

1 Basics

We create `rdwr.c` including functions `block_read` and `block_write`. Both of them have 3 parameters: `file_name`, `block_size`, `block_count`. In `block_read`, we read `buf` sized `block_size` `block_count` times from the file named `file_name`. And so on in `block_write`.

In `run.c`, we parse the parameters and choose to use `block_read` or `block_write`. We record the start time and the end time to get the runtime.

2 Measurement

Firstly, we parse the parameters to get the `file_name` and `block_size`.

Then, we start with `block_count=1`. In each loop, we double `block_count` and run

```
block_read(file_name, block_size, block_count);
```

We record the runtime and quit the loop when `runtime>5` (second)

Finally, we print `block_size * block_count` as the `file_size`

Extra Credit: explore `dd`

We use "`dd`" to copy 2GB data (`block_size` 4096) from the iso file and compare its runtime with "`run`". We find that their runtimes are almost the same. "`dd`" takes 50 seconds and "`run`" takes 49 seconds total.

```
cbxoi@cbx:/mnt/e/Programing Environment/os-final/test$ dd if=test.iso of=test2.iso bs=4096 count=524288
524288+0 records in
524288+0 records out
2147483648 bytes (2.1 GB, 2.0 GiB) copied, 50.3539 s, 42.6 MB/s
```

```
cbxoi@cbx:/mnt/e/Programing Environment/os-final$ ./run ./test/test.iso -r 4096 524288 && ./run ./test/
test3.iso -w 4096 524288
Runtime: 24.017378
Runtime: 25.299178
```

Extra credit: learn about Google Benchmark — see if you can use it.

We've tried Google Benchmark on this part. Benchmark is a lightweight library, and it is a bit awkward to use it to calculate a proper `file_size`. (much simpler just in C code)

However, it is a nice tool to monitor performance. We've included an example below, where we compare dd and our implementation of block_read + block_write. dd is still slightly faster with different block sizes. (the format BM_dd/2048 means that dd is tested using block size of 2048 Bytes)

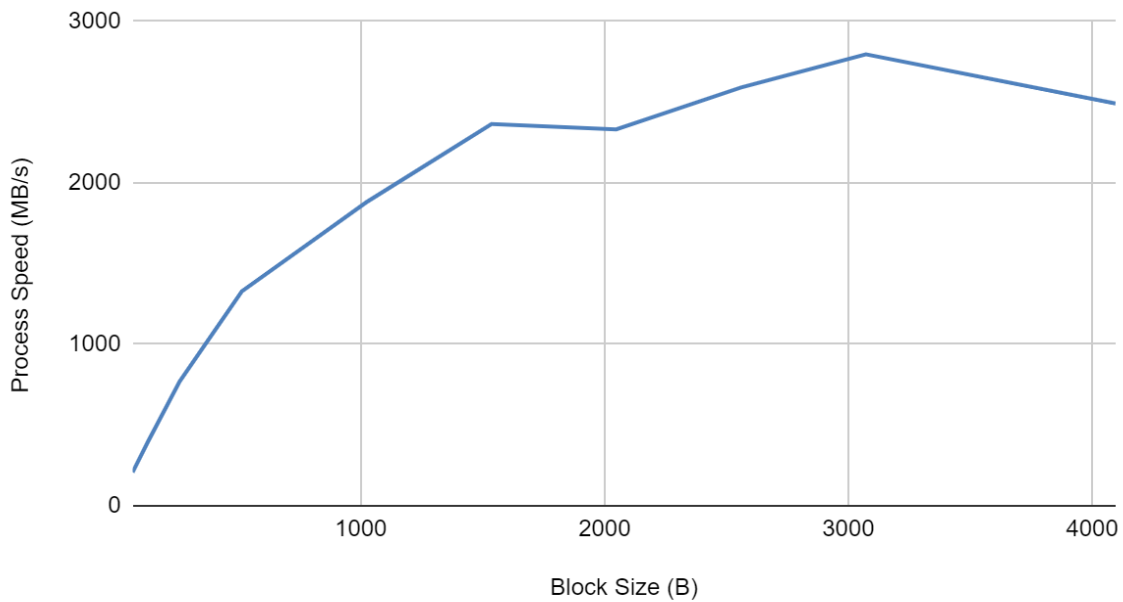
Benchmark	Time		CPU		Iterations
BM_read_and_write/512	1.2601e+10	ns	181507	ns	1
BM_read_and_write/1024	8060205591	ns	67053	ns	1
BM_read_and_write/2048	6564216220	ns	117544	ns	1
BM_read_and_write/4096	5702071011	ns	196581	ns	1
BM_dd/512	9296277143	ns	63828	ns	1
BM_dd/1024	6122079373	ns	168342	ns	1
BM_dd/2048	5338524615	ns	169010	ns	1
BM_dd/4096	4366228072	ns	90932	ns	1

3 Raw Performance

- Make a graph that shows its performance as you change the block size.

As we can see from the graph, the process speed (performance) increases as the block size increases until reaching around 2.5GB/s. Please also refer to our spreadsheet in data.xlsx.

