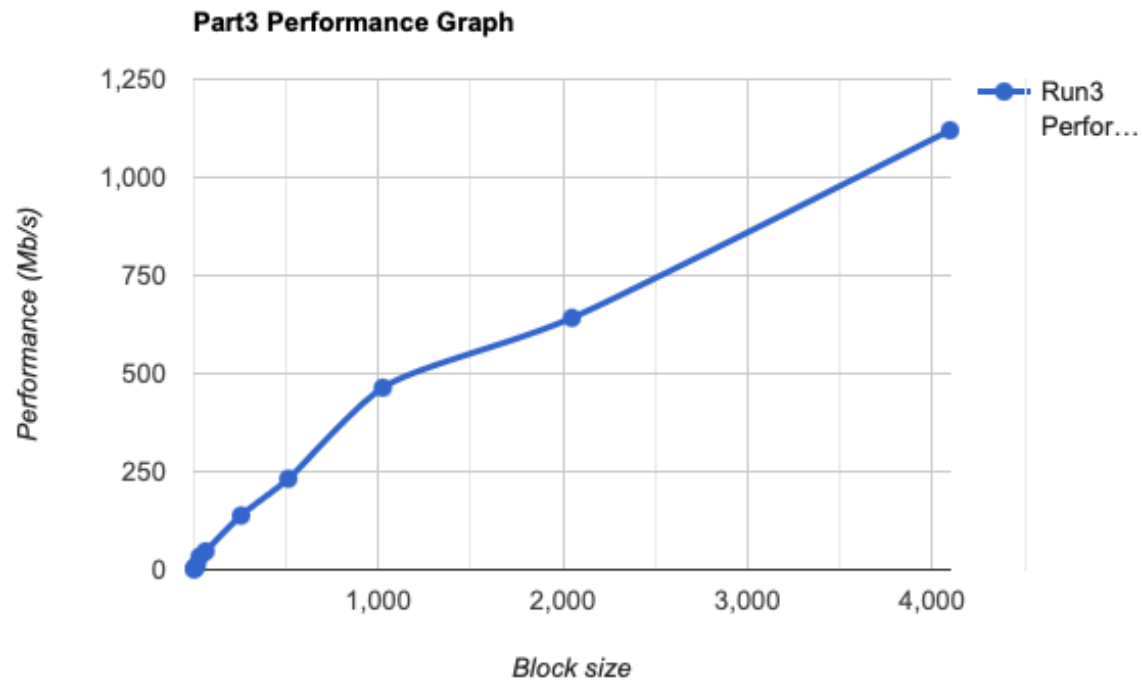
root@5ff6ce9d53c6:/home#
```


After performing run3.c against iso file(2GB) across different block sizes, below are the values in the table.

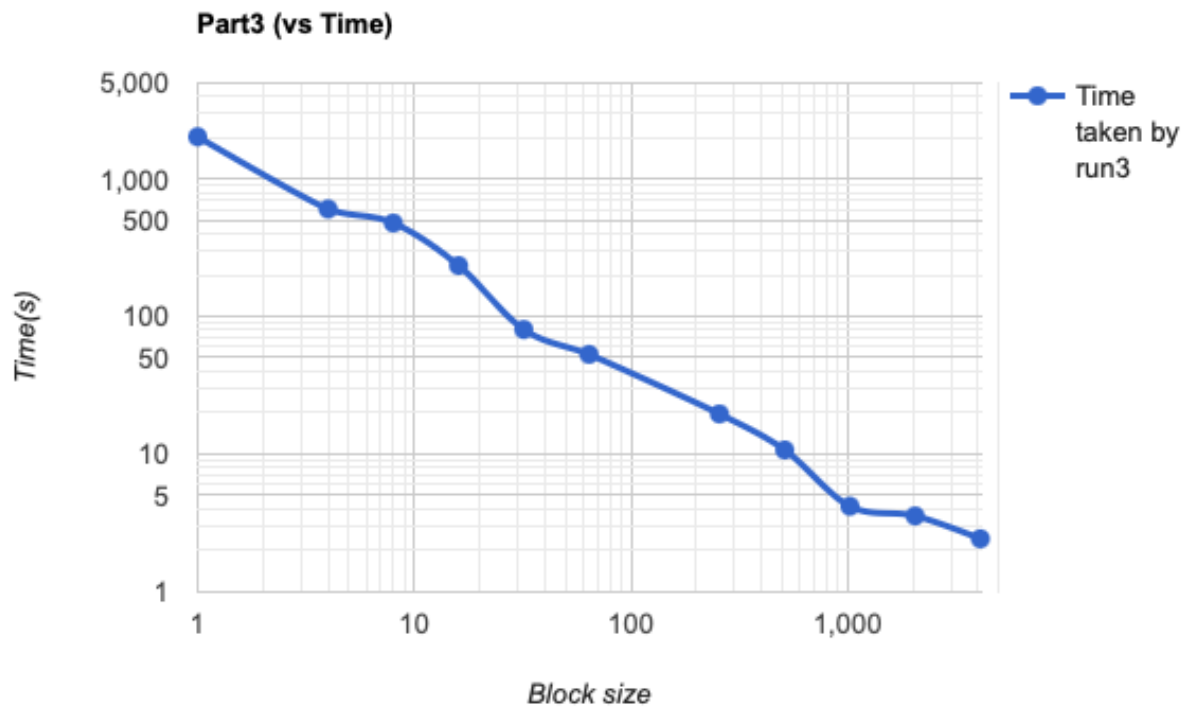
Block size (In bytes)	Block count	Speed in Mib/s	Time in sec
1	2818738098	1.08 MB/s	2032.22s
4	704684544	3.05 MB/s	705.32s
8	352342272	5.62 MB/s	478.22s
16	176171136	11.00 MB/s	244.45s
32	88085568	33.53 MB/s	80.20s
64	44042784	36.92 MB/s	72.81s
256	11010696	138.06 MB/s	19.47s
512	5505348	231.95 MB/s	11.59s
1024	2752674	463.87 MB/s	4.14s
2048	1376337	641.87 MB/s	3.52s
4096	688169	1119.99 MB/s	2.40s

Graph:

These graphs are generated based on the values we got in the above table.



Observation: When the block size is increased gradually, Performance increased significantly, and we can see that in the above graph.



Observation: When the block size is increased gradually, Time taken to read the file has decreased significantly from almost 2000 seconds to 4 seconds and we can see that in the above graph.

Part-4 Caching

After performing run3.c against iso file(2GB) across different block sizes, below are the values in the table.

Before Clearing the cache:

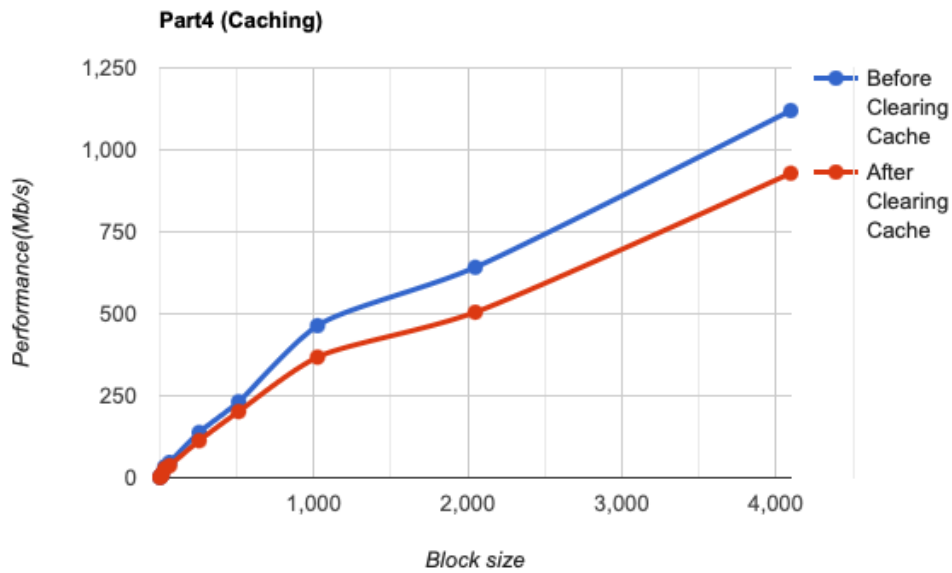
Block size (In bytes)	Block count	Speed in Mib/s	Time in sec
1	2818738098	1.08 MB/s	2032.22s
4	704684544	3.05 MB/s	605.32s
8	352342272	5.62 MB/s	478.22s
16	176171136	11.00 MB/s	234.45s
32	88085568	33.53 MB/s	80.20s
64	44042784	46.92 MB/s	52.81s
256	11010696	138.06 MB/s	19.47s
512	5505348	231.95 MB/s	10.64s
1024	2752674	463.87 MB/s	4.14s
2048	1376337	641.87 MB/s	3.52s
4096	688169	1119.99 MB/s	2.40s

After clearing the cache:

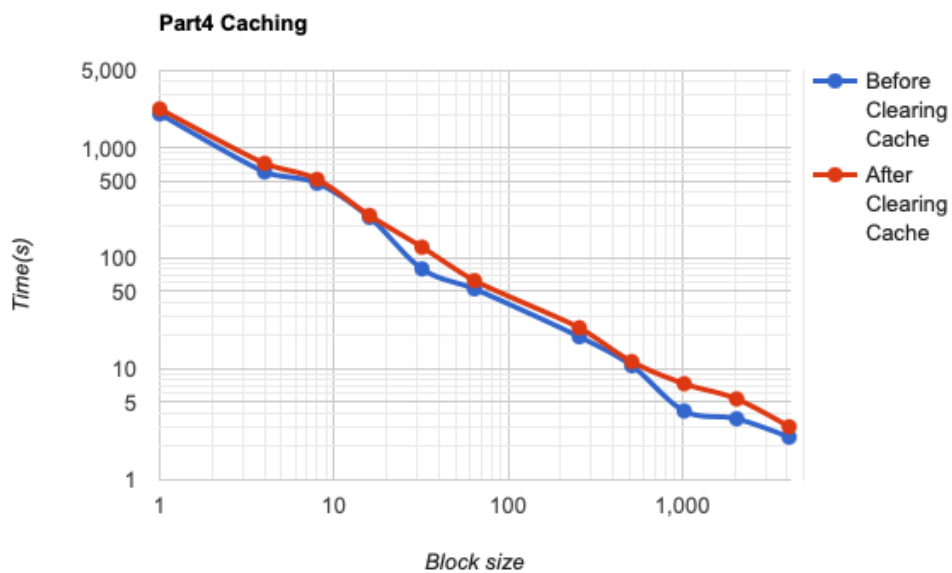
Block size (In bytes)	Block count	Speed in Mib/s	Time in sec
1	2818738098	1.06 MB/s	2252.22s
4	704684544	2.78 MB/s	723.32s
