Operating Systems: Project 5 Report

Courtney Kelly and Katie Schermerhorn

April 7, 2017

Purpose

The purpose of these experiments is to demonstrate and visualize how both the number of frames in a virtual memory system and the page replacement algorithm can greatly impact the overall performance of an Operating System. We did this my keeping track of the number of reads from disk, the number of writes to disk, and the number of page faults. These are all indicators of the number of cache misses.

Therefore, our goal in picking a replacement policy for this cache is to minimize the number of cache misses, i.e., to minimize the number of times that we have to fetch a page from disk. Knowing the number of cache misses let us calculate the average memory access time (AMAT) for a program, which is a way to measure the performance of an Operating System. So, having low numbers of reads, writes, and page faults means a low number of cache misses and therefore a low AMAT (good!).

Program Setup

To set up this experiment, we first built the `main.c` program. This required us to modify two functions: the main function and the page fault handler. In the main function, we first get a time seed, which we later need to implement the random page replacement policy. Then we parse the command line arguments and initialize the summary components (i.e., the number of disk reads, writes, and page faults) to 0. We do some error checking to make sure valid values are passed through the command line.

The third argument of the command line calls for the page replacement algorithm that the user wants the program to use. In the case where there are no open frames available for a new page, this algorithm tells the program which page to evict. In RAND mode, the program will choose a frame at random. In FIFO mod, the program will utilize the "first in, first out" method to evict the page which was put into the frame first. The algorithm we implemented "CUSTOM" is similar to the FIFO algorithm but also checks if a page has been recently written to. If the CUSTOM mode is chosen, the program will initialize a array of size of the amount of frames. We then filled with -1s. In the handler function, when a write is made to a page, the -1 is changed to a 0 in the "writtento" array for that specific frame. When a page needs replaced, it will run the same FIFO algorithm, but if that frame equals 0 in the "writtento" array, then it will move to the second-in page. It will then change the 0 to a -1, so that the next time the algorithm tries to evict that page, it can.

Experimental Setup

To measure the performance of each page replacement policy on each program, I used a shell script to run all three programs on a specific policy. This shell script iterates through each program, iterates through ten possible frames numbers 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, and runs `virtmem` with those parameters. The output is then searched, and written to csv file. The script is as follows:

```
#!/bin/bash
```

```
pol=$1
echo "# Frames, # Reads, # Writes, # Page Faults, Algorithm" >> ${pol}.csv
for alg in "sort" "scan" "focus"; do
    for i in 10 20 30 40 50 60 70 80 90 100; do
        out=$(./virtmem 100 $i $pol $alg)
        faults=$(echo "$out" | grep "page faults" | cut -d " " -f 5)
        reads=$(echo "$out" | grep "reads" | cut -d " " -f 5)
        writes=$(echo "$out" | grep "writes" | cut -d " " -f 5)
        echo "$i,$reads,$writes,$faults,$alg" >> ${pol}.csv
    done
done
```

I ran this scripts three times on each page replacement policy as follows:

```
./run_tests rand
./run_tests sort
./run_tests custom
```

This generated three .csv files, which I was then easily able to upload to Google Sheets and visualize the results. You can see these figures in the following sections, where there will each be discussed.

FIFO Performance

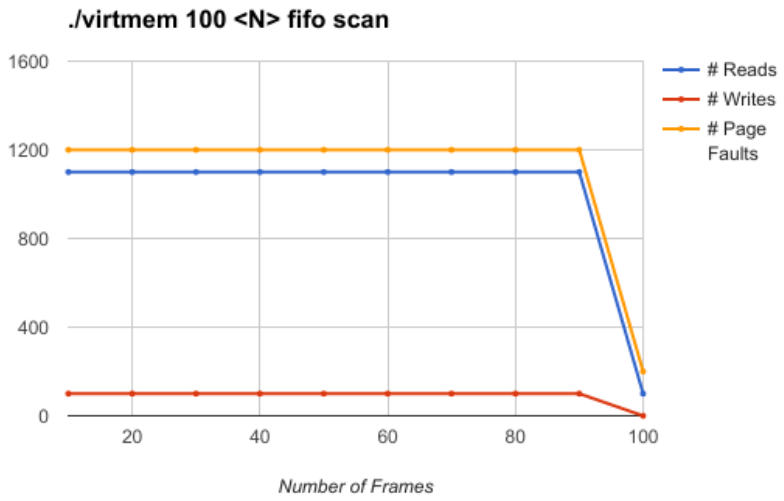

Figure 1: FIFO Page Replacement on Scan Program Graph

As you can see when we run the fifo page replacement algorithm with the scan program, the number of reads, writes, and page faults stays fairly constant. From this we can infer, that the scan program is accessing memory in a way that is counter-acting the fifo algorithm. The scan program is probably full of compulsory misses, so the program consistently page faults, and then either reads or writes to disk. The performance is only improved when the number of frames increases enough to accommodate all of the cache misses. From Figure 1 we can see this occurs at frame sizes greater than 90.