Process Speed (MB/s) vs. Block Size (B)

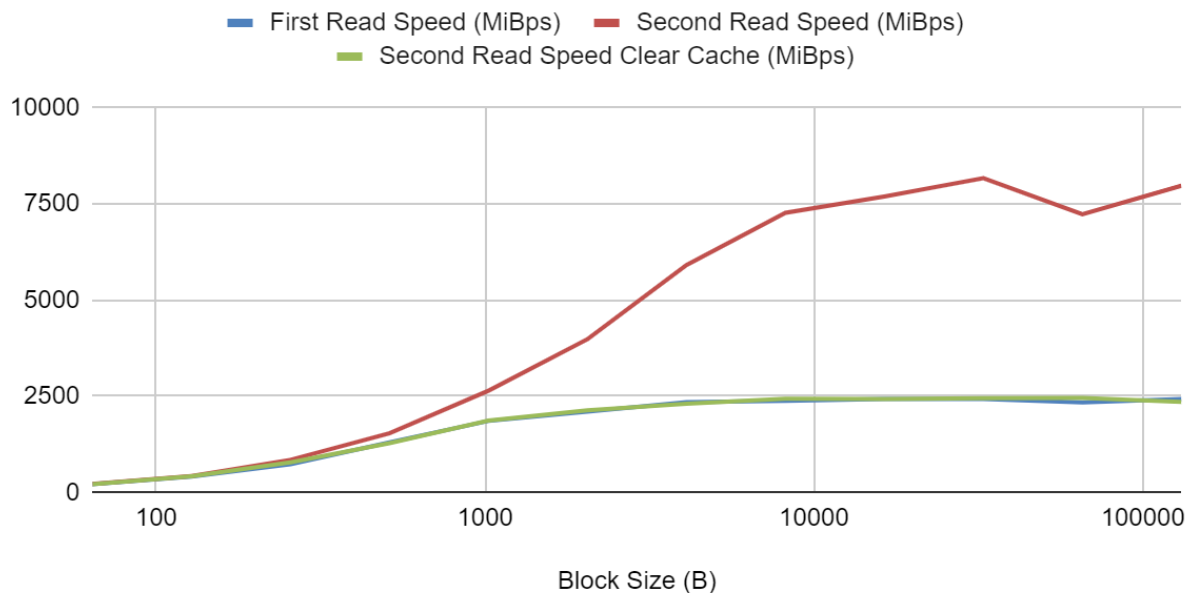


4 Caching

We read the file once, clear the cache, and then read the file again. Their data has been recorded in data.xlsx. Below is a graph of their speed in different block sizes. (Note that x axis is in log scale)

From this graph, we observed that, second read speed is significantly faster than the first read and second read cache cleared. The difference of speed grows larger and larger as the block size increases, until it becomes stable around block size 8192 Bytes.

First Read Speed (MiBps), Second Read Speed (MiBps) and Second Read Speed Clear Cache (MiBps)



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

The following comes from the documentation of Linux:

To free pagecache:

```
echo 1 > /proc/sys/vm/drop_caches
```

To free reclaimable slab objects (includes dentries and inodes):

```
echo 2 > /proc/sys/vm/drop_caches
```

To free slab objects and pagecache:

```
echo 3 > /proc/sys/vm/drop_caches
```

Using 3 remove both the slab objects and pagecache. Slab is a memory management mechanism used in Linux for commonly used objects. Page cache is part of the VM system. It is a cache of pages in RAM.

5 System Calls

Please refers to `system_call.c`

- Measure performance MiB/s when using block size of 1 byte

3.680185 MiB/s

- Measure performance in B/s. This is how many system calls you can do per second.

3858954 B/s, or 3858954 systemcalls per second

- Try with other system calls that arguably do even less real work (e.g. `lseek`)

We have compared system calls `read` and `lseek(system_call.c)`, here's the printing result:

Number of read per second: 3844769.056968

Number of `lseek` per second: 10240425.108437

6 Raw Performance

In this part we use multiple threads.

We use the `block_size` 4096. (Basely)

For the thread numbered `i`, we open a `fd` and read the blocks numbered

`i, i+num_threads, ... , i+k*num_threads`

Use `lseek()` to jump to the right place. After reading, we combine 4 characters into an unsigned int and XOR them.

Finally, we output the total runtime (This runtime includes the time to calculate XOR), the runtime per MB and the result of XOR

For those files whose size is not a multiple of 4, we add `'\0'` to its tail when XOR and make sure each byte is included.

Multiple threads works well. One thread takes approximately 4 times runtime compared to 16 threads.

We also try to increase the `block_size` and find that it makes `fast.c` faster. `block_size 4096` takes approximately 8 times runtime compared to `block_size 65536` (216). After that, the increase of `block_size` hardly causes the development of efficiency.

Extra Try:

We try to use `mmap` instead of reading to input the data but we find that it slows down. It almost takes double the time. We guess that under a small `**block_size`, `mmap` takes more time when mapping such a small bunch. The code is in `fast_mmap.c`