8	352342272	4.82 MB/s	518.22s
16	176171136	9.89 MB/s	244.45s
32	88085568	27.53 MB/s	126.10s
64	44042784	36.92 MB/s	62.81s
256	11010696	113.06 MB/s	23.47s
512	5505348	201.95 MB/s	11.59s
1024	2752674	367.92 MB/s	7.31s
2048	1376337	504.85MB/s	5.32s
4096	688169	928.81 MB/s	2.98s

Graphs

With respect to performance (MB/s):



With respect to Time(seconds):



Observation:

In the first graph, we can see that the performance before clearing cache is more than the performance after clearing the cache.

In the second graph, we can see that time taken to read the file before clearing cache is less than the time taken to read the file after clearing the cache.

Performance:

Before clearing the Cache > After clearing the cache.

Time:

After clearing the cache > Before clearing the Cache.

Extra credit: Why "3"? Read up on it and explain.

We can manage the Linux kernel's page cache, dentries, and inodes using the command.

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

My Understanding after I used and explored the above command:

The number in the command corresponds to the option passed to the file, indicating what to clear.

The values that we can give in the place of number to this file are:

- **1 (free pagecache):** This option frees up the page cache, which is the cache used by the kernel to store pages of files.

- **2 (free dentries and inodes):** This option frees up the dentry and inode caches. Dentries represent directory entries, and inodes represent information about files, such as ownership and permissions.

- **3 (free pagecache, dentries, and inodes):** This option combines the effects of options 1 and 2, clearing the page cache as well as the dentry and inode caches.

How 3 in the command helped me to obtain accurate measurements:

