COP4600
Wanwan Li
Project 2
Date: 10/15/2022

GROUP 22
Team Members:
Uliana Ozerova, U17965448.
Time spent working on the project: ~15-22 hours(including diagrams)
Helga Mustali, U74694583.
Time spent working on the project: ~20-23 hours
Pedro Mussi, U18147051.
Time spent working on the project: ~10-15 hours

Project 2
Memory Virtualization

Simulation of Memory Page Replacement Algorithms

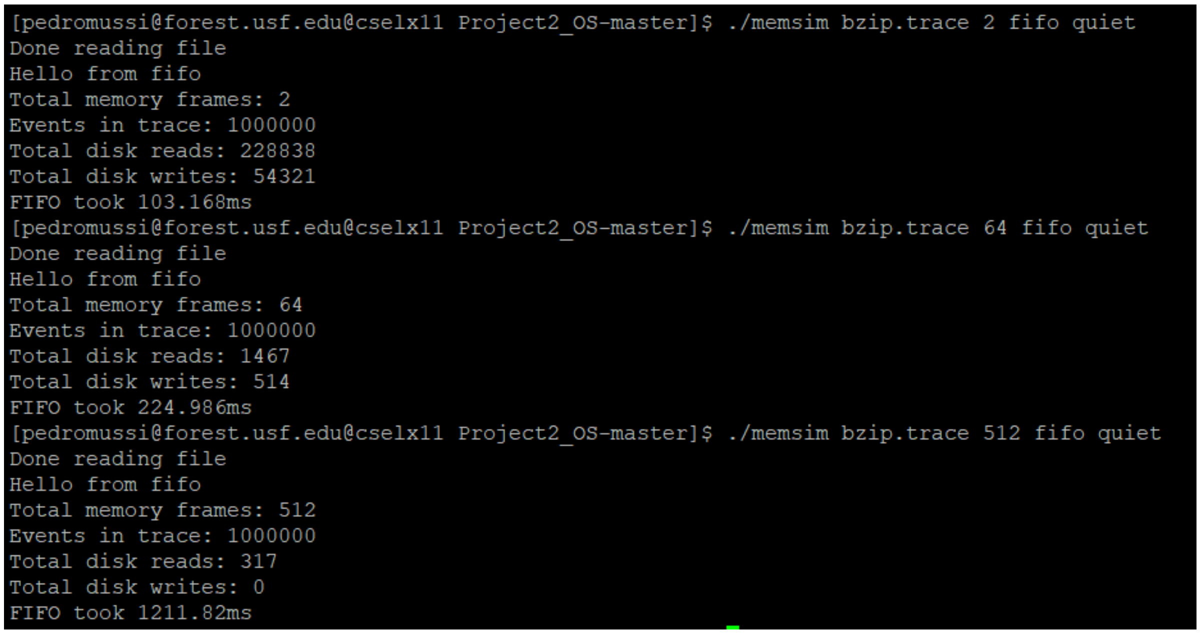
First-In First-Out, Least Recently Used, Segmented FIFO

Introduction

The OS is responsible for deciding which pages to evict from memory whenever a new page comes in, and it does this by using a replacement policy. Though there are several page replacement policies, for this project we implemented and studied the First-In-First-Out, Least Recently Used, and the Segmented First-In-First-Out replacement algorithms. Page replacement policies allow us to find a page that's in memory that is not being used and page it out, and each of these algorithms have different performances and ways on how these pages are replaced. Using memory traces from a real recording of a running program, which was taken from the SPEC benchmarks, we were tasked with building a simulator that reads these traces and performs each of the mentioned page replacement algorithms. Through the terminal, the user will be able to run a specific trace file, the number of memory frames they want to test, the type of algorithm, and the ability to see details about each trace or just the end results for each run. With these page replacement policies, the goal is to choose a policy that minimizes the number of misses and maximizes the amount of hits and by implementing these algorithms we were able to learn a lot about the performances of these algorithms.

