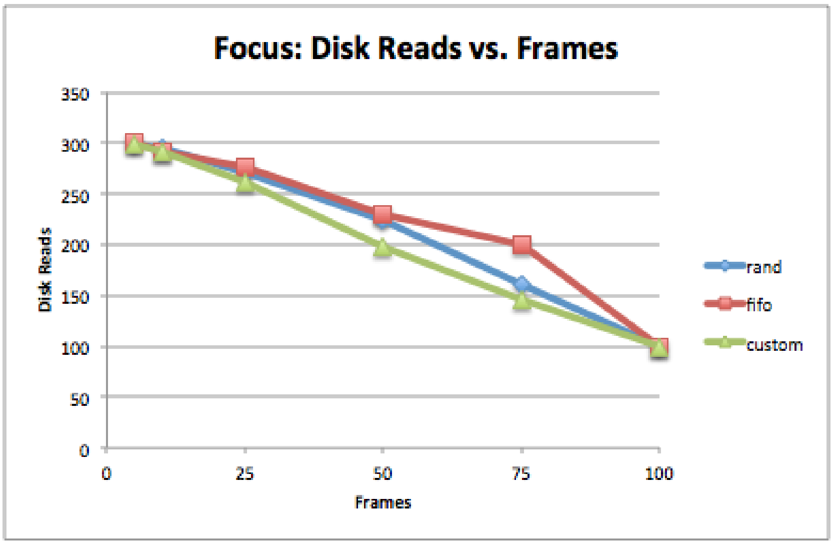
Teddy Brombach
Tristan Mitchell

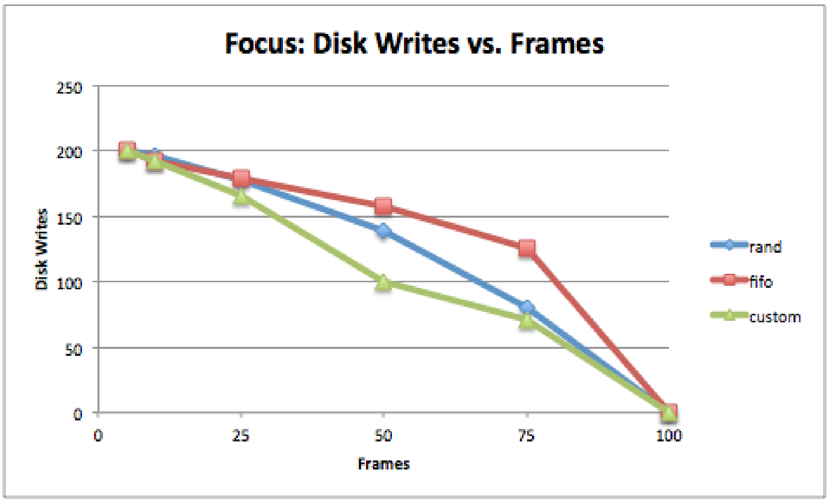
Project 5 Report

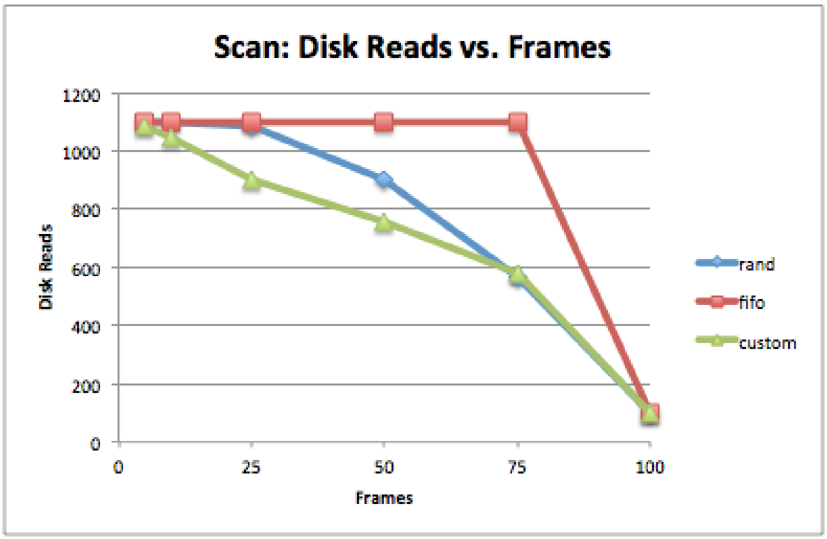
In this project, our goal was to write a page fault handler for a virtual memory simulation. Given a certain number of pages of virtual memory and a certain number of frames of physical memory, our handler's job was to map the pages into the available frames in order to allow the running program to read and write data to and from the disk appropriately. Our experiments for this project were performed while ssh-ing into student02@cse.nd.edu from our laptops, and our command line arguments were formatted exactly as laid out in the project document ("./virtmem <npages> <nframes> <rand|fifo|custom> <scan|sort|focus>" - after running "make").

