OS Project 05 Report

Brittany DiGenova and Collin Klenke

I. Purpose:

The overall goal of this project is to determine how organization of virtual memory can improve execution time of a program. It is importance to have an effective amount of frames in memory such that page faults are minimized (in order to reduce costly disk accesses) while also balancing space demands of the program. Even though running ./virtmem with the same number of pages and frames resulted in no faults, it also eliminated the memory saving effects of virtual memory access. By designing FIFO, rand and an algorithm of our own we gained a better understanding of efficiently designing memory in an operating system.

II. Setup:

Machine – Student01
Command Line Args – ./virtmem npages nframes algorithm program
(Detailed in the Results section)

III. Custom Algorithm:

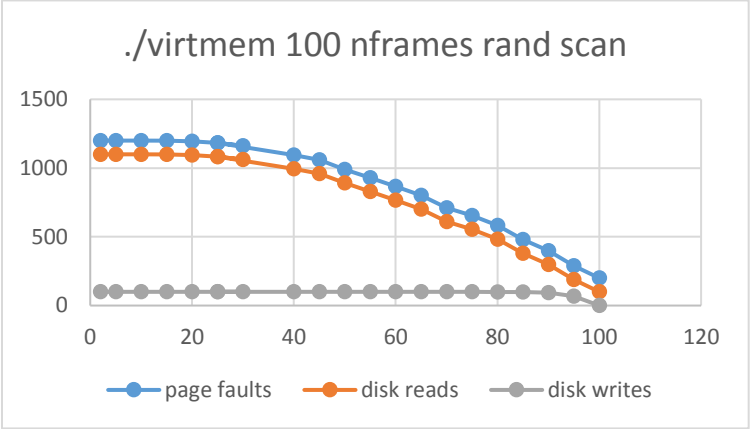
For our custom algorithm implementation we designed a least recently used model with a twist to favor keeping files with dirty bits in memory. By keeping frames with dirty bits (write bit set) in memory, we reduced the number of writes to disk. When the read only frames are kicked out they do not have to write to disk and therefore are more efficient. We joined this concept with the idea of least recently used to create our final algorithm.

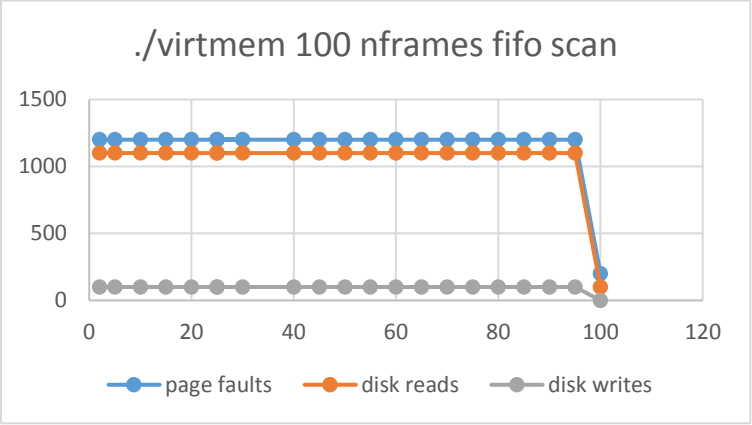
Pages stored in memory are kept track of in a linked list. Pages near the front of the list are less likely to be ejected on a page fault, and pages near the back of the list are more likely to get ejected on a page fault. On a read fault, the list kicks out the last node in the list (which represents the least recently used, non-written node). The new page is then placed in the middle of the linked list. It should begin in a neutral location since it is read only (making it more likely to be kicked out), but also recently used (making it less likely to be kicked out).

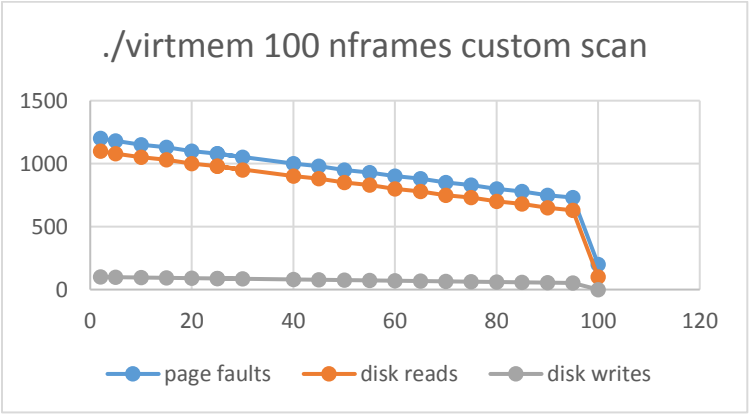
On write faults the node containing the page to be written to is moved to the front of the list. This gives preference to it both for being written to, and for being recently used. The algorithm performed better in almost every case for both disk reads and disk writes.