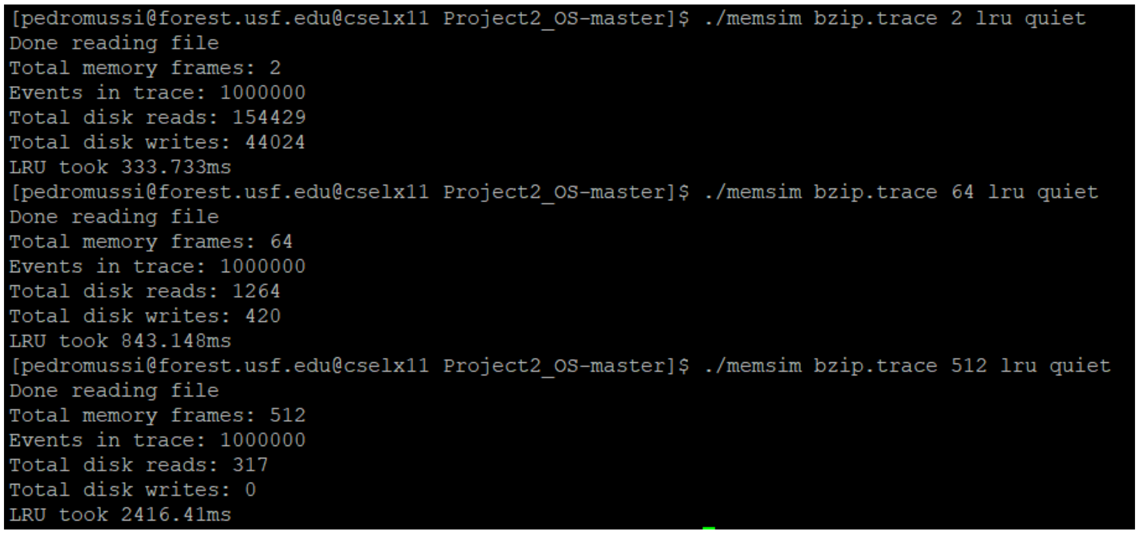
Methods

The first algorithm we started working on was First-in, First-out (FIFO). The first step is to check if the page table is not full and if the current input has not been loaded into memory. If both conditions are met we load it into the empty frame. The algorithm runs a function on the current page and creates a current state for the page table. After that it searches through the page table and the current input to check if the page table was loaded into memory. In case we get a hit, the hit counter is incremented and we continue to the next page. If we get a miss, we increment the miss counter, the algorithm deletes the top page from the queue and loads the current input into the queue. We ran simulations for frames from 2 to 4096, but to communicate how we implemented each algorithm we can simply show frames 2, 64 and 512 of our algorithm working.



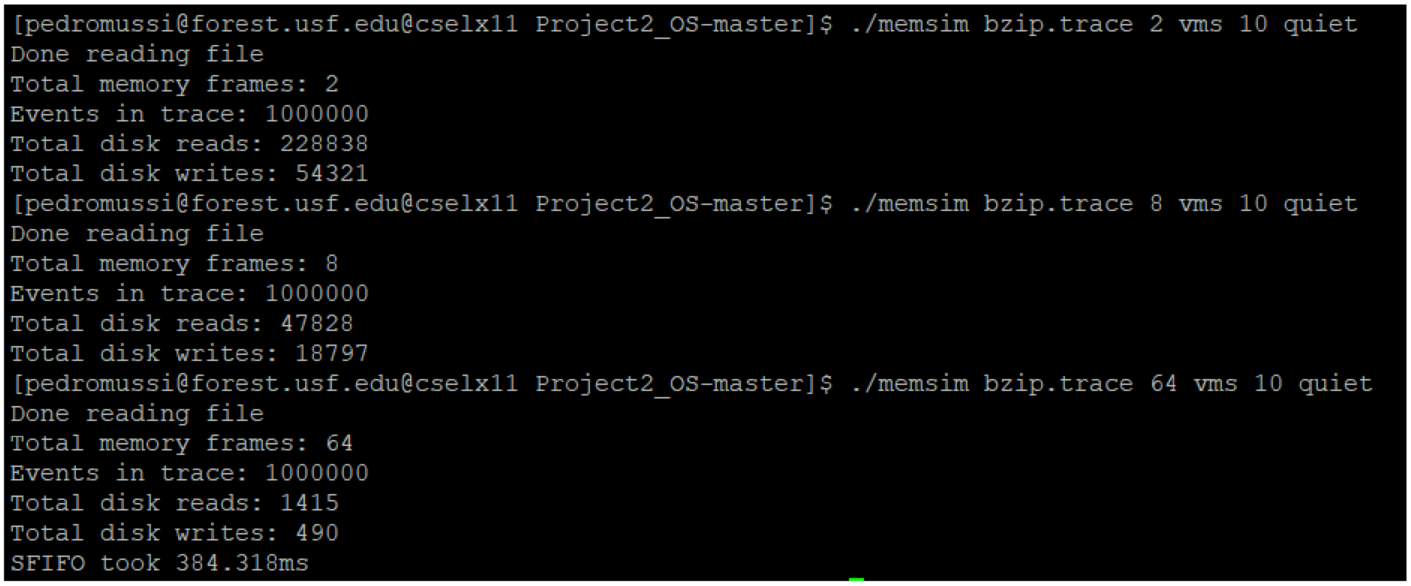
```
[pedromussi@forest.usf.edu@cselx11 Project2_OS-master]$ ./memsim bzip.trace 2 fifo quiet
Done reading file
Hello from fifo
Total memory frames: 2
Events in trace: 1000000
Total disk reads: 228838
Total disk writes: 54321
FIFO took 103.168ms
[pedromussi@forest.usf.edu@cselx11 Project2_OS-master]$ ./memsim bzip.trace 64 fifo quiet
Done reading file
Hello from fifo
Total memory frames: 64
Events in trace: 1000000
Total disk reads: 1467
Total disk writes: 514
FIFO took 224.986ms
[pedromussi@forest.usf.edu@cselx11 Project2_OS-master]$ ./memsim bzip.trace 512 fifo quiet
Done reading file
Hello from fifo
Total memory frames: 512
Events in trace: 1000000
Total disk reads: 317
Total disk writes: 0
FIFO took 1211.82ms
```

