Olivia Hess
Operating Systems
Thursday, November 29th
Final report

The question I set out to answer is: How consistent is data reading and writing within an OS?

The way I aim to answer this question is by setting up an experiment in which I write a bit (for example, 0) into a sequential set of 10,000 memory locations, and time how long it takes. I will then read a sequential set of 10,000 memory locations and time how long it takes. Next, I will randomly generate 10,000 memory locations, and keep track of those locations in an array. I will then repeat the previous experiment, writing the same value into each 10,000 memory locations and timing how long it takes. I will do the same for reading from each one.

Because I would be running this experiment in user space, then all of the addresses I would be getting are *virtual addresses* and not *physical addresses.* More so, with virtual memory, you cannot assume any address to be a valid address, as the memory manager returns valid addresses to the compiler, which gives it to the user program- not the other way around.

In C++, it is certainly possible to write and read from specific memory addresses with the proper implementation, but after experimenting it became clear that I would be getting segfaults the majority of the time, because the program won't initially have access to the memory address I am manually specifying.

As a result, I decided to *simulate* memory locations using an array. I created to arrays, int sequential[] and int random[], both of size 100,000. These arrays simulate my range of addressable memory locations.

To simulate reading and writing from a sequential set of memory locations, I had two loops- one read 10,000 elements from the array sequential[], and the other wrote 10,000 "0"s into the first 10,000 indices (aka memory locations) of sequential[]. I used the C++ library <time.h> to count the number of clock cycles that each loop took.

To simulate reading and writing from a randomly generated set of memory locations, I had two similar loops. But first, I had a loop that generated 10,000 random numbers from 0-99,999. I filled an array with these 10,000 numbers.

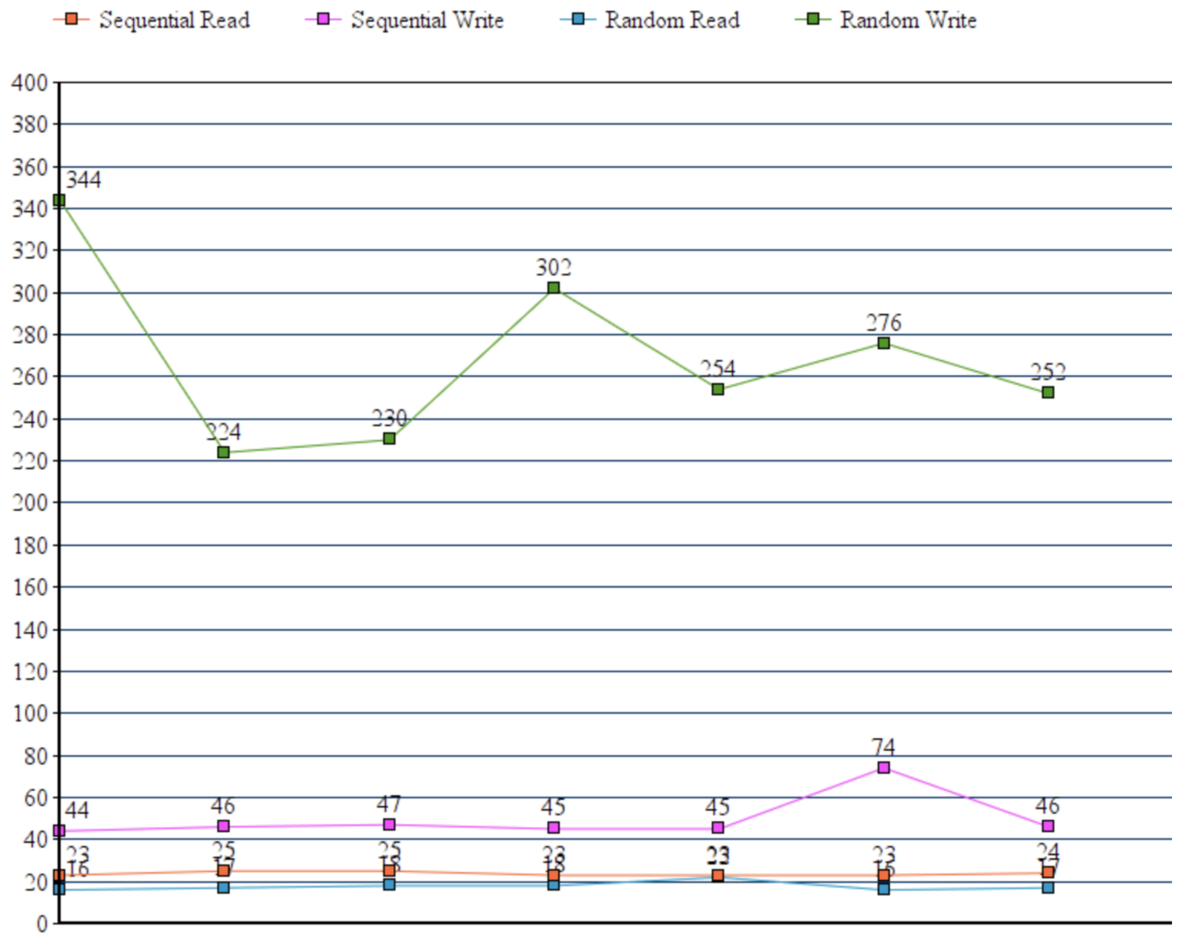
To write into random[], I used the array of randomly generated numbers as indices. I used a similar process as above.

Results

Sequentially writing to 10000 locations, It took **44 clock ticks** and **4.4e-05 seconds**
Sequentially reading from 10000 locations, It took **23 clock ticks** and **2.3e-05 seconds**
Randomly writing to 10000 locations, It took **344 clock ticks** and **0.000344 seconds**
Randomly reading from 10000 locations, It took **16 clock ticks** and **1.6e-05 seconds**

Analysis

The following graph represents 8 trials of running the code.



After running the experiment multiple times, it becomes clear that randomly writing to a set of memory locations takes the most amount of time. Sequentially writing takes significantly less time, however as we have learned in class, it doesn't make as much sense for the operating system to write into memory locations sequentially because those memory locations will eventually be freed up.

I was surprised that reading from both sequential and random indices took around the same number of clock cycles. I expected reading from random indices to take more time, as writing to them took so much time. In a real simulation of memory units, this would likely be different, I suspect that my method of "reading" was quite trivial and the majority of the clock cycle was spent on the loop rather than the actual reading.

Nonetheless, this little experiment was a great way for me to think about the way virtual memory and physical memory works on the computer, and I came to the conclusion that writing to memory locations is **not** consistent within an OS, as I determined from the wide range of values I got for my clock cycles. This is due to small interrupts that the OS experiences when running programs.