Project 2 Report: Memory Visualization

By

Henri Berisha and Darshan Vala

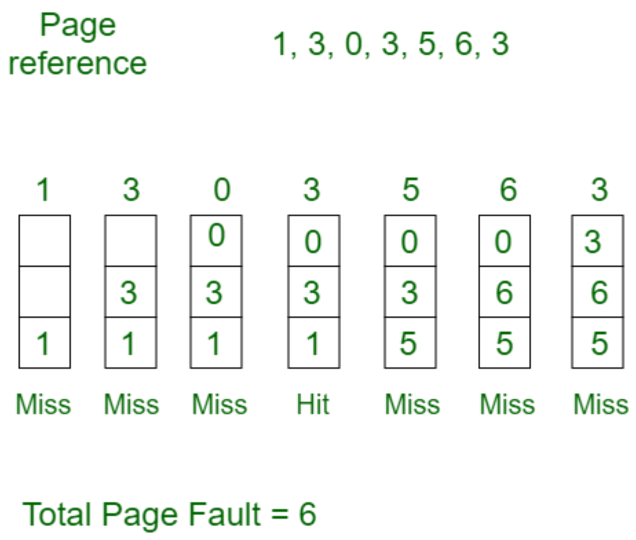
1.0 Introduction

What is the essential problem of page replacement and the structure of your simulator?

The essential problem with page replacement is that optimal page replacement is perfect but it is not realistically possible since the operating system has no way of knowing future requests. The operating system (OS) uses paging to manage memory, so a page replacement algorithm (PRA) is necessary in deciding which page to replace when a new page is coming in. To use optimal page replacement, you must set up checkpoints/benchmarks so that other replacement algorithms can analyze it. These PRAs consist of: First-In-First-Out (FIFO), Least-Recently-Used (LRU), and Segmented FIFO (VMS). A simulator called memsim.cpp was created and utilizes the previously stated PRAs and does the following: reads memory traces and replicates what virtual memory would do with a single level page table, keeps track of pages loaded in memory, keeps track of duplicate loaded pages, saves dirty bit pages to disk, and loads new pages into memory from the disk. The simulator increments a counter to keep track of reads and writes.

1.1 Methods

1.11 FIFO



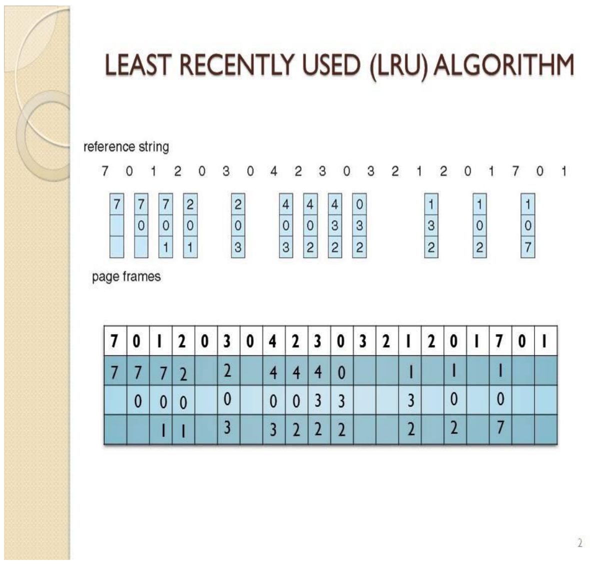
The FIFO page replacement algorithm has the OS keep track of all of the pages in memory by utilizing a vector, in C++ respectively. The oldest page in the queue would be at the front of the queue while a new incoming page would remove the oldest page and then replace it (as seen in the example image above from GeeksforGeeks). The goal of all PRAs is to reduce the number of page faults. Our simulator utilized a vector called tbl to store traces and if we found a hit then the simulator notifies the user if the debug mode is selected. Otherwise we would progress through the inputted trace file. If we come across a trace with a ‘read’ then we would simply increment our read counter by 1. However, if we come across a trace with ‘write’ then we first increment our writes counter by 1. Next we erase the first trace in tbl and push back the current trace that is entering. The simulator then tells the user that a miss occurred and a new entry will be added by eviction. This is assuming that the size of tbl is the same as the number of frames. If this occurs and tbl has more space than the number of frames, eviction is not performed.

