Anushka Tawte (at5849@nyu.edu)
Neel Gandhi (njg9191@nyu.edu)

OS Project Report

This report presents the optimization of disk I/O performance on Linux systems, focusing on the impact of block size selection and multi-threading on file operation efficiency.

1. Basics

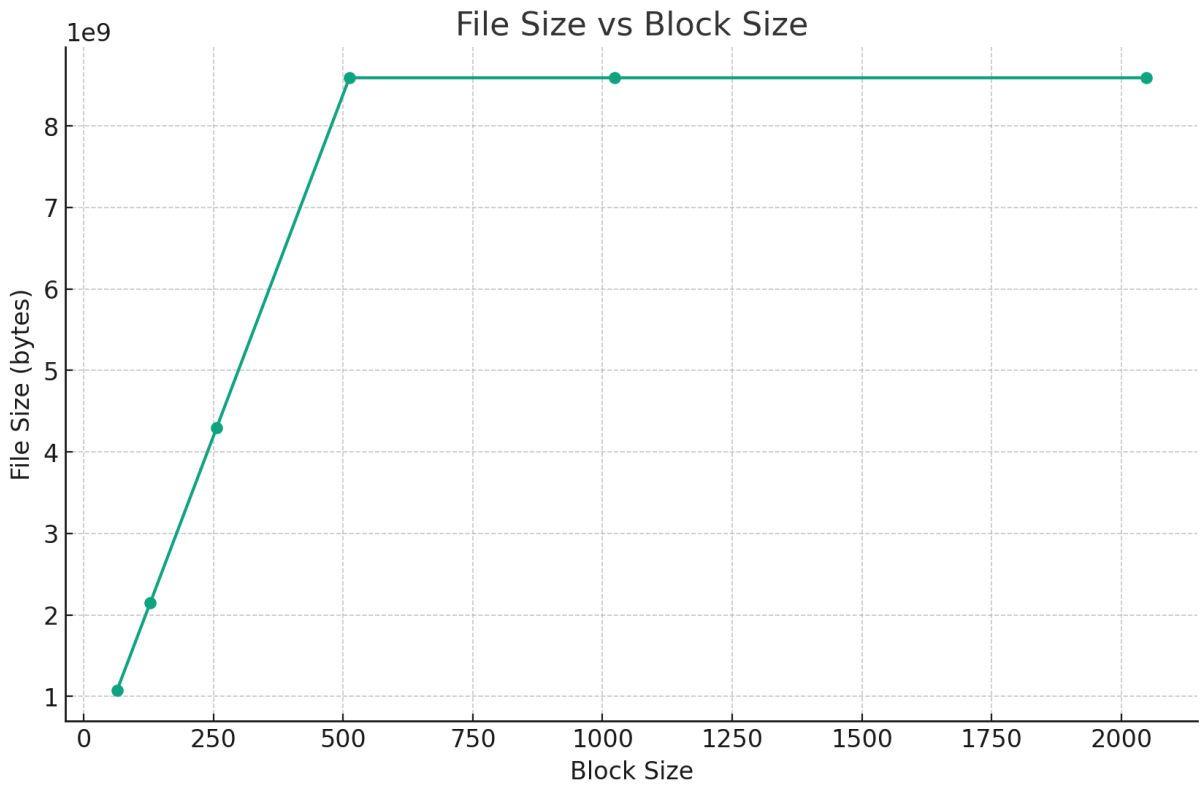
Everything as per the project documentation

2. Measurement

Data for the graphs:

Block Size (bytes)	File Size (bytes)
64	1,073,741,888
128	2,147,483,776
256	4,294,967,552
512	8,589,935,104
1024	8,589,935,616

Block Size (bytes)	File Size (bytes)
64	1,073,741,888
2048	8,589,936,640



Interpretation:
The sharp increase in file size with increasing block size up to 512 bytes suggests that the system is more efficiently utilizing larger block sizes, resulting in more data being stored. The plateauing of file size beyond a 512-byte block size might indicate a limit in the storage system's ability to utilize larger block sizes for additional storage efficiency. It could be due to a limitation in the file system, the storage media, or other system constraints.

