Released: **Nov 5th, 2023**

Due: **Dec 10th, 2023 (11:59PM EST)**

CS-GY 6233 - Fall 2023

Final Project

General Notes

Goal

Breakdown

1. Basics
2. Measurement
3. Raw Performance
4. Caching
5. System Calls
6. Raw Performance
7. Submission Requirements
8. Conclusion

Frequently Asked Questions

General Notes

- Read the requirements carefully, especially what to include in your final submission for each part.
- Refer to the submission instructions on the bottom of this document.
- Anticipating some of your questions, there is a section on FAQs.
- Working together with at most one partner allowed (working alone also allowed)
- The environment for this project is real Linux (We recommend a recent Ubuntu, but really any recent distribution should work). You can use C or C++.

Goal

The project is centered around performance.

We will try to get disk I/O as fast as possible and evaluate the effects of caches and the cost of system calls. In the end you should have a good understanding of what sits in the way between your process requesting data from disk and receiving it.

Breakdown

1. Basics

- Write a program that can read and write a file from disk using the standard C/C++ library's `open`, `read`, `write`, and `close` functions.

  - Add parameter for the file name;

  - Add parameter for how big the file should be (for writing);

  - Add a parameter to specify how much to read with a single call (block size);

Way to execute: `./run <filename> [-r|-w] <block_size> <block_count>`

2. Measurement

When measuring things, it helps if they run for "reasonable" time. It is hard to measure things that run too fast as you need high-precision clocks and a lot of other things can affect the measurement. It is also annoying to wait for a long time for an experiment, especially if you have to do many experiments. For this reason you should make sure that your experiments take "reasonable" time. We recommend something between 5 and 15 seconds.

- Write a program to find a file size which can be read in "reasonable" time.
  - Input: block size
  - Output: file size

Way to execute: `./run2 <filename> <block_size>`

Returns `block_count` (We will also accept `file_size`)

Hint: You can start with a small file size and use your program from (1) above to read it and measure the time it takes. You can keep doubling your file size until you get between 5 and 15 seconds.

Extra credit idea: learn about the "`dd`" program in Linux and see how your program's performance compares to it!

Extra credit idea: learn about Google Benchmark – see if you can use it (note: I have not tried).

3. Raw Performance

Because you would be looking at different file sizes when using different block sizes, it makes sense to use units for performance that are independent of the size. Use MiB/s (this is megabytes per second).

- Make your program output the performance it achieved in MiB/s

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

We will not mandate a way to run this... anything that works for you to produce the graphs.

4. Caching

Once you have a file of "reasonable" size created, reboot your machine.

Call your program to read the file and note the performance.

Call your program to read the file again (immediately after) and note the performance.

Ideally you should observe the second read be much faster, assuming the file can fit in your physical memory. This is the effect of caching.

Experiment with clearing the caches and not.

- Make a graph that shows performance for cached and non-cached reads for various block sizes.

NOTE: 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"`.

Extra credit: Why "`3`"? Read up on it and explain.

We will not mandate a way to run this... anything that works for you to produce the graphs.

5. System Calls

If you have a very, very small block size (e.g. 1 byte), most of the time would be spent trapping into the kernel. We talked in class about how system calls are expensive. Let's quantify it!

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

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

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

We will not mandate a way to run this... anything that works for you to produce the graphs.

6. Raw Performance

Try to optimize your program as much as you can to run as fast as it could.

- Find a good enough block size?

- Use multiple threads?

- Report both cached and non-cached performance numbers.

Way to run this should be `./fast <file_to_read>`. Output should be the xor value.

Extra credit idea: Figure out how to do this on high-end hardware. You can try "m5d.metal" instance on AWS cloud. This normally costs money ($5/hour) but if you use spot pricing you can get it as cheap as $1/hour. Also it is charged by the minute, so you can bring it up, run your experiment, shut it down for cents on the dollar. Also, Amazon has a program that gives free credits to students... Not sure what their turnaround time is, but you can try!

Make sure you use the NVMe drive which is attached to it. You may find this article I wrote a while back useful:

Configure AWS EC2 Automation

Submission Requirements

- PDF of the report

- zipped file of the code (for each part)

- the regular "partner" stuff to make sure we know who worked with whom

- Instructions on how to run. If the program does NOT run or has errors, it will not be credited.

Conclusion

This is a living document and will be updated as many of you have questions. Don't hesitate to put comments right on the document. Ideas for extensions are also welcome!

Frequently Asked Questions

Q. Is it necessary to use block count when in read mode?

Yes!

`./run <filename> [-r|-w] <block_size> <block_count>`

Always use 'block_size' and 'block_count'` as specified in the command line above.

File size is just 'block_size' * 'block_count'!

If you are reading, make sure you have a file big enough for that... it's size should be at least `block_size * block_count`! One way to do this is to create a large file first and then use it for all your reads.

Maybe add a check in the beginning of the program to see if the file has enough bytes and exit if the file is not big enough to accommodate reading that many blocks of the specified size!

If you did your analysis and produced numbers in a different way, as long as you used the right block size you should be fine for graphs and performance metrics. But make sure that you have a program that reads just the right amount for grading purposes so the TAs can check your XOR values.

Q. Can you provide a sample file and the correct XOR result so we can test our program with it?

Maybe many people can download a commonly available large file like this one ... and compare their XOR values

Q. Is the blockSize argument always a multiple of 4? Since an integer is 4 bytes?

We will always use multiples of 4, but you will have to run some experiments with `block_size` for one of the problems... just make sure your code does not crash on it. When we run for performance we will always use a multiple of 4.

Q. Does it matter what content we write to the files?

No. You can write whatever happens to be in memory inside your buffer – no need to put in anything special in there.

Q. Should we be measuring CPU time or wall time?

Wall time! CPU time is likely to be close to 0 since the problem is I/O Bound... unless you use a really small block size.

Q. If the inputs are then same, is it okay that xor is different?

No, XOR is a deterministic function and produces the same output for the same input.

Q. Does the file we read have to be in a specific format?

No special format required, but you should be reading the file as a binary... every 4 bytes being treated as an integer. Linux already does that with `open`, `read`, etc... Don't use `fopen`, `fread` and such — those are not system calls they are C library functions which internally call `open` and `read` but might be doing their own buffering and other trickery.

Q. Raw Performance — how to?

Let's have everybody produce a program or a script called `fast` which accepts a single parameter — a file name. The program should read the file as fast as it can, compute the XOR for the whole file and print it out. The size of the file will be a multiple of 4.

```
./fast <file_to_read>
```

If the correct XOR number is printed we will compute the performance of the read as part of grading by dividing the size of the file to the time taken and will create a wall of fame for the whole class. We will publish the top 10 results.

Q. Example code for XOR

```
#include <stdio.h>
unsigned int xorbuf(unsigned int *buffer, int size) {
    unsigned int result = 0;
    for (int i = 0; i < size; i++) {
        result ^= buffer[i];
    }
    return result;
}
#define SIZE 100
```

```
int main()
{

    unsigned int buffer[SIZE] = {};

    // random initialization
    for (int i = 1; i < SIZE; i++) {
        buffer[i] = buffer[i - 1] * 31 + 17;
    }

    // compute xor one way...
    printf("%08x\n", xorbuf(buffer, SIZE));
    // compute xor another way... (as if 2 threads to half each)
    unsigned int xor1 = xorbuf(buffer, SIZE / 2);
    unsigned int xor2 = xorbuf(buffer + SIZE / 2, SIZE / 2);
    printf("%08x = %08x ^ %08x\n", xor1 ^ xor2, xor1, xor2);

    return 0;
}
```