OPERATINGSYSTEMS FINALPROJECT

By

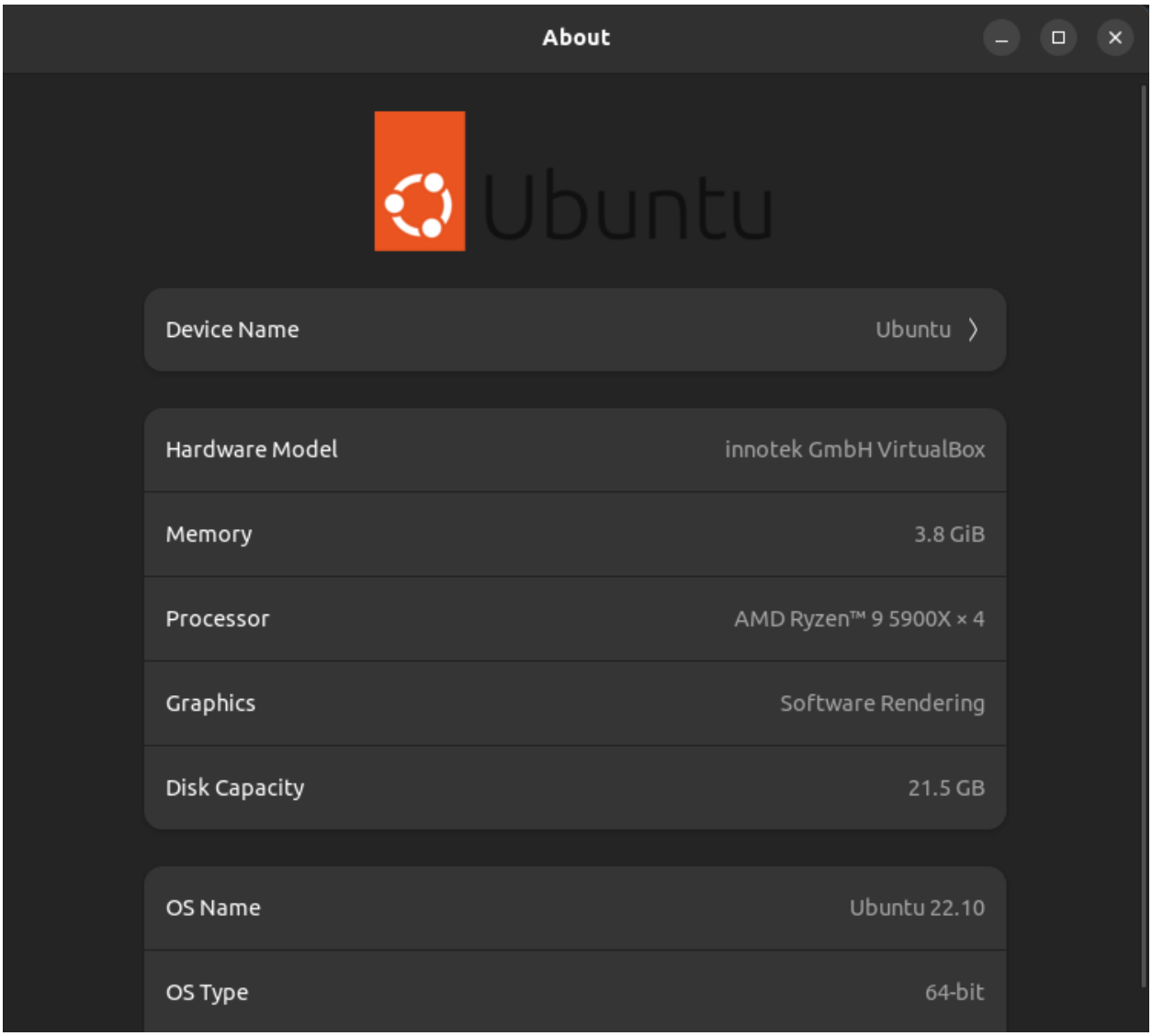
Thejus Rao – th2833@nyu.edu

Stephanie Hou - sh6403@nyu.edu

Table of Contents

Working environment

All tests in this document were performed on a 64-bit version of Ubuntu 22.10 in a virtual machine environment. The system has the following specifications – 4GB of RAM, disk capacity of 20GB, and a Ryzen 5900X x86 64-bit processor with 4 threads.

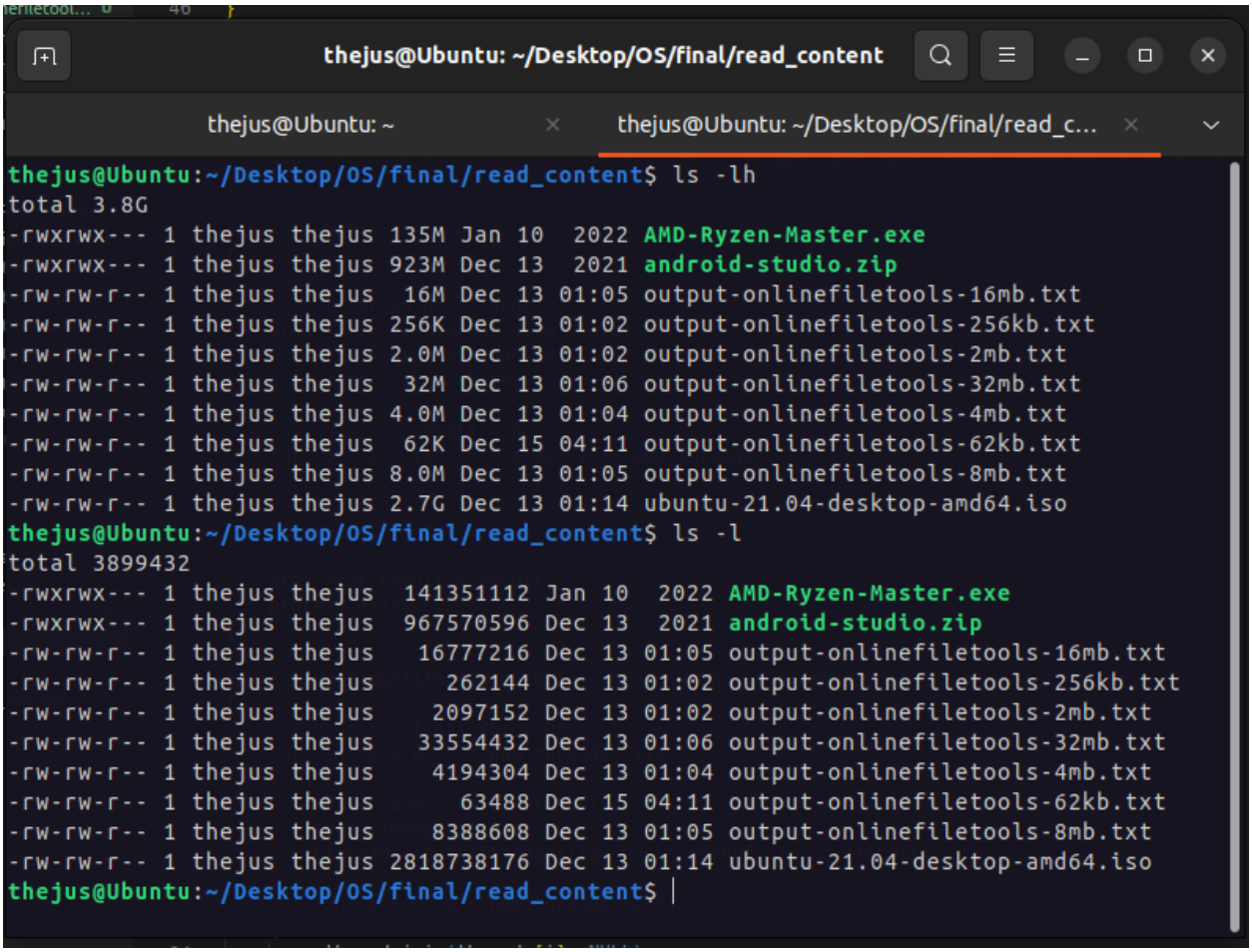


Experiments

A bunch of randomly generated text files, in addition to

- `ubuntu-21.04-desktop-amd64.iso` (provided in the question statement),
- `AMD-Ryzen-Master.exe`,
- `android-studio.zip`

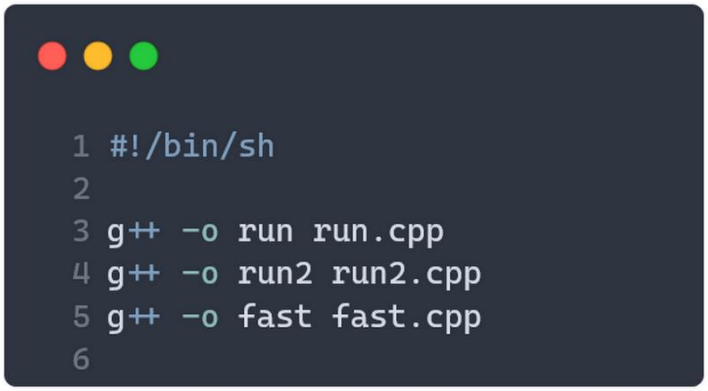
were used to check the performance of the program. The file sizes are given below (`ls -lh` lists the sizes in megabytes for reference, `ls -l` lists it in bytes):



The Build Code

Since the question statement quoted we could use either C/C++, and bash for the build script, we used C++ to code `run`, `run2`, and `fast` bash for the `build.sh` file.