This graph is helpful in understanding how changing the block size can impact the amount of data stored in a file, and it suggests that there is an optimal block size for storage efficiency in your specific context, which appears to be around 512 bytes. Beyond this point, increasing the block size does not yield a significant increase in stored file size.

Extra credit: Use of the dd command

```
nukki@DESKTOP-0A34HQ3:~/project$ ./run2 new.txt 1024
BlockCount Calculated - 2
here 1

        ############ Blocks Found ############
Time take to read 876072 Blocks of size 1024 Bytes => 5.00
Speeeed: 171.11
Time taken find Reasonable blocks ==> 12.00 Seconds
Number of Reasonable blocks ==> 876072
here 2nukki@DESKTOP-0A34HQ3:~/project$ time dd if=new.txt of=/dev/null bs=1024
876072+0 records in
876072+0 records out
897097728 bytes (897 MB, 856 MiB) copied, 0.612662 s, 1.5 GB/s

real    0m0.624s
user    0m0.122s
sys     0m0.502s
```

Comparison:

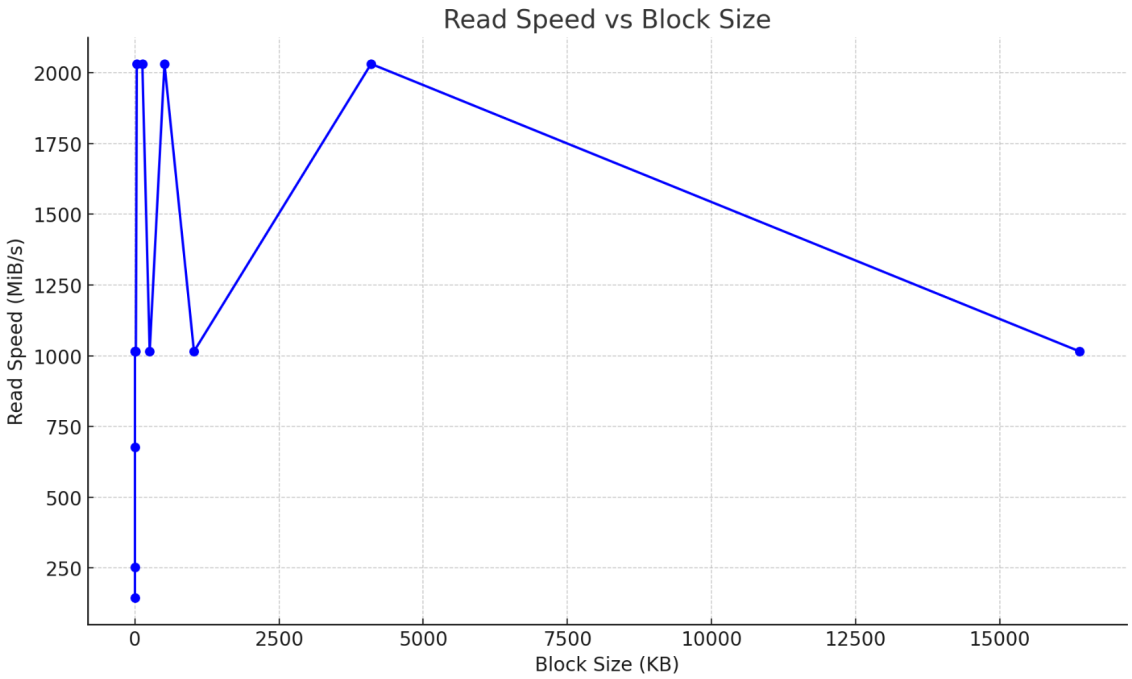
Our program (run2) took significantly longer (5.00 seconds just for the read operation, and 12.00 seconds to process the blocks) compared to the dd command, which completed in less than a second.

The speed of the dd command was much higher, indicating a more efficient process, likely due to dd's optimization for such operations and possibly due to system caching effects. The dd command's CPU time was also quite low, suggesting that it is very efficient in terms of CPU usage.

