Sophia Hostetler

Kamryn Suh

2/24/2022

COP 4600

Abstract

This project required demonstration of page replacement algorithms on real traces. Using the page replacement algorithms FIFO, LRU, and Segmented FIFO, we were able to emulate a memory simulator to track the costs of each algorithm as well as the effect each has on system performance.

Introduction

In an operating system, paging is used to share memory across different processes and to manage the memory of the system. Errors called page faults can occur in paging when a program accesses a memory page that is mapped into the virtual address space yet has not been loaded in physical memory. When a page fault occurs the operating system may replace an existing page with a new page that is needed. There are multiple algorithms used in order to determine which process is the best fit for replacing pages when a new page comes in- the main ones being FIFO, LRU and segmented FIFO- with the main goal being a reduction in the number of page faults. FIFO stands for first in first out- basically meaning the first page in the queue will be the first page to be removed when replacing a page. Page faults occur in this algorithm when new references are being added into the queue and when they are being replaced. In FIFO, it is to be noted that there is also a possibility that the number of page faults can increase when increasing the amount of page frames that can be in memory at a time. This is a variation called Belady's Anomaly. LRU stands for least recently used-meaning the page that has not been used in the most amount of time is replaced with the new page. Page faults occur in LRU when new references are added and replaced, but this algorithm does not suffer from Belady's Anomaly due to it being a stack algorithm. Segmented FIFO basically mixes the two previous algorithms together and incorporates a secondary buffer. The primary buffer runs under FIFO, while the secondary buffer utilizes LRU. When a page fault occurs in this algorithm three instances could happen- the primary buffer will move the oldest page to the secondary buffer if it is full, if the secondary buffer has a page reference made to it the page will move to the front of the primary buffer, or if both buffers are full the least recently used page in the second buffer is removed. In addition, if the secondary size cache (LRU) is size 0, then it degenerates into a FIFO algorithm. If the primary size cache (FIFO) is size 0, then it degenerates into an LRU algorithm. This

project utilizes all three algorithms in order to build a simulator that will read a given memory trace and simulate paging. The simulator will provide the number of frames, events in the trace, disk reads, and disk writes in order to demonstrate the processes that are occurring within the virtual memory system.

Methods

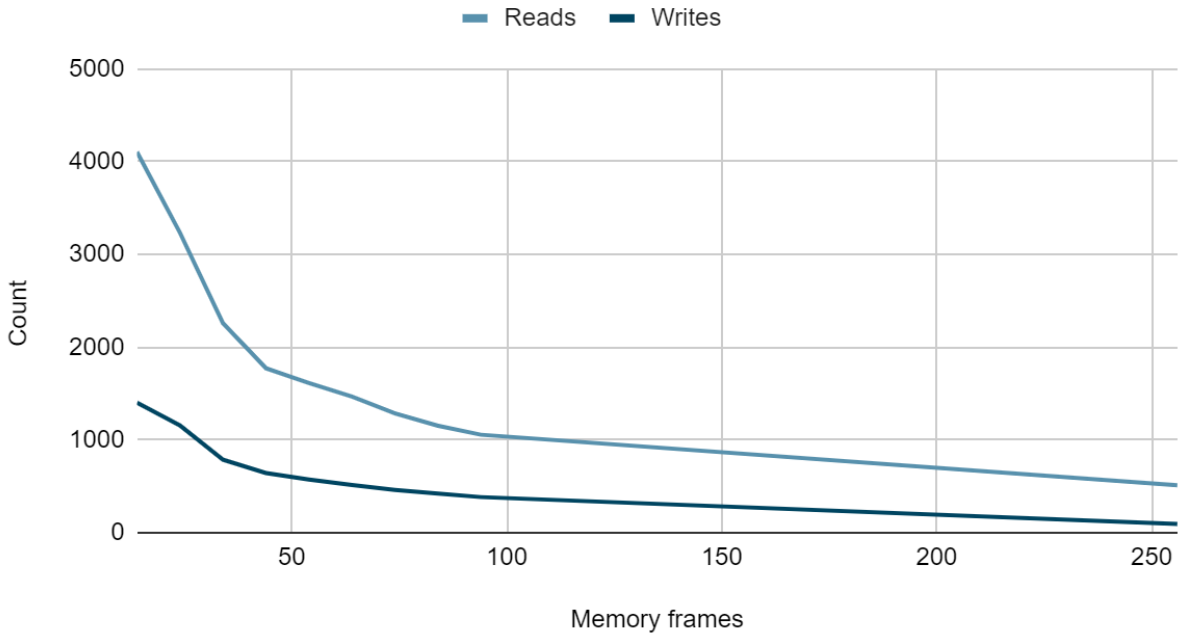
In order to test different functionalities of our simulator we ran different tests for each memory trace. We tested multiple values for FIFO, LRU, and VMS that were either higher, lower, or closer to what was needed to run a proper implementation. We tested LRU and FIFO with frames ranging from 14 through 256 (with increments of 10, for example: 14, 24, 34, 44, etc) to determine what was occurring when each algorithm was implemented. For VMS, we implemented with a p value ranging from 0 through 100 percent and 14 and 64 frames. Our VMS algorithm was not 100% complete, however it dealt with the edge cases correctly, where the primary and secondary cache was 0 for both. We implemented a graph with the usage of 14 and 64 frames with various p values to showcase a similar directional line graph as the FIFO and LRU.

