An Analysis of Page Replacement Algorithms in Modern Operating Systems

Steven Faulkner
COP 4600
Project2
sfaulkner1@mail.usf.edu

Abstract-To compare the overhead between three different page replacement algorithms written in ANSI C. The Algorithms tests are traditional First In First Out (FIFO), Least Recently Used (LRU) and, segmented FIFO, a variation of a second chance algorithm used in the VMS Operating System. We introduce our methodology and design principles, then discuss the triumphs and shortcomings of each algorithm, and conclude our findings with any anomalies, limitations or challenges that were presented during our findings.

I. Introduction

The goal of this project was to gain a deeper understanding of common page replacement algorithms used in modern operating systems. To achieve this goal, a paging simulator was created, many algorithms exists some with multiple variations, each of these algorithms trade off space efficiency for time efficiency or vice versa, thus each algorithm could potentially be suited for a unique purpose. With this in mind, testing three popular algorithms across a wide range of feasible page sizes could potentially reveal the strengths and weaknesses of each algorithm.

II. Methodology

Our design for this problem revolved around a linked list data structure, that would be utilized as a queue, this would be beneficial and cost efficient in our algorithms and insertion at one end and removal at the other end can be performed in constant time. Something we tried to be conscious of since our test range would involves creating up to 32,000 frames which would drastically reduce performance on even the most optimized algorithms.

To test our implementation, a varying range of table sizes were used, ranging from 2^2 to 2^{15} pages. We believed that this range of table sizes could sufficiently cover most everyday scenarios an average user would experience during use.

III. Results

Testing was done using the 'gcc.trace' file provided, which contains 1,000,000 traces from a real-world example. Since this trace file was needed for VMS to run properly we deemed it appropriate to test for the other algorithms as this would allow a more deterministic and accurate result versus testing against swim, where page faults were lower with both FIFO and LRU algorithm.

Results in respect to cache miss percentage were as respected, FIFO performed marginally better than LRU when the number of frames were relatively small, however as the number of frames grew, the LRU drastically better than FIFO, which can be seen in Figure 1 and Figure 2. As to be expected, our variation of the Second Chance Algorithm (VMS) outpaced the other algorithms by a large margin, with cache misses sometimes being half of the other algorithms for a given frame number. Our specific variation of VMS also improved the time spent 'writing' back to disk since clean pages would be evicted before dirty pages. Yet, given the astronomical run time of our implementation, we question if such a trade off is worth it.

Running time for each algorithm was directly proportional to how complex you might perceive the algorithm to be, since FIFO is easy to implement and required less instructions it was out fastest running algorithm, with LRU being close behind, lastly VMS took magnitudes longer to complete. We believe the run time performs as poorly as it does due to a combination of two factors, the sheer number of branches involved with checking where a frame might lie in memory, and the linear time of checking a linked list, using a hash map in conjuncture to a linked list would, in our opinion, give the best results since checking if the page is already in memory would be done in ***O(1)*** time versus ***O(n)*** for iterating over a linked list, this combined with our ***O(1)*** time for head insertion and tail deletion would drastically improve overall performance.

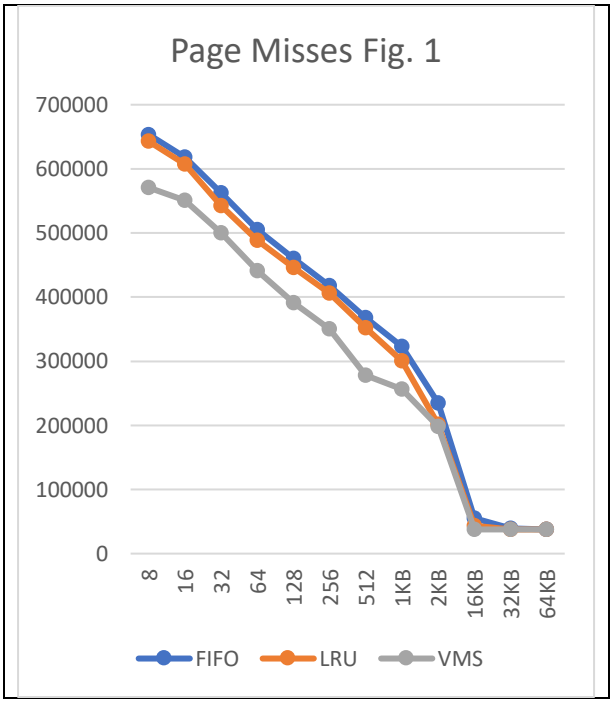

Fig. 2
Number of Writebacks per Algorithm

nFrames	FIFO	LRU	VMS
16	68542	65636	28530
64	51098	48737	23500
256	39232	36883	22129
1KB	29137	25999	19241
16KB	8167	7644	231
32KB	2840	2839	13

As. seen in the chart above, the amount of page misses decreases along a fairly linear path until 16KB of page frames are used, this is largely due to the page table being large enough to hold all the frames in memory without having to read from the disk, and the page misses seen at these sizes are due to the initial page faults. Figure 2 details the number of write operations performed on dirty pages by each algorithm, the unique handling of VMS shines here since the algorithm chooses to evict unwritten references over written ones.

IV. Conclusions

As stated in the methods, use of any specific algorithm is a holistic choice that should consider architecture of hardware, amount of memory available, etc. If speed is of upmost priority with a relatively small amount of memory the LRU algorithm is the best choice, since it performs as quickly as FIFO in most cases will reducing the number of page faults that occur, which would reduce the overall time in any real-world scenario. On the other hand, if memory space is freely available a highly optimized VMS or other second chance algorithm would suit best.

Testing all three algorithms against one data set allowed us to have a normalized test set to compare each algorithms results against, but limited our testing in total scope, testing FIFO and LRU against the other trace files widened our scope of how the algorithms handled higher/lower page faults and write backs, which would vertically shift or numerical results up or down, but the overall trend remained the same.

We also noted earlier that performance deteriorated rapidly as the page table grew, and that could be solved by a secondary data structure, a hash map, that would grant constant lookup times versus linear lookup times in large scale page tables.