COP 4600 Operating Systems – Project 2 Report

Group 17

Siwon Kim - Leader - CS - 60% of work (FIFO implementation, Methods, Results)

Jacob Kearschner - CS - 40 % of work (LRU implementation, Introduction, Conclusions)

1. Introduction

The main motive of this project is to study the different methods of page replacement algorithms. We will examine and implement two algorithms for handling page replacement: First In First Out and Least Recently Used.

In all the memory simulators, the first step is to go through the page table and check whether the memory trace that has been given is already in the table. On a hit, the simulator will update the page's dirty bit if and only if the new page is marked with a 1 for the bit value. On a miss, each algorithm will attempt to insert the new page into its respective data structure. If the data structure is full (a page fault), the algorithm will then make a decision on what existing page should be removed. The core difference between each algorithm is how the algo handles removing a page from its data structure. Standard FIFO removes the oldest page by time of initial insertion while LRU considers what page has had the most time since being added or recalled.

2. Methods

Here's the output of FIFO and LRU experiments depending on the number of frames applied for each bzip.trace and sixpack.trace.

bzip.trace	FIFO				LRU			
nframes	total read times	total write times	total hit times	total fault times	total read times	total write times	total hit times	total fault times
2^0	629737	118475	370263	629737	629737	118475	370263	629737
2^1	228838	54321	771162	228838	154429	44024	845571	154429
2^2	128601	38644	871399	128601	92770	35650	917230	92770
2^3	47828	18797	952172	47828	30691	11092	969309	30691
2^4	3820	1335	996180	3820	3344	1069	996656	3344
2^5	2497	851	997503	2497	2133	702	997867	2133
2^6	1467	514	998533	1467	1264	420	998736	1264
2^7	891	305	999109	891	771	224	999229	771
2^8	511	125	999489	511	397	48	999603	397
2^9	317	0	999683	317	512	0	999683	512
2^10	317	0	999683	317	1024	0	999683	1024

sixpack.trace	FIFO				LRU			
nframes	total read times	total write times	total hit times	total fault times	total read times	total write times	total hit times	total fault times
2^0	792379	159542	207621	792379	792379	159542	207621	792379
2^1	529237	136484	470763	529237	483161	135857	516839	483161
2^2	351810	94710	648190	351810	282620	71260	717380	282620
2^3	230168	57121	769832	230168	176496	32717	823504	176496
2^4	140083	31314	859917	140083	108682	19342	891318	108682
2^5	85283	18805	914717	85283	67747	13730	932253	67747
2^6	48301	11936	951699	48301	41186	9672	958814	41186
2^7	27778	8346	972222	27778	21090	6526	978910	21090
2^8	15440	5426	984560	15440	11240	4092	988760	11240
2^9	8089	3353	991911	8089	5823	2444	994177	5823
2^10	5492	2368	994508	5492	4468	1846	995532	4468

a. When the process doesn't have enough physical memory

In the orange-colored section case for bzip.trace table, the number of frames which means the capacity of the physical cache memory is extremely small compared to the number of events in trace (for total events is 1,000,000). This causes many read, write, hit, and fault times in both FIFO and LRU. Especially, for the total page fault time, a lot of pages are missed (not hit) and this page fault is the applications have sustained issues with trying to make data available in memory. This can involve disk I/O and impact performance.

b. When the process' working set fits in the memory

The working set size of bzip.trace case is how much memory an application needs to keep working. In this case, when the number of frames reaches 2^9, it can be when the working set fits in the memory because after reaching 2^9, it shows consistent hit and writes times.

The working set size of sixpace.trace is explained below.

c. When increasing the allocated memory does not improve performance significantly

In the blue-colored section case for bzip.trace table, a relatively large number of frames has been applied and this large capacity of physical memory can be expected to have higher performance since the page fault times is smaller. However, as it is shown in the table when the number of frames exceeds a certain point, it does not change any of the total times in FIFO and doesn't change the write and hit times in LRU. This means it is not true that the more capacity the physical memory has, the more improved performance is expected.

d. How much memory does each traced program actually need?

