

CS-GY 6233 FINAL PROJECT

Partner 1 – Jay Mahesh Sarvaiya: js12178

Partner 2 – Harsh Jagdish Harwani: hh2752

Part 1 - Basics:

Write a Program that can read, write a file from disk using standard C/C++ library's open, read, write, close:

We Implemented the Program using four system calls: -

- 1) Open - The Open system call opens the file name specified by the path. The return value of the system call is a file descriptor(fd). The argument flag O_RDONLY signifies Read-Only.

```
fd=open(argv[1],O_RDONLY);
```

- 2) Read – The Read system call reads up to the count bytes from the fd into the buffer. If the count is zero then the system call will throw an error.

```
bytes_read=read (fd,buffer,block_size* sizeof(unsigned int));
```

- 3) Write – The write system call writes up to the count bytes from the buffer to the file described in the fd.

```
exit(0);}  
write(fd_w, buffer, block_size);
```

- 4) Close – The close system call is used to close the file descriptor so that it no longer refers to any file and reused.

```
close(fd);
```

Part 2 - Measurement:

Write a program to find block Count which can be read in “reasonable” time:

Our approach to find the block count which can be read in reasonable time is to first note down the start time of the file and then the end time of the file and later subtract start from end time to get the duration of the time the file is in read mode.

Now, we limit the time between 5seconds and 15seconds to get the corresponding Block Count.

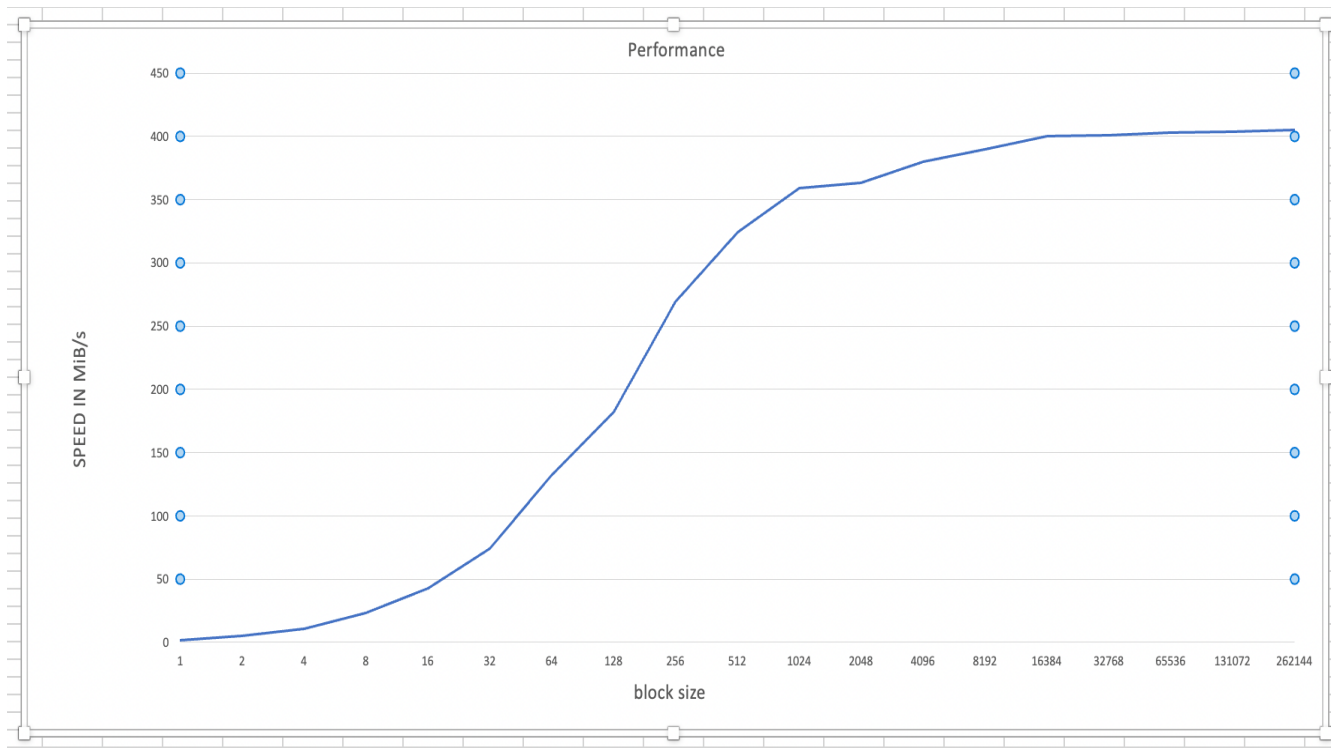
Extra Credit – Learn about dd program in Linux and see how your program compares to it:

- Its primary objective is to convert and copy files.
- dd is used simply for copying of data at low level.
- Dd command can also be used to perform read and write operations the only thing that we need to make sure is that the it should be executed in the correct driver.

Please note that one gigabyte was written for the test and 135 MB/s was server throughput for this test. Where,

<https://kenichishibata.medium.com/test-i-o-performance-of-linux-using-dd-a5074f1de9ce>

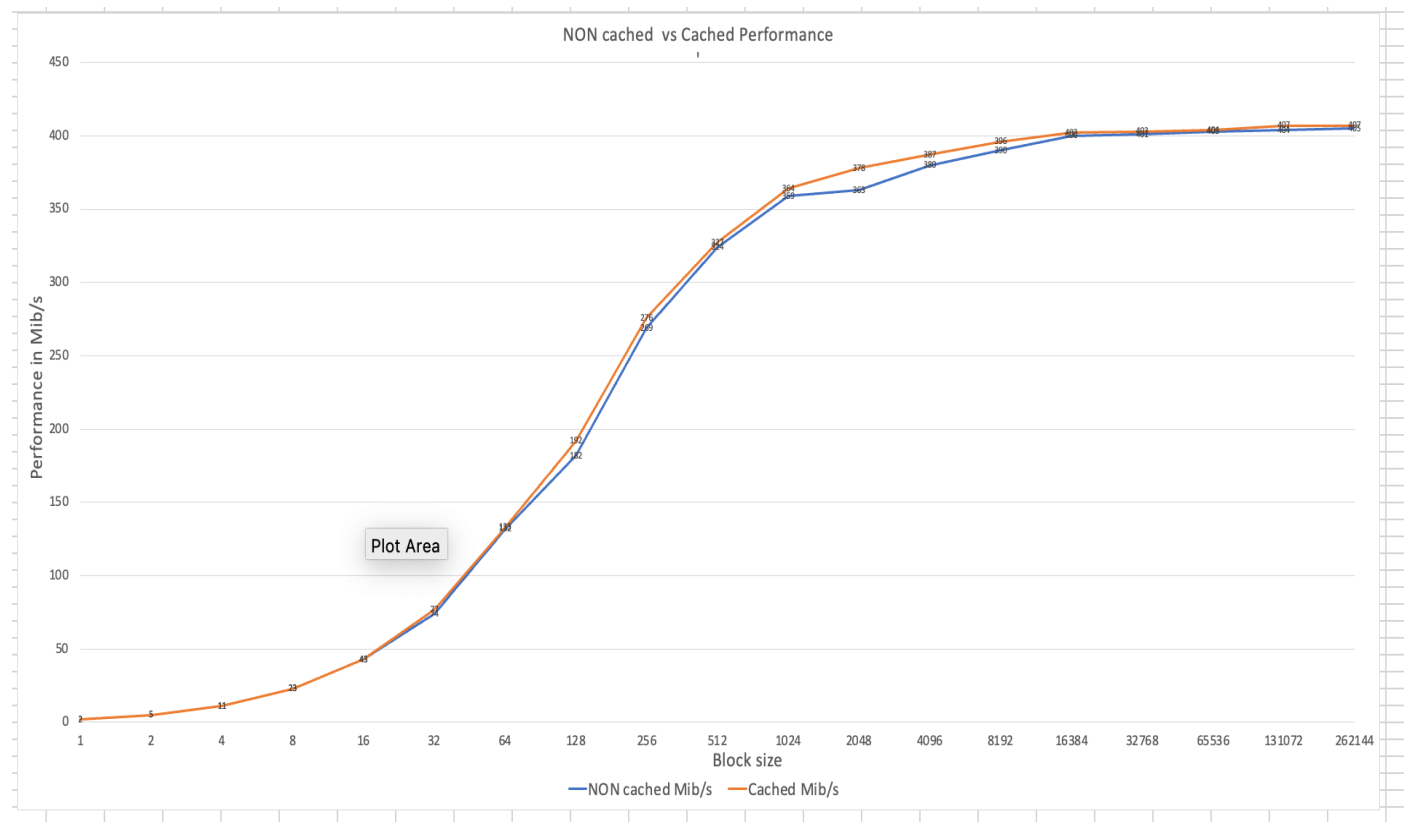
Part 3 – Raw Performance:



For this our approach is that given a block size for example 1, we will output the corresponding block count that is read in reasonable time (5-15 seconds). To show our results we tested with block size as input 1 and then kept doubling it (i.e block size input range is 1,2,4,8... upto 262144) and we have plotted the corresponding Performance for that respective block size inputs.

X axis block size and Y axis is performance in Mib/s. To get the performance we are dividing the bytes read (block_count * block_size) by the time taken to read them in reasonable time (5-15 seconds).

Part 4 - Caching:



For this our approach is like part 3 but additionally we are testing the Performance for example block size 1 again after the non-cached performance to get the cached performance. And we repeat this step for all block size inputs. We have plotted our results on graph to make it easily readable.

X axis block size and Y axis is performance in Mib/s

Blue line is Non-Cached performance

Yellow line is Cached Performance

Extra Credit – Why “3”? Read up and explain:

There are three types of cache and they store different types of data so that the computer can access the files faster to improve the overall performance of the system.

The three types are: -

- Page cache
- Dentry Cache
- Inode Cache

Page Cache – Stores Files that were recently accessed.

Dentry Cache - Does not conclude the contents of the recently accessed file but contains the directory attributes.

Inode Cache - Does not conclude the contents of the recently accessed file but contains the file attributes.

Echo “3” is used to clear Page, Dentry, Inode cache all together.

Echo “1” is used to clear just the Page Cache.

Echo “2” is used to clear Dentry and Inode cache.

Part 5 – System Calls:

```
lseek(int fd, off_t offset, int whence);
```

First parameter here is the file descriptor of the file, which you can get using open () system call. Second parameter specifies how much we want the pointer to move and the third parameter is the reference point of the movement i.e., beginning of file (SEEK_SET), current position (SEEK_CUR) of pointer or end of file(SEEK_END).

Example: - lseek (fd,5, SEEK_SET) – this moves the pointer 5 positions ahead starting from the beginning of the file

In this our approach was to first read the with block size 1 and see the performance it gives for the number of bytes read in reasonable time (5-15 seconds). Then we measure the number of system call lseek does in reasonable time (5-15 seconds)

Read performance-

Block count IS 16777216

Time taken is 5.379

Performance in MiB/s is = 2 MiB/s

Performance in B/s is = 3355443

For block size 1 the performance in B/s is the number of system call read can do.

Lseek performance–

Time taken for lseek is 9.206

System calls per seconds is = 7289687.50

Thus, we can observe that read does less system call than lseek because lseek does very less work. Lseek does almost double the number of system calls that read does.