The second algorithm we started working on was Least Recently Used (LRU). The first step is to check if the page table is not full and if the page has not been loaded to memory. After checking that we add the page to the empty frame and check the current input in the page table to see if the current input has already been loaded into memory. In case we get a hit, the hit counter is incremented, but we must keep track of the page table and access the least recently used frame. In case we get a miss, the miss counter is incremented, and we will search the page table and compare the least recently used with the current input and replace the page with the current input. We ran simulations for frames from 2 to 4096, but to communicate how we implemented each algorithm we can simply show frames 2, 64 and 512 of our algorithm working.



```
[pedromussi@forest.usf.edu@cselx11 Project2_OS-master]$ ./memsim bzip.trace 2 lru quiet
Done reading file
Total memory frames: 2
Events in trace: 1000000
Total disk reads: 154429
Total disk writes: 44024
LRU took 333.733ms
[pedromussi@forest.usf.edu@cselx11 Project2_OS-master]$ ./memsim bzip.trace 64 lru quiet
Done reading file
Total memory frames: 64
Events in trace: 1000000
Total disk reads: 1264
Total disk writes: 420
LRU took 843.148ms
[pedromussi@forest.usf.edu@cselx11 Project2_OS-master]$ ./memsim bzip.trace 512 lru quiet
Done reading file
Total memory frames: 512
Events in trace: 1000000
Total disk reads: 317
Total disk writes: 0
LRU took 2416.41ms
```

The last algorithm we started working on was Segmented FIFO (VMS). Segmented FIFO is a variant of FIFO that utilizes a secondary buffer which provides performance curves that use both FIFO and LRU. The algorithm relies on the use of a portion of the program memory as a secondary buffer, and a parameter P determines the percentage of total program memory to be used in the secondary buffer. The first step is to check if the page is in the FIFO page table, and update the ‘W’ bit. In the case that the page is not in FIFO and FIFO is not full, we add a page to the back of FIFO. In the case the page is not in FIFO and FIFO is full, but LRU is not full, we then eject the front of FIFO and move into LRU, making the most recent LRU page. In case the page is in LRU and FIFo is full, we update the ‘W’ bit, move the page to the back of FIFO and move the front of FIFO to LRU, making the most recent LRU page. The last case is that the page is not in FIFO and not in LRU, and both are also full, this is where we eject the oldest LRU page, move the front of FIFO to LRU, making the most recent LRU, and add a new page to the back of FIFO. We ran simulations for frames from 2, 4, 8, 16, 64, and 256, and with LRU percentages of 95%, 40% and 10%, but to communicate how we implemented each algorithm we can simply show frames 2, 8 and 64 and LRU 10% for our algorithm working.