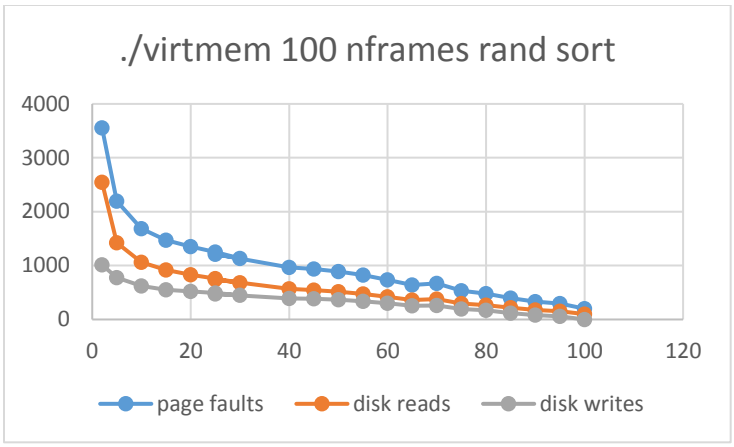
IV. Results

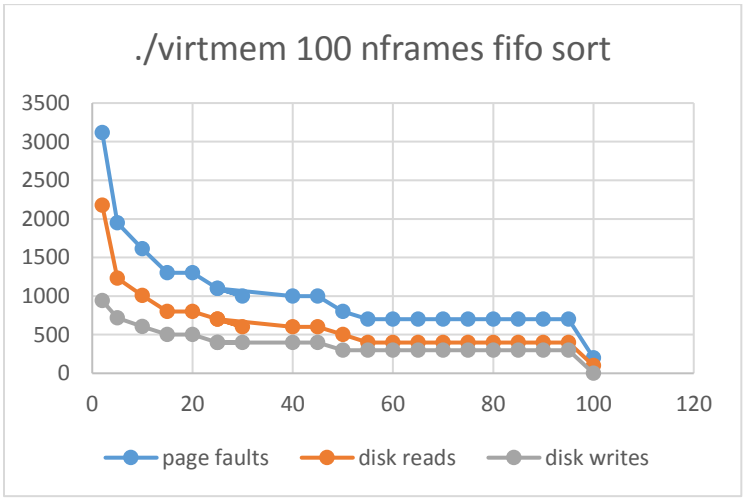
Note: In all graphs X axis is number of frames and Y axis is memory access information