The `build.sh` file:



```
#!/bin/sh

g++ -o run run.cpp
g++ -o run2 run2.cpp
g++ -o fast fast.cpp

```

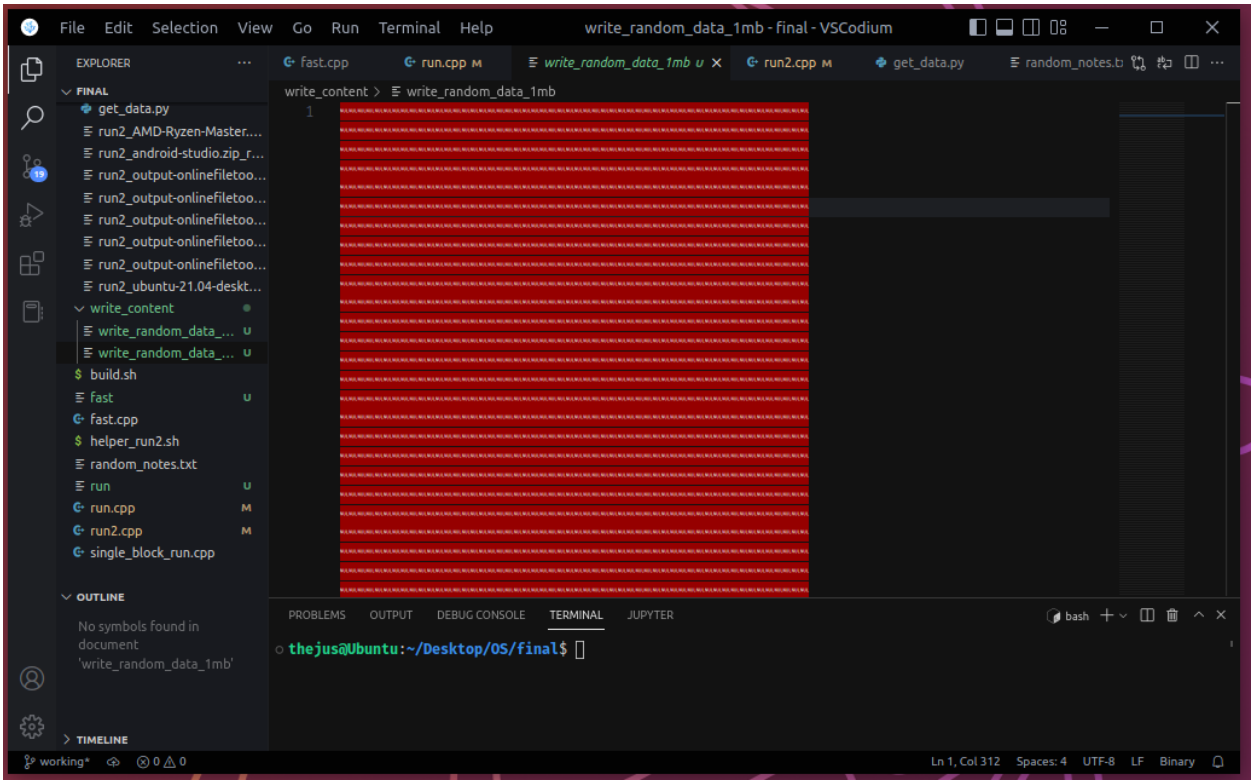
Part one - Basics

The program is run according to the following parameters: `./run <filename> [-r|-w] <block_size> <block_count>`. Upon running we get the following outputs:

In write mode

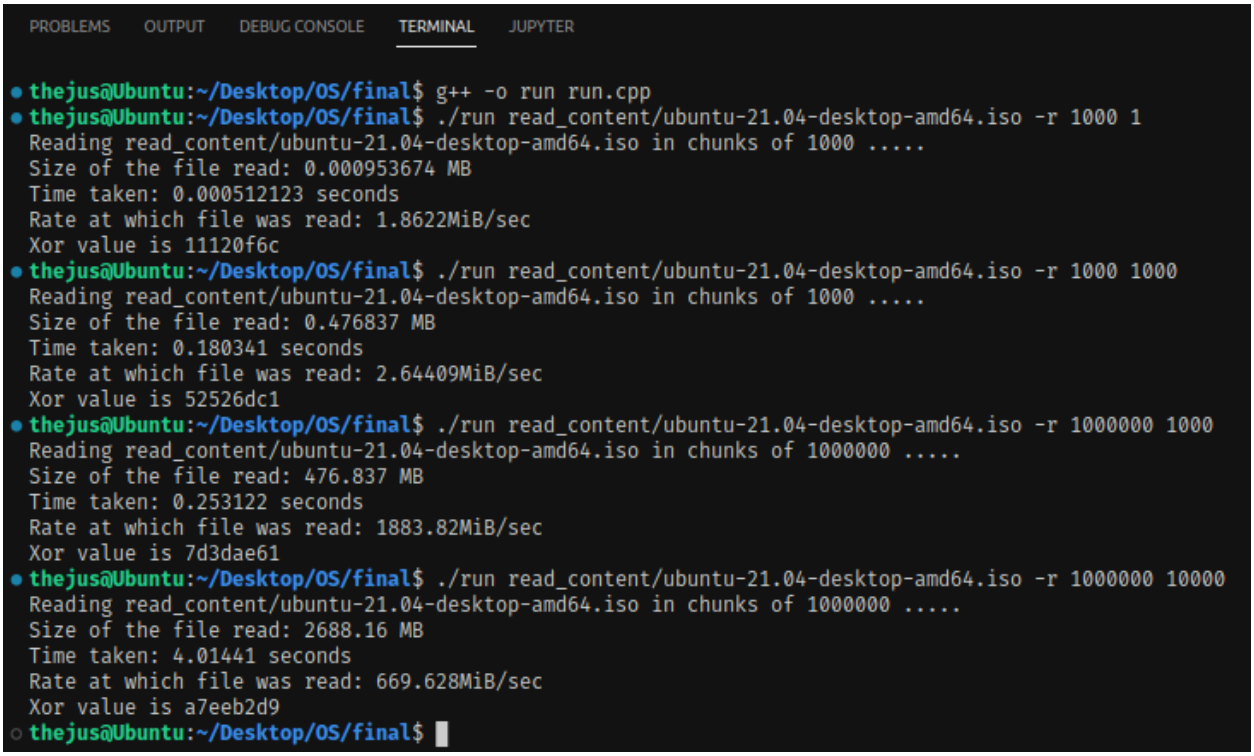
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER

thejus@Ubuntu:~/Desktop/OS/final$ g++ -o run run.cpp
thejus@Ubuntu:~/Desktop/OS/final$ ./run write_content/write_random_data_1mb -w 1000 1000
Size of the file that was written: 0.953674 MB
Time taken: 0.000729799 seconds
Rate at which file was written: 1306.76MiB/sec

thejus@Ubuntu:~/Desktop/OS/final$ ./run write_content/write_random_data_1gb -w 10000 100000
Size of the file that was written: 953.674 MB
Time taken: 1.30925 seconds
Rate at which file was written: 728.413MiB/sec

thejus@Ubuntu:~/Desktop/OS/final$ ls -l write_content/
total 977548
-rw-rw-r-- 1 thejus thejus 1000000000 Dec 17 23:42 write_random_data_1gb
-rw-rw-r-- 1 thejus thejus    1000000 Dec 17 23:41 write_random_data_1mb
thejus@Ubuntu:~/Desktop/OS/final$
```

The contents of the file will be garbage values, i.e the contents inside memory. Here is a sample (binary data inside, so rendering is flawed):



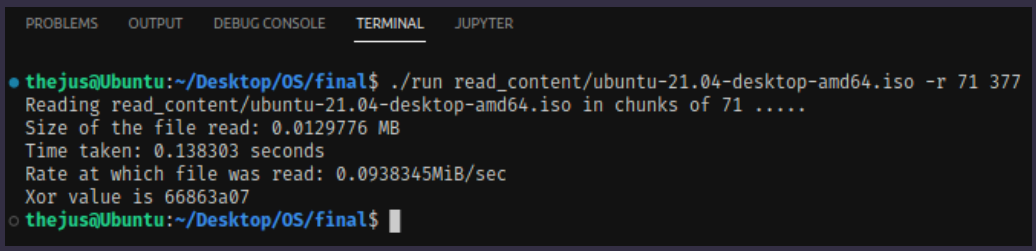
Note: In write mode, we are using a `char` array, unlike in read mode where we use an `int` array. This is because we would cast the `int` buffer into a `char *` anyway, as `the fstream object write` function takes only a `char*` type argument (along with bytes to write as the next argument)

In read mode



Here, we are varying different block counts and block sizes. We can see that **the xor values will be different depending on the number of bytes it reads from the file**. `block_size*block_count` will give the total number of bytes read. If `block_size*block_count > size-of-file`, then it will read `size-of-file` bytes. We can see above from the last execution of `./run` that the xor value of `a7eeb2d9` is the correct, despite `block_size*block_count` slightly exceeding the size of the file.

Note: If `block_size` is not divisible by four, then it will default to rounding upwards. The program works fine with no issues, this case is handled.



Code – **run.cpp**

```cpp
#include <iostream>
#include <cstring>
#include <fstream>
#include <string>
#include <sys/time.h>
#include <pthread.h>

using namespace std;

int num_threads = 4;
char *buffer;
unsigned int *buf;
pthread_t *threads;

struct thread_data
{
    unsigned int thread_id;
    unsigned int size;
    unsigned int xor_result;
};

double now()
{
    struct timeval tv;
    gettimeofday(&tv, 0);
    return tv.tv_sec + tv.tv_usec / 1000000.0;
}

double get_rate(double size, double start, double end)
```

```
{
    return size / ((end - start) * 1024 * 1024);
}

void perror(string s)
{
    cout << "Error! " << s << endl;
    exit(0);
}
void print_performance(double size, double start, double end, unsigned int
block_count, unsigned int final_xor)
{
    cout << "Number of blocks read: " << block_count << " blocks" << endl;
    cout << "Size of the file read: " << (size / (1024 * 1024)) << " MB" << endl;
    cout << "Time taken: " << (end - start) << " seconds" << endl;
    cout << "Rate at which file was read: " << get_rate(size, start, end) <<
"MiB/sec" << endl;
    printf("Xor value is %08x", final_xor);
}

void print_performance_w(double size, double start, double end)
{
    cout << "Size of the file that was writte: " << (size / (1024 * 1024)) << " MB"
<< endl;
    cout << "Time taken: " << (end - start) << " seconds" << endl;
    cout << "Rate at which file was written: " << get_rate(size, start, end) <<
"MiB/sec" << endl;
}

void *xorbuf(void *arg)
{

    struct thread_data *args;
    args = (struct thread_data *)arg;
    long tid = args->thread_id;
    long size = args->size;

