OS Project 2 – Performance of Various Paging Algorithms

The fundamental problem of memory management is limited resources, in this case main memory, RAM. How to manage the RAM demands of 10s to 100s of processes is an important subject. When RAM is not limited, a new process only needs to load each requested page from the disk once. When this isn't the case, a process can only keep loaded so many different pages at once. When space runs out, what page do we choose to get rid of to make room? The goal of this paper is to compare several page replacement algorithms primarily by number of disk reads/writes but also in complexity. These algorithms being random, first in first out, least recently used, and VMS.

The page size for this experiment is always 4096KB. The virtual addresses are 32 bits, so the VPN is 20 bits. The number of frames available is user specified for each run.

Description of the implementation of each algorithm is as follows:

Random: This algorithm uses 2 arrays, each nframes in size. One holds the VPN of the pages, and the other holds the clean/dirty indicator at the same index. When the array is full, a random index is selected using the C function rand() with a seed of int 1. The new page is swapped in place of the old page. Evicted pages with a 'W' cause the writes variable to be incremented.

FIFO: This algorithm uses a linked list of structs. Each struct holding both the VPN and the clean/dirty indicator. As each new page is loaded it is appended to the back of the list. When the list is full and a new page must be loaded, the page at the front of the list is deleted. Evicted pages with a 'W' cause the writes variable to be incremented.

LRU: This algorithm also uses a linked list but unlike FIFO, if a page is used while it is in the list it is removed and reinserted into the back of the list. Similarly, pages at the front of the list are evicted to make room.

VMS: This algorithm assumes VPNs starting with a 3 belong to process 1 and all others belong to process 2. Two linked lists are needed, one for each process. Each list can expand up to nframes long, but the total length of both lists cannot exceed nframes. The eviction policy checks which process is asking for more room. If this process is using a greater or equal number of frames than the other, it must evict from its own list. Else it can evict from the other list. The eviction policy also prioritizes clean pages above dirty, skipping dirty pages that are older than clean ones.

Methods:

