FINAL PROJECT REPORT

Operating Systems CS GY 6233

MEASURING DISK I/O PERFORMANCE IN LINUX

Team

Syed Ahmad Taqi – st4324

Tejas Sateesh – ts4044

Introduction

In this project, we measure the disk I/O performance in Linux by working with different System Calls to understand what sits in the way between a process requesting data from disk and receiving it.

To achieve this, we have written different C Programs which read and write from the disk (with or without caching) through means of System Calls such as *read, write, pread. lseek and mmap*. We measure the performance of these by changing parameters like block size and block count to find out how the I/O performance varies and what is the reason behind it.

System Specifications

All tests mentioned hereafter are performed on our personal laptop with following hardware and software specifications:

- OS: Ubuntu 21.10, 64-bit.
- RAM: 16GB DDR4
    - 3200 MHz Clock Speed
    - SO-DIMM
- Processor: AMD Ryzen 5900HS
    - 3.4GHz
    - 8 Core, 16 threads
    - Cache size: L1 512 KB, L2 4 MB, L3 20 MB
- Disk Drive: 1 TB NVMe M.2
    - PCIE 3.2
    - 3900 MiB/s Serial Read

Part 1 - Basics

For this part we have written a C Program which reads and writes from and to the disk using the standard C file functions Open, Read, Write and Close which translate directly into System Calls in Linux. The program is accompanied with the report with the name “part1.c”.

The parameters for the program are:

- filename: Name of file to use for reading or writing. If the filename doesn’t exist when we write, we create a new one.
- -r | -w : flags which tell whether to read from or write to a file.
- Block size: Size in Bytes which will be read or written onto a file with every invocation of system call.
- Block count: Number of blocks of size Block count which will be read or written onto the given file.

Part 2 - Measurements

Here we try to find a file size (or Blockcount), corresponding to various Block sizes, which can be read in reasonable time (5 to 15 seconds) using the logic we wrote for Part 1. This is to ensure when we measure the performance in the latter parts, the largest proportion of the time should be when I/O is happening, hence running the program long enough will make sure time taken for other steps such as XOR calculation is negligible in the calculation.

To make this work, we came up with another program, "part2.c" which is invoked through a Python script for each of the block size. For block size, measured in Bytes, we go with powers of 2 starting with 1 byte to 134217728 (134 MB).

Blocksize (Bytes)	Blockcount	Size (Bytes)
1	20741013	20741013
2	20700171	41400342
4	21138212	84552848
8	21043198	168345584
16	21304770	340876320
32	21377053	684065696
64	20452464	1308957696
128	20609924	2638070272
256	20716632	5303457792
512	20132829	10308008448
1024	19824821	20300616704
2048	19796181	40542578688
4096	18702348	76604817408
8192	15593761	127744090112
16384	15593761	255488180224
32768	6234611	204295733248
65536	3586009	235012685824
131072	1864093	244330397696
262144	101997	26737901568
524288	50031	26230652928
1048576	24398	25583157248
2097152	11870	24893194240
4194304	5621	23576182784
8388608	2607	21869101056
16777216	1318	22112370688
33554432	681	22850568192
67108864	354	23756537856
134217728	183	24561844224

Figure 1: Table illustrating the various block count found for each of the block size by running the program for about 8 seconds.