Our data structures for this project consisted of:

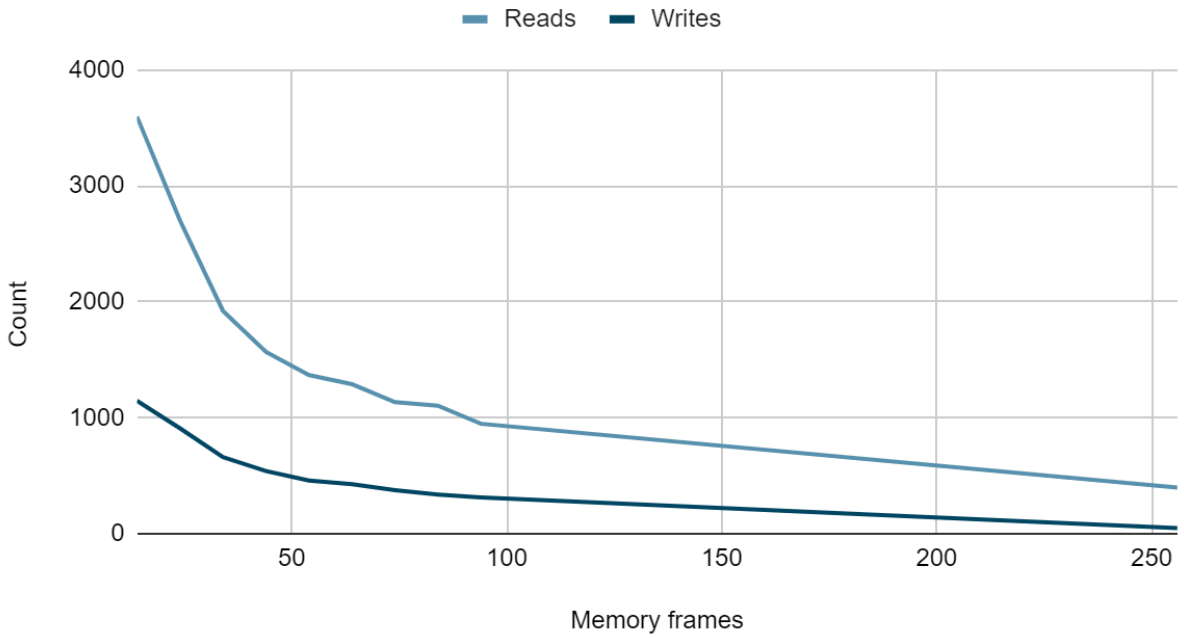
- Unordered_set for both FIFO and LRU to represent current pages
- Unordered_map for LRU to represent the least recently used elements of pages
- Unoredered_map for both FIFO and LRU to represent the trace array and the RW (read and write) array
- Vector<int> for FIFO to store pages in FIFO manner
- Variables for read and write to count
- Vector<unsigned int> and vector<char> for the all the addresses passed in and the RW's attached to the address