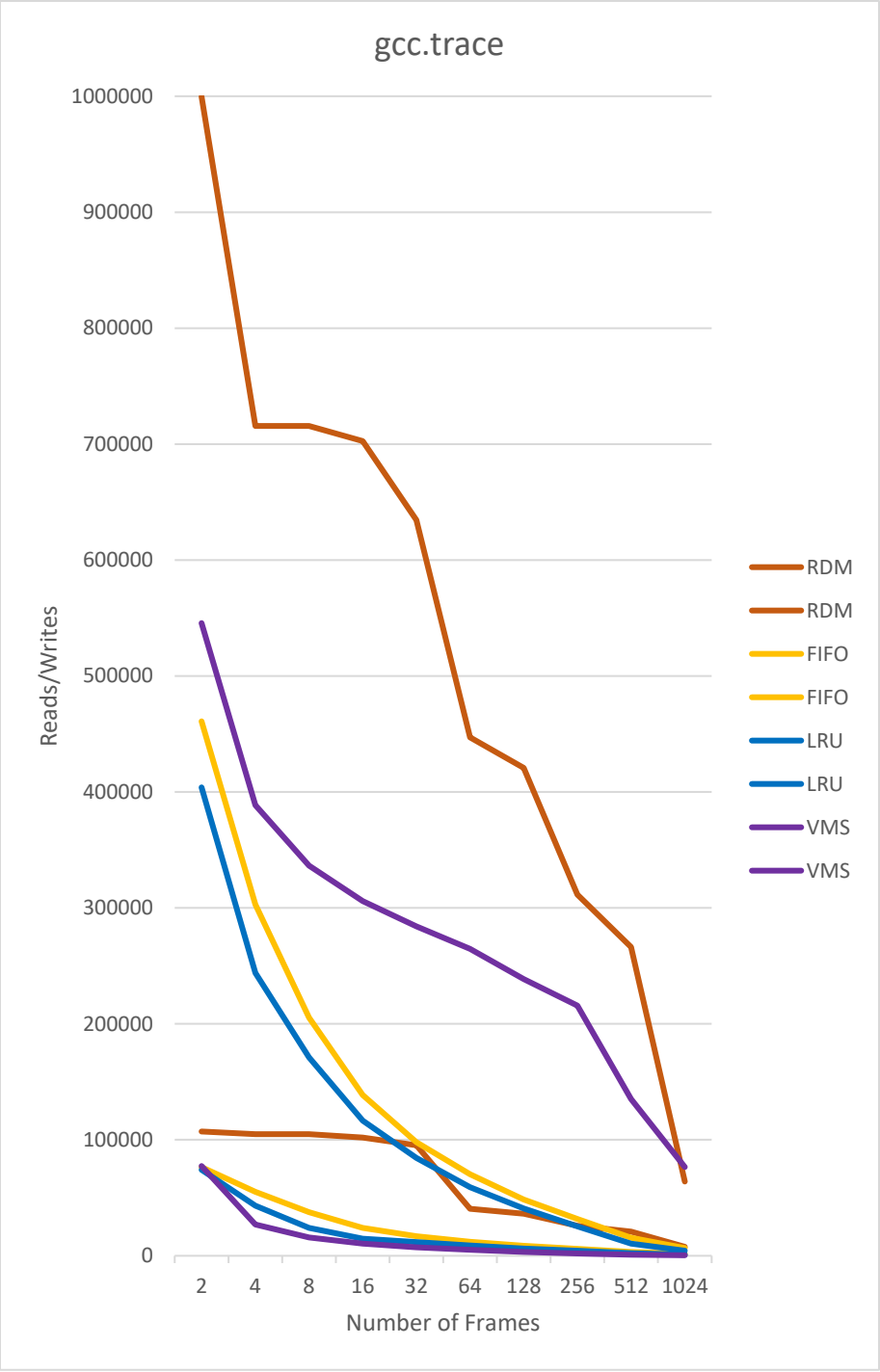
Each algorithm is ran for all powers of 2 starting from 2 and ending where the entirety of the process fits inside the given number of frames. This is repeated for gcc, sixpack, swim, and bzip trace files. This will provide data for the range of extreme shortage of memory to completely sufficient memory.

Results:

Table for the gcc.trace file with nframes ranging from 2-4096, for every algorithm.

gcc.trace	random		FIFO		LRU		VMS	
nframes:	RDM reads	RDM writes	FIFO reads	FIFO writes	LRU reads	LRU writes	VMS reads	VMS writes
2	999700	107180	460912	76432	403955	74158	545640	77389
4	715743	104907	302860	55013	243809	43187	388568	26954
8	715721	104894	205368	37524	171186	23983	336525	15909
16	702468	101870	138539	24043	116604	14749	305768	10489
32	634403	95235	98067	16759	84401	11737	284110	7225
64	447146	40508	70315	12053	59089	8863	264663	5122
128	420733	36159	48526	8487	40821	6131	238586	3257
256	311520	26017	31698	5945	25308	4231	215724	2015
512	266155	20813	15609	3425	10425	2173	135154	1028
1024	64012	7732	6673	1747	4391	955	76523	427
2048	13052	3452	3432	700	2904	427	3564	0
4096	2851	0	2852	0	2852	0	2852	0

Graphical representation of previous table:



Each color corresponds to one algorithm. The first line of the same color is the number of reads, it is always higher than the number of writes of the same algorithm.

Conclusions:

Comments for each algorithm:

Random: Performs quite poor compared to every other algorithm. The only positive aspects are simplicity and minimal overhead.