When "3" is used, it is a more comprehensive cache clearing operation, freeing up more system resources. This helped me in clearing system's caches to obtain accurate measurements and to release memory resources.

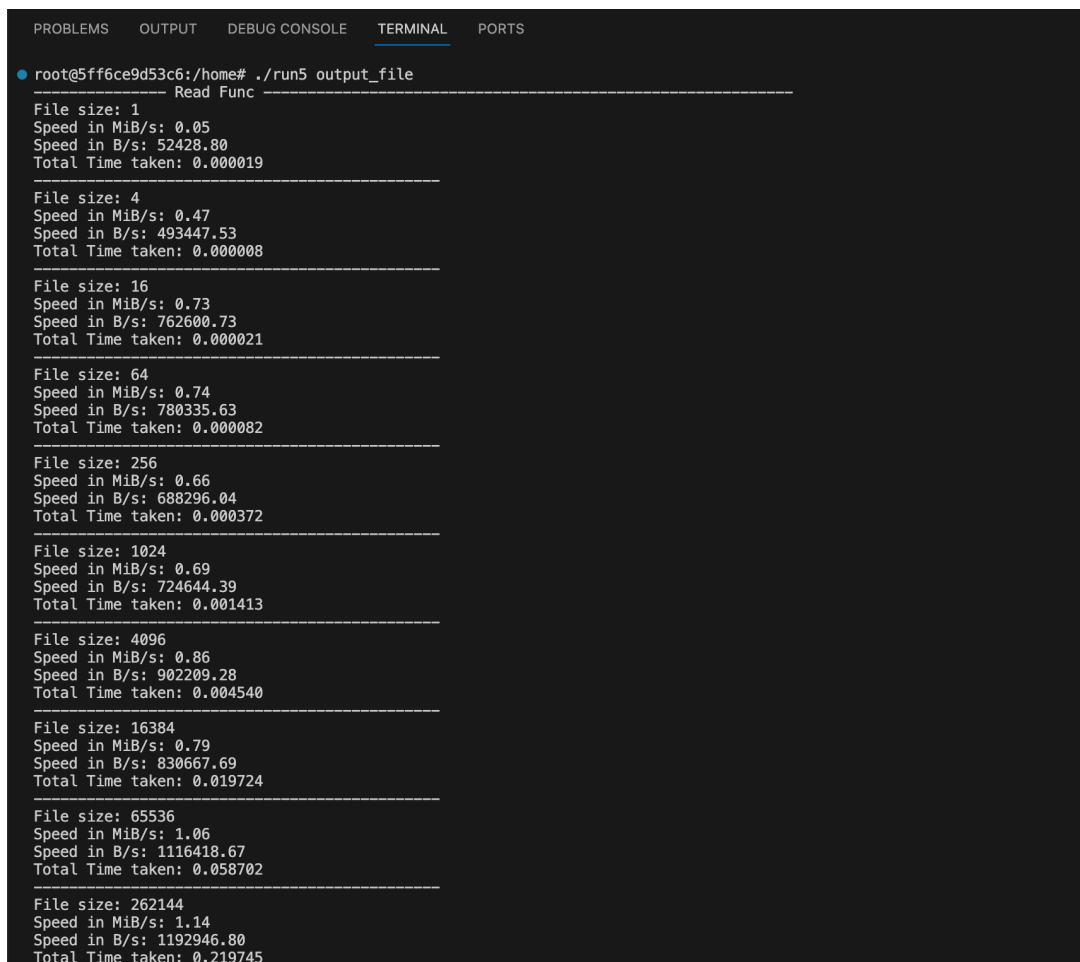
Part -5

I have used **Lseek, open and read** system calls for Part5.

OUTPUT:

The following screenshots includes:

1. Results of filesize, Speed in MB/s, Speed in B/s and Time taken for Read system Call.
2. Results of filesize, Speed in MB/s, Speed in B/s and Time taken for Lseek system Call.
3. Results of filesize, Speed in MB/s, Speed in B/s and Time taken for Open system Call.



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

● root@5ff6ce9d53c6:/home# ./run5 output_file
----- Read Func -----
File size: 1
Speed in MiB/s: 0.05
Speed in B/s: 52428.80
Total Time taken: 0.000019
-----
File size: 4
Speed in MiB/s: 0.47
Speed in B/s: 493447.53
Total Time taken: 0.000008
-----
File size: 16
Speed in MiB/s: 0.73
Speed in B/s: 762600.73
Total Time taken: 0.000021
-----
File size: 64
Speed in MiB/s: 0.74
Speed in B/s: 780335.63
Total Time taken: 0.000082
-----
File size: 256
Speed in MiB/s: 0.66
Speed in B/s: 688296.04
Total Time taken: 0.000372
-----
File size: 1024
Speed in MiB/s: 0.69
Speed in B/s: 724644.39
Total Time taken: 0.001413
-----
File size: 4096
Speed in MiB/s: 0.86
Speed in B/s: 902209.28
Total Time taken: 0.004540
-----
File size: 16384
Speed in MiB/s: 0.79
Speed in B/s: 830667.69
Total Time taken: 0.019724
-----
File size: 65536
Speed in MiB/s: 1.06
Speed in B/s: 1116418.67
Total Time taken: 0.058702
-----
File size: 262144
Speed in MiB/s: 1.14
Speed in B/s: 1192946.80
Total Time taken: 0.219745
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

----- LSEEK Func -----
File size: 1
Speed in MiB/s: 1.00
Speed in B/s: 1048576.00
Total Time taken: 0.000001
-----
File size: 4
Speed in MiB/s: 1.33