Part 3 – Raw Performance

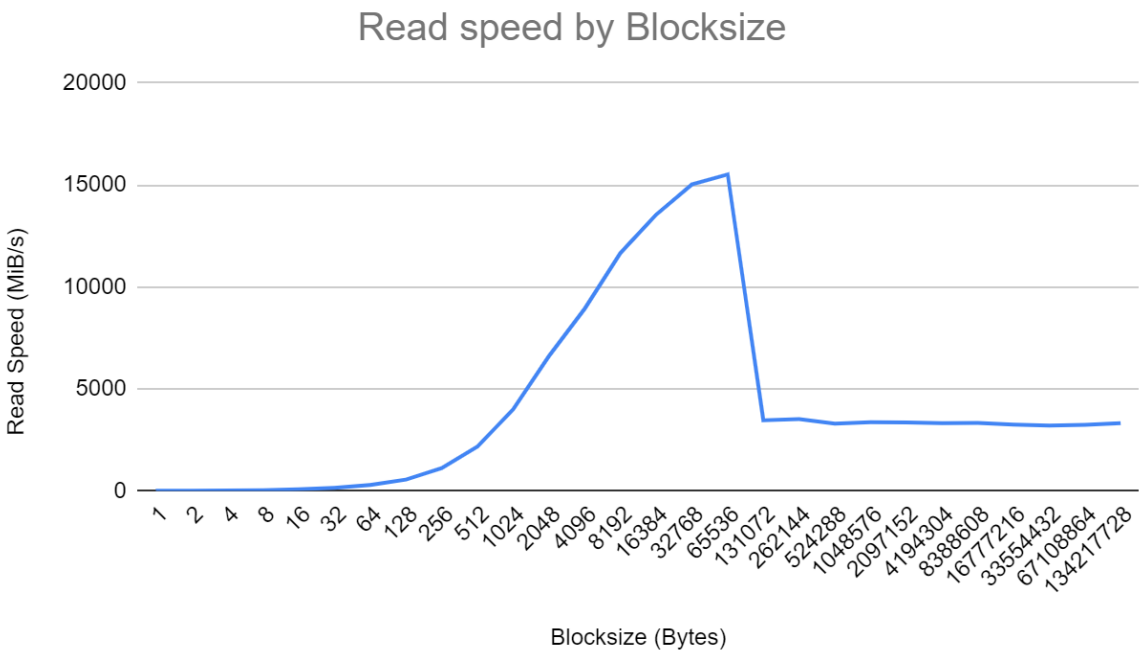

Figure 2: Average read speed over varying block size

We ran the program written in Part 1 using the Block Size to Block Count chart (Figure 1) obtained from Part 2. Figure 2 shows the average of read speed taken over 4 runs for each block size. The timings and hence read speeds are inclusive of cached reads in this case.

We can observe that the performance increases with increasing block size until 65536 Bytes (64 KiB), after which it decreases and saturates. This is because the system has a cache of 8 Pages*8 KiB, i.e., 64 KiB effective L1 cache. Since only 64 KiB of data can be cached for a system call, if the block size exceeds 64 KiB, page faults occur which reduces the effective reading time which can be observed in the graph. Once the cache is exceeded, the performance reduces to the disk read speeds, which has a manufacture provided read rate of 3900 MiB/s for sequential reads and saturates there.

Part 4 – Caching

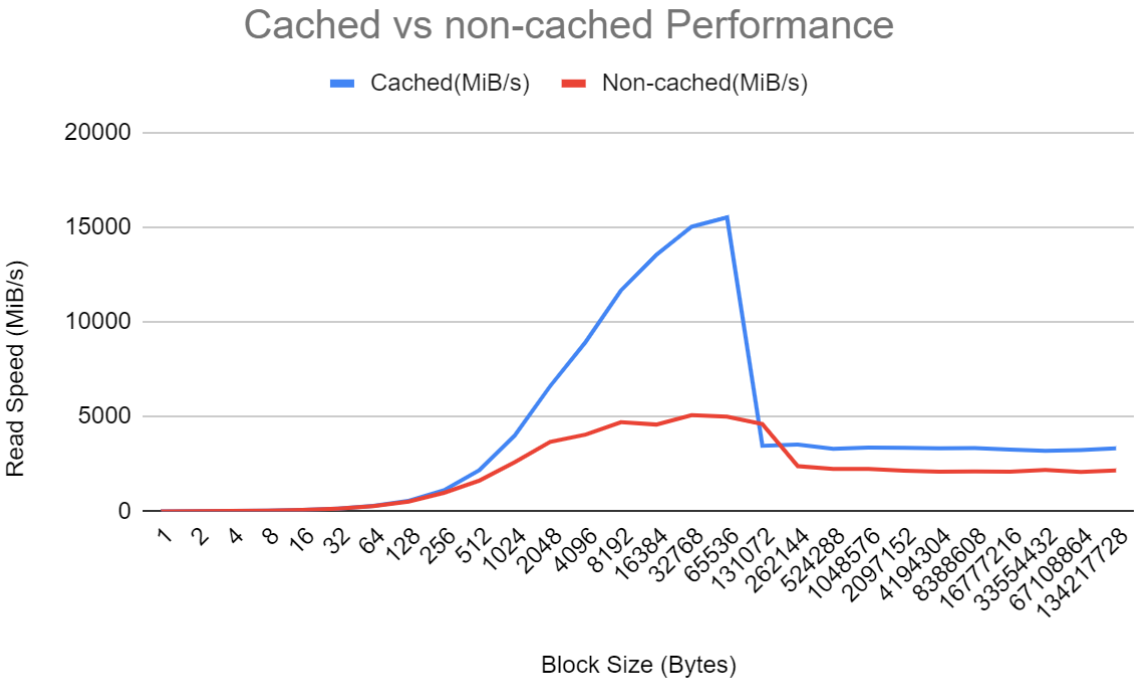

Figure 3: Cached vs non-cached read speeds

Here, we ran the program written for Part 1 the same way as we did for Part 3, but this time before each invocation, we cleared the cache. To do this we used the following command - *sudo sh -c "/usr/bin/echo 3 > /proc/sys/vm/drop_caches"* before every run of the program.