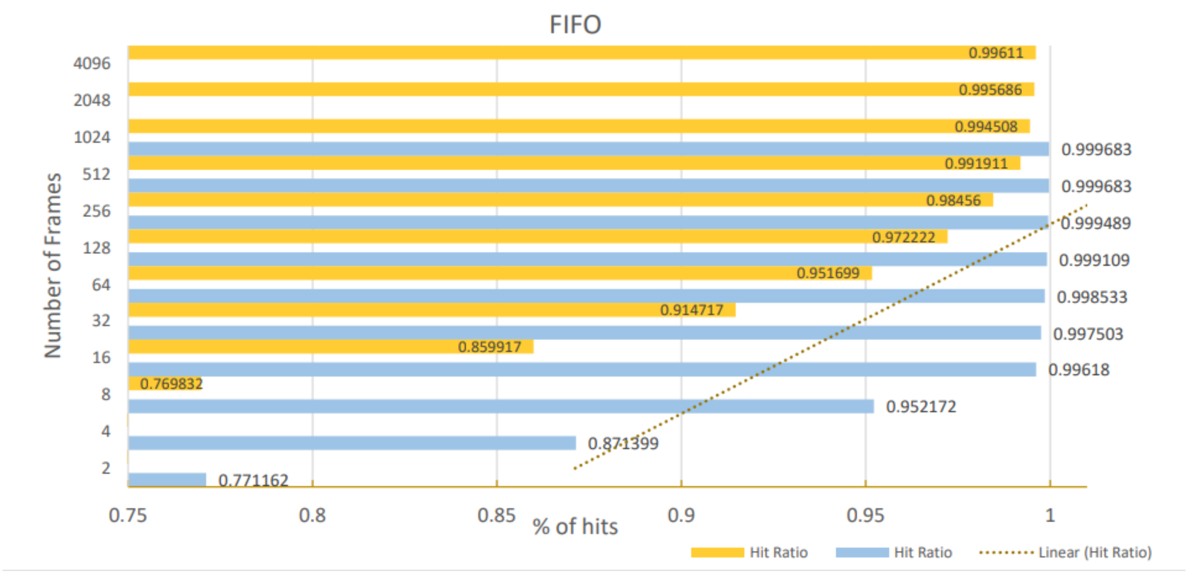
```
[pedromussi@forest.usf.edu@cselx11 Project2_OS-master]$ ./memsim bzip.trace 2 vms 10 quiet
Done reading file
Total memory frames: 2
Events in trace: 1000000
Total disk reads: 228838
Total disk writes: 54321
[pedromussi@forest.usf.edu@cselx11 Project2_OS-master]$ ./memsim bzip.trace 8 vms 10 quiet
Done reading file
Total memory frames: 8
Events in trace: 1000000
Total disk reads: 47828
Total disk writes: 18797
[pedromussi@forest.usf.edu@cselx11 Project2_OS-master]$ ./memsim bzip.trace 64 vms 10 quiet
Done reading file
Total memory frames: 64
Events in trace: 1000000
Total disk reads: 1415
Total disk writes: 490
SFIFO took 384.318ms
```

We performed the same tests using the sixpack.trace and it presented different behaviors that can be observed in our results. The result for page faults were considerably larger all throughout the experiment; the hit ratio was lower and it consistently took a larger amount of frames for the hit ratio to approach 99%. Overall when compared to bzip.trace, sixpack.trace had worse hit ratio efficiency while producing more page faults.

Results

FIFO

		BZIP				SIXPACK			
i	Num of Frames	Page Faults	Num of Writes	Hit Ratio	Runtime(ms)	Page Faults	Num of Writes	Hit Ratio	Runtime(ms)
0	2	228838	54321	0.771162	106.21	529237	136484	0.470763	142.015
1	4	128601	38644	0.871399	107.152	351810	94710	0.64819	143.002
2	8	47828	18797	0.952172	111.448	230168	57121	0.769832	147.003
3	16	3820	1335	0.99618	138.986	140083	31314	0.859917	164.438
4	32	2497	851	0.997503	221.97	85283	18805	0.914717	219.951
5	64	1467	514	0.998533	298.666	48301	11936	0.951699	330.837
6	128	891	305	0.999109	472.089	27778	8346	0.972222	604.156
7	256	511	125	0.999489	1560.12	15440	5426	0.98456	1154.28
8	512	317	0	0.999683	1757.91	8089	3353	0.991911	2200.47
9	1024	317	0	0.999683	1743.33	5492	2368	0.994508	4495.09
10	2048	-	-	-	-	4314	1561	0.995686	7489.47
11	4096	-	-	-	-	3890	0	0.99611	9222.77



The diagram presents the hit ratio for *bzip.trace* and *sixpack.trace* files.

