Project 2: Page Table Simulation

William Horn & Jessic Tran
University of South Florida
Operating Systems

Introduction

For this project, we were tasked to implement a page table simulator which compares different page replacement policies on traces of actual programs. Page replacement is an important problem because the time penalty for having to fetch information from disk is magnitudes larger than when a process is able to retrieve the information from memory. Choosing a page replacement policy which is able to reduce the number of page faults while also not being costly to implement or requiring an exorbitant amount of pages is obviously the best solution to this problem. Simulating these page replacement policies thus provides us with a good estimation of the real world performance of these policies, and better allows operating system developers to decide which policy to enforce.

For our implementation, we implement two major data structures to simulate the four replacement policies: a PageTable struct which contains a dynamically allocated array of PageTableEntry structs, as well as a DLinkedList struct used to know what order PageTableEntries were placed into the table for VMS, and LRU (for FIFO, we choose a simpler approach which we discuss below). Below is a high level explanation of how we implemented our four replacement policies:

- RDM

  To simulate random replacement is fairly trivial. We add entries at the next available position in the page table, incrementing the number of reads until the page table is full, at which point we swap to random replacement. At that point, we simply select a random index within the page table and evict the page, incrementing the number of reads, as well as the number of writes if the entry was modified prior to it being evicted from the page table. Through this entire process process we modify dirty bits as necessary, i.e. when a page is immediately written to when pulled into the page table, or when modified on a page hit. In the description of our other page replacement policies it may be assumed we properly handle counting reads / writes, and dirty bits modification throughout their implementation for the sake of brevity.

- LRU

  Simulating least recently used does require the use of the prior mentioned DLinkedList in order to keep track of how recent page table entries are accessed / pulled into the page table. Similarly to random eviction, we simply add entries at the next available position, but when a page is added to the page table, we add that page to the front of the DLinkedList. Thus, pages at the front of the linked list were accessed most recently, while those at the end were accessed the least recently and are the candidates for eviction. Once the page table is full, we simply reference the DLinkedList for the page at the end, evict it, and update the recency of that page entry by moving it to the front. If a page hit occurs, we simply update the recency without evicting the page.

- FIFO

  For our simulation of first-in-first-out replacement, we took a simplified approach to emulate how FIFO would occur without having to explicitly keep track of the order pages were added to the page table via the DLinkedList method. Essentially, we add pages to the page table from indices 0 to numFrames-1 to initially fill it. Because we add them in this order, when the page table is full, we begin using an index variable initialized to index 0, and every time we remove a page we simply increment this index variable, setting it to zero once we've exceeded the bounds of the page table. While this definitely is not exactly how it would be simulated in an actual page table, it does maintain the FIFO property, while also having just as fast of a time complexity if we were to implement a queue.

- VMS

  Our VMS implementation was definitely the most difficult to implement due to having the multiple data structures and conditions we had to account for. We will dissect our logic for our VMS implementation into three scenarios for this report: page hits, page faults when the page table has empty entries, and page faults when the page table is full.

  - Hits

    On a hit, we first check if the page is in the clean list or dirty list. If it is, we wish to reclaim the page so we remove it from the respective list. We then do a check to see if the page is in the respective process' FIFO list. If it is, we simply check if the page is being written to and set its dirty bit accordingly. If it is not in the FIFO

list, then it must've been in either the clean or dirty list. In this scenario, we add the page back to the respective process' list.

- Fault with empty entries

  If the page is not found in the page table, and there are empty entries, we add the page to the next available entry. We then do a check for the respective process' FIFO list to see if adding the page to the page table will exceed its RSS. If so, we remove the first-in page and add it to either the clean or dirty list depending on its dirty bit. After adding every page, we check to see if the page table is full. Once it is, we swap to evicting pages.

- Fault with full page table

  If the page table is full, we check to see if there are pages within the clean list. If so, we remove the first-in. Otherwise, we check the dirty list, and similarly remove a page if it has any listed. If both are empty, then we evict the first-in page from the respective process' FIFO list. We then place the new page within the page table at the evicted page's position.

