Operating System Project 2 Memory Simulator – Ahmad Sibai

Introduction:

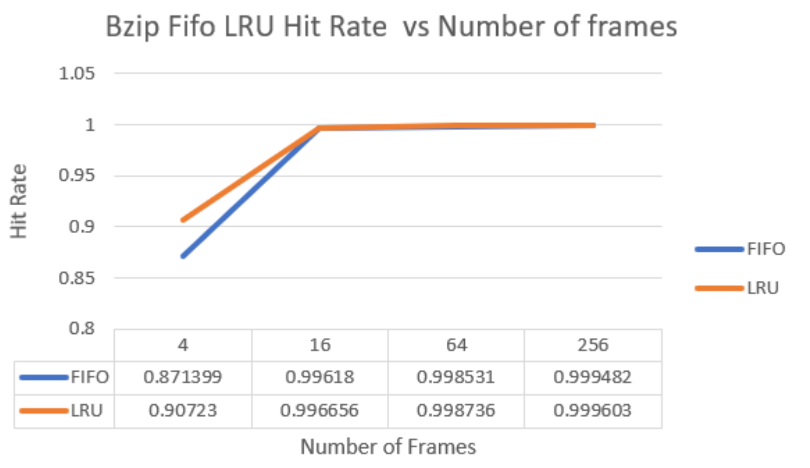
Page replacement is basically a way in which you deal with limited page table and number of frames within memory. For this specific project, we are going to be looking at the following three algorithms: fifo which stands for first in first out, lru which stands for least recently used, and vms otherwise known as segmented fifo which operates based off 2 buffers, a primary which is the fifo buffer, and the secondary which is the lru buffer. So vms is sort of in a way, a combination of both. Paging is a memory management scheme where pages are used to retrieve and store data as another part of the memory hierarchy. The simulator I have coded revolves around the C language with a unique approach to the data structures I used as C makes it much more challenging to adhere to the original data structures of each algorithm while also being as efficient as possible. Tables and graphs are provided as well to help you visualize the comparisons. The program consists of three main files, memsim.c which takes in the users' arguments and stores those characters into the proper variables, algo.h which is the header file and includes the function calls for the last files functions, algo.c which includes the 3 functions each representing an algorithm. I spent a lot of hours over many days, I would estimate in total so including the report and everything, over 24 hours working on this project.

Methods

Due to us being required to use C, I was a little limited in how I wanted to implement each of the algorithms. At first, I wanted to use a queue for fifo as that would make the most sense given that if the table was full, the oldest address was removed and replaced by the new one. But I faced difficulties and quickly realized that an STL Queue using C++ would have been much easier, so I resorted to using arrays. Lru was a similar situation where I wanted to use a linked list but then I figured I would stay consistent throughout all three and focus on the different ways of implementing each algorithm, so I resorted to using arrays for all three as my data structure. For fifo, I used an array for the page table and another array to represent the memory. And then I implemented the function in a way where if memory was full, then the first/oldest trace to have been stored would be removed and replaced by the new trace which was to be loaded. The function functioned like a queue and checked if the page was already present in the table before replacing the oldest page in memory frames. As for lru, I had four arrays, The first two represented the page table addresses and the page table read/write operations. The last two arrays represented the memory addresses and the memory read/write operations. The lru function replaces the least recently used page, which is often the oldest, but not necessarily. An example is when the oldest is called when the page table is full and then the next trace is loaded. Rather than replacing the oldest, it was recently called therefore making it the most recently used. The page replaced would be the page loaded after that trace's original load. Finally, vms also used four arrays. The first two represented the primary page table for fifo and the secondary page table for lru. The last two represented the memory frames allocated for fifo and then the same but for lru. Vms had 5 cases and 5 solutions, so I coded it to work around those while using the two buffers for fifo and lru based on the users input of the percentage. To accurately observe and analyze each algorithm's performance, I used 4 different number of frames which were 4, 16, 64, and 256 to look at how that would affect the hit rate for each algorithm. In addition to that, I built graphs when testing the same trace file so fifo and lru were compared on a graph. And vms's three different percentages which were 25, 50, and 75 were also shown and compared on a graph. This enabled the data to show how the number of reads and writes would change based on the different input and ultimately how that would affect the hit rate.