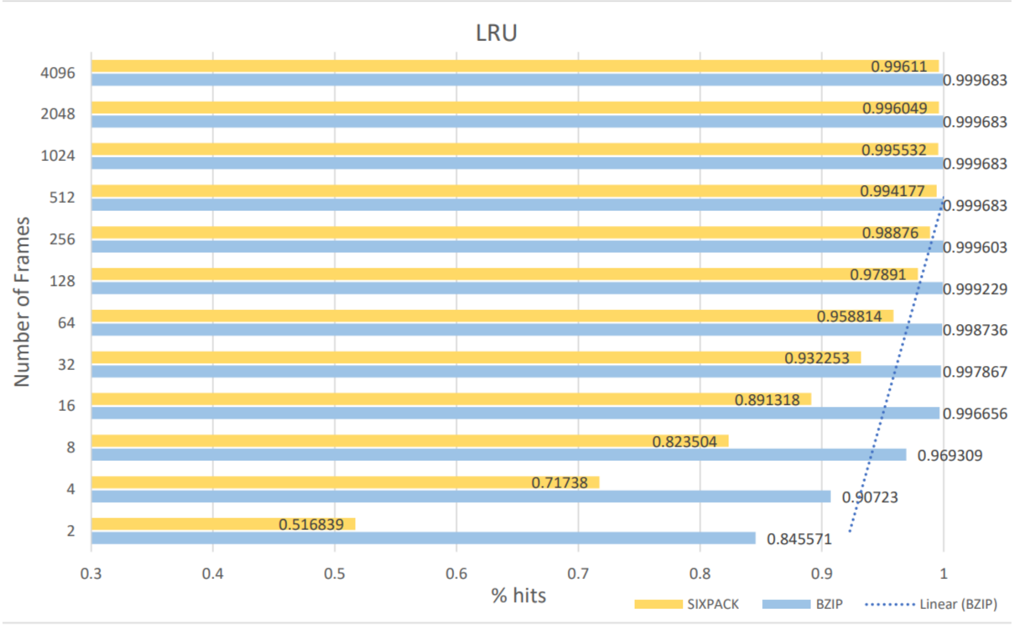
For bzip.trace file, the relationship between the number of frames, hit ratio, and runtime demonstrates that the more memory available for the processes or requests, the more successfully it will be delivered, with a 77% hit ratio for 2 available frames, and 99% for at least 16. Interestingly, every memory larger than 16 frames does not greatly affect the number of hits but suffers performance issues. When we give it 1024 frames, the hit ratio stays at 99%, but the runtime goes from 138ms to 1745ms. At 128 frames, the runtime makes a big jump and more than triples, going from ~472ms to ~1560, but stays somewhat steady afterward.

For sixpack.trace file, it is different. This time we start at less than 50% hit ratio for 2 frames. In the meanwhile, the runtime doubles, going from ~330ms with 64 frames to ~604ms with 128 frames. And then, it doubles every time we double the number of frames. Hit ratio jumps in the beginning but calms down after 32 frames.

In conclusion, considering the outcomes of the algorithm mentioned above and the goal to get the highest hit ratio, I think that 16 frames are the optimal memory size for the data in the bzip.trace file while 512 is good for sixpack.trace, although it is 32 times larger. It goes to show how much the overall memory performance depends on the type of application or data in question.

LRU

		BZIP				SIXPACK			
i	Num of Frames	Page Faults	Num of Writes	Hit Ratio	Runtime(ms)	Page Faults	Num of Writes	Hit Ratio	Runtime(ms)
0	2	154429	44024	0.845571	332.68	483161	135857	0.516839	261.559
1	4	92770	35650	0.90723	396.46	282620	71260	0.71738	357.86
2	8	30691	11092	0.969309	438.533	176496	32717	0.823504	426.22
3	16	3344	1069	0.996656	498.471	108682	19342	0.891318	502.686
4	32	2133	702	0.997867	642.044	67747	13730	0.932253	651.335
5	64	1264	420	0.998736	841.978	41186	9672	0.958814	869.98
6	128	771	224	0.999229	1264.02	21090	6526	0.97891	1288.5
7	256	397	48	0.999603	2069.58	11240	4092	0.98876	2115.43
8	512	317	0	0.999683	2413.2	5823	2444	0.994177	3743.67
9	1024	317	0	0.999683	2415.1	4468	1846	0.995532	6971.54
10	2048	317	0	0.999683	2414.38	3951	1356	0.996049	13001.6
11	4096	317	0	0.999683	2414.59	3890	0	0.99611	17018.9



The diagram represents the hit ratio for bzip.trace and sixpack.trace files when using the LRU algorithm.