As apparent by the graph in Figure 3, caching and reading the file contents from the main memory helps us achieve very large performance gains when compared to directly reading from the Disk (SSD in our case). The non-cached speeds are almost near the manufacturer states read speed of the disk which is around 3900 MiB/s.

Extra – Comparison with DD

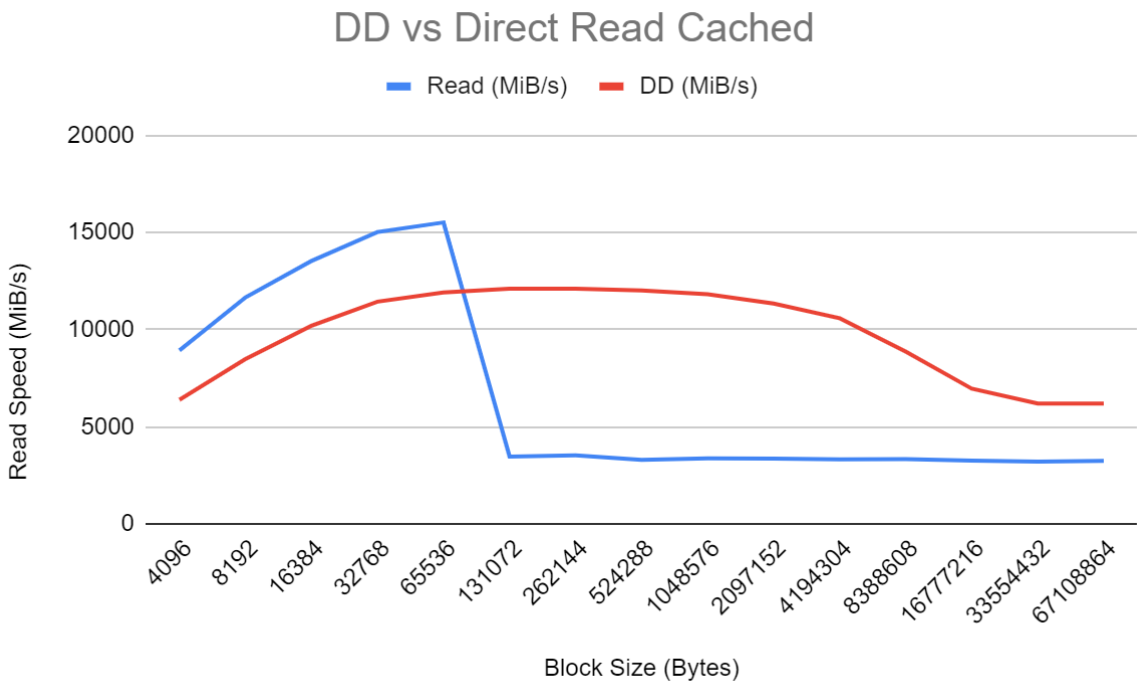

Figure 4: Comparison of read speed of DD vs our program for various block size without clearing the cache