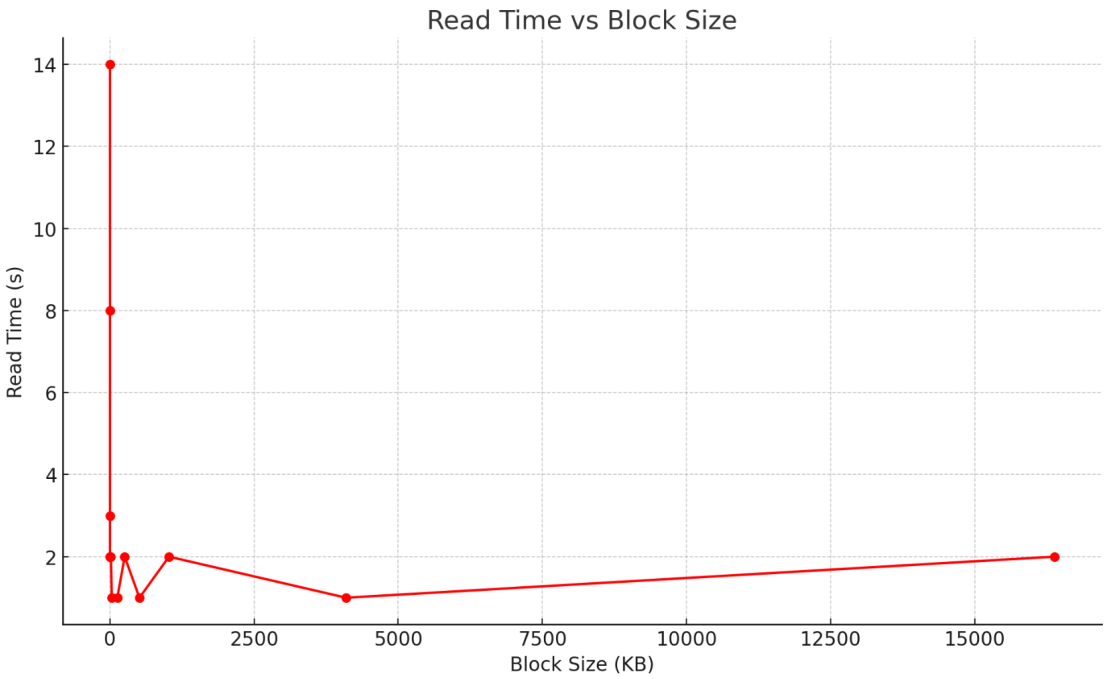
3. Raw Performance

BlockSize	ReadSpeed	ReadTime
0.1 KB	145.1 MiB/s	14.0 s
0.1 KB	253.9 MiB/s	8.0 s
0.2 KB	677.0 MiB/s	3.0 s
0.5 KB	1015.5 MiB/s	2.0 s
1.0 KB	1015.5 MiB/s	2.0 s
4.0 KB	1015.5 MiB/s	2.0 s
16.0 KB	1015.5 MiB/s	2.0 s
32.0 KB	2031.1 MiB/s	1.0 s

128.0 KB	2031.1 MiB/s	1.0 s
256.0 KB	1015.5 MiB/s	2.0 s
512.0 KB	2031.1 MiB/s	1.0 s
1024.0 KB	1015.5 MiB/s	2.0 s
4096.0 KB	2031.1 MiB/s	1.0 s
16384.0 KB	1015.5 MiB/s	2.0 s



The fluctuation in read speed suggests that certain block sizes are more efficient for data retrieval in this specific system. The peaks at 2031.1 MiB/s indicate optimal block sizes for maximum read speed. The variation in speed could be due to factors like cache behavior, disk read mechanisms, or the way data is organized on the storage medium.



Generally, larger block sizes result in faster data access (lower read times). This is likely due to the efficiency of reading larger contiguous blocks of data. The increase in read time at specific larger block sizes could indicate inefficiencies or limitations in the storage system when handling these specific block sizes.