Part 6 – Raw Performance:

To optimize the performance as fast we hard coded the block count to 32768 as the complete file was read fastest at this block size. And to optimize the the performance more we are compiling using the following syntax

```
gcc -O3 fast.c -o fast
```

We were able to read the complete file in 1.078 seconds. Below are the details for the fast read

Time– 1.078 seconds elapsed

XOR: a7eeb2d9

Performance is = 2614785024.0 Bytes/s

We also compared our read with the performance by dd

dd performance

```
dd if=xyzj.iso of=/dev/null bs=32768
```

here xyzj.iso is the ubuntu.iso file given by professor. We just renamed it for

Time – 1.129388 secs

Dd command performance–2495810068 bytes/sec

We observed the fastest read at block size 32768 for the completely reading the file. And using the syntax gcc -O3 fast.c -o fast. We were able to optimize performance to an extend that it's slightly better than dd commands performance. We did not use multiple threads as using multiple threads for reading would not optimize performance. Considering we have one file descriptor and different threads might fight for the same file descriptor and this would be inefficient.

Non cached performance- 2087954176.000000 Bytes/s

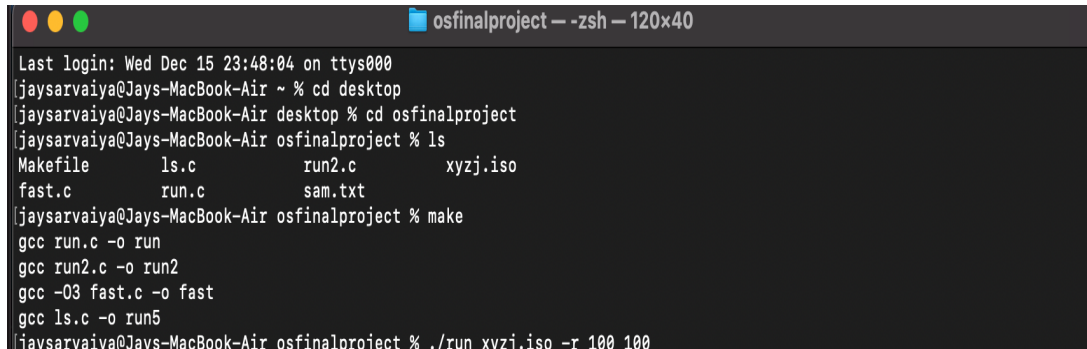
Cached performance– 2908914688.000000 Bytes/s

HOW TO RUN?

- Q1: ./run <filename> [-r|-w] <block_size> <block_count>
- Q2: ./run2 <filename> <block_size>

- Q6: `./fast <file_to_read>`

To Run 1,2,6 using the above format just go the directory where the files are and type “make” on the terminal and it should compile everything and you can use the above format to run the programs.



```

osfinalproject - zsh - 120x40
Last login: Wed Dec 15 23:48:04 on ttys000
[jaysarvaiya@Jays-MacBook-Air ~ % cd desktop
[jaysarvaiya@Jays-MacBook-Air desktop % cd osfinalproject
[jaysarvaiya@Jays-MacBook-Air osfinalproject % ls
Makefile      ls.c          run2.c        xyzj.iso
fast.c        run.c         sam.txt
[jaysarvaiya@Jays-MacBook-Air osfinalproject % make
gcc run.c -o run
gcc run2.c -o run2
gcc -O3 fast.c -o fast
gcc ls.c -o run5
[jaysarvaiya@Jays-MacBook-Air osfinalproject % ./run_xyzj.iso -r 100 100

```

Above is a picture to make things clear. In case if anything is not working or giving you the desired result, please contact us to know to compile and run, my net id-js12178 as it might be the case compilation is not going correctly and hence, you’re not getting the desired output. We have run all the parts (1-6) and they are providing the desired results so please contacts us if anything is not working or producing the correct results.

Or you could manually compile all the code using the gcc command for each program and run them.

Since there was not written a way to run part5 the make command will compile the part 5

In case you want to run part 5

`./run5 <filename> 1.`

Here we are typing 1 because in part 5 first we are getting the output for normal read using block size 1 and after this it will output lseek performance.