Speed in B/s: 1398101.33
Total Time taken: 0.000003
-----
File size: 16
Speed in MiB/s: 1.52
Speed in B/s: 1597830.10
Total Time taken: 0.000010
-----
File size: 64
Speed in MiB/s: 1.42
Speed in B/s: 1491308.09
Total Time taken: 0.000043
-----
File size: 256
Speed in MiB/s: 1.54
Speed in B/s: 1619520.10
Total Time taken: 0.000158
-----
File size: 1024
Speed in MiB/s: 1.65
Speed in B/s: 1732540.26
Total Time taken: 0.000591
-----
File size: 4096
Speed in MiB/s: 1.65
Speed in B/s: 1729749.21
Total Time taken: 0.002368
-----
File size: 16384
Speed in MiB/s: 1.63
Speed in B/s: 1710418.32
Total Time taken: 0.009579
-----
File size: 65536
Speed in MiB/s: 1.65
Speed in B/s: 1731187.22
Total Time taken: 0.037856
-----
File size: 262144
Speed in MiB/s: 1.64
Speed in B/s: 1724110.20
Total Time taken: 0.152046
-----
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

----- Open Func -----
File size: 1
Speed in MiB/s: 0.07
Speed in B/s: 71089.90
Total Time taken: 0.000014
-----
File size: 4
Speed in MiB/s: 0.24
Speed in B/s: 250406.21
Total Time taken: 0.000016
-----
File size: 16
Speed in MiB/s: 0.26
Speed in B/s: 271695.81
Total Time taken: 0.000059
-----
File size: 64
Speed in MiB/s: 0.27
Speed in B/s: 279620.27
Total Time taken: 0.000229
-----
File size: 256
Speed in MiB/s: 0.29
Speed in B/s: 304090.01
Total Time taken: 0.000842
-----
File size: 1024
Speed in MiB/s: 0.32
Speed in B/s: 334004.77
Total Time taken: 0.003066
-----
File size: 4096
Speed in MiB/s: 0.30
Speed in B/s: 316050.43
Total Time taken: 0.012960
-----
File size: 16384
Speed in MiB/s: 0.32
Speed in B/s: 336662.14
Total Time taken: 0.048666
-----
File size: 65536
Speed in MiB/s: 0.34
Speed in B/s: 351767.18
Total Time taken: 0.186305
-----
File size: 262144
Speed in MiB/s: 0.34
Speed in B/s: 358523.74
Total Time taken: 0.731176
-----
```