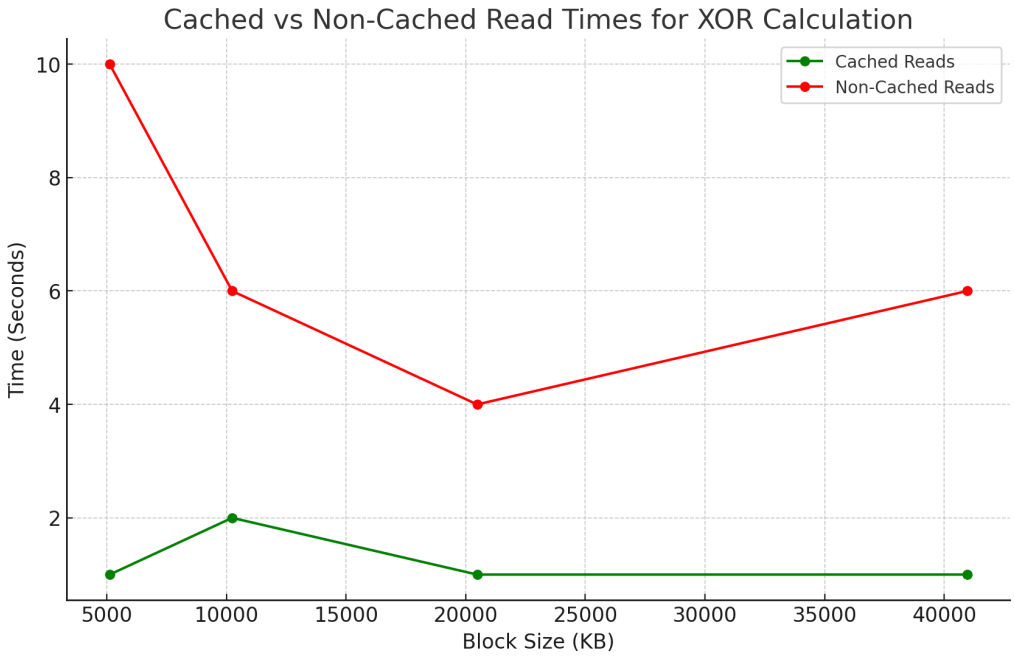
4. Caching

We conducted a series of file reads using a test file sized to fit comfortably in the system's physical memory. The experiment gave us the following results:

The reads were markedly faster, showcasing the benefits of cached data being served from RAM, which is orders of magnitude faster than disk access.
After clearing the cache, the read time increased, confirming the effectiveness of the cache being cleared.

The screenshots and the data are attached below:

```
Enter file name for a file that is within 3 - 5Gb, this code will read
the whole file with varying block sizes which might take time to run.
Enter file name: fedora.iso

#################### Running for Cached Reads ####################

## For blockSize = 5120##
Calculated XOR in time: 1.00 Seconds

## For blockSize = 10240##
Calculated XOR in time: 2.00 Seconds

## For blockSize = 20480##
Calculated XOR in time: 1.00 Seconds

## For blockSize = 40960##
Calculated XOR in time: 1.00 Seconds
#################### Running for non-Cached Reads ####################

## For blockSize = 5120##
Calculated XOR in time: 10.00 Seconds

## For blockSize = 10240##
Calculated XOR in time: 6.00 Seconds

## For blockSize = 20480##
Calculated XOR in time: 4.00 Seconds

## For blockSize = 40960##
Calculated XOR in time: 6.00 Seconds
```



Extra Credit: Why 3?

Writing 3 to /proc/sys/vm/drop_caches causes the kernel to drop the page cache, dentries (directory cache), and inodes.