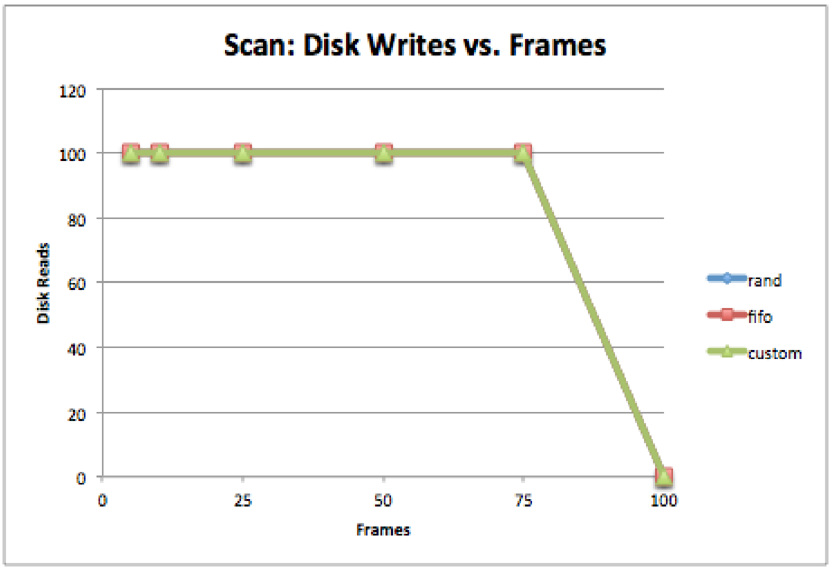
In addition to implementing simple random and fifo page replacement policies, we also developed our own custom page replacement algorithm that outperforms those two. The way our custom algorithm works is that the page fault handler keeps track of how many times each page has caused a fault before. When the physical frames are full but a new page needs to be pulled in, the frame currently in physical memory with the fewest number of previous faults is removed. The idea behind this is that if a certain page of data causes a lot of faults, the program is trying to access it a lot, so such pages should be prioritized for holding frames in physical memory.

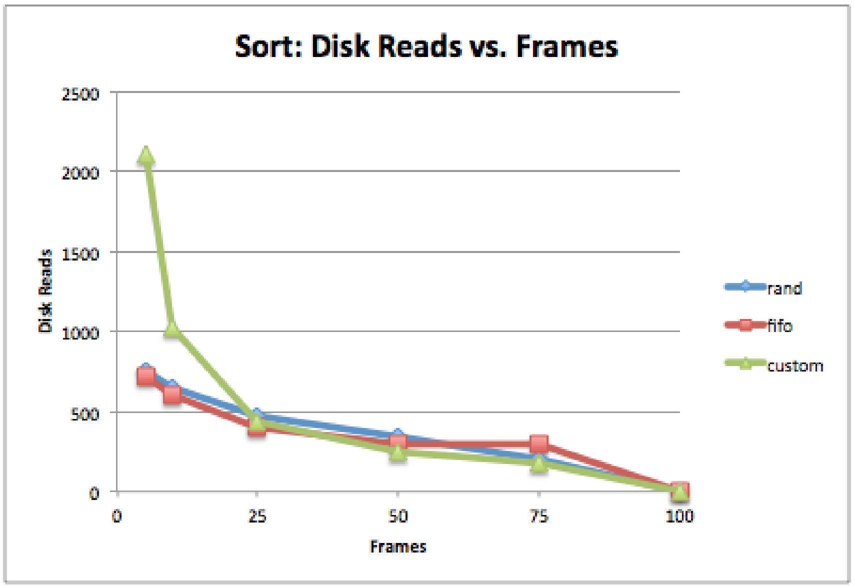
We assessed the performance of our algorithm by comparing the number of disk reads and writes required by each one for all three programs at npages = 100 and nframes = each of [5, 10, 25, 50, 75, 100]. Having fewer such disk operations was taken to mean better performance. Our test data is illustrated in the graphs below:

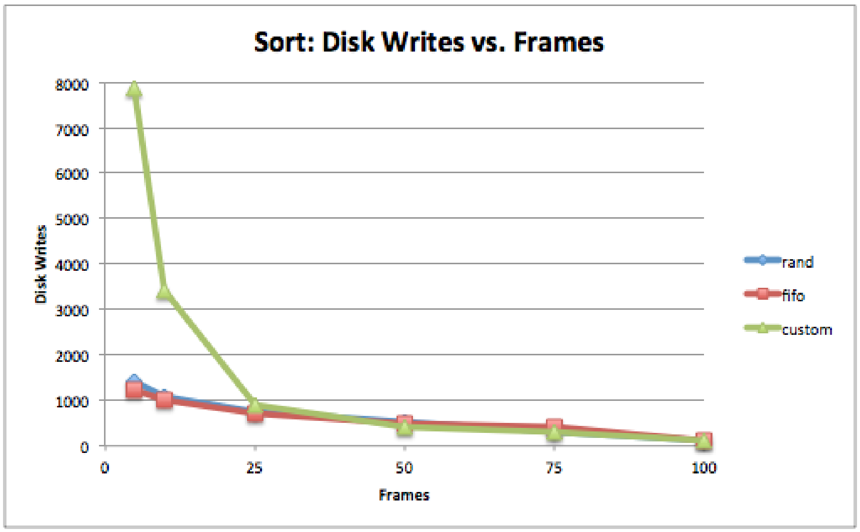



Focus: Disk Writes vs. Frames
Disk Writes
250
200
150
100
50
0
0
25
50
75
100
Frames
rand
fifo
custom


Scan: Disk Reads vs. Frames
Disk Reads
1200
1000
800
600
400
200
0
0
25
50
75
100
Frames
rand
fifo
custom


Scan: Disk Writes vs. Frames
Disk Reads
120
100
80
60
40
20
0
0
25
50
75
100
Frames
rand
fifo
custom





From these results, one can see that our custom algorithm was almost always the best of the three. It was either better or just as good as the others for every test except for when the sort program was run with fewer than 25 frames. The differences between the graphs can be explained by the different memory access patterns used by each of the three programs. Sort, for example, used a call to qsort, which works on small contiguous sections of memory at a time but moves around often. For small numbers of frames under our custom algorithm, this means that the physical memory would hold on to the recently used pages and wait until the pages in the new region caused a sufficient number of faults before evicting the old ones, leading to the very large numbers of both reads and writes seen above. Focus, on the other hand, only looked through small chunks of data at a time, which gave our algorithm the biggest advantage, and led to the smallest number of faults of any program. Finally, scan looked across all pages of data sequentially multiple times, causing the flat FIFO reads graph and the flat disk writes graphs.

This is the worst pattern of accesses for FIFO, because every time it kicks out a page it is actually kicking out the most likely page to be used next.

Another pattern that might seem strange in the graphs above is that all the algorithms always converge to a point when there are 100 frames, but this is expected. When the number of physical frames is the same as the number of virtual pages, every page can be stored at once, requiring only (in this case) 100 reads to pull everything in, and 0 writes back to disk (nothing gets kicked out). All these things considered, the results still show that our custom algorithm is on average better than both of the other two. By prioritizing the pages that have caused the most faults in the past, our algorithm keeps the most important program data in physical memory.