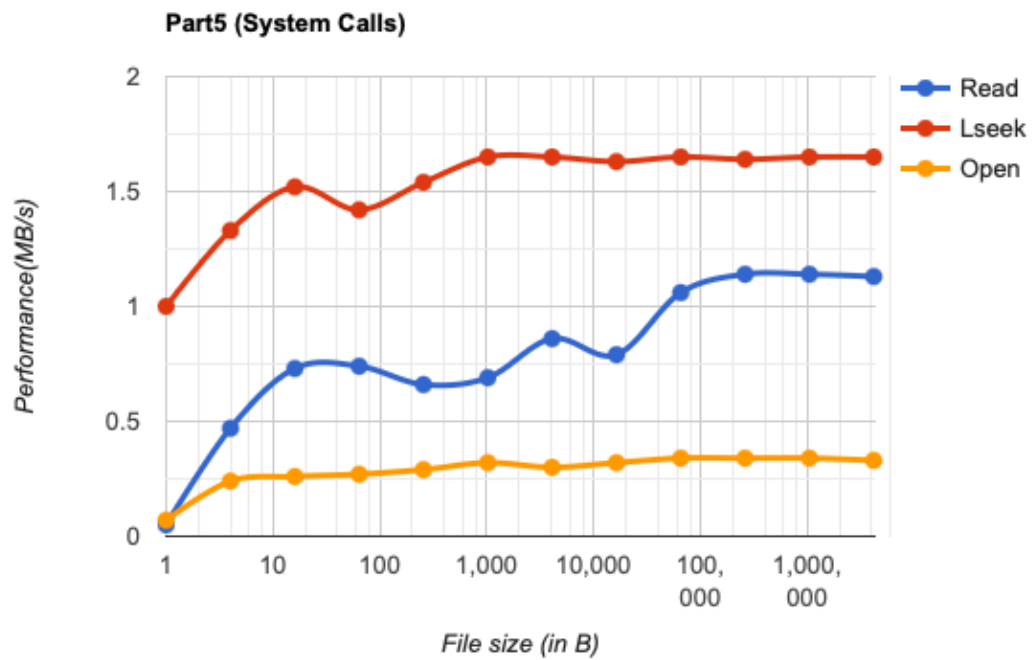
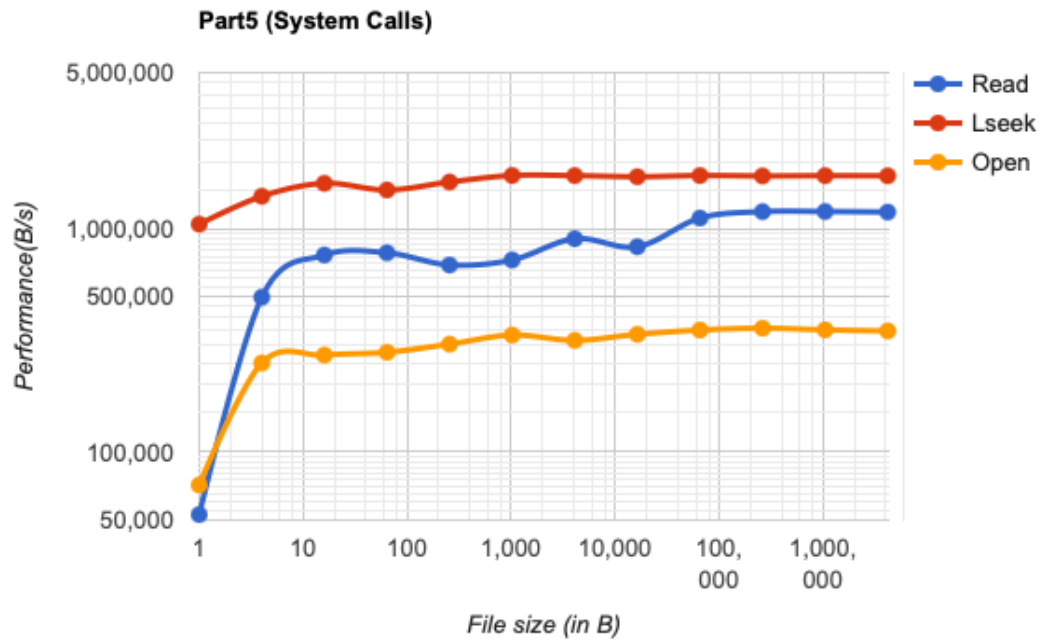
Table values for Read, Lseek, Open Systemcalls:

File Size	Speed (MiB/s)	Speed (B/s)	Operation Type
1	0.05	52428.80	Read Func
4	0.47	493447.53	Read Func
16	0.73	762600.73	Read Func
64	0.74	780335.63	Read Func
256	0.66	688296.04	Read Func
1024	0.69	724644.39	Read Func
4096	0.86	902209.28	Read Func
16384	0.79	830667.69	Read Func
65536	1.06	1116418.67	Read Func
262144	1.14	1192946.80	Read Func
1048576	1.14	1194571.77	Read Func
4194304	1.13	1187860.70	Read Func

File Size	Speed (MiB/s)	Speed (B/s)	Operation Type
1	1.00	1048576.00	LSEEK Func
4	1.33	1398101.33	LSEEK Func
16	1.52	1597830.10	LSEEK Func
64	1.42	1491308.09	LSEEK Func
256	1.54	1619520.10	LSEEK Func
1024	1.65	1732540.26	LSEEK Func
4096	1.65	1729749.21	LSEEK Func
16384	1.63	1710418.32	LSEEK Func
65536	1.65	1731187.22	LSEEK Func
262144	1.64	1724110.20	LSEEK Func
1048576	1.65	1728657.33	LSEEK Func
4194304	1.65	1726558.34	LSEEK Func

