COMP2240 – A3 Report – Page Replacement Algorithm Analysis

By Matthew Corbett
October 2023

Introduction

The purpose of this report is to compare the performance of the clock policy replacement algorithm under two different scopes. The first is fixed allocation with a local replacement scope (FALR), which will choose to replace a page only from the frames allocated to that process. The second is variable allocation with a global replacement scope (VAGR), which will choose to replace a page from all available frames in main memory. The processes are run through a round-robin scheduling algorithm until all are completed.

Testing Approach and Methodology

When testing the program, I had temporary outputs to console for three occasions: when a page fault occurred, when a page is executed, and when a process finishes its RR time quantum. Each of these outputs also contained the current algorithm time and related process name. This allowed me to see a complete timeline of the algorithm at every relevant time unit to ensure it was running as intended. I also created a sample set of processes to specifically pinpoint an advantage that VAGR has over FALR. This sample set contained 3 processes each with 5 unique pages, a fourth process with 10 unique pages, and a total frame size of 35. This resulted in FALR allocating 8 frames to each process, which caused process4 to fill every frame and need to swap pages out, while the other 3 processes had 3 frames left unused. Thus, while VAGR only had the minimum 25 page faults while only needing to transfer pages into empty frames, FALR had 29 page faults (due to page order) with an extra 2 unused frames. The only other sample sets I created were simply used to test edge cases such as a set with a large amount of processes and pages. I also created and tested some process.txt files that contained data written as a single line without spaces to ensure I was reading from the files correctly.

Comparison of Page Replacement Methods

The results of the page replacement methods showed a relationship between the size of the datasets (the number of unique pages in each process) and the amount of frames in main memory that were allocated to this set of processes. For FALR, the total frames were split evenly between each process and any extra frames were left unused. If each process had a total of unique pages equal to or less than its allocated frames, it would only need to place pages into empty frames thus generating the minimum amount of page faults required. This would produce the exact same results of VAGR given the same total frames and set of processes. The two scopes only begin to diverge when at least one process in FALR has more unique pages than its allocated frames, and thus must replace a page already occupying an allocated frame even if there are still empty frames in main memory. VAGR can utilise these empty frames and will only start swapping out pages when all frames are full, thus generating less page faults than FALR in these situations.

Edge Cases Considered and Related Behaviour of Algorithm

I used a sample set to test if the program could handle 10 processes each with at least 20 unique pages, and it had no issues. I also made a sample set with processes containing 2

pages allocated a single frame that needed to swap them back and forth. This worked correctly and resulted in a very long turnaround time due to replacement time, which I can imagine would have been even worse if dispatch time was accounted for. I couldn't think of any other edge cases to test.

Issues During Development and Unexpected Behaviours

First thing I encountered was needing a way to store the page(int) and a use-bit(int) together, and I initially decided to use the Point class with x being the page and y being the use-bit. This was a little messy at first as I found out it didn't have methods to change only y. I finished FALR and VAGR with points but realised I wasn't getting the output I wanted with VAGR. This was because when I wanted to see if a processes page was already in main memory I was accidentally scanning through every process frame in the frame array. This wasn't an issue in FALR as I had a 2D frame array that only checked frameArray[PID][here]. So, I then wanted to keep a process ID with each frame/page so I could check if it matched the current process when replacing a page. I couldn't do this with the Point class, so I got rid of it and made a Frame class that held the page(int), use-bit(int) and PID(int) then swapped out the Points with this class.

As for unexpected behaviours, I encountered two that I recall. With the first, I was following the spec and moving blocked processes that finished page swapping during a current process's runtime, back to the ready queue before the current running process expired its RR time quantum. Because I didn't update the current algorithm time before moving those processes, they were seeing that the ready queue was empty (as I take the current job out of ready queue when 'running' it) and moving the algorithm time forward to when they were ready (to simulate that wait time passing). Thus, the current expiring process was put behind those in the ready queue instead of just running itself again as it should have been the only process ready at that time.

The second was similar, I was moving a process that finished its round back to the ready queue before I checked if it had any pages left, which resulted in incorrect turnaround times.

Review of Results (Conclusion)

FALR seems to suffer from the fact that you can't have a perfect fixed allocation size of frames when processes might have largely varying number of frames needed. You may have processes with unused frames while other processes are needing to swap pages in and out as they do not have enough frames, leading to more processor idle time and overhead (due to swapping). The even distribution of frames in our simulation also results in there being free frames that will never be used.

Meanwhile, VAGR can use every free frame in main memory, which is an advantage in large and highly varying sets of processes and pages. Also, due to global scope never being a fixed allocation, VAGR can dynamically 'give' processes that are page thrashing more frames when needed, taken from processes that are not efficiently utilising them.

In conclusion, I believe global scope policies will always perform better than local scope policies that have a fixed allocation.