For bzip.trace, it requires 2^0 ~ 2^9 size for both FIFO and LRU cache as seen in the table above.

For sixpack.trace, it requires 2^0 ~ 2^12 size for both FIFO and LRU cache

e. What is the working set of each program?

For bzip.trace, the working set is when the number of frames is 2^9

For sizpack.trace, the working set is when the number of frames is 2^12

f. Which algorithm works best?

Only between FIFO and LRU, LRU is better than FIFO. FIFO can cause Belady's Anomaly which means adding more frames can cause more page faults. However, LRU does not suffer from this theoretically.

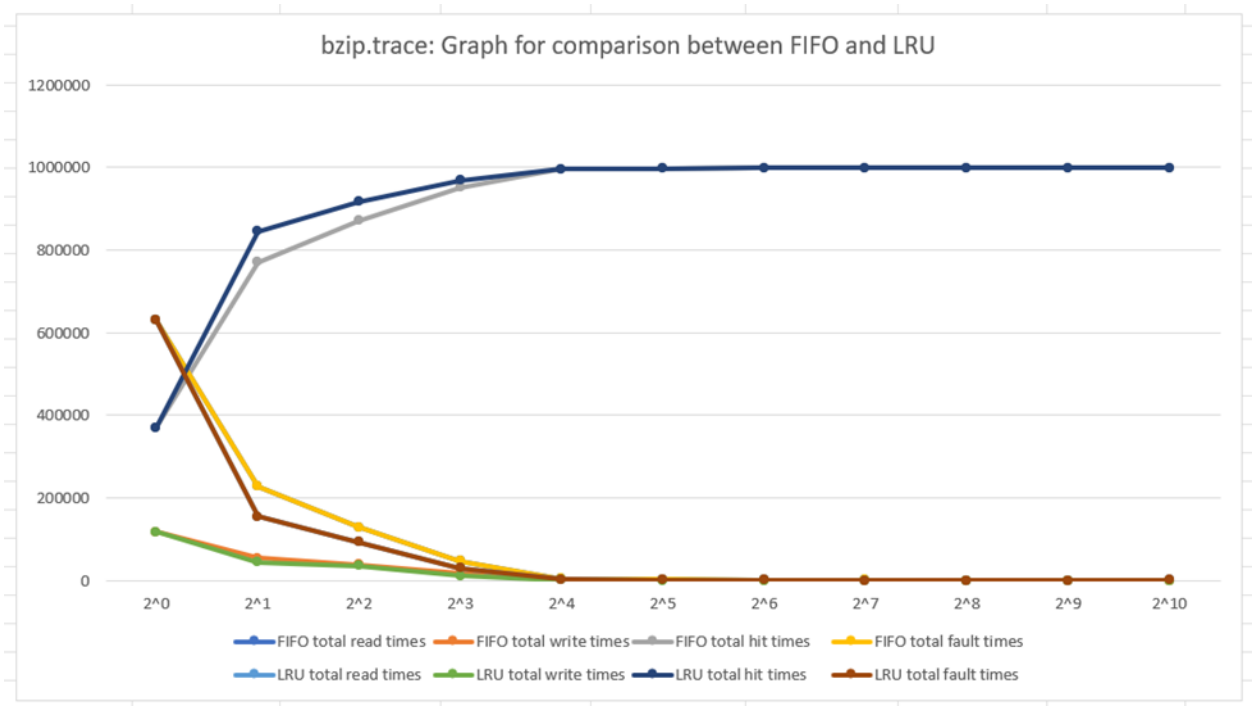
From the experimental results, results from the graph below show that the total page hit times of LRU increase rapidly and the total page fault times of LRU decrease rapidly compared to the case of FIFO. This means LRU has better performance in the page replacement process.