For the number of frames, we decided to set them equal to powers of 2: 0, 1, 2, 4, 6, 7, 8, 9, 10, 11, and 12 when using bzip.trace. These led to the number of frames being: 1, 2, 4, 16, 64, 128, 256, 512, 1024, 2048, and 4096 respectively. When we had 1 frame, the shortage of memory was apparent as we had 370263 hits and 629737 misses. When the number of frames was between 2 and 512 we experienced a significant decrease in the amount of misses and an increase in hits (as seen in the Results section). However, everything after 2 to the power of 9 (512) did not increase significantly. It plateaued and did not increase in any area.

For the number of frames, we decided to set them equal to powers of 2: 0, 1, 2, 4, 6, 7, 8, 9, 10, 11, 12, and 15 when using sixpack.trace. These led to the number of frames being: 1, 2, 4, 16, 64, 128, 256, 512, 1024, 2048, 4096, and 32768 respectively. When total frames were between 1 and 2 , the shortage of memory was apparent as we had 207621 hits and 792379 misses and 470763 hits and 529237 misses respectively. This shortage of memory led to a lot more misses than hits. When the number of frames was between 4 and 2048 we experienced a significant decrease in the amount of misses and an increase in hits (as seen in the Results section). However, everything after 2 to the power of 11 (2048) did not increase significantly. It plateaued and did not increase in any area.

1.12 LRU

The LRU page replacement algorithm decides that the pages that have been used a lot in the previous instructions will be used again in the upcoming. It utilizes a clock that increments with each trace when a memory reference is made. The time for the frame that was last accessed is known as well. Every time a page is referenced, the register is updated (as seen in the image to the right by Tami Sorgente). Our simulator utilizes vector tbl, which stores traces from an input file. If we find an incoming trace that has occurred already in tbl, we look at the time correlates with that trace. The trace with the same value simply replaces itself. The algorithm identifies this as a hit. If a trace is not found in tbl and it is a 'read' trace, then we increment our read counter. If tbl is full then the simulator will check and see which of its traces has the smallest time. That trace is identified as the least recently used, thus the trace is set to be evicted. It is erased from tbl and the new trace takes its place. If the trace is a 'write' then we increment our write counter.



For the number of frames, we decided to set them equal to powers of 2: 0, 1, 2, 4, 6, 7, 8, 9, 10, 11, and 12 when using bzip.trace. These led to the number of frames being: 1, 2, 4, 16, 64, 128, 256, 512, 1024, 2048, and 4096 respectively. When we had 1 frame, the shortage of memory was apparent as we had 370263 hits and 629737 misses. When the number of frames was between 2 and 512 we experienced a significant decrease in the amount of misses and an increase in hits (as seen in the Results section). However, everything after 2 to the power of 9 (512) did not increase significantly. It plateaued and did not increase in any area.

For the number of frames, we decided to set them equal to powers of 2: 0, 1, 2, 4, 6, 7, 8, 9, 10, 11, 12, and 15 when using sixpack.trace. These led to the number of frames being: 1, 2, 4, 16, 64, 128, 256, 512, 1024, 2048, 4096, and 32768 respectively. When total frames were 1, the shortage of memory was apparent as we had only 207621 hits and 792379 misses. This shortage of memory led to a lot more misses than hits. When the number of frames was between 2 and 2048 we experienced a significant decrease in the amount of misses and an increase in hits (as seen in the Results section). However, everything after 2 to the power of 11 (2048) did not increase significantly, in terms of hits. There was a slight increase in misses which plateaued at 3890 for 2 to the power of 12 and beyond.

1.13 VMS

The Segmented-FIFO (VMS) page replacement algorithm is a variant of FIFO but it incorporates a secondary buffer. This secondary buffer offers results that are between the results of FIFO and LRU. The primary buffer works as FIFO and the secondary buffer works as LRU. So VMS can be seen as a combination of both. This algorithm requires an additional parameter P, which determines the percentage (out of 100) of memory to be used in the second buffer. If there is a fault while the primary buffer is full, the oldest trace is moved to the secondary buffer. If a reference to a trace happens in the secondary buffer, then that trace is moved to the front of the primary buffer. Lastly, if a fault occurs when both buffers are full, then the least recently used trace in the second buffer is removed. If our process does not have enough physical memory, meaning that our primary and secondary buffers are full, then our simulator evicts from the secondary and adds traces to the primary. This would mean our variable P would be set to 0, which means our simulator runs VMS as if it was FIFO. If our process has enough memory then we have less misses and the secondary buffer complements the primary very well. This would mean our variable P would be set to 100, which means our simulator runs VMS as if it was LRU.