File Size	Speed (MiB/s)	Speed (B/s)	Operation Type
1	0.07	71089.90	Open Func
4	0.24	250406.21	Open Func
16	0.26	271695.81	Open Func
64	0.27	279620.27	Open Func
256	0.29	304090.01	Open Func
1024	0.32	334004.77	Open Func
4096	0.30	316050.43	Open Func
16384	0.32	336662.14	Open Func
65536	0.34	351767.18	Open Func
262144	0.34	358523.74	Open Func
1048576	0.34	352445.56	Open Func
4194304	0.33	348463.77	Open Func

Graphs:



Observation:

From the above two graphs, we can conclude that performance is better when lseek is used and then in read and then the open system call.

Performance:

Lseek > Read > Open.

Part-6

Try to optimize your program as much as you can to run as fast as it could.

- Find a good enough block size?
- Use multiple threads?
- Report both cached and non-cached performance numbers.

Block Size:

- I have used the formula to calculate block size dynamically based on the file size using the formula **(long)(6 * pow(fsize, 0.45))**.
- The calculated block size is then divided by the number of threads (**blockSize = blockSize/numThreads**) to distribute the workload among threads.
- The block size is adjusted to be a multiple of 4 (**blockSize = blockSize – blockSize % 4**) for efficient processing with **unsigned int**.
- The final block size is used for dividing the file into chunks for each thread to process.

Multiple Threads:

- Our program uses the POSIX threads library (**pthread.h**) to create multiple threads.
- The number of threads is set to 2 (**int numThreads = 2;**), and a loop is used to spawn and manage these threads.
- Each thread reads a specific portion of the file concurrently, contributing to an overall parallelized computation.