Methods

For our experiment we decided to look into how the performance of different page replacement policies compare as they are provided different sized page tables. We wanted to see if one particular page replacement policy improved more substantially than others based on the number of page table entries, so that an obvious choice for a replacement policy could be made for page tables of certain size. We were also curious if we could deduce any information about the traces based on the information provided by the simulations (such as whether a process is large taking up a large portion of the VAS). We decided to run our test with the number of page table entries increasing by powers of 2 from 4 entries to 1024 entries. Note that this corresponds to 16 KB of memory allocated for each process up to 4 MB per process, respectively. We believe these are ranges which are reasonable for modern computers (with the assumption that modern computers have anywhere from 4 GB - 64 GB of RAM).

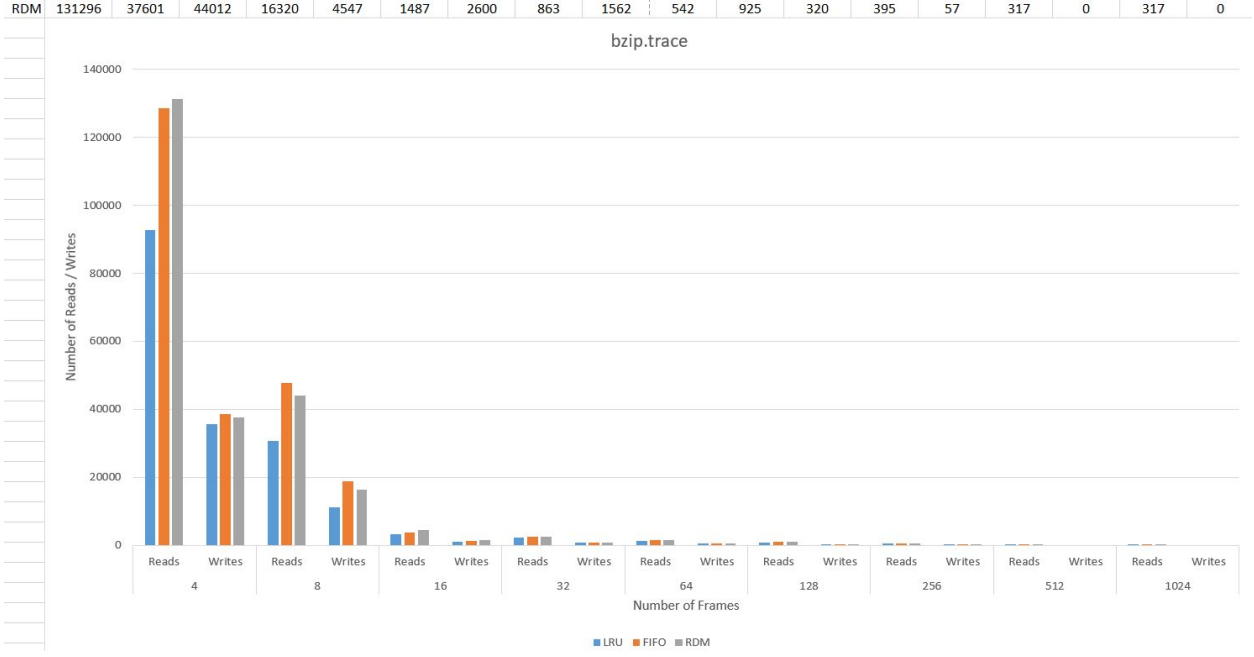
Results

Our results below show that for every trace, the different replacement policies all perform within margin of error of each other, so there is no definitive choice when given the option to implement page tables of larger and larger sizes. It also shows that definitively for these

particular traces that LRU is the highest performing replacement policy, beating all other replacement policies in terms of page faults (i.e. the number of disk reads). Universally, it was shown that page tables of size 4 and 8 are not large enough to be used in actual systems, with hit rates of sub 80% for gcc and sixpack, and sub 70% hit rates for swim. For the bzip trace, the page table began to perform incredibly well with only 16 page table entries; any more only slightly improved performance. We believe this shows that bzip is a relatively small process, taking up only a small fraction of the VAS. However, for all other traces, increasing the number of page table entries improved performance by about 30% for every doubling of page table entries. Thus, we believe that page tables of sizes 256 - 1024 are reasonable for most processes with hit rates of above 95%. These values can be decided depending on the amount of memory provided to the operating system / the amount of memory available to the hardware. Below are bar graphs showing the number of reads / writes for each respective process:

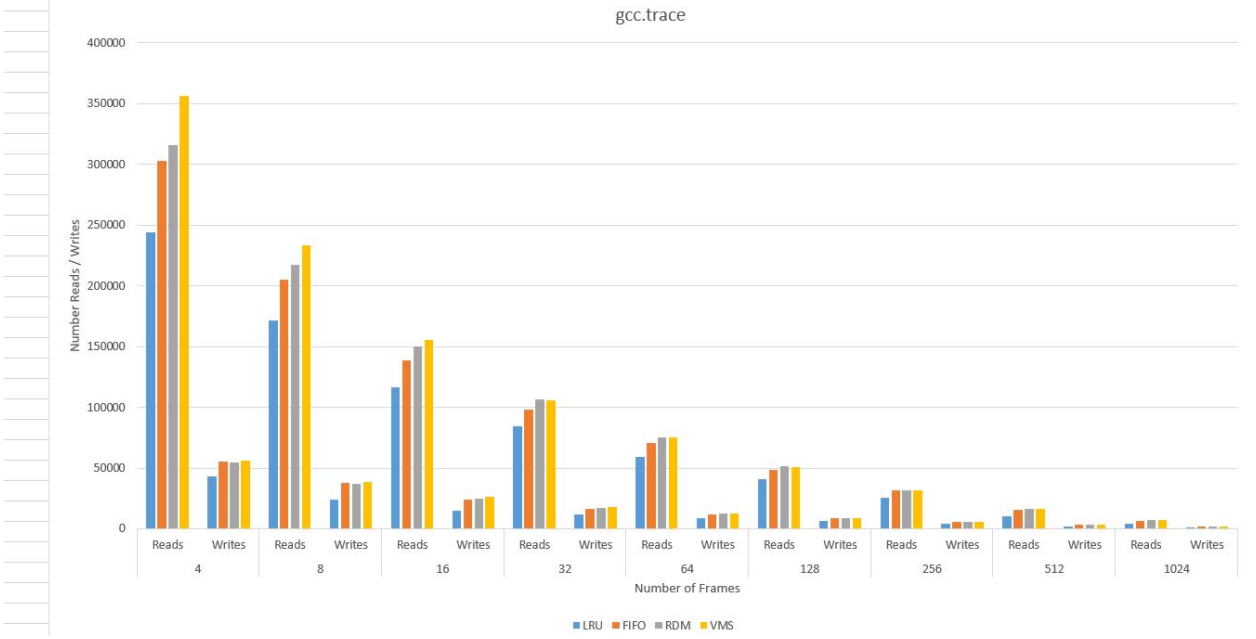
- bzip.trace

NumFrames	4		8		16		32		64		128		256		512		1024	
	Reads	Writes	Reads	Writes	Reads	Writes	Reads	Writes	Reads	Writes	Reads	Writes	Reads	Writes	Reads	Writes	Reads	Writes
LRU	92770	35650	30691	11092	3344	1069	2133	702	1264	420	771	224	397	48	317	0	317	0
FIFO	128601	38644	47828	18797	3820	1335	2497	851	1467	514	891	305	511	125	317	0	317	0
RDM	131296	37601	44012	16320	4547	1487	2600	863	1562	542	925	320	395	57	317	0	317	0