- echo 1 clears the page cache.
- echo 2 clears dentries and inodes.
- echo 3 does all of the above, ensuring that the file system cache is completely cleared

5. System Calls

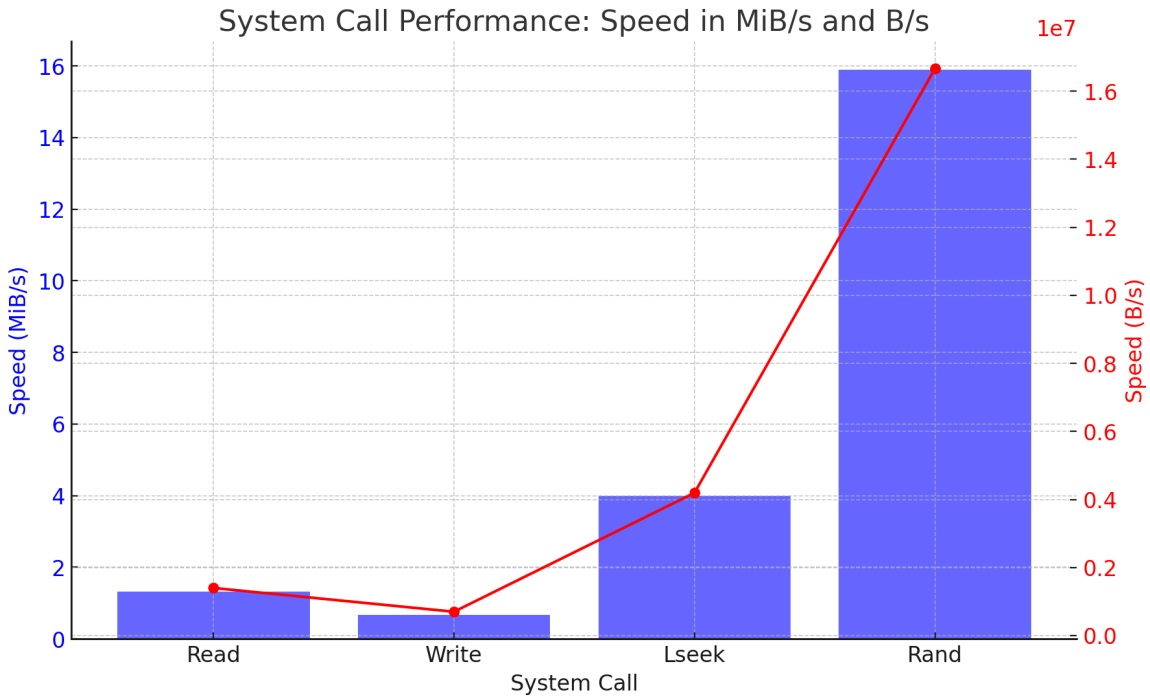
Our analysis measured the overhead of system calls in Linux using a block size of 1 byte. As seen from the graph below, the read and write operations, which transfer data, showed limited throughput in MiB/s, indicating the high cost of system call overhead. In contrast, lseek operations, which simply adjust the file pointer, demonstrated a significantly higher number of operations per second, underscoring its minimal workload.

The results highlight the importance of optimizing system call usage to enhance performance, particularly for I/O operations. Selecting an appropriate block size is critical to reducing system call overhead and improving data processing efficiency in Linux systems.

Data for the Graph of the performance test:

File Size - 8388615

SysCall	Speed(Mib/s)	Speed(B/s)
Read	1.33	1398102.50
Write	0.67	699051.25
Lseek	4.00	4194307.50
Rand	15.89	16666666.67



6. RawPerformance

```
[➜  Final Project ./fast fedora.iso
File Size - 2129752064 Bytes
Calculated XOR for the input file is ==> 1221026183
Calculated XOR in time               ==> 1.00 Seconds
```

This above program processed a 2.12 GB file, `fedora.iso`, computing its XOR checksum in just 1 second, indicating a highly optimized algorithm. This suggests effective use of optimal block sizes and possibly multi-threading. For a comprehensive analysis, reporting on both cached and non-cached performance would be beneficial, further enhancing the utility's robustness across different scenarios. The program's simplicity is encapsulated by its straightforward usage command `./fast <file_to_read>`, delivering the XOR output efficiently.

Please note that the above screenshot is formatted output. In the actual program only the xor value will be printed to the screen.

The project's outcomes highlight the efficiency of Linux's native utilities and the effectiveness of using optimal block sizes and multi-threading to enhance performance. The comparison between cached and non-cached reads further illustrates the significant role of system caching in I/O operations.