Results

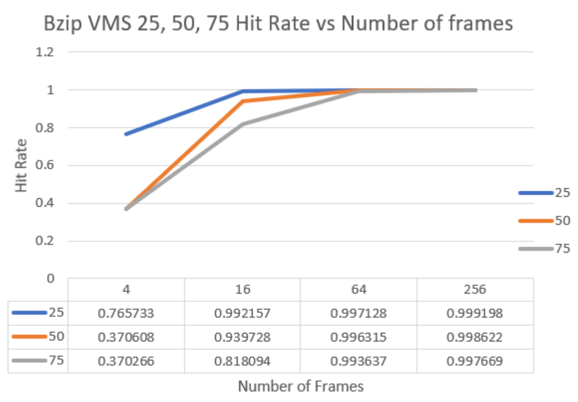
bzip	bzip	bzip	bzip
fifo	lru	lru	lru
hit rate	reads	writes	hit rate
0.871399	92770	24390	0.90723
0.99618	3344	726	0.996656
0.998531	1264	284	0.998736
0.999482	397	34	0.999603



	4	16	64	256
FIFO	0.871399	0.99618	0.998531	0.999482
LRU	0.90723	0.996656	0.998736	0.999603

To start off, this is the table and graph for fifo and the lru algorithms when they read traces from the bzip file. There is a significant hit rate difference when the number of frames per page is 4, coming in at a 3% difference but as that number increases to 16, then 64, and finally 256, we see that margin decrease and become barely noticeable at the end. Once that hit rate is .99 at 16 frames per page, we start to see a plateau from both algorithms where the change decreases heavily. What we can conclude from this is that the more frames in a page, the higher the hit rate and that ultimately makes more sense because more frames mean more room in memory for an address to have been already stored or to get stored.

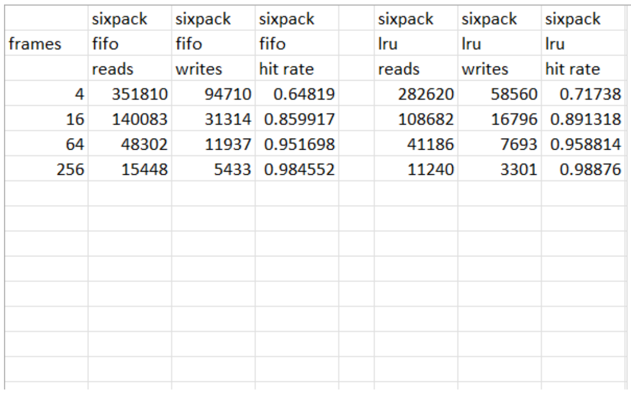
bzip	bzip	bzip	bzip	bzip	bzip	bzip	bzip	bzip
vms 25	vms 25	vms 25	vms 50	vms 50	vms 50	vms 75	vms 75	vms 75
reads	writes	hit rate	reads	writes	hit rate	reads	writes	hit rate
234267	31586	0.765733	629392	106380	0.370608	629734	106455	0.370266
7843	1127	0.992157	60272	8785	0.939728	181906	27666	0.818094
2872	440	0.997128	3685	494	0.996315	6363	878	0.993637
802	184	0.999198	1378	217	0.998622	2331	375	0.997669



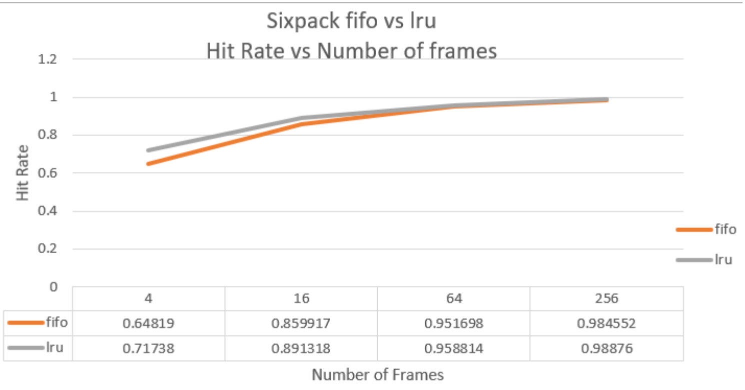
	4	16	64	256
25	0.765733	0.992157	0.997128	0.999198
50	0.370608	0.939728	0.996315	0.998622
75	0.370266	0.818094	0.993637	0.997669

With regards to this table and graph, I tested out three different vms percentages against the same number of frames as the previous graph. And as you can see once again, all three lines had the lowest hit rate out of all the different frame numbers for four frames. And 25 percent was the highest which meant that 75%

of the traces were called by fifo and the remaining 25% by lru. Again, all three lines saw an increase in hit rate as the number of frames increased leading to a connection between the two attributes.