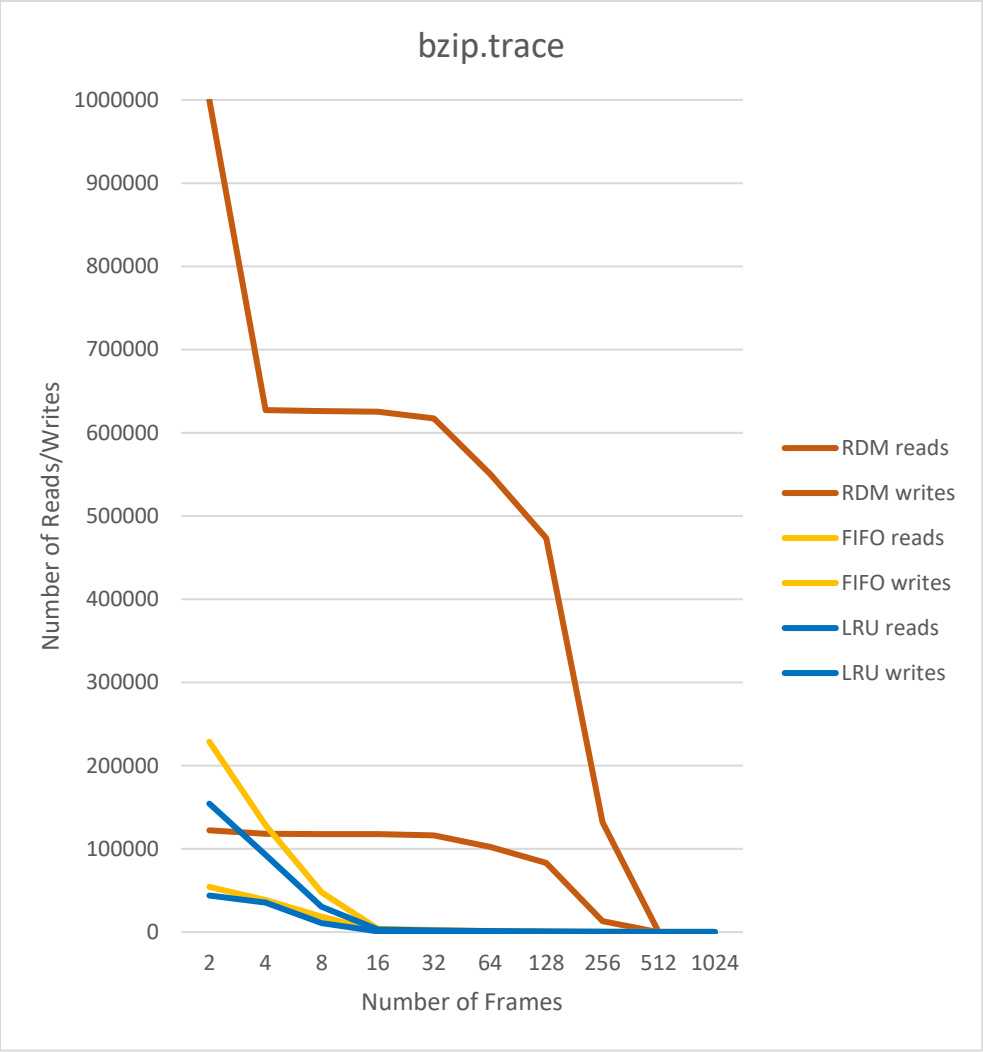
FIFO: Substantial improvement over random. This algorithm is more complex to implement, but the speed of execution is similar to random.

LRU: Surprisingly minimal improvement vs pure FIFO, but still a consistent drop. This algorithm shows its complexity with noticeably longer run time.

