An Analysis of Memory Management Policies and their Efficiency

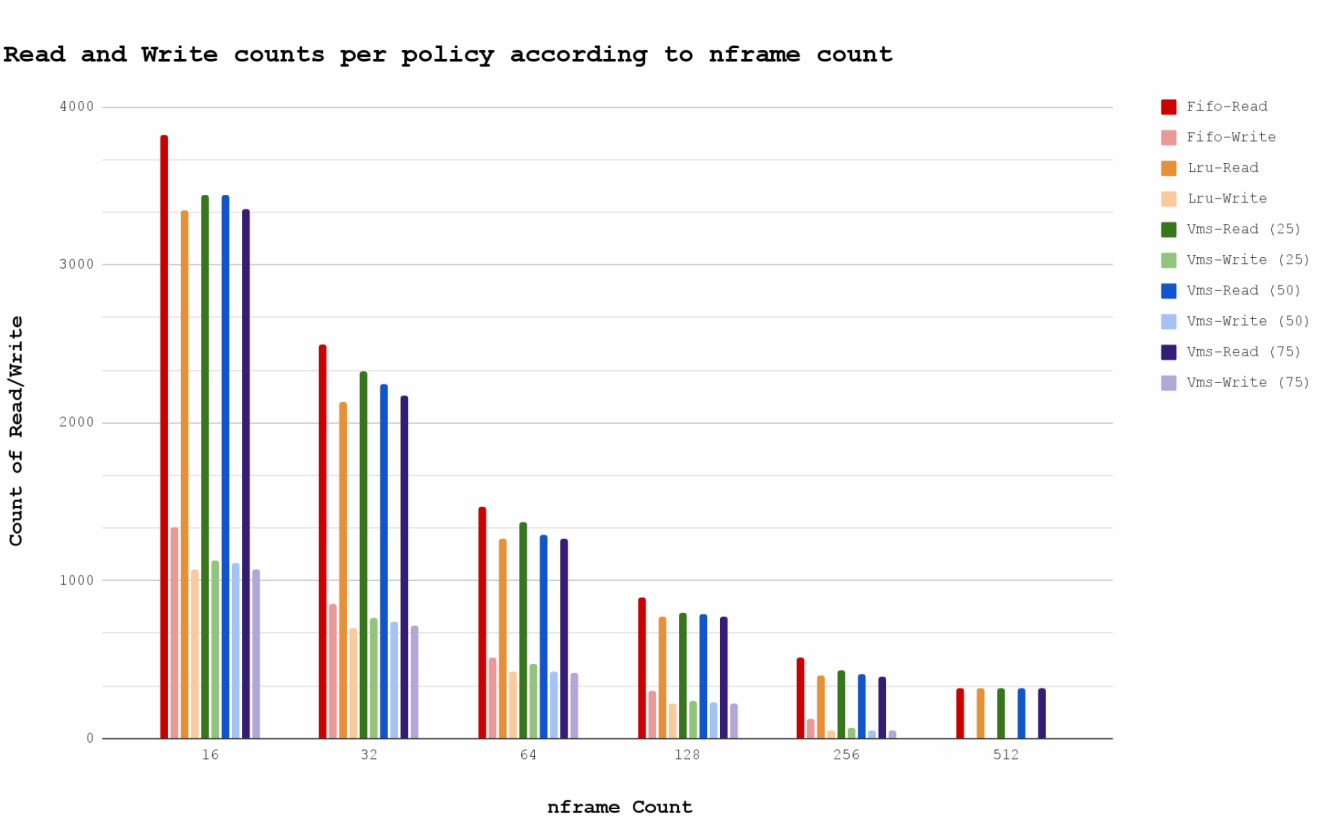
Sidharth Menon
COP4600 Section 01

In this report, I have simulated read and write counts with three different memory management policies and with variable frame count, First-in First Out, Least Recently Used, and Segmented FIFO combining two frames of FIFO and LRU. Tested using trace files of 1,000,000 entries of addresses and a Read or Write command, the data acquired from these tests have been compiled into the table and graph shown below.

Simulating real read/write counts that our operating systems function by every time we use computers, this report is helpful in anlazying which memory management policies are most efficient and why they are. As it is a simulator, we can extrapolate the objective information that is received to how real systems efficiently access information for the user to access.

The table below represents the relationship between a selected frame count and each policy's read and write counts for the test trace file bzip.trace. VMS is segmented-fifo and the number in parentheses represents the percentage by which the frames are split into two buffers.

Policy-> / nframes	FIFO Read	FIFO Write	LRU Read	LRU Write	VMS Read (25%)	VMS Write (25%)	VMS Read (50%)	VMS Write (50%)	VMS Read (75%)	VMS Write (75%)
16	3820	1335	3344	1069	3445	1124	3439	1112	3351	1069
32	2497	851	2133	702	2329	765	2243	740	2168	715
64	1467	514	1264	420	1367	471	1290	427	1267	416
128	891	305	771	224	795	239	785	229	772	225
256	511	125	397	48	432	65	406	54	393	49
512	317	0	317	0	317	0	317	0	317	0



According to the graph, we can see that First-In First-Out (FIFO) representation is consistently the highest read/write count, showing how its simplistic implementation leads to it performing the worst. Due to this policy never accounting for how often a page may appear, there is unnecessary reading of files that may have just very recently been popped from the buffer.

Jumping ahead to the VMS/Segmented-FIFO implementation, the most important takeaway seems to be that the larger the secondary buffer is allowed to be, the more efficient the program is. This can be attributed to the secondary buffer implementing LRU functionality which we will discuss next. The performance of VMS is right in between purely FIFO and LRU implementation, which is understandable as it is a mix of both policies.

The clear winner in terms of lowest read and write counts is the Least Recently Used (LRU) policy. Both counts are much lower than FIFO's output and the bigger the LRU secondary buffer is, the better VMS performs as well. LRU functions by keeping track of a counter that ticks up every time a page in the buffer is not called, and gets reset to 0 whenever it gets a call. When a page must be popped, instead of removing the oldest in entry order, such as in FIFO, the element with the largest counter value is removed instead, essentially removing the "least recently used."

Between the three memory management policies simulated in this experiment, Least Recently Used was the most efficient implementation across all nframe values. This is due to it accounting for the frequency of appearances by pages, and shows the importance of forethought in matters of efficiency. While the test trace files were a million entries long, real systems have billions upon billions of entries, so the time differential that may be minor at a smaller scale grows immensely and can heavily impact the speed of the overall system, if not for an efficient memory management policy implementation. And as LRU consistently was the most efficient throughout all nframe counts, we can safely assume that upon extrapolating this information to billions of billions of entries, LRU implementation would still be the most efficient policy out of these three.

Estimated work time:
Initial FIFO functionality took me at least 3-4 hours to fully set up and working including the general memsim.c setup and user input functionality.
LRU implementation was very fast, and under an hour as I modified my FIFO code and it managed to work perfectly on first try.
Segmented-FIFO took me the longest out of the three as it took me a while to first understand exactly how this policy functioned. Once I understood it properly, the most difficult part was connecting the two buffers together and ensuring Dirty bits do not get accidentally overwritten. I had implemented the LRU by attaching the trace count at which each page entered the secondary buffer to each element, but for some reason, this value would randomly affect other values and break the LRU from functioning properly. I had to spend some time rewriting the code until it worked as intended. This overall must have taken me 4-5 hours.
Report writing took about 2 hours.

Overall completed over 5 days, totalling ~12 hours of work.