For the number of frames, we decided to set them equal to powers of 2: 0, 1, 2, 4, 6, 7, 8, 9, 10, 11, and 12 when using bzip.trace. These led to the number of frames being: 1, 2, 4, 16, 64, 128, 256, 512, 1024, 2048, and 4096 respectively. We set our variable P to 50, since if it was 0 it would be equivalent to FIFO and if it was 100 it would be equivalent to LRU. Picking 50, a number in between, would show results true to VMS. When we had 1 frame, the shortage of memory was apparent as we had 370263 hits and 629737 misses. When the number of frames was between 2 and 512 we experienced a significant decrease in the amount of misses and an increase in hits (as seen in the Results section). However, everything after 2 to the power of 9 (512) did not increase significantly. It plateaued and did not increase in any area.

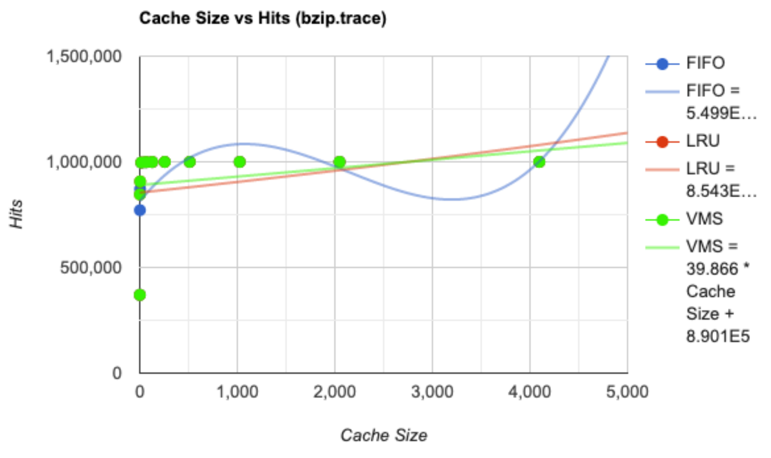
For the number of frames, we decided to set them equal to powers of 2: 0, 1, 2, 4, 6, 7, 8, 9, 10, 11, 12, and 15 when using sixpack.trace. These led to the number of frames being: 1, 2, 4, 16, 64, 128, 256, 512, 1024, 2048, 4096, and 32768 respectively. We set our variable P to 50, since if it was 0 it would be equivalent to FIFO and if it was 100 it would be equivalent to LRU. Picking 50, a number in between, would show results true to VMS. When total frames were 1, the shortage of memory was apparent as we had only 207621 hits and 792379 misses. This shortage of memory led to a lot more misses than hits. When the number of frames was between 2 and 4096 we experienced a significant decrease in the amount of misses and an increase in hits (as seen in the Results section). However, everything after 2 to the power of 12 (4096) did not increase significantly, in terms of hits. It plateaued and did not increase in any area.

1.2 Results

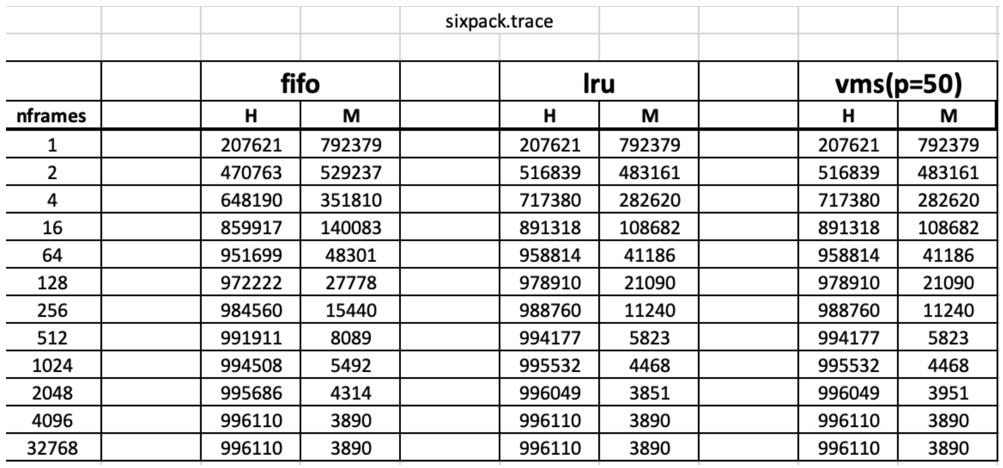
After running the experiments and analyzing the graph and tables (as seen to the right), some interesting thoughts come to mind. First we'll talk about bzip.trace when the variable P for VMS is 50%. Something interesting that we notice in table 1(to the right) is that when the number of frames is 1, the hits and misses are the same, no matter which algorithm is used. Another interesting result is that once the number of frames is 512, every single algorithm reaches its best ratio of hits to misses before plateauing. LRU performed better than VMS, however it was by a very thin margin, with less than 1% of a difference. FIFO proved to be efficient but it showed the most amount of misses before reaching the equilibrium point along with VMS and LRU at point 512 frames. The plot above also shows that LRU was the most efficient at getting the most hits, followed closely by VMS and then FIFO.