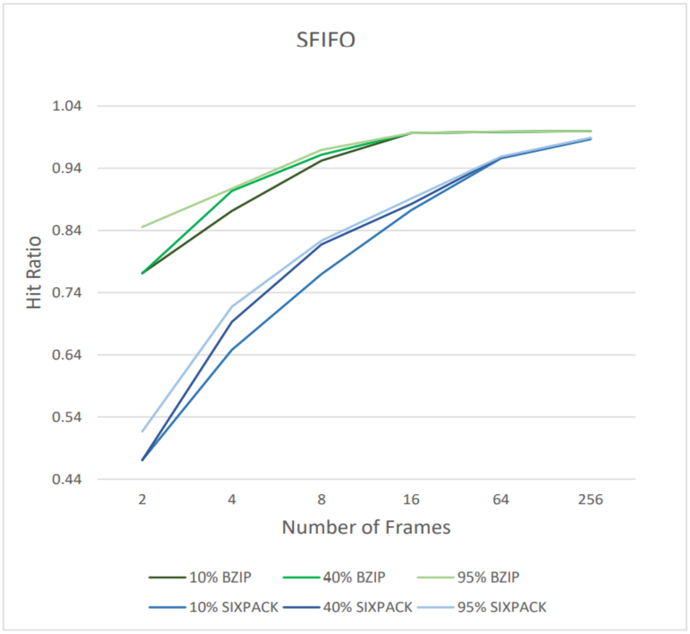
For bzip.trace, as the amount of memory available to be used in the processes gets higher, the hit ratio also gets higher. This can be shown by looking at the ratio for 2 available frames, where the hit ratio is 84%, and comparing it to the ratios past 16, where the hit ratio is around 99%. As the number of frames increases, we can see a big difference in the runtime of the program, but the number of hits remains practically unchanged. This can be analyzed by looking at 32 frames, where the hit ratio is 99% and the runtime is ~642ms, and comparing it to 512 frames, where the hit ratio is still the same at 99% but the runtime is increased all the way to ~2413ms. However, from 512 onwards the runtime is almost the same for all the remaining frames, around 2414ms or so.

For sixpack.trace the hit ratio starts lower than the bzip.trace with a hit ratio for 2 frames of 51%. Moreover, it only reaches a 99% hit ratio at 512 frames, compared to the bzip.trace which reaches around the same hit ratio at 16 frames. The runtime never becomes stable and keeps increasing all the way to 4096 frames, and we can see that by comparing 1024 frames at a ~6971ms runtime to 2048 frames at a ~13001ms runtime. All the while the hit ratio says around the same hit ratio of 99% from 512 frames onwards.

From the data we were able to gather with our experiment and weighing the hit ratio with the runtime at each frame, the best memory size for the data in the bzip.trace file would be at 16 frames because it has the best balance between good hit ratio at 99% with also a fairly low runtime at ~498ms. For the data in the sixpack.trace file, it will be a bit hard to find such balance since the runtime increases so greatly for each frame being added. The best balanced option would probably be somewhere between 256 and 512 frames at a hit ratio of ~99%, with a runtime between 2115ms and 3743ms.