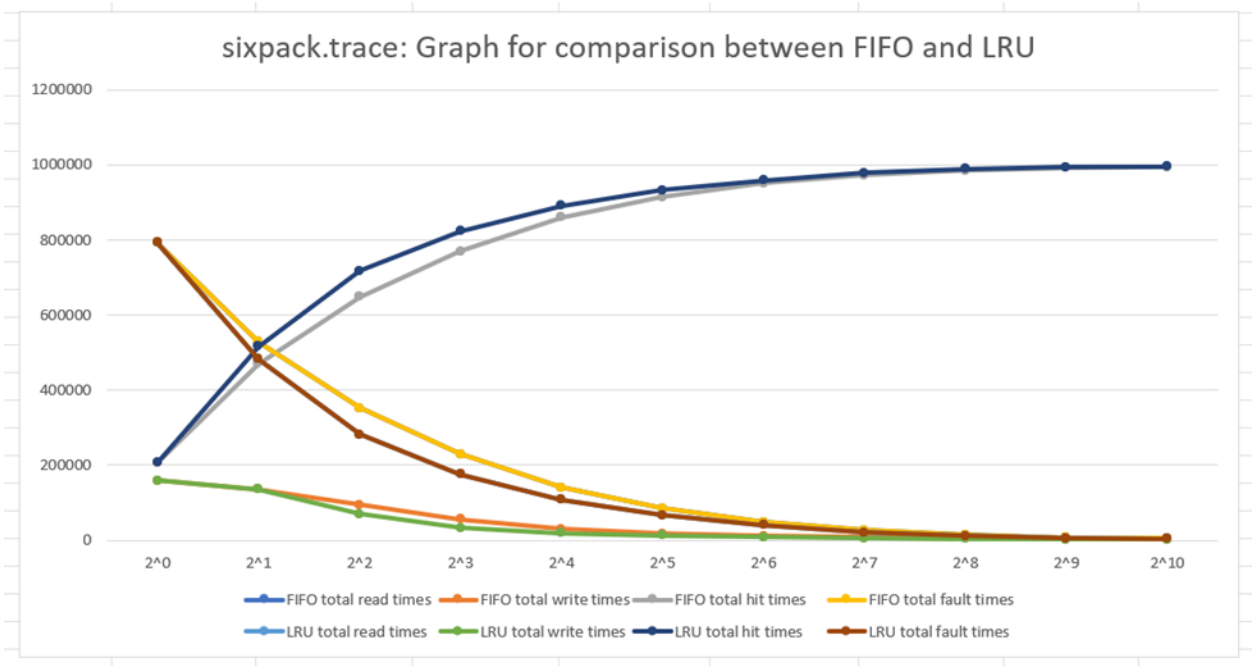
From the running time perspective, when running FIFO with a large input of nframes, the time is very long to get the output compared to LRU.

g. Does one algorithm work in all situations?

LRU outperforms FIFO in the experiment tracing all two trace files and real-world usage. However, there is no single algorithm that is the best in page-replacement

3. Results





From the experimental results, the results from both graphs above show that the total page hit times of LRU increase rapidly and the total page fault times of LRU decrease rapidly compared to the case of FIFO. This means LRU has better performance in the page replacement process. The results are similar to the expected output; that is, LRU outperformed FIFO.

4. Conclusions

In addition to following the expected pattern of LRU outperforming FIFO, since the LRU cache data structure abstracts most of the complexities of the algorithm, implementation was a relatively simple task, even compared to FIFO. Given the amount of benefits at a debatable negative cost to implement, LRU is definitely the better algorithm choice among the two.

Regardless of algorithm choice, the ultimate decider in performance falls upon the allocation of frame space. When running bzip.trace, results for both FIFO and LRU converged at with an nframes of 2^4, while sixpack.trace converged at 2^8. If it can be made available, increasing frame space will have a much larger effect on performance than algorithm choice.