Operating Systems Project 2 Report: Jayce Houghton, Mark Yo

Introduction:

The primary problem of page tabling is implementing a paging policy that updates a page table in the most efficient way possible, such that you minimize the number of page faults. The easiest to implement but laziest method is using a simple first in first out policy, though this method leads to the most page faults due to not considering any information about a page's usage other then when it entered the queue. The next policy implemented is LRU or Least Recently Used, which considers what pages are being constantly accessed. Finally, we implemented the VMS policy, which is a version of fifo where two processes share the same memory space. In our implementation we manipulate arrays for LRU and Fifo and implement a linked list for VMS. The purpose of this experiment is to determine the best policy for the most situations and compare their effect on multiple memory traces.

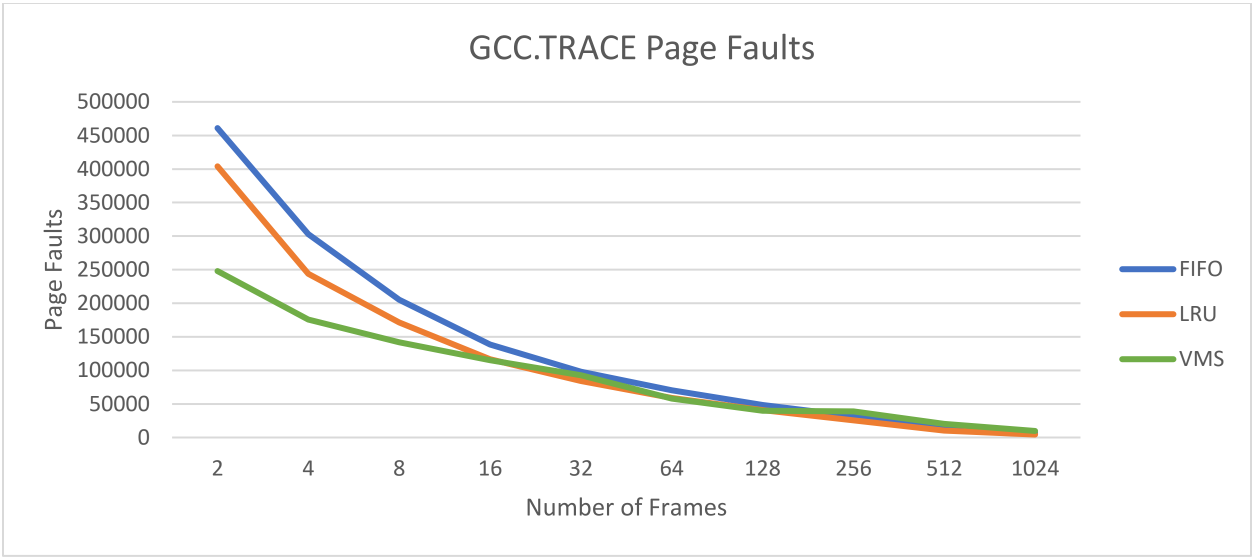
Methods:

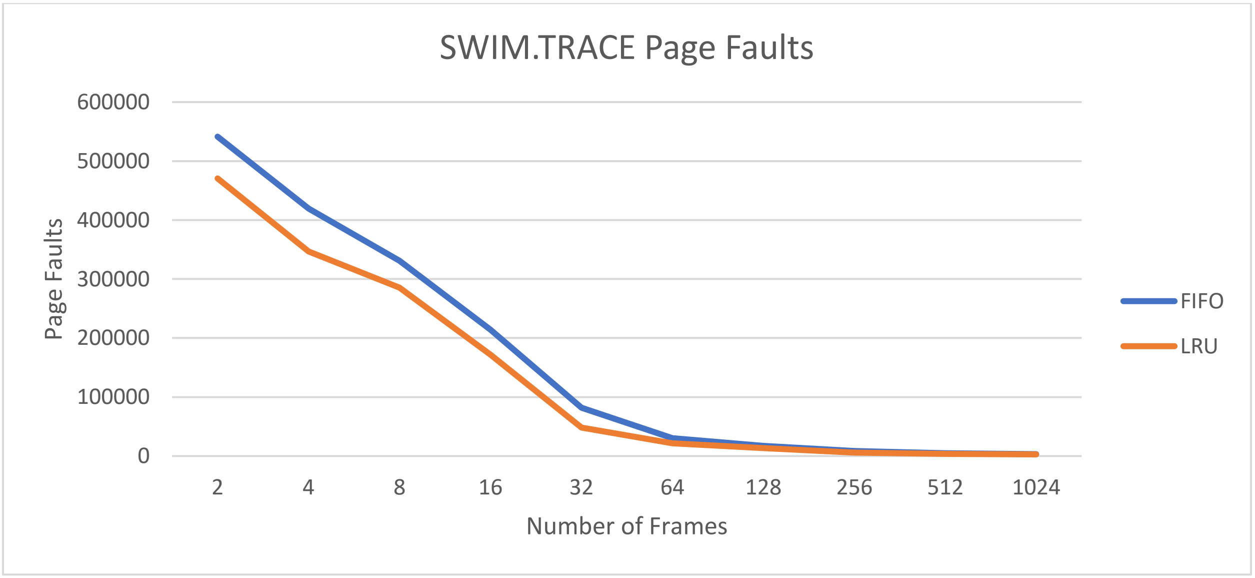
To obtain information about the provided memory traces we implemented FIFO, LRU, and VMS policies, when in VMS we considered pages starting with 3 as belonging to a separate process. We first needed to determine the amount of page bits allocated to the page number, which we determined using the length of the memory reference along with the provided information on page size. We used this information to extract the page number when looping through the memory traces. To learn about the memory traces, we ran each of our policies using nframes of size 2 to 2^10 in increments of powers of 2. In this way we were able to see how the number of frames effected the number of page faults, and how policy effected these results as well. We were also able to use this information to determine how much memory our processes needed as well as when increasing the number of frames no longer improved performance.

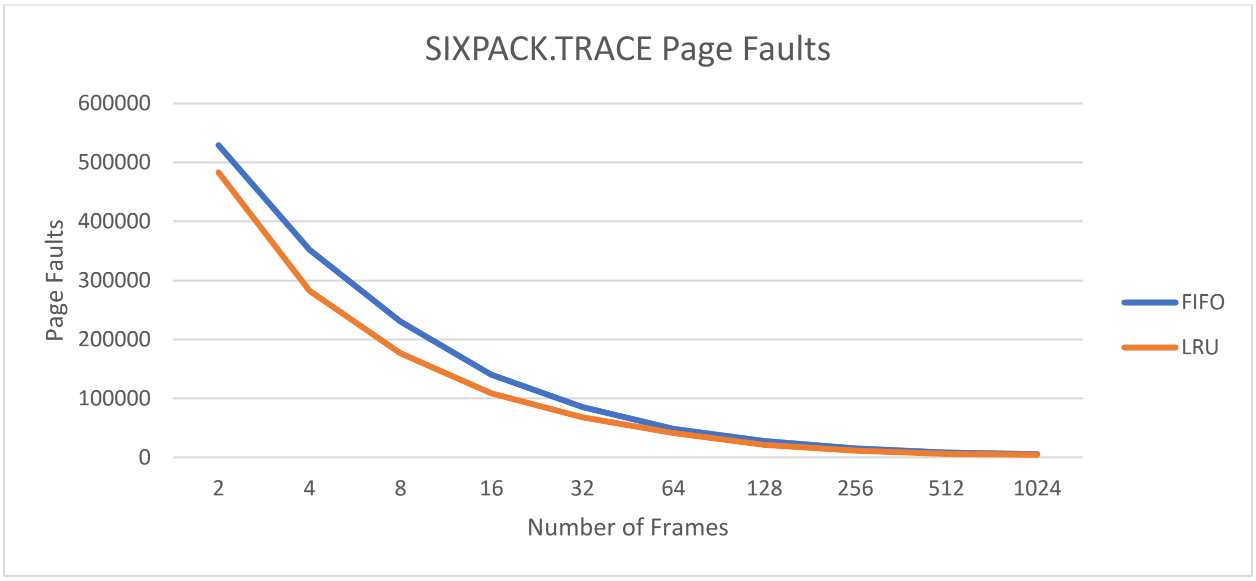
Results:

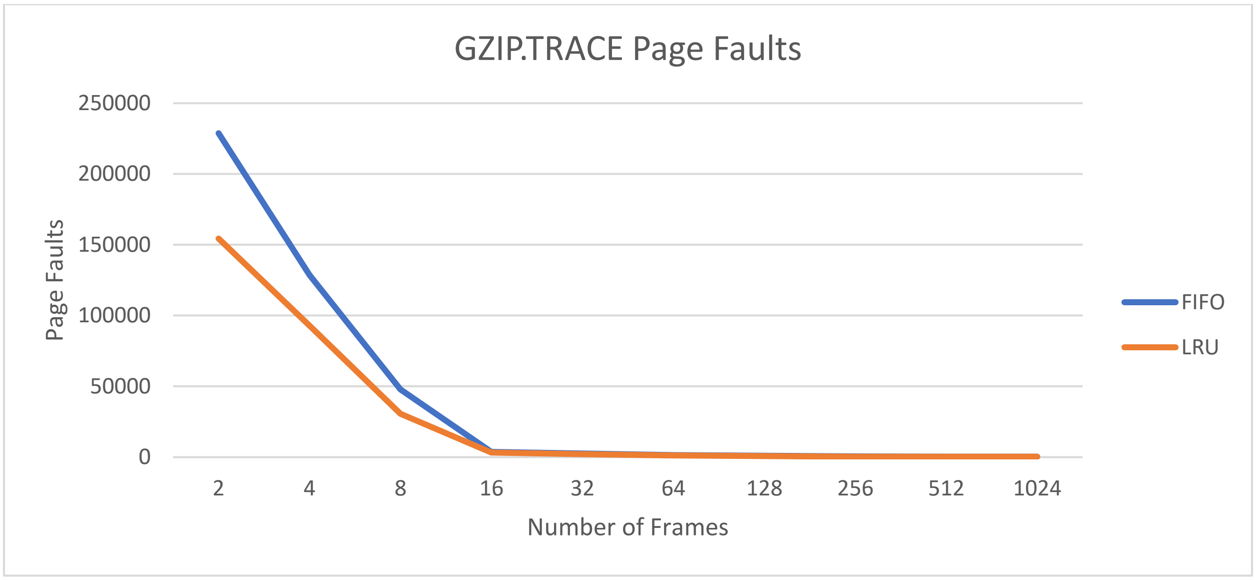
Graphs:

Note: We analyzed the number of page faults instead of the hit rate. The hit rate of each algorithm is simply the opposite of the page faults, and we found that analyzing our data in terms of page faults was equally as meaningful as we can see how increasing the number of frames effects (in most cases decreases) the number of page faults. It also helps to show how the different algorithm's when compared influence the number of page faults.









FIFO:

FIFO	Page Faults			
Number of Frames	gcc.trace	swim.trace	sixpack.trace	bzip.trace
2	460912	541458	529237	228838
4	302860	419509	351810	128601
8	205368	330893	230168	47828
16	138539	214295	140083	3820
32	98067	81638	85283	2497
64	70315	30422	48301	1467
128	48526	17523	27778	891
256	31698	8551	15440	551

512	15609	4850	8089	317
1024	6673	3264	5492	317

We tested FIFO on each of the 4 traces and counted the number of page faults at different intervals of nframes. FIFO lead to a logarithmic slope with the number of page faults decreasing faster during the first few increases of nframes but plateauing somewhat near the end. From this, we can analyze each of the traces. The gcc.trace seems to plateau the least out of all of the traces, suggesting that he has significantly more then 1024 pages, possibly even another factor of 2 nframes are necessary. Swim.trace on the other hand plateaus much faster than gcc.trace and likely less than another factor of 2 nframes are necessary to fit all pages in memory. Sixpack.trace's number of page faults is almost identical except for the fact that it begins with more page faults and ends with less despite showing an extremely similar curve. Likely the end of the memory trace progresses it begins accessing the same memory, but during the beginning it's access of memory is extremely variable. Finally, Bzip.trace plateaus starting at 16 frames, and completely levels starting at 512 frames. From this can we tell that bzip has less than 512 pages and once you pass this number all pages can fit in the memory space.