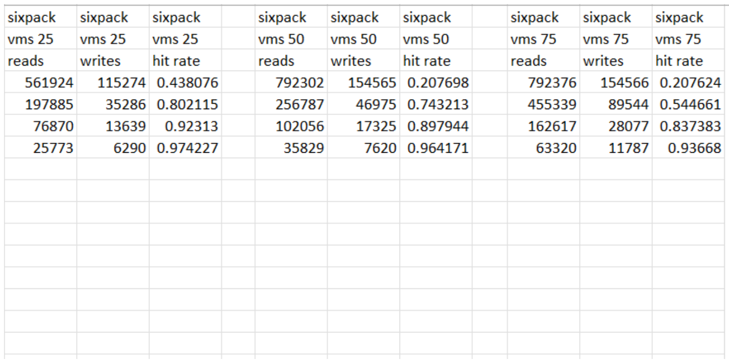


	sixpack	sixpack	sixpack	sixpack	sixpack	sixpack
frames	fifo	fifo	fifo	lru	lru	lru
	reads	writes	hit rate	reads	writes	hit rate
4	351810	94710	0.64819	282620	58560	0.71738
16	140083	31314	0.859917	108682	16796	0.891318
64	48302	11937	0.951698	41186	7693	0.958814
256	15448	5433	0.984552	11240	3301	0.98876

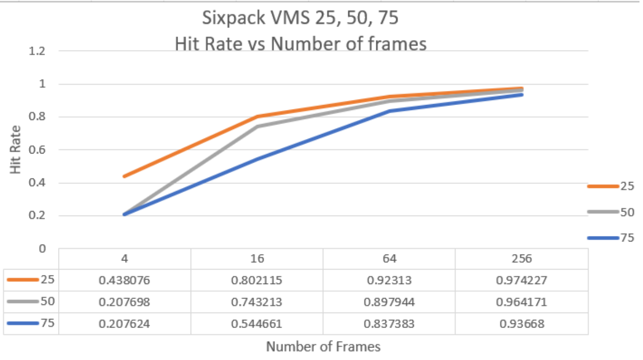


	4	16	64	256
fifo	0.64819	0.859917	0.951698	0.984552
lru	0.71738	0.891318	0.958814	0.98876

Compared to the bzip file, sixpack caused the algorithms to produce way more reads and writes and so ultimately, this is why the hit rate was generally way lower than when we ran the same tests on the bzip file. However, yet again, we see a steady increase in the hit rate as the algorithms go from working with 4 frames in a page to 256 frames in a page. This time, the graph doesn't even plateau which means that the rate of the increase isn't decreasing heavily.



sixpack	sixpack	sixpack	sixpack	sixpack	sixpack	sixpack	sixpack	sixpack
vms 25	vms 25	vms 25	vms 50	vms 50	vms 50	vms 75	vms 75	vms 75
reads	writes	hit rate	reads	writes	hit rate	reads	writes	hit rate
561924	115274	0.438076	792302	154565	0.207698	792376	154566	0.207624
197885	35286	0.802115	256787	46975	0.743213	455339	89544	0.544661
76870	13639	0.92313	102056	17325	0.897944	162617	28077	0.837383
25773	6290	0.974227	35829	7620	0.964171	63320	11787	0.93668



	4	16	64	256
25	0.438076	0.802115	0.92313	0.974227
50	0.207698	0.743213	0.897944	0.964171
75	0.207624	0.544661	0.837383	0.93668

Finally, we have the last table and graph which compares the three different vms percentages (25, 50, 75) when the file was sixpack. Clearly, the initial hit rate at 4 frames per page for all three lines is the lowest at around 20-40%. But after that, we yet again witness a steady increase as the jump of 25 from 4 frames to 16 doubles the hit rate. For 50, the hit rate goes from around .2 to .75, so more than triples and for line

75, from .2 to .54 which is more than double. There is a clear correlation between an increase in frames per page in memory and an increase in hit rate.

Comparing the two trace files, bzip's trace formation causes a way higher hit rate especially for fifo and lru when there are 4 frames per page while sixpacks produces way more reads and writes which means an initial very low hit rate. Though both manage to reach a .99 hit rate as they move toward 256 frames per page.

Conclusion:

After thorough analysis, I can confidently say that the size of available memory via the page table affects the memory performance. And that has a positive effect on their relationship because as seen throughout each graph, the hit rate increases as the number of frames per page table increases. At 16 frames when using bzip, both fifo and lru managed to get a hit rate above a .995 which when rounded is 1 1.0 hit rate, they had very few misses. Vms's 25 and 50 percent trials managed to catch up with fifo and lru at 64 frames per page, where they both managed to get over a .995 hit rate. Finally, the vms 75 line needed 256 frames per page to reach above .995 for the hit rate which means very few misses. When comparing these three, initially lru had the highest hit rate, and after that came fifo, and last, vms. So lru started with a higher hit rate than all of them and kept on seeing an improvement in the hit rate as the number of frames increased. When vms saw an increase in percentage, meaning more taken on by fifo, the results became more like lru. This proposes the fact that vms is a solution between both other algorithms as it uses both and enables flexibility when it came to who would take on more. Not only did we clearly see that increasing the frame size increases the hit rate, but also that different paging algorithms certainly work differently but also very in terms of sufficiency and effectiveness when it comes down to memory traces. An unorganized formation of a trace meaning never having the same trace in a row proves to result in a lower hit rate and when referencing the 1981 paper, vms turned out to be the algorithm to offer a solution which was similar to lru's performance whilst it not having to deal with lru's storing of each memory access.