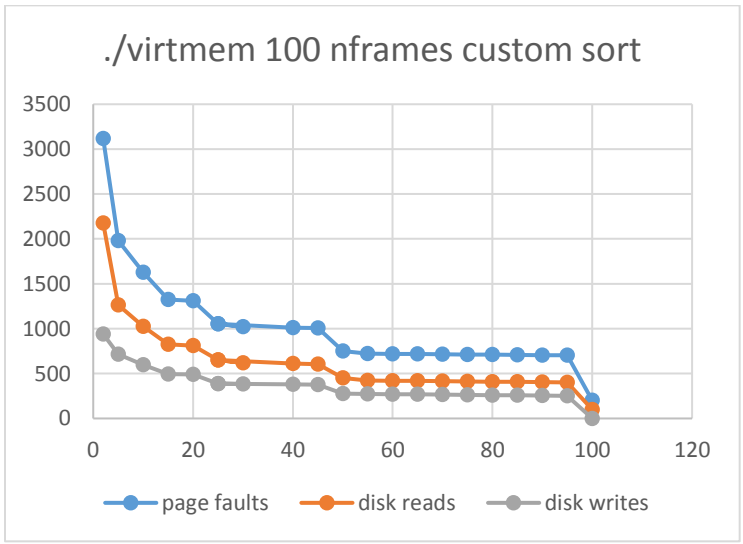


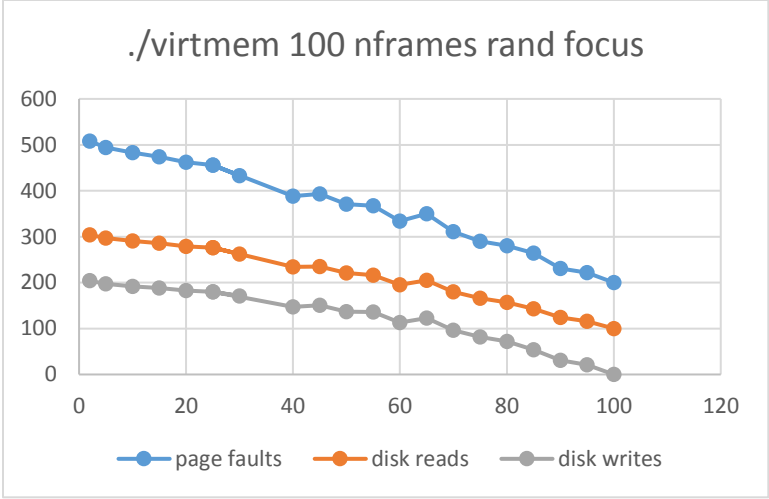


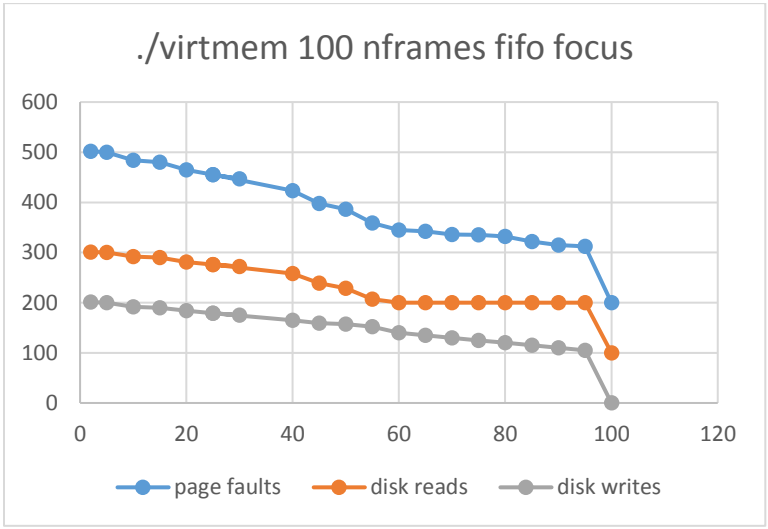


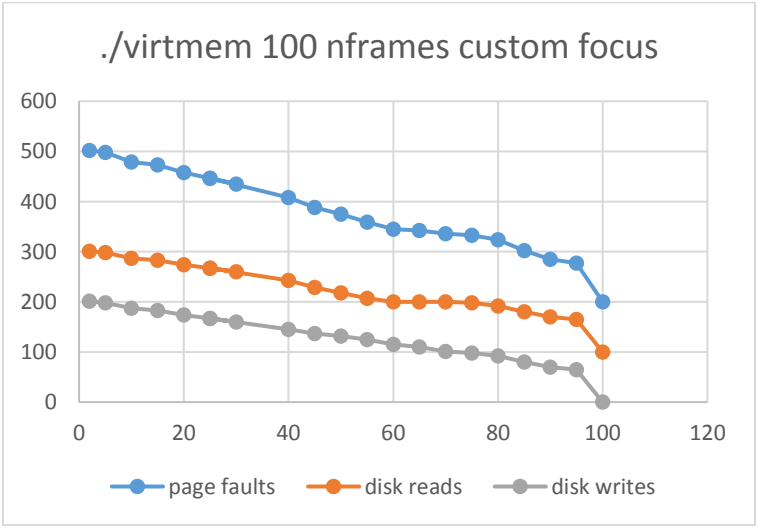

./virtmem 100 nframes rand sort
4000
3000
2000
1000
0
0
20
40
60
80
100
120
page faults
disk reads
disk writes


./virtmem 100 nframes fifo sort
3500
3000
2500
2000
1500
1000
500
0
0
20
40
60
80
100
120
page faults
disk reads
disk writes


./virtmem 100 nframes custom sort
3500
3000
2500
2000
1500
1000
500
0
0
20
40
60
80
100
120
page faults
disk reads
disk writes


./virtmem 100 nframes rand focus
600
500
400
300
200
100
0
0
20
40
60
80
100
120
page faults
disk reads
disk writes


./virtmem 100 nframes fifo focus
600
500
400
300
200
100
0
0
20
40
60
80
100
120
page faults
disk reads
disk writes


./virtmem 100 nframes custom focus
600
500
400
300
200
100
0
0
20
40
60
80
100
120
page faults
disk reads
disk writes

V. Discussion:

In all the tests there were significantly lower amounts of disk writes than disk reads or page faults. This is to be expected because write backs to disk only occur when pages being replaced have a dirty “write” bit. There are far more page faults than disk reads in our tests because of the way we handle page faults. If a page fault occurs on a read only page, write access is added and the page returns which means there can be more page faults than disk reads.

Sort, which was the program with the highest number of page faults, performed comparably for all three algorithms. For this program it would make sense to have about half as many frames as pages. The performance increases only slightly adding frames after this point. However, the performance increases rapidly when going from two to five frames. I would assume that this id due to the fact that this program implements a sorting algorithm. Therefore at the beginning it will be looking at all the data and page faulting often, and as it sorts, and more of the data has been processed it will look at less and less of the data to sort in the future creating the inversely exponential graph.

The second program, scan, appears to have gone repeatedly through the data in the same sequence. This is evident because the fifo algorithm performs the same for any amount of frames less than pages. The page that is needed will never be in memory if there are less frames than pages when using fifo with scan. Our custom algorithm got better linearly, outperforming random page replacement up until a 6/10 frame to page ratio. This is due to the way that we implement our algorithm by inserting pages in the middle of the linked list. If nothing is being written to then the behavior is partially fifo resulting in little increase in performance.

For the third program focus all algorithms followed a similar pattern of improvement when adding frames. Our algorithm did slightly better in almost every case. All cases improved only gradually when adding frames. Fifo also showed no improvement once it reached a 6/10 frame to page ratio. This is most likely for similar reasons to it’s failure in scan. Overall our custom algorithm outperformed the others consistently. When only using 100 pages it is hard to see a drastic difference but it was better in almost every case. Some trouble occurs when no writes are done as that is a large portion of our improvement algorithm. In future cases where we understand the types of programs being used better, we could better predict the pages that will be used frequently