    unsigned int result = 0;
    for (int i = tid; i < size; i += num_threads)
    {
        // if(buf[i]!=0) cout<<buf[i]<<" thread "<<tid<<" "<<i<<endl;
        result ^= buf[i];
    }
    args->xor_result = result;
    pthread_exit(NULL);
```

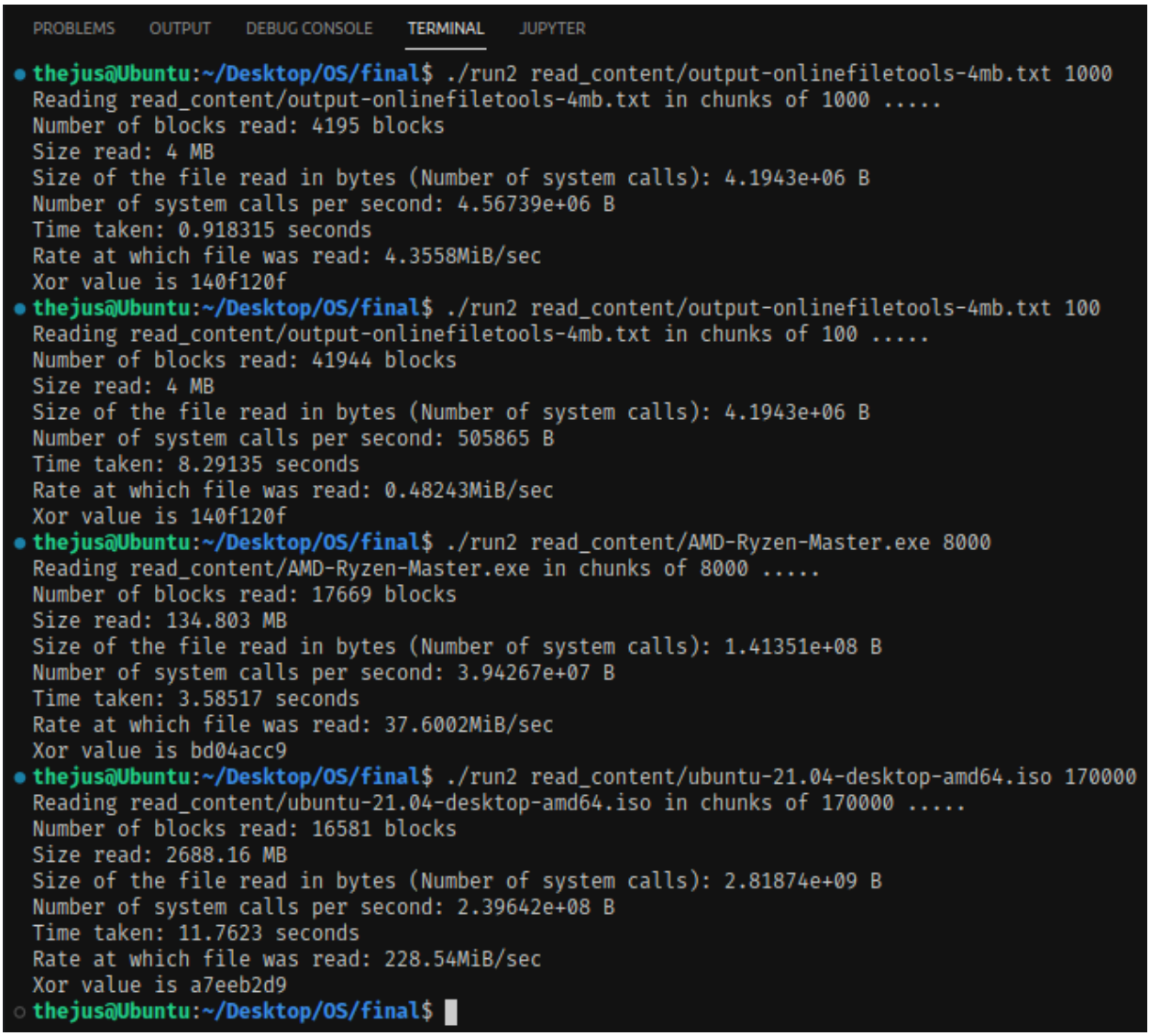
```
}

unsigned int multithreaded_xor(unsigned int no_of_elements, struct thread_data td[])
{
    unsigned int final_xor = 0;
    for (int i = 0; i < num_threads; i++)
    {
        td[i].size = no_of_elements;
        td[i].thread_id = i;
        pthread_create(&threads[i], NULL, xorbuf, (void *)&td[i]);
    }
    for (int i = 0; i < num_threads; i++)
    {
        pthread_join(threads[i], NULL);
    }
    for (int i = 0; i < num_threads; i++)
    {
        final_xor = final_xor ^ td[i].xor_result;
    }
    return final_xor;
}

int main(int argc, char *argv[])
{
    unsigned int block_size = 0, block_count = 0, size=0;
    bool read_mode = false, write_mode = false;
    double start, end;
    string file_name = "";
    struct thread_data td[num_threads];

    if (argc != 5)
        perror("Too few arguments!");

    else
    {
        string s = argv[2];
        file_name = argv[1];
        read_mode = ("-r" == s || "-R" == s);
        write_mode = ("-w" == s || "-W" == s);
        block_size = (unsigned int)stoi(argv[3]);
        block_count = (unsigned int)stoi(argv[4]);
        // cout<<"---"<<block_size<<"--"<<block_count<<endl;
    }

    srand(time(NULL));
```

```
    size = block_count * block_size;

    if (read_mode)
    {
        unsigned int no_of_blocks_elapsed = 0, final_xor = 0, size_of_buf;
        unsigned int no_of_elements = (unsigned int)(block_size / sizeof(int) +
block_size % sizeof(int));
        size_of_buf = no_of_elements * sizeof(int);
        buf = (unsigned int *)malloc(size_of_buf);

        start = now();
        ifstream object;
        object.open(file_name, ios::binary);
        if (object.fail())
            cout << "Can't read file " << file_name;
        else
        {
            cout << "Reading " << file_name << " in chunks of " << block_size << "
..... " << endl;
            threads = (pthread_t *)malloc(sizeof(pthread_t) * num_threads);
            if (!threads)
                perror("out of memory for threads!");

            while (object.read((char *)buf, size_of_buf))
            {
                final_xor ^= multithreaded_xor(no_of_elements, td);
                if (block_size * ++no_of_blocks_elapsed >= size)
                    break;
            }
            if (object.gcount() < block_size && object.gcount() > 0)
            {
                final_xor ^= multithreaded_xor(object.gcount() / sizeof(unsigned
int), td);
            }
            // cout<<"----"<<object.gcount()<<endl;
            end = now();
            print_performance(size, start, end, block_count, final_xor);
        }
    }
    else if (write_mode)
    {
        start = now();
        ofstream object(file_name);
        buffer = new char[size];
        for (unsigned int i = 0; i < block_count; i++)
```

```
        {
            object.write(buffer, size);
        }
        end = now();

        print_performance_w(size, start, end);
    }
    else
    {
        perror("Please check your arguments and specify if its -r/-w");
    }
    cout << "\n";
    return 0;
}
```

Part two - Measurement

The program is run according to the following parameters: `./run2 <filename> [-r|-w] <block_size>` Upon running we get the following outputs:



The program takes in a file and a given `block_size`, and reads accordingly. It reads the entire file, it finishes execution and gives us the output, as shown above.

Code – run2.cpp

```
#include <iostream>
#include <cstring>
#include <fstream>
#include <string>
#include <sys/time.h>
#include <pthread.h>

using namespace std;

int num_threads = 4;
char *buffer;
unsigned int *buf;
pthread_t *threads;

struct thread_data
{
    unsigned int thread_id;
    unsigned int size;
    unsigned int xor_result;
};

double now()
{
    struct timeval tv;
    gettimeofday(&tv, 0);
    return tv.tv_sec + tv.tv_usec / 1000000.0;
}

double get_rate(double size, double start, double end)
{
    return size / ((end - start) * 1024 * 1024);
}

void print_error(string s)
{
    cout << "Error! " << s << endl;
    exit(EXIT_FAILURE);
}

void print_performance(double size, double start, double end, unsigned int
block_count, unsigned int final_xor, int flag)
{
    cout << "Number of blocks read: " << block_count << " blocks";
```

```
    if (flag == 1)
    {
        cout << " (file not fully read, program exceeded time limit of 15s and was
terminated)";
    }
    cout << endl;
    cout << "Size read: " << (size / (1024 * 1024)) << " MB" << endl;
    cout << "Size of the file read in bytes (Number of system calls): " << (size) <<
" B" << endl;
    cout << "Number of system calls per second: " << (size/(end-start)) << " B" <<
endl;
    cout << "Time taken: " << (end - start) << " seconds" << endl;
    cout << "Rate at which file was read: " << get_rate(size, start, end) <<
"MiB/sec" << endl;
    printf("Xor value is %08x", final_xor);
}

void *xorbuf(void *arg)
{

    struct thread_data *args;
    args = (struct thread_data *)arg;
    long tid = args->thread_id;
    long size = args->size;

    unsigned int result = 0;
    for (int i = tid; i < size; i += num_threads)
    {
        // if(buf[i]!=0) cout<<buf[i]<<" thread "<<tid<<" "<<i<<endl;
        result ^= buf[i];
    }
    args->xor_result = result;
    pthread_exit(NULL);
}

unsigned int multithreaded_xor(unsigned int no_of_elements, struct thread_data td[])
{
    unsigned int final_xor = 0;
    for (int i = 0; i < num_threads; i++)
    {
        td[i].size = no_of_elements;
        td[i].thread_id = i;
        pthread_create(&threads[i], NULL, xorbuf, (void *)&td[i]);
    }
    for (int i = 0; i < num_threads; i++)
```

```
    {
        pthread_join(threads[i], NULL);
    }
    for (int i = 0; i < num_threads; i++)
    {
        final_xor = final_xor ^ td[i].xor_result;
    }
    return final_xor;
}

int main(int argc, char *argv[])
{
    unsigned int block_size = 0, block_count = 0, final_xor = 0, size = 0;
    double start, end;
    string file_name = "";
    struct thread_data td[num_threads];

    if (argc != 3)
        print_error("Check arguments!");

    else
    {
        file_name = argv[1];
        block_size = (unsigned int)stoi(argv[2]);
    }

    srandom(time(NULL));

    unsigned int no_of_elements = (unsigned int)((block_size + sizeof(int) - 1) /
sizeof(int));
    unsigned int size_of_buf = no_of_elements * sizeof(int);
    int flag = 0;
    buf = (unsigned int *)malloc(size_of_buf);
    // memset(buf,0,no_of_elements*sizeof(int));
    // cout<<no_of_elements<<" ----- "<<size<<" ----- "<<sizeof(buf)<<"-----
"<<(no_of_elements * sizeof( unsigned int))<<endl;

    start = now();
    ifstream object;
    object.open(file_name, ios::binary);
    if (object.fail())
        print_error("Cannot read file!");
    else
    {
```