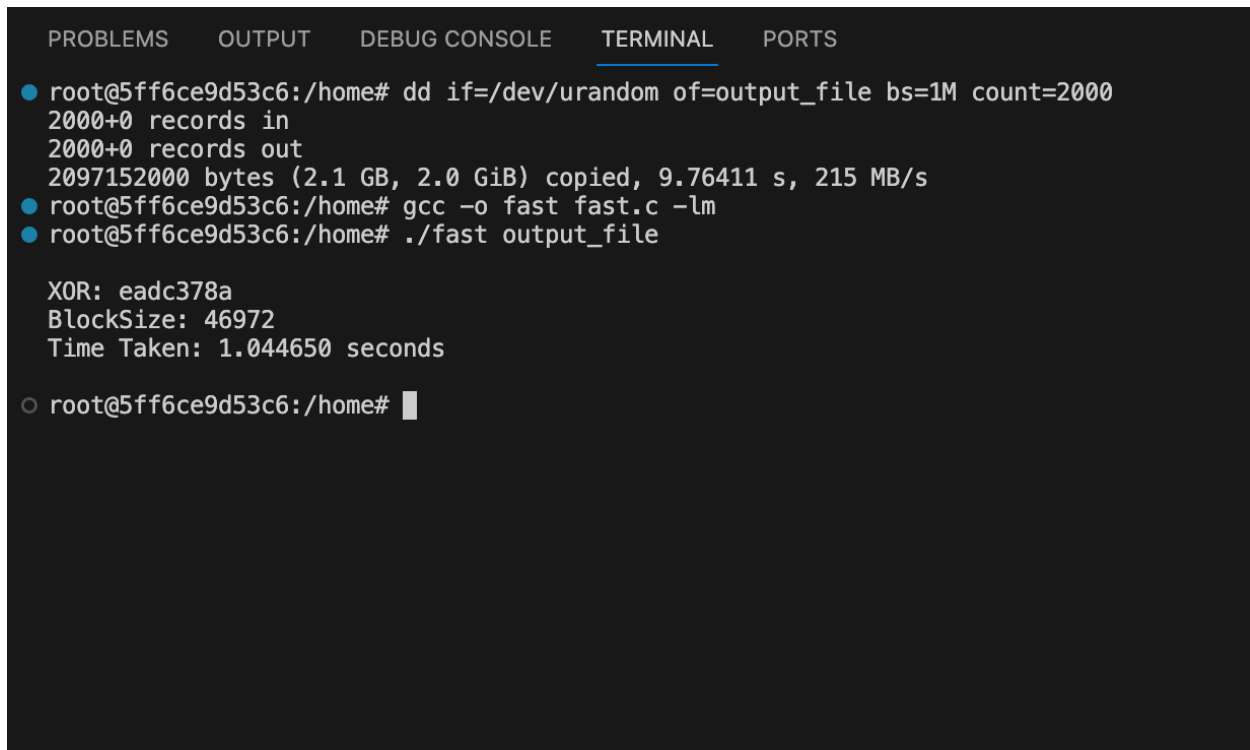
Cached and Non-Cached Performance:

Our code measures the total runtime using the **now ()** function and calculates the performance in terms of megabytes per second (**performance = (fsize) / (1024 * 1024 * runtime)**).

OUTPUT:

The following screenshot includes:

1. Output of XOR.
2. Block size Used.
3. Time taken to read the file.



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

● root@5ff6ce9d53c6:/home# dd if=/dev/urandom of=output_file bs=1M count=2000
2000+0 records in
2000+0 records out
2097152000 bytes (2.1 GB, 2.0 GiB) copied, 9.76411 s, 215 MB/s
● root@5ff6ce9d53c6:/home# gcc -o fast fast.c -lm
● root@5ff6ce9d53c6:/home# ./fast output_file

XOR: eadc378a
BlockSize: 46972
Time Taken: 1.044650 seconds

○ root@5ff6ce9d53c6:/home#
```

BY

HARI KISHAN REDDY ABBASANI (ha2755)
BHARANI KUMAR REDDY (bb3722)