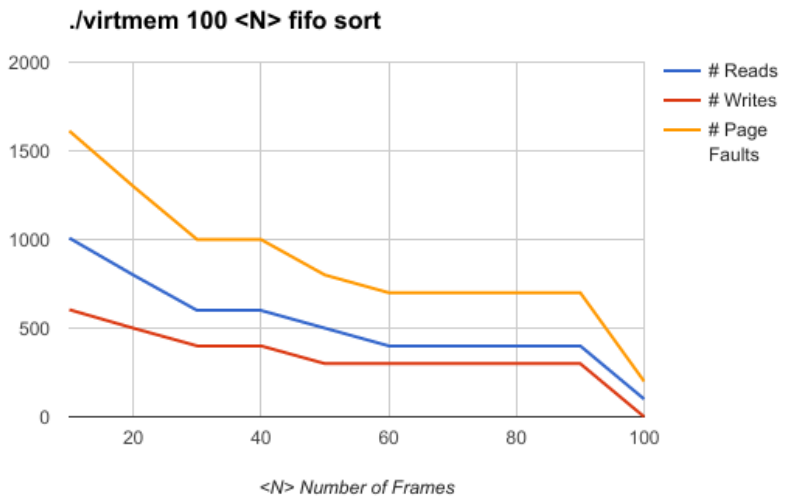

Figure 2: FIFO Page Replacement on Sort Program Graph

Here we can see that when we run the fifo page replacement algorithm with the sort program there is a downward trend as the number of frames increases. This makes sense because when there are more frames, there's more room for pages, and therefore less cache misses. The performance seems to plateau between 60 and 90 frames. Similar to the scan program, the sort program probably has a chunk of compulsory misses that can only be resolved by a large enough frame size to accommodate them all, i.e. greater than 90 frames.

Figure 3: FIFO Page Replacement on Focus Program Graph

Here we can see that when we run the fifo page placement algorithm with the focus program also trends downward as the number of frames increases. However, the trend is much smoother than we've seen before. This is because of the way the focus program accesses memory. It's cache misses are independent and can be resolved a little bit at a time when you increase the number of frames.