```
        cout << "Reading " << file_name << " in chunks of " << block_size << " .....
" << endl;
        // cout<<"buf4-----"<<buf[4]<<endl;
        threads = (pthread_t *)malloc(sizeof(pthread_t) * num_threads);
        if (!threads)
            perror("out of memory for threads!");

        while (object.read((char *)buf, size_of_buf))
        {
            final_xor ^= multithreaded_xor(no_of_elements, td);
            block_count++;
            size += object.gcount();
            // if ((end = now()) - start > 15)
            // {
            //     flag = 1;
            //     break;
            // }
        }
        size = block_count*block_size;
        if (object.gcount() < block_size && object.gcount() > 0 && flag == 0)
        {
            final_xor ^= multithreaded_xor(object.gcount() / sizeof(unsigned int),
td);
            block_count++;
            size += object.gcount();
        }
        // cout<<"----"<<object.gcount()<<endl;
        end = now();
        print_performance(size, start, end, block_count, final_xor, flag);
    }

    cout << "\n";
    return 0;
}
```

Part three – Raw Performance

In order to avoid program biases and variations, and mostly in order to test out different scenarios, we used different files. The files we used are:

```
bytes          file   name
================================================
2097152        2.0M   output-onlinefiletools-2mb.txt
4194304        4.0M   output-onlinefiletools-4mb.txt
8388608        8.0M   output-onlinefiletools-8mb.txt
16777216        16M   output-onlinefiletools-16mb.txt
33554432        32M   output-onlinefiletools-32mb.txt
141351112      135M   AMD-Ryzen-Master.exe
967570596      923M   android-studio.zip
27 2818738176  2.7G   ubuntu-21.04-desktop-amd64.iso

```

The graphs for performance (MB/sec) vs `block_size` (bytes) for each of the files are given below. X-axis corresponds to the `block_size` and Y-axis gives the performance.

For each of the above files, we have collected the following data:

- Block size in bytes
- Rate in MB/sec
- Time taken in seconds
- Size in Megabytes

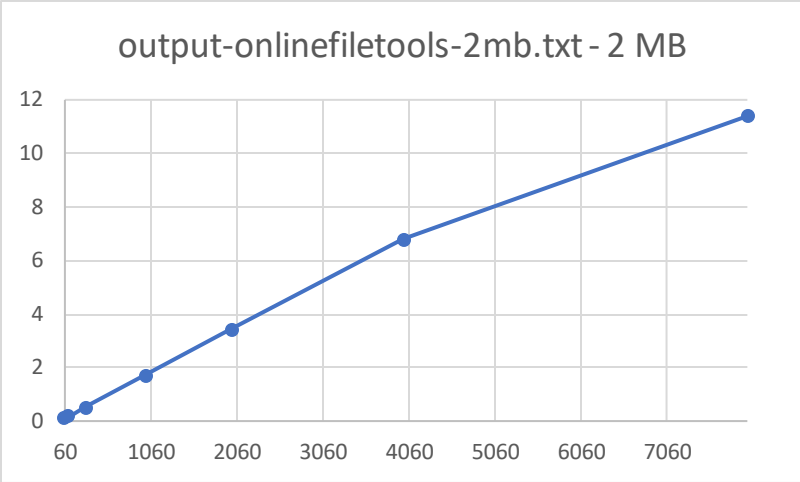
For reference, 1 MB = 1024x1024 bytes.

Care has been taken to ensure other I/O tasks were not taking place at the same time, CPU was free the time of operation and all the available memory was not otherwise occupied in order to obtain accurate results.

Testing Files

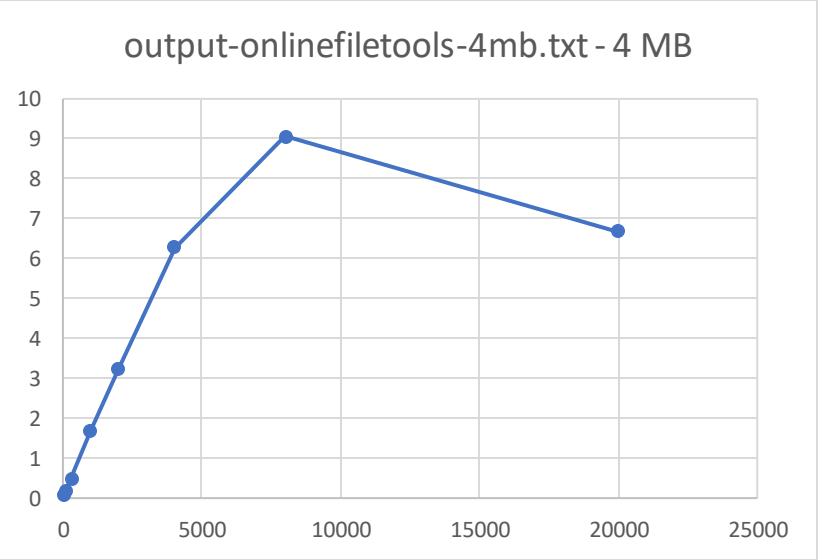
run2_output-onlinefiletools-2mb.txt

Block Size in Bytes	Rate in MB/sec	Time taken in sec
60	0.0980585	20.396
100	0.164238	12.1774
300	0.500106	3.99915
1000	1.7063	1.17212
2000	3.40903	0.586677
4000	6.78633	0.29471
8000	11.3935	0.175538



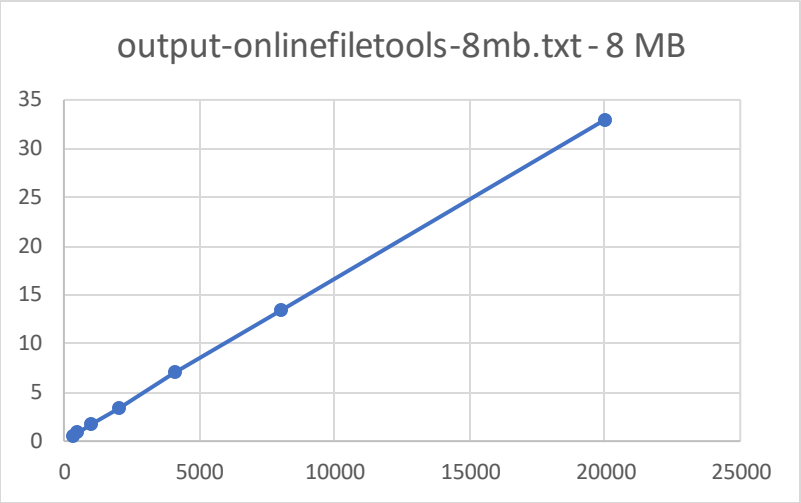
run2_output-onlinefiletools-4mb.txt

Block Size in Bytes	Rate in MB/sec	Time taken in sec
60	0.101735	39.318
100	0.173402	23.0678
300	0.502205	7.96487
1000	1.69449	2.36059
2000	3.24666	1.23203
4000	6.27926	0.637018
8000	9.06908	0.441059
20000	6.66799	0.599881