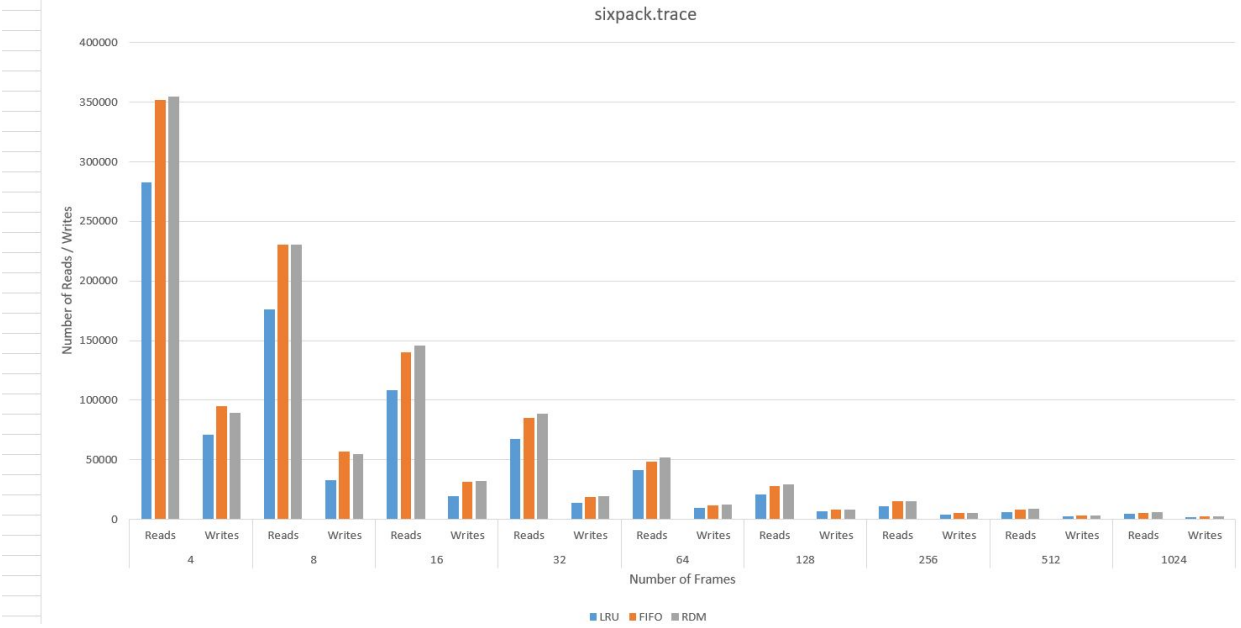
- gcc.trace

	NumFrames																	
	4		8		16		32		64		128		256		512		1024	
	Reads	Writes	Reads	Writes	Reads	Writes	Reads	Writes	Reads	Writes	Reads	Writes	Reads	Writes	Reads	Writes	Reads	Writes
LRU	243809	43187	171186	23983	116604	14749	84401	11737	59089	8863	40821	6131	25308	4231	10425	2173	4391	955
FIFO	302860	55013	205368	37524	138539	24043	98067	16759	70315	12053	48526	8487	31698	5945	15609	3425	6673	1747
RDM	315981	54680	217511	37274	150375	24737	106572	17267	74933	12239	51254	8554	31983	5696	16323	3385	7151	1704
VMS	356224	55899	233401	38638	155145	26328	106063	17961	75015	12720	50915	8879	31392	6025	16109	3656	7446	1932