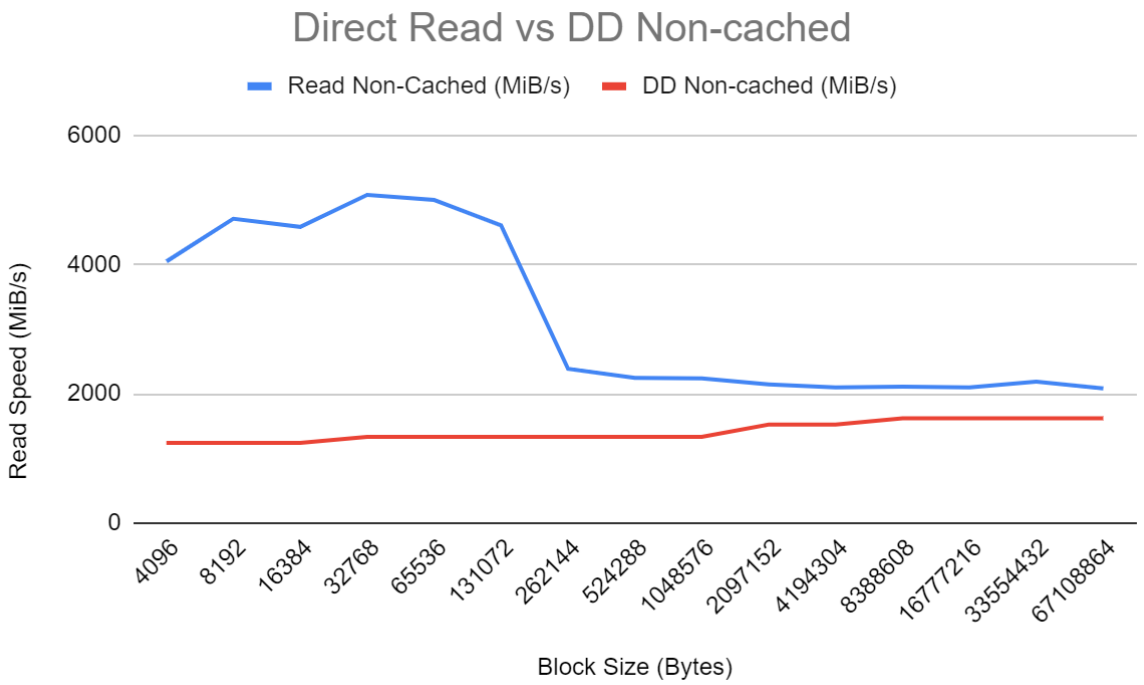

Figure 5: Comparison of read speed of DD vs our program for various block size when cache is

dd is a command line utility program available in Linux which is used to for Disk performance benchmarking. The GNU dd syntax is as follows –

dd if=/dev/input.file of=/path/to/output.file bs=block-size count=number-of-blocks

Using this we ran the test for various block size in our table from Part 2 and made the Graph in Figure 4 and Figure 5. The results indicate that until 64KiB Block size our program performs much better than dd, but after that DD is optimizing the reads much better; consistently maintaining high read speeds.

Extra – Clearing cache in Linux

On Linux there is a way to clear the disk caches without rebooting your machine. E.g. sudo sh -c "/usr/bin/echo 3 > /proc/sys/vm/drop_caches".

Why do we use '3' in this?

If we use '1' only Page Cache is cleared. If we use '2' only Dentries and inodes cache is cleared. But if we use '3' all of Page Cache, Dentries and inodes cache are cleared which is what we require in our case to start from a clean slate.

To explain it further, Dentries and inodes cache is directory and file attributes. This cache helps to cut down on disk I/O operations and goes hand in hand with the Page Cache. Hence when we clear Page Cache we also want the other two to be cleared so that we can be sure when we run our program to test disk read speeds the cache is not involved at all.

Part 5 - System Calls

Blocksize	read (Bytes/s)	pread (Bytes/s)	lseek (Bytes/s)
1	4771454.48	4931348.23	15015713.94
2	4893047.76	4886780.62	15575232.26
4	4851067.38	4904992.74	15519660.3
8	4846149.29	4891408.29	15564444.58

Figure 6: Comparison of different system calls for various block sizes

The table above indicates that when run with a block size of 1, lseek runs the greatest number of times every second. This may be because lseek is used to only change the location of read/write pointer of a file descriptor and does not actually read from the file.

Part 6 - Fast read

For this part we have come up with another program which uses mmap instead of using plain read and write which we used for previous parts. We noticed that mmap gives much better-read speeds, especially when file size is within the range of available Main Memory.

Mmap creates one to one mapping of a file on secondary storage onto primary storage and returns the address of the memory location. We have used certain flags provided by mmap like MAP_POPULATE which fetches data eagerly and adds it to the ramdisk that Linux Kernel manages. Thus, the performance for a file of size lesser than the available system main memory would be closer to the fastest possible IO read speeds of the system. Once we cross the main memory size for file size, we end up having to load the pages at every page limit, thus decreasing the read speed due to page reload. But the read speeds are still faster than the read() system call for a large file.

The system call for mmap() is as follows:

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);

Where, addr indicates the position where we need our mapping to be created at, specifying NULL lets the decision to the OS. Length is the size of memory we need for the mapping. Prot is the flag to indicate the type of file operations – PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE. Flags indicate additional parameters for the OS to consider while allowing access to this memory location. MAP_ANON, MAP_SHARED, MAP_PRIVATE, MAP_FILE, MAP_FIXED are some of the values that can

be used. Fd indicates the file descriptor for which we are creating the mapping. Offset indicates the position from which we need to read the file contents.

Once the file is read and processed upon, we can unmap the address using munmap(). Thus, mmap is a powerful tool which can be used to access data on IO devices, when the files are comparable to main memory size.

Our program "part6.c" compiles to the executable, "fast" which can be used to check the read speeds for a file using mmap.

References

[1] DD Source Code: https://github.com/coreutils/coreutils/blob/master/src/dd.c

[2] Open man page: https://man7.org/linux/man-pages/man2/open.2.html

[3] readahead man page: https://man7.org/linux/man-pages/man2/readahead.2.html

[4] Fread Source Code: https://github.com/lattera/freebsd/blob/master/lib/libc/stdio/fread.c

[5] DD guide: https://www.cyberciti.biz/faq/howto-linux-unix-test-disk-performance-with-dd-command/

[6] Unit converter: https://www.convertunits.com/from/megabyte/second/to/MiB/s

[7] Linux cache guide: https://www.geeksforgeeks.org/how-to-clear-ram-memory-cache-buffer-and-swap-space-on-linux/