Rand Performance

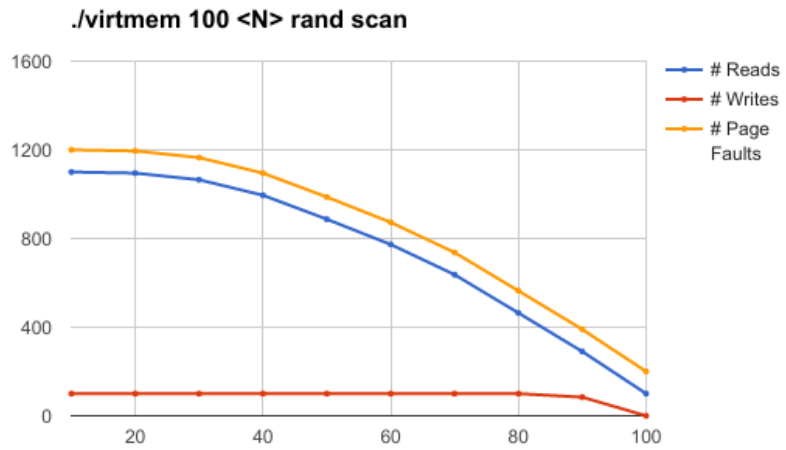

Figure 4: Random Page Replacement on Scan Program Graph

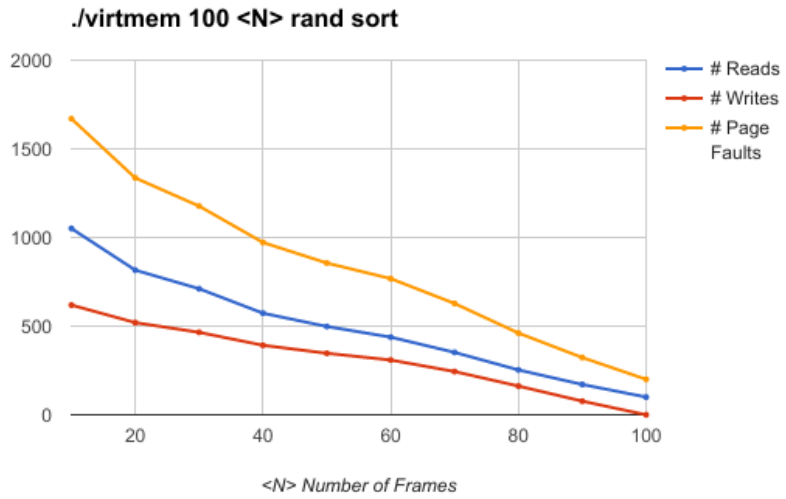

Figure 5: Random Page Replacement on Sort Program Graph

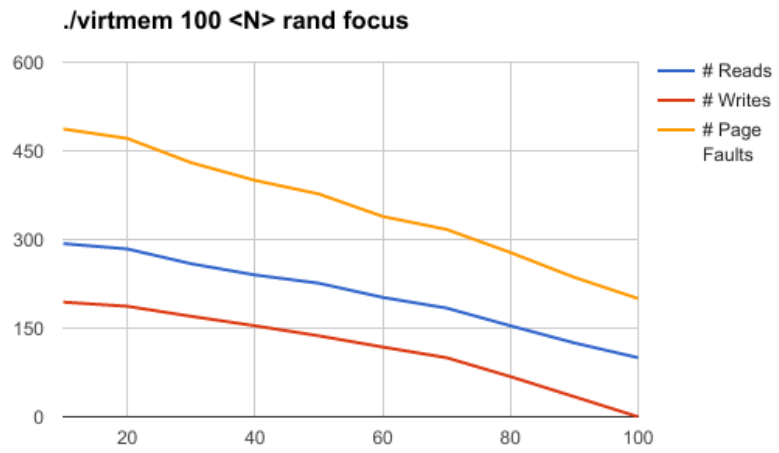

Figure 6: Random Page Replacement on Focus Program Graph

In Figures 4, 5, and 6, we can see the trend of the random page replacement algorithm is pretty consistent. It steadily trends downward as the number of frames increases for all three programs: scan, sort, and focus. As stated before, this makes sense because as the number of frames increases, there is more space to accommodate all the needed pages, and thus we get more cache hits and less cache misses. The trendlines are smooth because there isn't anything in the programs to counteract the random policy as seen with the fifo policy.

Custom Performance

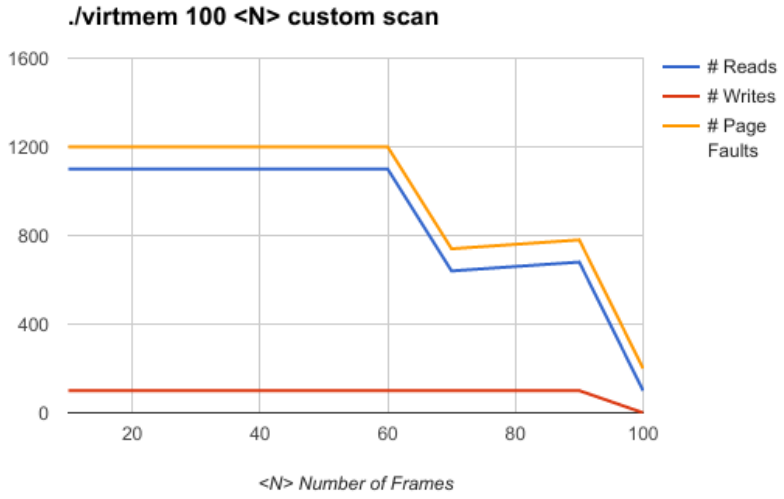

Figure 7: Custom Page Replacement on Scan Program Graph

Figure 8: Custom Page Replacement on Sort Program Graph

Figure 9: Custom Page Replacement on Focus Program Graph

In Figures 7, 8, and 9, we can see the downward trend of the custom page replacement algorithm is pretty consistent for the sort and focus programs. But what is going on with the scan program? As you can see in Figure 7, there are a very low number of disk writes. The custom program functions by popping off a page that hasn't been written too. But if no pages have been written too, it'll pop off the first frame every time. This results in a pretty horrible performance; one that can only be improved with a larger number of frames. The custom algorithm does much better on the sort and focus programs. This makes sense, because there are a larger number of reads and writes.

Evaluation

As we can see in the above results, there is no page replacement algorithm that consistently performs better than the others. Performance depends on how the program running accesses memory and the number of frames. Overall, all the graphs show having more frames improves performance. However, we're only running one program at a time. Only having a really large number of frames to improve performance might become infeasible when running multiple processes at a time. Therefore, we also need a solid page replacement algorithm to also help improve performance. As you can see in Figures 1, 4, and 7, the random page replacement algorithm performs best with the scan program. This is because the scan program accesses in a way such that the fifo and custom algorithms are don't improve performance. We can also see form these graphs that there are a very low number of writes to disk. The custom algorithm is based on the which pages have recently written to disk, so if there aren't that many writes to disk in the first place, it makes sense that the custom algorithm would be ineffective.

As you can see in Figures 2, 5, and 8, all three page replacement algorithms perform equally well with the sort program. It could be argued that the custom algorithm performs best because the trendline is a bit steeper than the others. This means that as the number of frames increases the custom algorithm has a greater impact on performance. This again relates to how the sort program accesses memory. We can see from the Figures that there are a good number of reads and writes from/to disk. Since the custom algorithm takes into account which pages have recently been written too (but not recently read), it makes sense that this algorithm would perform slightly better.

As you can see in Figures 3, 6, and 9, all three page replacement algorithms perform similarly with the focus program. Fifo performs slightly worse than the custom and rand algorithms. This again has to do with the way the focus program accesses memory. It seems to function similarly to the sort program in terms of the number of reads and writes. So it makes sense that the random and custom algorithms would perform better. The fifo algorithm performs worse because the way focus accesses memory must be in a way that counteracts the first in, first out method.