LRU:

LRU	Page Fault			
Number of Frames	gcc.trace	swim.trace	sixpack.trace	bzip.trace
2	403955	470679	483161	154429
4	243809	346936	282620	92770
8	171186	285375	176496	30691
16	116604	171961	108682	3344
32	84401	48254	67747	2133
64	59089	21656	41186	1264
128	40821	13250	21090	771
256	25308	5673	11240	397
512	10425	3632	5823	317
1024	4391	2803	4468	317

We analyzed LRU in a similar way to FIFO. The results and what they tell us about each trace are very similar outside of LRU being shown to be significantly more efficient in the grand scheme of things, so we will highlight biggest differences. The first notable finding is that the LRU policy leads to at least 50,000 less page faults on smaller nframe sizes but begins to align with FIFO on larger sizes. This leads us to the conclusion that the more pages you can fit in memory, the less important replacement policy becomes. One of the most interesting places we can see this is in the Bzip.trace. In this trace LRU is significantly faster up until 16 frames, where it plateaus in exactly the same as FIFO. Once we reach over 512 frames, the line becomes flat, showing that if all pages fit into the memory, it make no difference which page replacement policy is used. Another interesting thing that LRU analysis on bzip.trace shows us is that the number of frames in bzip is likely very close to 397 as we see this value a 256 right before the line becomes straight, so LRU gave us more information here than FIFO. In all other traces the path of the LRU line follows the exact same curve as FIFO down to practically being a ratio better. This can best be seen looking at the swim.trace where the line for LRU moves in almost exactly the same way as the FIFO line, simply being lower that it.

VMS:

VMS	Page Faults
Number of Frames	gcc.trace
2	247926
3	175663
8	142009
16	115159
32	92815
64	58055
128	39822
256	38932
512	20455
1024	9700

In our VMS test, we ran through the same tests as FIFO and LRU but only ran on gcc.trace. This was due to gcc.trace being the only trace file that was designed to work with the specifications we were asked of in the design. As you notice, the amount of page faults decreases at a rate similar to that of FIFO. However, we initially start with less page faults since we have the second-chance FIFO queue. In addition, as the number of frames increases, the proportion of drops in page faults decreases. This is most likely due to the fact there are less values being sent to the second-chance lists so page faults become more likely to occur, even though the frequency is still less since the memory allowed is larger.

Conclusion:

From our results we can conclude that Least Recently Used is the best paging algorithm for single process paging while VMS is best when there are multiple processes using the same memory. We also found that the larger your memory space gets, the less important the paging algorithm becomes, while at significantly small memory sizes, it is very important which algorithm is used. We were able to determine a lot of information about each trace, such as the fact that the bzip trace likely has less than 512 pages due to our results with LRU and FIFO. We were also able to determine that gcc.trace and sixpack.trace have similar memory reference styles except for the fact that sixpack seems to reference more unique pages during the beginning of the trace that gcc, and less unique pages than gcc near the end of it's memory trace. Also, a finding not mentioned in results is that the larger the memory space, the longer it takes for the paging algorithm to run. With scientifically large amounts of memory space algorithms such as Least Recently Used requires a lot more work to keep in order. This effect is not as significant in FIFO since the algorithm is fairly simple. This leads me to believe that in cases such as bzip where the number of page faults begins to plateau at 16 frames, it begins to become detrimental to increase the memory size as the runtime begins to outweigh the number of page faults. (INSERT DISCUSSION ABOUT VMS HERE THEN FINAL REMARKS).