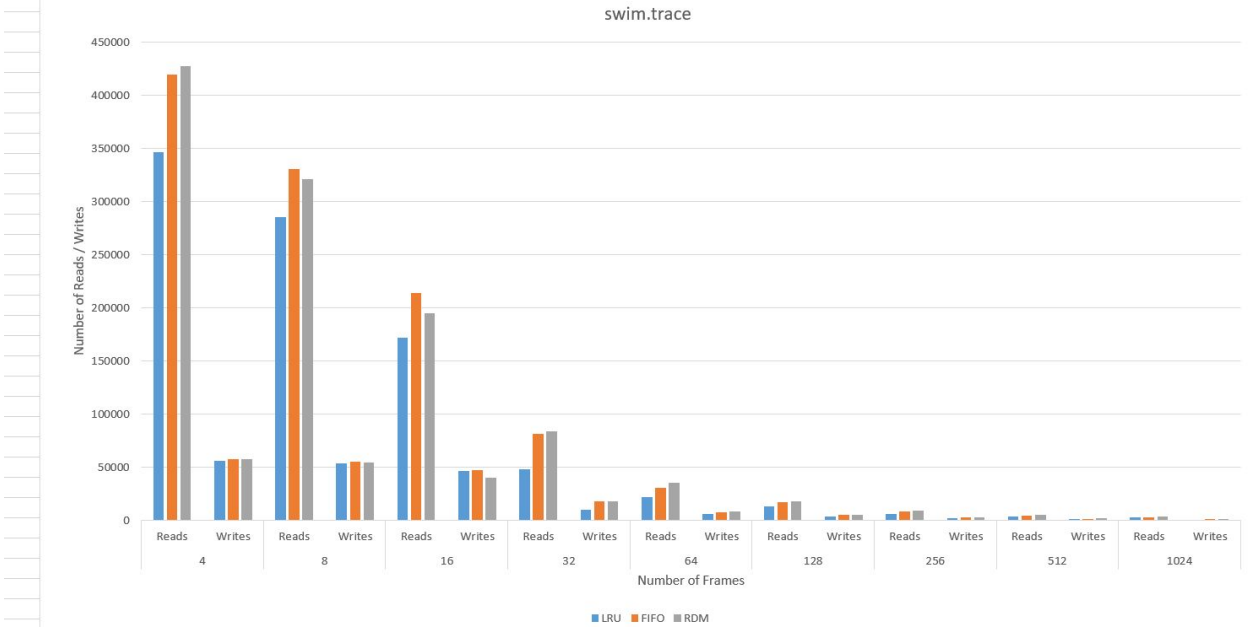
- sixpack.trace

	NumFrames																	
	4		8		16		32		64		128		256		512		1024	
	Reads	Writes	Reads	Writes	Reads	Writes	Reads	Writes	Reads	Writes	Reads	Writes	Reads	Writes	Reads	Writes	Reads	Writes
LRU	282620	71260	176496	32717	108682	19342	67747	13730	41168	9672	21090	6526	11240	4092	5823	2444	4468	1846
FIFO	351810	94710	230168	57121	140083	31314	85283	18805	48301	11936	27778	8346	15440	5426	8089	3353	5492	2368
RDM	355019	89563	230424	54481	145477	31975	88937	19480	51895	12763	29231	8474	15564	5361	8790	3463	5945	2446



- swim.trace

	NumFrames																	
	4		8		16		32		64		128		256		512		1024	
	Reads	Writes	Reads	Writes	Reads	Writes	Reads	Writes	Reads	Writes	Reads	Writes	Reads	Writes	Reads	Writes	Reads	Writes
LRU	346936	56045	285375	53614	171961	46293	48254	9674	21656	5687	13250	3812	5673	1866	3632	1159	2803	789
FIFO	419509	57750	330893	55488	214295	47434	81638	18172	30422	7585	17523	4932	8551	2836	4850	1624	3264	969
RDM	427569	57783	321152	54341	195240	40237	83569	18069	35383	8500	18102	4970	9435	2994	5180	1727	3487	1039



Conclusion

From the results we learned that processes can vary dramatically in the number of page table entries required to suit their needs, particularly when comparing the gcc and bzip traces. They also vary drastically in how memory efficient they are, with some traces having to write to disk far less frequently. Actually implementing the paging algorithms definitely solidified our understanding of paging, through having to properly extract the page numbers, considering paging scenarios with parallel processes, the different methods of page eviction, among other small details. One large misconception I had prior to this project was that page entries could directly be written to without having to be taken into the page table, which is retrospect is obviously incorrect because address translation cannot occur without the page table. We also confirmed the performance of the algorithms as we have discussed in lecture: FIFO sounds great on paper but does not perform well on paper. Similarly, random replacement sounds like it would be a poor replacement policy, but it shows to perform just as well as FIFO. LRU was told to perform well due to time locality, which we definitely confirmed from our findings.