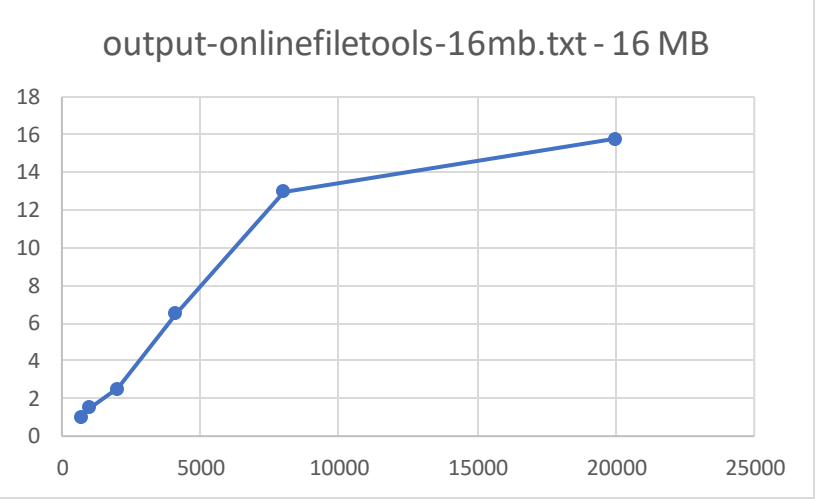
run2_output-onlinefiletools-8mb.txt

Block Size in Bytes	Rate in MB/sec	Time taken in sec
300	0.504379	15.8611
512	0.840117	9.52248
1000	1.68361	4.7517
2000	3.35301	2.38592
4096	7.01725	1.14005
8000	13.4298	0.595691
20000	32.9639	0.24269



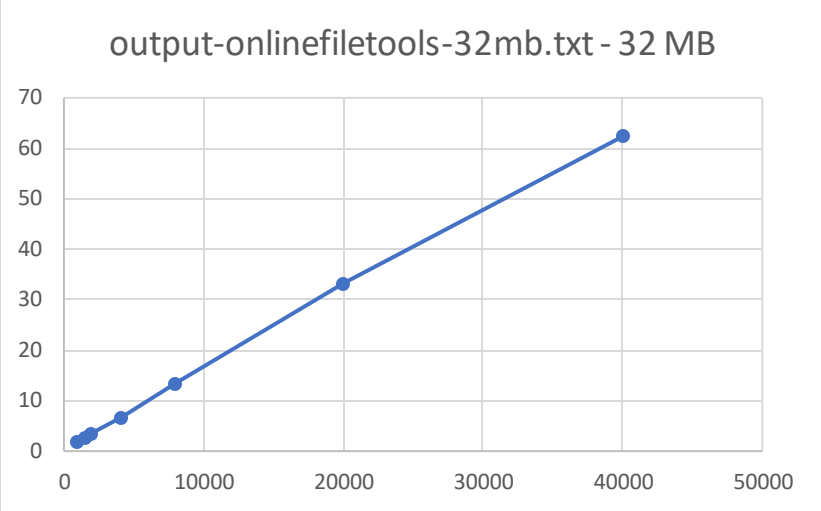
run2_output-onlinefiletools-16mb.txt

Block Size in Bytes	Rate in MB/sec	Time taken in sec
680	1.02944	15.5424
1000	1.51158	10.585
2000	2.5552	6.26175
4096	6.49084	2.46501
8000	12.9742	1.23321
20000	15.7881	1.01342



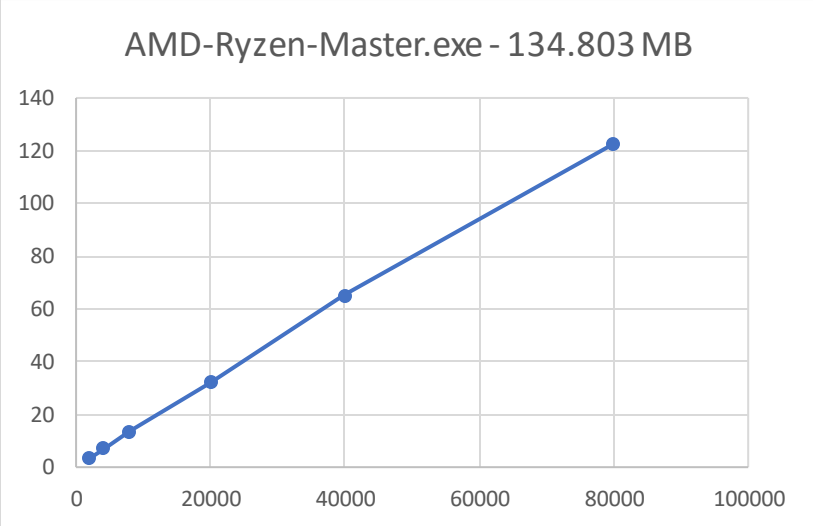
run2_output-onlinefiletools-32mb.txt

Block Size in Bytes	Rate in MB/sec	Time taken in sec
1000	1.69476	18.8818
1500	2.50447	12.7772
2000	3.46356	9.23905
4096	6.66941	4.79802
8000	13.3097	2.40427
20000	33.3101	0.96067
40000	62.3464	0.513261



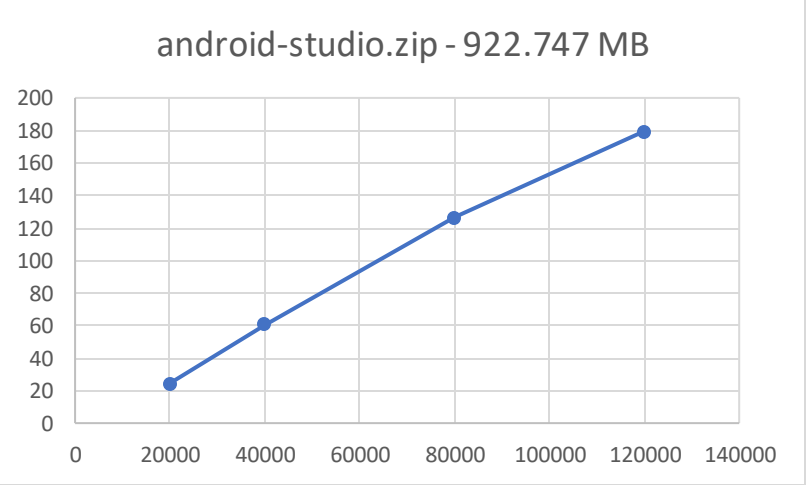
AMD-Ryzen-Master.exe

Block Size in Bytes	Rate in MB/sec	Time taken in sec
2000	3.29764	40.8786
4096	7.03181	19.1704
8000	13.3513	10.0966
20000	32.188	4.18798
40000	64.9727	2.07476
80000	122.63	1.09926



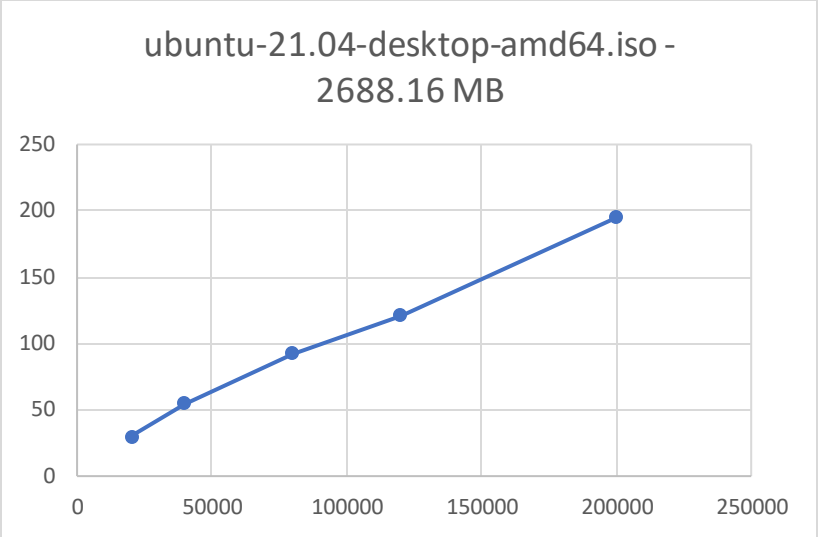
android-studio.zip

Block Size in Bytes	Rate in MB/sec	Time taken in sec
20000	24.531	37.6156
40000	60.504	15.251
80000	126.447	7.2975
120000	179.527	5.13987



ubuntu-21.04-desktop-amd64.iso

Block Size in Bytes	Rate in MB/sec	Time taken in sec
20000	30.1109	89.2754
40000	54.8533	49.0063
80000	92.5068	29.059
120000	120.994	22.2174
200000	194.823	13.7979



Observations

We can see from the above data that the performance increases almost linearly with block size. There were a couple of variations, such as:

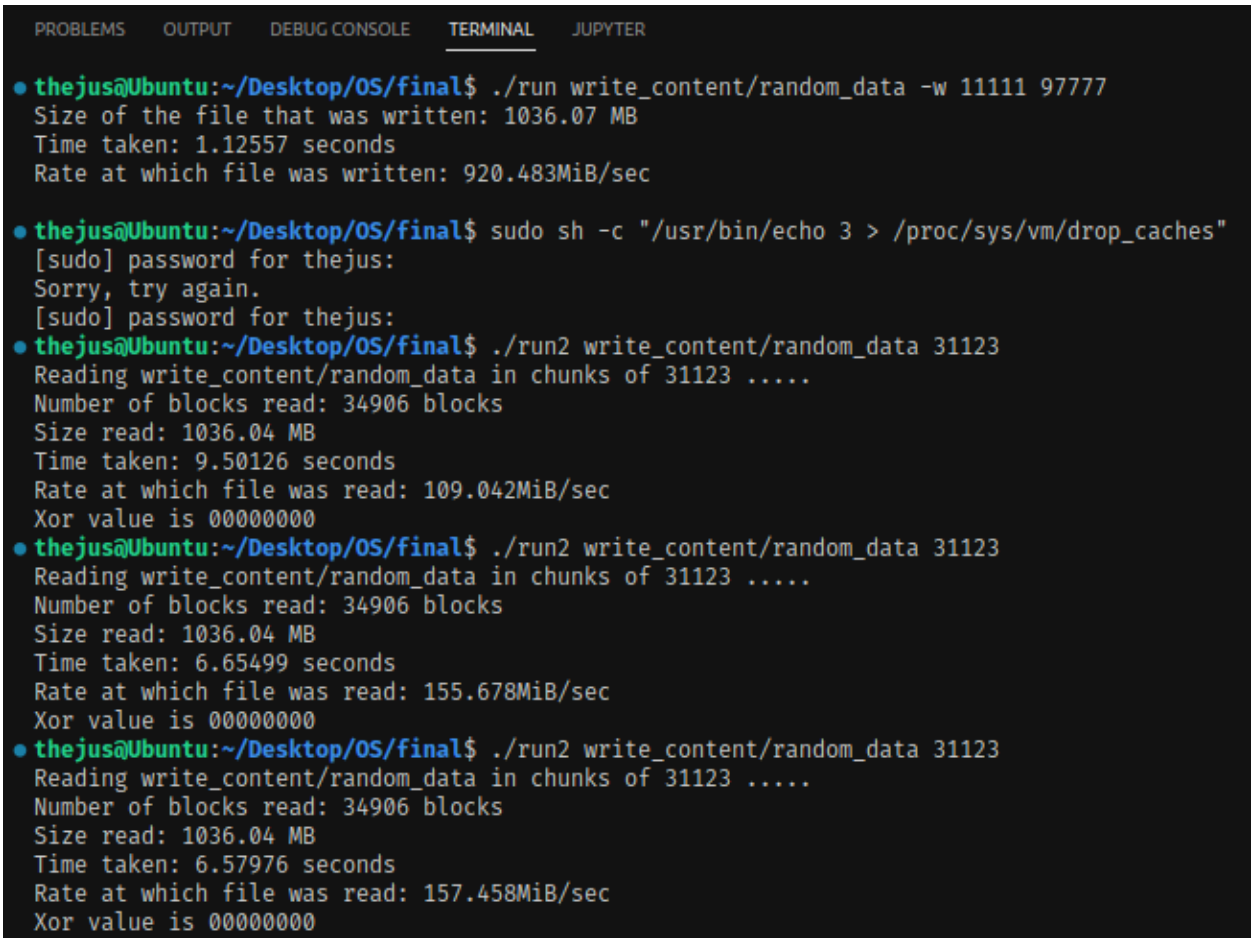
- `run2_output-onlinefiletools-4mb.txt` - We can chalk this down to run-to-run variance, as the files bigger/smaller and block sizes bigger/smaller do not exhibit the same behaviour. Its quite likely that the system was performing that some other I/O task and as a result performance dipped.
- `run2_output-onlinefiletools-16mb.txt` - This is likely either the same case as above, or could be that performance plateau over time with increasing block sizes, or a combination of the two. Or perhaps the file was in cache at the block size of 8000, resulting in diminishing gains on increasing block size.

There are a couple of things to keep in mind, such as

- Higher and higher values of block sizes lead to the program using more memory than it should, and also causes memory fragmentation.
- At some point increasing the block size does not linearly improve performance, i.e. we get diminishing returns.

Part four - Caching

First, we create a file of variable size (say, 1GB) using `./run`. Then, since we are in a linux environment, instead of rebooting, we clear the cache using the given command and read the file.



The first time it reads the file, it takes about 9.5 seconds to read the file. The second time, it takes about 6.65 seconds. If we were to increase the block size from 31,123 to 70,000, then:

```
thejus@Ubuntu:~/Desktop/OS/final$ sudo sh -c "/usr/bin/echo 3 > /proc/sys/vm/drop_caches"
thejus@Ubuntu:~/Desktop/OS/final$ ./run2 write_content/random_data 70000
Reading write_content/random_data in chunks of 70000 .....
Number of blocks read: 15521 blocks
Size read: 1036.07 MB
Time taken: 6.14644 seconds
Rate at which file was read: 168.565MiB/sec
Xor value is 00000000
thejus@Ubuntu:~/Desktop/OS/final$ ./run2 write_content/random_data 70000
Reading write_content/random_data in chunks of 70000 .....
Number of blocks read: 15521 blocks
Size read: 1036.07 MB
Time taken: 3.08563 seconds
Rate at which file was read: 335.773MiB/sec
Xor value is 00000000
thejus@Ubuntu:~/Desktop/OS/final$ ./run2 write_content/random_data 70000
Reading write_content/random_data in chunks of 70000 .....
Number of blocks read: 15521 blocks
Size read: 1036.07 MB
Time taken: 2.99522 seconds
Rate at which file was read: 345.909MiB/sec
Xor value is 00000000
```

The first time it reads the file, it takes about 6.146 seconds to read the file. The second time, it takes about 3.08 seconds, and the third, 2.99 seconds. That's an improvement of a factor of more than 2X. On increasing it from 70,000 to 311,232, we get:

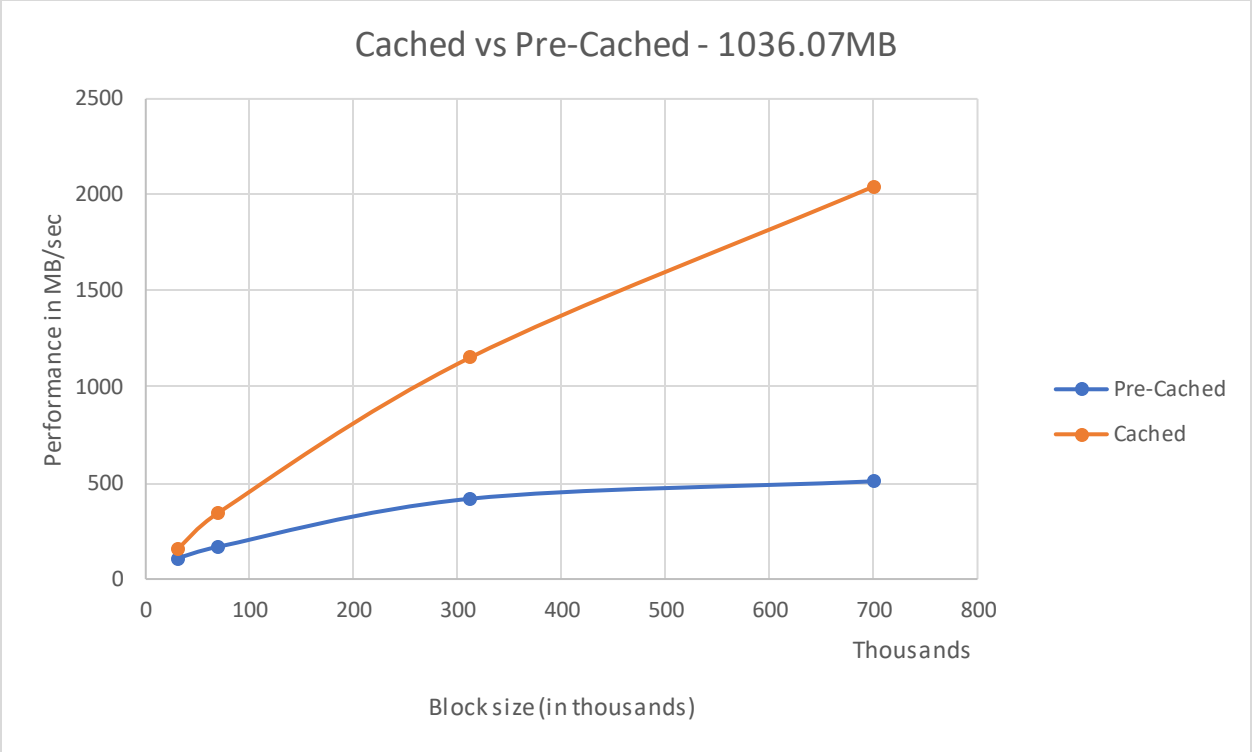
```
thejus@Ubuntu:~/Desktop/OS/final$ sudo sh -c "/usr/bin/echo 3 > /proc/sys/vm/drop_caches"
thejus@Ubuntu:~/Desktop/OS/final$ ./run2 write_content/random_data 311232
Reading write_content/random_data in chunks of 311232 .....
Number of blocks read: 3491 blocks
Size read: 1036.07 MB
Time taken: 2.48053 seconds
Rate at which file was read: 417.682MiB/sec
Xor value is 00000000
thejus@Ubuntu:~/Desktop/OS/final$ ./run2 write_content/random_data 311232
Reading write_content/random_data in chunks of 311232 .....
Number of blocks read: 3491 blocks
Size read: 1036.07 MB
Time taken: 0.900104 seconds
Rate at which file was read: 1151.06MiB/sec
Xor value is 00000000
```

It takes 2.48 seconds on the first try, and on the second it takes 0.9 seconds. That's an even bigger factor compared to 2X. Finally, on increasing it to 700,000 blocks, we get:

```
thejus@Ubuntu:~/Desktop/OS/final$ ./run2 write_content/random_data 700000
Reading write_content/random_data in chunks of 700000 .....
Number of blocks read: 1553 blocks
Size read: 1036.07 MB
Time taken: 2.03629 seconds
Rate at which file was read: 508.804MiB/sec
Xor value is 00000000
thejus@Ubuntu:~/Desktop/OS/final$ ./run2 write_content/random_data 700000
Reading write_content/random_data in chunks of 700000 .....
Number of blocks read: 1553 blocks
Size read: 1036.07 MB
Time taken: 0.537982 seconds
Rate at which file was read: 1925.85MiB/sec
Xor value is 00000000
thejus@Ubuntu:~/Desktop/OS/final$ ./run2 write_content/random_data 700000
Reading write_content/random_data in chunks of 700000 .....
Number of blocks read: 1553 blocks
Size read: 1036.07 MB
Time taken: 0.507029 seconds
Rate at which file was read: 2043.42MiB/sec
Xor value is 00000000
```

	Pre-cached		Cached	
Block size	**Performance in MB/sec**	**Time taken in sec**	**Performance in MB/sec2**	**Time taken in sec3**
31123	109.0142	9.50126	157.458	6.57926
70000	168.565	6.14644	345.909	2.99522
311232	417.682	2.48053	1151.06	0.900104
700000	508.804	2.03628	2043.42	0.507029

On plotting a block size vs performance graph for the file tested, for cached vs pre-cached performance, we get the below graph:



Trends

We can see from the above that the larger the block size is relative to the file, the greater the performance. However, while the benefits of larger block sizes slow down over time, the performance the file is cached increases linearly. We will attempt to replicate this with a file of smaller file size and see if the results hold true

Note: Why the Xor value of the above file gives 0

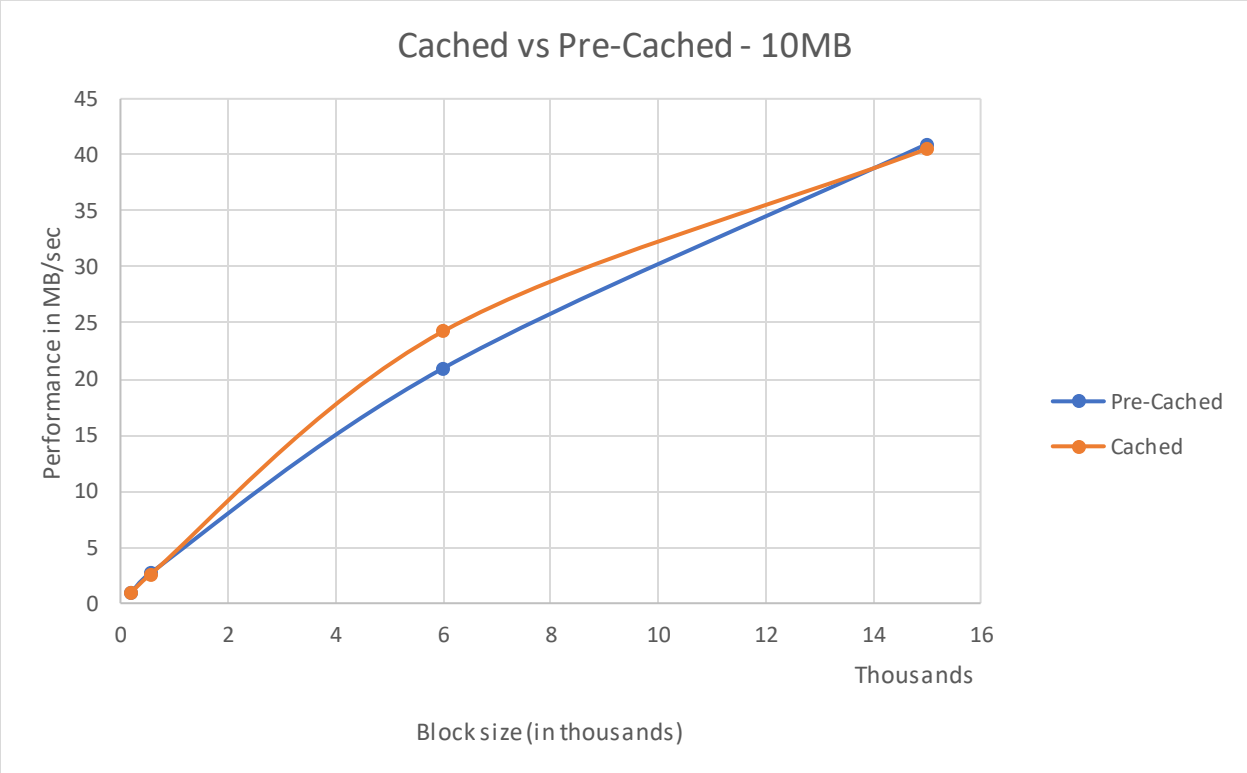
We are writing via a `char` array will be populated with the `null char` literal. The value of which is `\u0000`.

`char` is a primitive type. This means that it can never hold `null`, so like `int`, `double` and the rest, it needs some starting value. For `int` it's 0, for `char` it's `\u0000`, which evaluates to 0. And **`0 XOR 0 = 0`**

write_content/10mb_file