SFIFO

			BZIP				SIXPACK			
i	Frames	lru p (%)	Page Faults	Num of Writes	Hit Ratio	Runtime(ms)	Page Faults	Num of Writes	Hit Ratio	Runtime(ms)
1		95%	154429	44024	0.845571	443.371	483161	135857	0.516839	447.656
2	2	40%	228838	54321	0.771162	fifo	529237	136384	0.470763	fifo
3		10%	228838	54321	0.771162	fifo	529237	136384	0.470763	fifo
4		95%	92770	35650	0.90723	440.883	282620	71260	0.71738	
5	4	40%	96509	35951	0.903491	334.362	306866	79965	0.693134	383.886
6		10%	128601	38644	0.871399	fifo	351810	94710	0.64819	fifo
7		95%	30691	11092	0.969309	474.83	176496	32717	0.823504	584.455
8	8	40%	38290	14314	0.96171	346.132	182379	35838	0.817621	404.258
9		10%	47828	18797	0.952172	fifo	230168	57121	0.769832	fifo
10		95%	3344	1069	0.996656	540.744	108682	19342	0.891318	643.718
11	16	40%	3431	1119	0.996569	415.253	117333	20005	0.882667	477.798
12		10%	3620	1198	0.99638	353.24	127230	26008	0.87277	472.55
13		95%	1263	418	0.998737	1037.69	41197	9673	0.958803	1078.34
14	64	40%	1308	441	0.998692	890.938	41881	10063	0.958119	945.748
15		10%	1415	490	0.998585	620.689	44073	10647	0.955927	715.723
16		95%	398	48	0.999602	4234.86	11245	4092	0.988755	3130.55
17	256	40%	409	49	0.999591	2777.67	11581	4234	0.988419	2658
18		10%	480	99	0.99952	2573.18	13388	4777	0.986612	2018.24



For both bzip.trace and sixpack.trace files, we can observe that with a such small number of frames as 2, it does not matter much we will utilize the secondary memory, the miss ratio will stay very high(less than 50% for sixpack, 25-25% for bzip traces).

When compared to FIFO, we can see that, if given 8 frames and more than 10% of secondary buffer, the number of page faults decreases - and goes from 47828 faults for FIFO to 30691 for SFIFO. However, yet again, it happens for the cost of the runtime.

On the diagram, we observe that the curve runs more smoothly starting at 4 frames for sixpack, but there is still a huge jump going from 2 to 4 for both files.

Also, if we look at the data for 16 frames, we`ll see that the number of page faults decreased by a factor of 10 for bzip, and only by a factor if a third for sixpack. It indicated that this algorithm is more effective for the data contained in the bzip file.

As evident from the diagram and the table, the more main and secondary memory available for the process(256 frames and 95% of secondary buffer), the better the performance of the algorithm. SFIFO has the best of two worlds: LRU and FIFO.

Conclusion

From our implementations and experiments, we found that the type of page replacement policy, the number of frames, as well as the trace files had an effect on performance. With each policy, as the number of frames increased, so did the hit ratio, but so did the runtime as well. For 2 frames used, the hit ratio for FIFO was 7.4% lower than LRU using the bzip trace, and it was 4.6% lower than LRU using the sixpack trace. For 256 frames used, the hit ratio for FIFO was 0.01% lower than LRU using the bzip trace, and it was 0.4% lower than LRU using the sixpack trace. So the hit ratio for LRU was higher than FIFO. The runtime for 2 frames used for FIFO was 226.5ms faster than LRU for the bzip trace and 119.5ms faster than LRU for the sixpack.trace and the runtime for 256 frames used for FIFO was 311.67ms faster than LRU for the bzip trace and 961.15ms faster than LRU using the sixpack trace.

Based on these results, the FIFO algorithm was faster than LRU, however, when testing both based on the same number of frames used, FIFO had more page faults and lower hit rate than LRU. Though LRU was a bit slower, it was more efficient than FIFO since it’s removing the least recently used page and keeping the most frequently used ones which causes the page fault to be lower since the page is in the page table already. FIFO has more page faults since it can remove pages that are frequently accessed and it will keep adding them and removing them since it just removes pages that were paged in first, hence first-in-first-out, so it’s not very efficient in the way it decides to replace the pages.

Additionally, even if two traces have the same amount of entries, the number of page faults as well as the runtime can vary, even though each algorithm showed that their hit rate was increasing. With each algorithm there was a difference between the results of the bzip trace and the sixpack trace, sixpack had a higher number of page faults and this also caused the runtime to be higher as well even though they both had 1 million entries. So the type of page access matters as well.

When the percentage for SFIFO was close to 100% then LRU was implemented but when the percentage was close to 0% then FIFO was implemented, so the page faults and hit rate for 95% were the exact as or very close to FIFO and for 10% results were the exact as or very close to LRU. However the runtimes varied, the segmented FIFO was slower than both policies. Our observations confirm the results of the paper because we also noticed that the hit rate can fall at any point between FIFO and LRU. Additionally, we also noticed as the pages used increased, the number of page faults decreased.