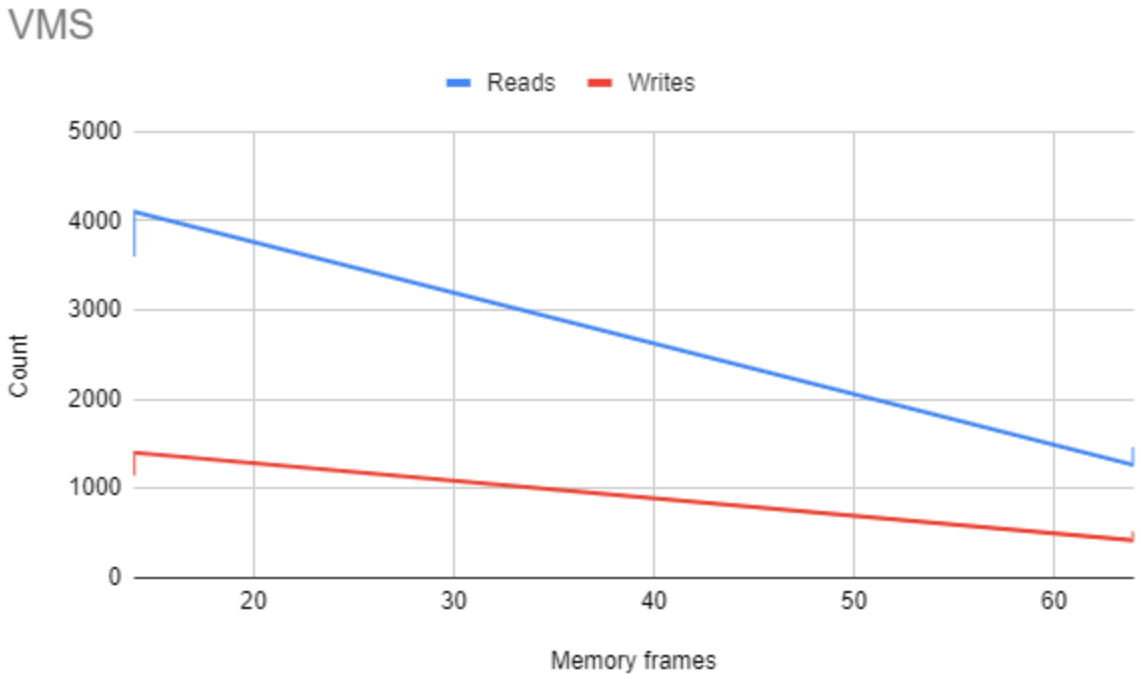
Results

FIFO


Reads
Writes
5000
4000
3000
2000
1000
0
Count
50
100
150
200
250
Memory frames

LRU


Reads
Writes
4000
3000
2000
1000
0
Count
50
100
150
200
250
Memory frames



For the three graphs, it consists of FIFO, LRU, and VMS. For all, the x-axis represents the memory frames and the y-axis represents the count of reads and writes. These results were tested with the bzip.trace file. FIFO and LRU showcase a similar line graph with a downward trend as the memory frames increase. VMS also follows a similar line graph with a downward trend with only frames 14 and 64 tested. With frames 14 and 64 tested, we used various p values ranging from 0 through 100. Unfortunately, our results for VMS aren't entirely accurate in the code implementation with an exception of the edge cases (primary and secondary cache size is 0). However, we tried our best to demonstrate how VMS results would look like with various percentage and frame values. The number of frames and percentage values used in all three algorithms are listed below:

- FIFO & LRU frames: 14, 24, 34, 44, 54, 64, 74, 84, 94, 256
- VMS frames: 14, 16
- VMS percentage values: 0, 1, 10, 20, 30, 40, 50, 95, 99, 100

Conclusions

Through this process we learned the differences in memory processing depending on which algorithm is selected. We realized the differences in efficiencies depending on which algorithm is being utilized. FIFO is least ideal when keeping a write count to a minimum, whereas LRU and VMS are better options. This was tracked by choosing a specific number of frames, in our case 64, to track the number of reads and writes that would occur when running each algorithm. However, there is not a single algorithm that works best for every instance that

could be encountered due to differences that could occur in the sizes of reference strings. We also observed that the size of available memory affects memory performance while conducting trials. As the size of the memory increases, the performance of memory is more efficient. For example, when the number of frames was set to 14 in FIFO the reads were 4105 and the writes were 1403. However, when the frames were increased to 256, the read count dropped to 511 and the write count became 94. By doing this project, we learned more about page replacement algorithms and the functionality behind them. Overall, this led us to discover the importance of the subjects of page replacement, virtual memory, and physical memory in disks.