```
thejus@Ubuntu:~/Desktop/OS/final$ ./run write_content/10mb_file -w 1024 10240
Size of the file that was written: 10 MB
Time taken: 0.02391 seconds
Rate at which file was written: 418.234MiB/sec
```

	Pre-cached		Cached	
Block size	Performance in MB/sec	Time taken in sec	Performance in MB/sec2	Time taken in sec3
200	0.923859	10.6719	0.962077	10.3942
550	2.67721	3.83027	2.54597	3.91354
6000	20.9551	0.477211	24.2486	0.412395
15000	40.9621	0.244128	40.5319	0.246719



Observations

Cache seems to come into play only at larger block sizes. At smaller sizes it seems to have minimal effect on the performance of I/O operations. To prove this, we ran it at an extremely large block size and got a reduction in running time.

```
thejus@Ubuntu:~/Desktop/OS/final$ ./run2 write_content/10mb_file 150000
Reading write_content/10mb_file in chunks of 150000 .....
Number of blocks read: 70 blocks
Size read: 10 MB
Time taken: 0.081625 seconds
Rate at which file was read: 122.512MiB/sec
Xor value is 00000000
thejus@Ubuntu:~/Desktop/OS/final$ ./run2 write_content/10mb_file 150000
Reading write_content/10mb_file in chunks of 150000 .....
Number of blocks read: 70 blocks
Size read: 10 MB
Time taken: 0.0318131 seconds
Rate at which file was read: 314.335MiB/sec
Xor value is 00000000
thejus@Ubuntu:~/Desktop/OS/final$ ./run2 write_content/10mb_file 150000
Reading write_content/10mb_file in chunks of 150000 .....
Number of blocks read: 70 blocks
Size read: 10 MB
Time taken: 0.0164571 seconds
Rate at which file was read: 607.641MiB/sec
Xor value is 00000000
```

We went from 0.0816 seconds to 0.016457 seconds. That's an improvement factor of 4.958.

Extra credit

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

Because `/proc/sys/vm/drop_caches` is a sysctl file that has 3 modes.

- Mode "1" will free the page cache - buffered I/O, other OS data
- Mode "2" will free slab objects, dentries and inodes – dnodes are representations of folders, inodes represent files, both of which store cache.
- Mode "3" will essentially free up all of the above.

So, we use mode 3 to clear all cache. As for the rest of the command, the `/usr/bin/echo 3` can be replaced simply with "echo 3" , and sh tells linux that we can execute drop_caches using the old unix command interpreter, even if the extension is not sh (my ubuntu recognizes it as a "plain text document".

Part five – System Calls

We modified the `./run2` code in order to measure system call overhead. We also removed the XOR function call as that incurs a large performance penalty and hence not conducive to measuring system call performance. The logic for measuring system calls is given below.

```
        cout << "Reading " << file_name << " in chunks of " << block_size
            << " ..... " << endl;
        cout << "Checking overhead of read system call...." << endl;
        start = now();
        while (object.read((char *)buf, size_of_buf))
        {
            block_count++;
            read_sys_calls += 1;
        }
        end = now();
        print_performance(read_sys_calls, start, end, block_count, flag);

        cout << "\nChecking overhead of seek system call...." << endl;
        start = now();
        for (int i = 0; i < block_count; i++)
        {
            object.seekg((i>>2));
            seek_sys_calls++;
        }
        end = now();
        print_performance_seek(seek_sys_calls,start,end,flag);

```

The above program seeks purely to measure the system call performance. On running certain tests, we can see that on average, the `seekg` system call is almost twice as fast as `read`, meaning it incurs less overhead.

Output is given below

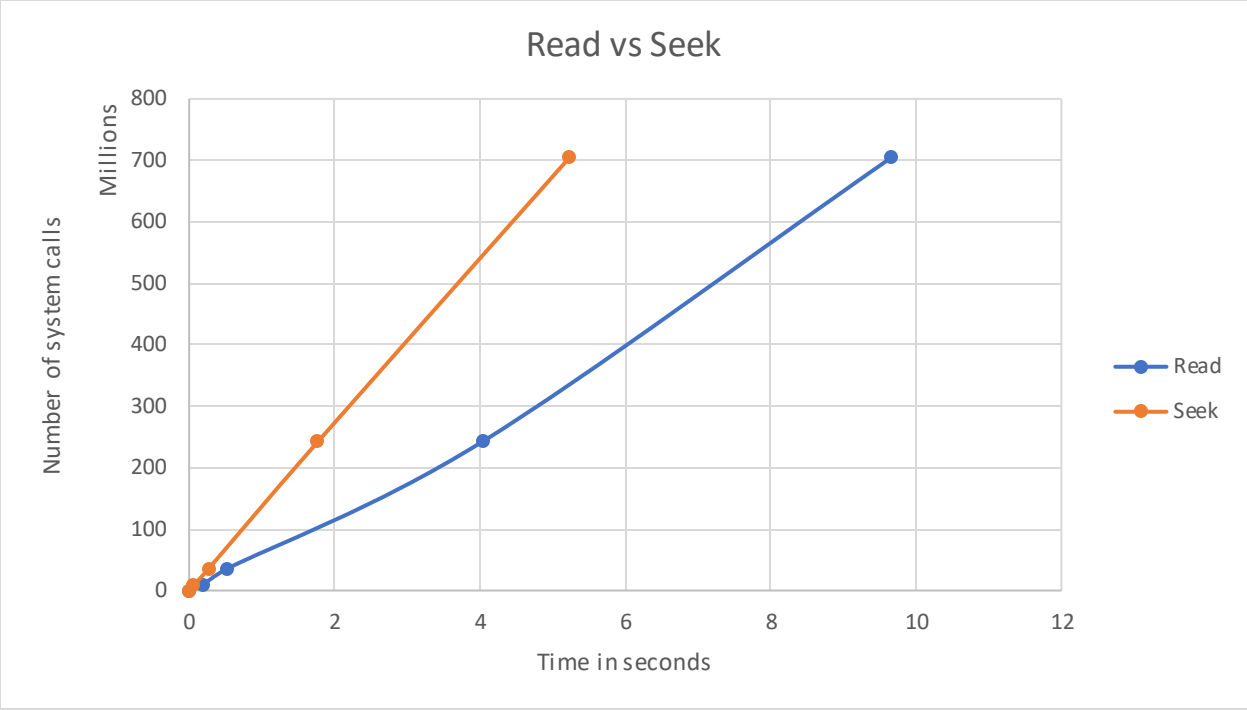
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER

thejus@Ubuntu:~/Desktop/OS/final$ sudo sh -c "/usr/bin/echo 3 > /proc/sys/vm/drop_caches"
[sudo] password for thejus:
thejus@Ubuntu:~/Desktop/OS/final$ g++ -o sys_calls sys_calls.cpp
thejus@Ubuntu:~/Desktop/OS/final$ ./sys_calls read_content/ubuntu-21.04-desktop-amd64.iso
Reading read_content/ubuntu-21.04-desktop-amd64.iso in chunks of 1 .....
Checking overhead of read system call....
Number of blocks read: 704684544 blocks
Number of system calls: 704684544 B
Number of system calls per second: 70726137.551003 B/sec
Time taken: 9.963566 seconds

Checking overhead of seek system call....
Number of system calls: 704684544.000000 B
Number of system calls per second: 135841779.758752 B/sec
Time taken: 5.187539 seconds

thejus@Ubuntu:~/Desktop/OS/final$
```

Observations

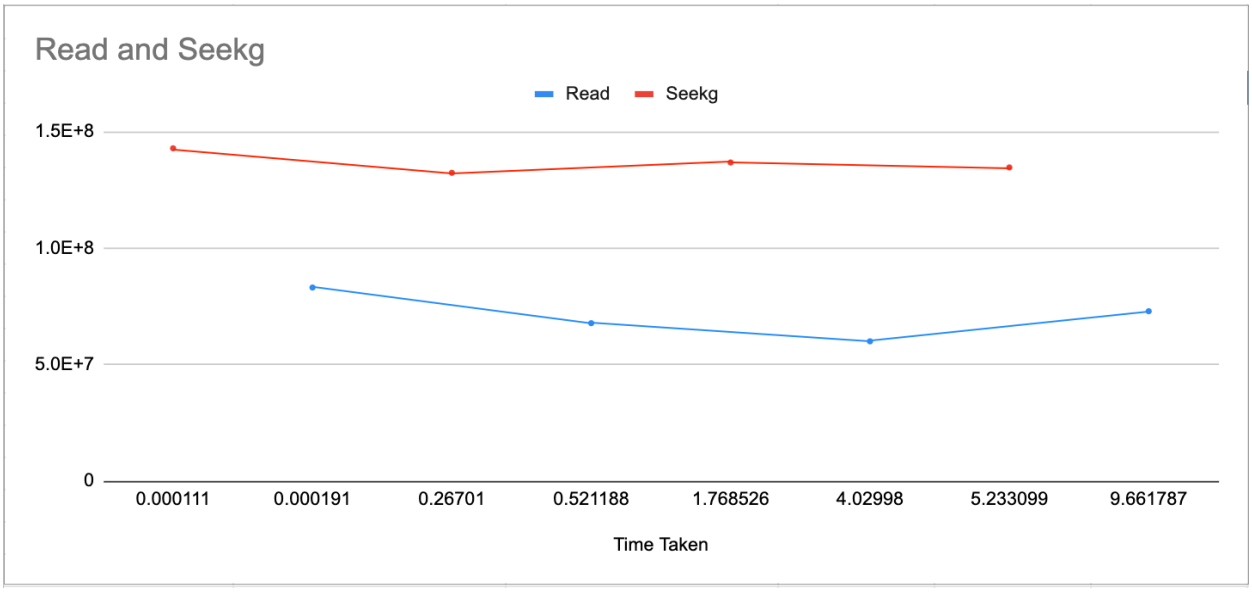
For the Read and seek system call tests

	Read		Seekg	
Number of system calls	System calls in B/sec	Time taken in sec	System calls in B/sec	Time taken in sec2
15872	83111102	0.000191	142858354	0.000111
35337778	67802360	0.521188	132346284	0.26701
241892649	60023284	4.02998	136776410	1.768526
704684544	72935218	9.661787	134659127	5.233099