bzip.trace

		fifo			lru			vms(p=50)	
nframes		H	M		H	M		H	M
1		370263	629737		370263	629737		370263	629737
2		771162	228838		845571	154429		845571	154429
4		871399	128601		907230	92770		907230	92770
16		996180	3820		996656	3344		996656	3344
64		998533	1467		998736	1264		998735	1265
128		999109	891		999229	771		999227	773
256		999489	511		999603	397		999600	400
512		999683	317		999683	317		999683	317
1024		999683	317		999683	317		999683	317
2048		999683	317		999683	317		999683	317
4096		999683	317		999683	317		999683	317



For sixpack.trace we used the same P for VMS, 50%. Something interesting that we see here is that when the number of frames is 1, the hits and misses are the same, no matter which algorithm is used. Another interesting result is that once the number of frames is 4096, every single algorithm reaches its best ratio of hits to misses before plateauing. This is different from bzip.trace, since that one was able to have all three algorithms produce the same results at 512 frames. Something odd does happen here, our LRU and VMS are the same for every single value except where the number of frames is 2048. LRUs misses decreased from 4468 to 3851, but then in the next frame the misses increased to 3890 instead of decreasing. The graph for this would have been the exact same as the previous, with there not being much difference between LRU and VMS.



sixpack.trace

		fifo			lru			vms(p=50)	
nframes		H	M		H	M		H	M
1		207621	792379		207621	792379		207621	792379
2		470763	529237		516839	483161		516839	483161
4		648190	351810		717380	282620		717380	282620
16		859917	140083		891318	108682		891318	108682
64		951699	48301		958814	41186		958814	41186
128		972222	27778		978910	21090		978910	21090
256		984560	15440		988760	11240		988760	11240
512		991911	8089		994177	5823		994177	5823
1024		994508	5492		995532	4468		995532	4468
2048		995686	4314		996049	3851		996049	3951
4096		996110	3890		996110	3890		996110	3890
32768		996110	3890		996110	3890		996110	3890

When comparing both tables of each trace, we can see that they are both very similar. They both show VMS(50) ≈ LRU. As memory is extremely limited, all three algorithms compute the same number of hits and misses. It also shows that, eventually, all the algorithms will reach the same plateau at the same time. The only difference seen in these datasets is that even with more memory, sixpack.trace makes the algorithms take up to 4096 frames until all algorithms reach the same point. Whereas for bzip.trace it only takes up to 512 frames. This shows that there is more happening in sixpack.trace and that it requires more memory to properly store all the reads and writes. The algorithm that shows the biggest difference is FIFO. Although it is a more simple algorithm, it takes longer for it to decrease the amount of misses when compared to the other two. This could be because the other two algorithms have additional factors such as a clock or a secondary buffer. With FIFO only having one buffer, it takes the OS a longer time to search and evict traces.

1.3 Conclusions

To conclude, our simulator shows that LRU is the most efficient algorithm by less than 1% of a margin, compared to VMS. VMS should be the most efficient paging algorithm, but we believe there is a margin of error where this very slight difference is occurring. We learned that the amount of memory storage given directly correlates with a better performance by each paging algorithm. This is seen because when there was not enough physical memory, each algorithm produced the same amount of hits and misses, regardless which trace file was used. As the algorithms go through the trace files, the problem remains that they do not know how many requests will occur. However, according to our experiment, VMS and LRU are interchangeably the most optimal paging algorithm. Depending on the P set for VMS, it could be even better.

References

1. https://www.geeksforgeeks.org/program-for-least-recently-used-lru-page-replacement-algorithm/?ref=gcse
2. https://www.geeksforgeeks.org/program-page-replacement-algorithms-set-2-fifo/
3. http://www.mathcs.emory.edu/~cheung/Courses/355/Syllabus/9-virtual-mem/LRU-replace.html
4. https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.youtube.com%2Fwatch%3Fv%3D8Z9-BvSXq_Q&psig=AOvVaw3pqFfle1QPWcmbrZ1Mti47&ust=1646694598287000&source=images&cd=vfe&ved=0CAsQjRxqFwoTCPiXqsvNsvYCFQAAAAAdAAAAABAW
5. https://dl.acm.org/doi/pdf/10.1145/800189.805473