On graphing it out, we get



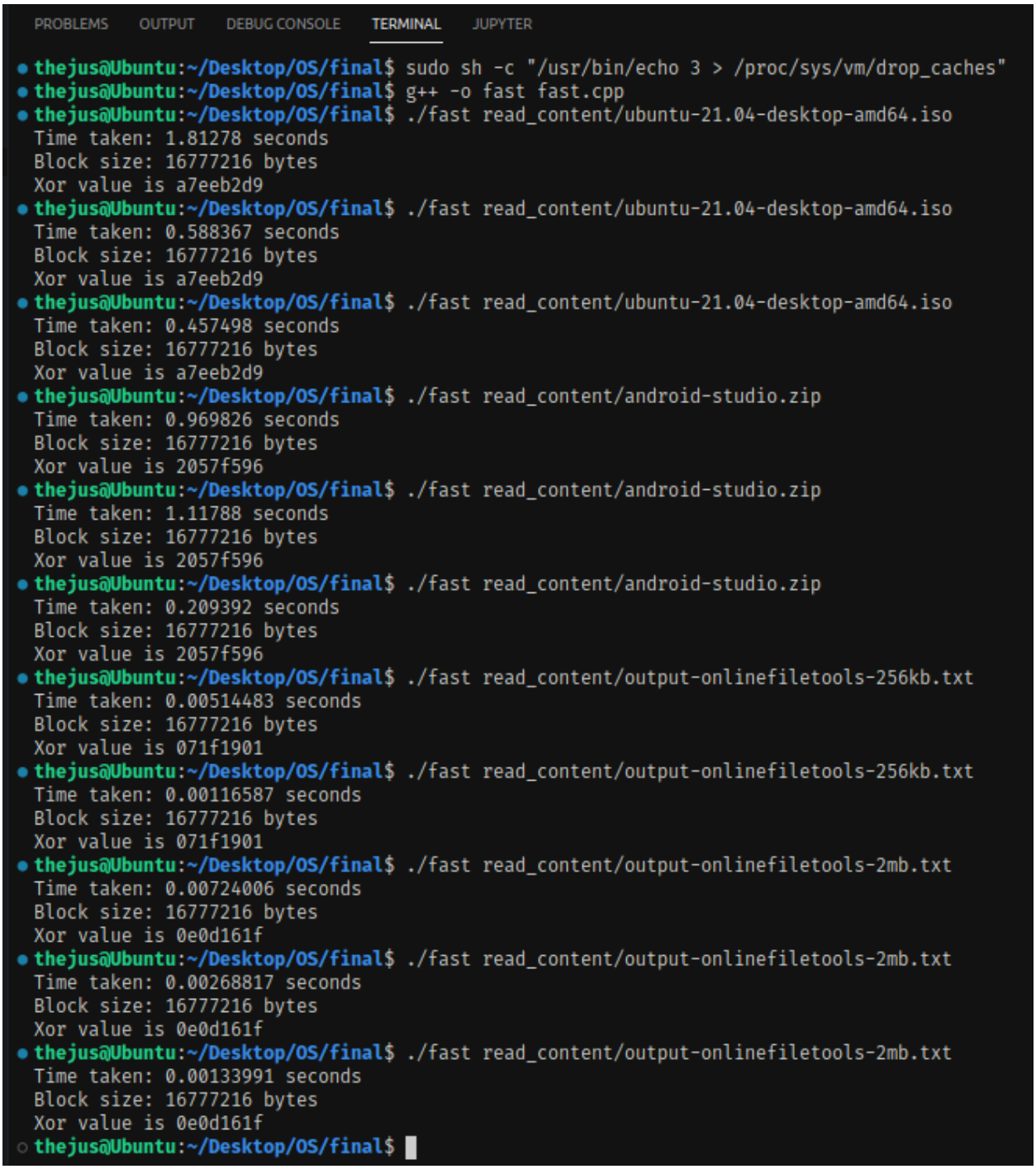
Looking at the rate of **syscalls per second vs time**, we get:



Seekg has almost half the overhead as read.

Part six – Raw Performance

The program is run according to the following parameters: `./fast <filename>`. Upon running we get the following output:



We can see from the above runs:

- `ubuntu-21.04-desktop-amd64.iso`
  - Pre-cached result: 1.81278 seconds
  - Cached result: 0.457498 seconds
- `android-studio.zip`
  - Pre-cached result: 0.969826 seconds
  - Cached result: 0.209392 seconds
- `output-onlinefiletools-256kb.txt`
  - Pre-cached result: 0.00514483 seconds
  - Cached result: 0.00116587 seconds
- `output-onlinefiletools-2mb.txt`
  - Pre-cached result: 0.00724006 seconds
  - Cached result: 0.00133991 seconds

Approach taken to extract performance from **`fast.cpp`**

- Consistent with our results from part 3, we used a big block size in order to extract the most performance that we could get. Our `block_size = 16777216` (16 megabytes, to be exact). We found that increasing this has no performance gain, and actually makes it slightly less performant. Another reason not to go with a huge block size is that it can cause memory fragmentation, and that is not ideal.

- Since XOR is purely a deterministic function, we opted to use multi-threading to calculate XOR value, one block at a time. Our multi-threaded function to calculate the XOR value looks like this

```c
void *xorbuf(void *arg)
{
    struct thread_data *args;
    args = (struct thread_data *)arg;
    long tid = args->thread_id;
    long size = args->size;

    unsigned int result = 0;
    for (int i = tid; i < size; i += num_threads)
    {
        result ^= buf[i];
    }
    args->xor_result = result;
    pthread_exit(NULL);
    return NULL;
}

unsigned int multithreaded_xor(unsigned int no_of_elements,
                               struct thread_data td[])
{
    unsigned int final_xor = 0;
    for (int i = 0; i < num_threads; i++)
    {
        td[i].size = no_of_elements;
        td[i].thread_id = i;
        pthread_create(&threads[i], NULL, xorbuf, (void *)&td[i]);
    }
    for (int i = 0; i < num_threads; i++)
    {
        pthread_join(threads[i], NULL);
    }
    for (int i = 0; i < num_threads; i++)
    {
        final_xor = final_xor ^ td[i].xor_result;
    }
    return final_xor;
}
```

The program uses 4 threads in order to get the best performance. Any more threads and performance decreases (possibly due to the overhead of context switches and syncing of the threads), and any less does not give us the most performance out of our system.

Code – `fast.cpp`

```c
#include <iostream>
#include <cstring>
#include <fstream>
#include <string>
#include <sys/time.h>
#include <pthread.h>
```

```
using namespace std;

int num_threads = 4;
unsigned int *buf;
pthread_t *threads;

struct thread_data
{
    unsigned int thread_id;
    unsigned int size;
    unsigned int xor_result;
};

double now()
{
    struct timeval tv;
    gettimeofday(&tv, 0);
    return tv.tv_sec + tv.tv_usec / 1000000.0;
}

double get_rate(double size, double start, double end)
{
    return size / ((end - start) * 1024 * 1024);
}

void print_error(string s)
{
    cout << "Error! " << s << endl;
    exit(EXIT_FAILURE);
}
void *xorbuf(void *arg)
{
    struct thread_data *args;
    args = (struct thread_data *)arg;
    long tid = args->thread_id;
    long size = args->size;

    unsigned int result = args->xor_result;
    for (int i = tid; i < size; i += num_threads)
    {
        // if(buf[i]!=0) cout<<buf[i]<<" thread "<<tid<<" "<<i<<endl;
        result ^= buf[i];
    }
    args->xor_result = result;
    pthread_exit(NULL);
```

```
    return NULL;
}

void multithreaded_xor(unsigned int no_of_elements, struct thread_data td[])
{
    unsigned int final_xor = 0;
    for (int i = 0; i < num_threads; i++)
    {
        td[i].size = no_of_elements;
        td[i].thread_id = i;
        pthread_create(&threads[i], NULL, xorbuf, (void *)&td[i]);
    }
    for (int i = 0; i < num_threads; i++)
    {
        pthread_join(threads[i], NULL);
    }
}

int main(int argc, char *argv[])
{
    unsigned int block_size = 16777216, final_xor = 0;  //16MB of buffer
    double start, end;
    string file_name = "";
    struct thread_data td[num_threads];

    for (int i = 0; i < num_threads; i++)
        td[i].xor_result = 0;

    if (argc != 2)
        print_error("Check arguments!");

    else
    {
        file_name = argv[1];
    }

    srand(time(NULL));

    unsigned int no_of_elements = (unsigned int)(block_size / sizeof(int));
    unsigned int size_of_buf = no_of_elements * sizeof(int);
    buf = (unsigned int *)malloc(size_of_buf);
    // memset(buf,0,no_of_elements*sizeof(int));
    start = now();
    ifstream object;
    object.open(file_name, ios::binary);
```

```
    if (object.fail())
        print_error("Cannot read file!");
    else
    {
        threads = (pthread_t *)malloc(sizeof(pthread_t) * num_threads);
        if (!threads)
            perror("out of memory for threads!");

        while (object.read((char *)buf, size_of_buf))
            multithreaded_xor(no_of_elements, td);

        if (object.gcount() < block_size && object.gcount() > 0)
            multithreaded_xor(object.gcount() / sizeof(unsigned int), td);

        for (int i = 0; i < num_threads; i++)
            final_xor = final_xor ^ td[i].xor_result;

        end = now();
        cout << "Time taken: " << (end - start) << " seconds" << endl <<
                "Block size: " << block_size << " bytes" << endl;
        printf("Xor value is %08x", final_xor);
    }

    cout << "\n";
    return 0;
}
```