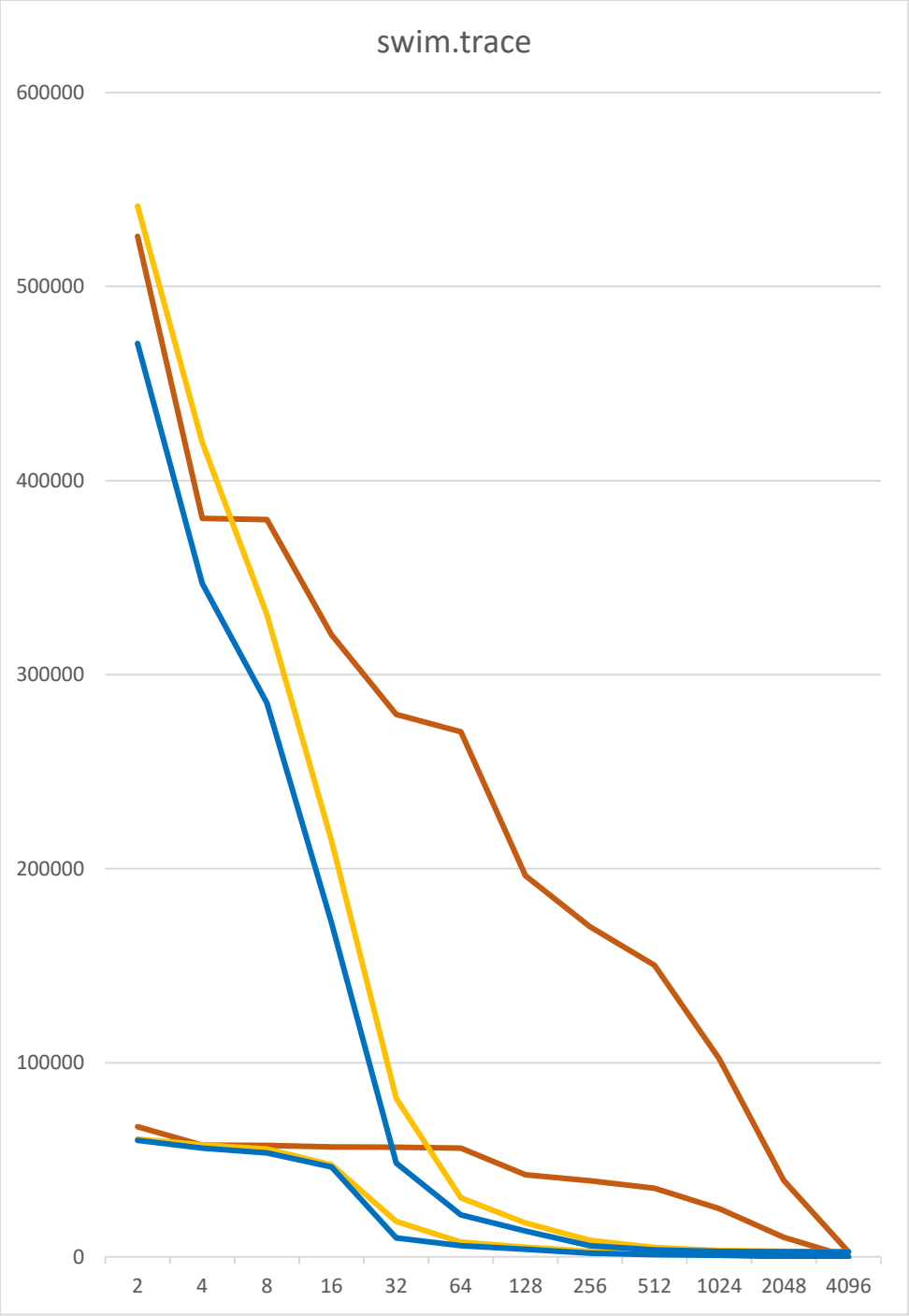
VMS: Here we see a special case where this algorithm reduces writes in exchange for more reads. In the case where this is desirable, this would be the algorithm of choice. This algorithm is the most complex in overhead.

General comments: With memory shortage we see the relative strength of each algorithm, but as each approaches having enough space to fit everything they converge. The results are consistent with expectations. FIFO considers time locality and performs better than random. LRU is a more intelligent FIFO that will hold relevant pages longer.

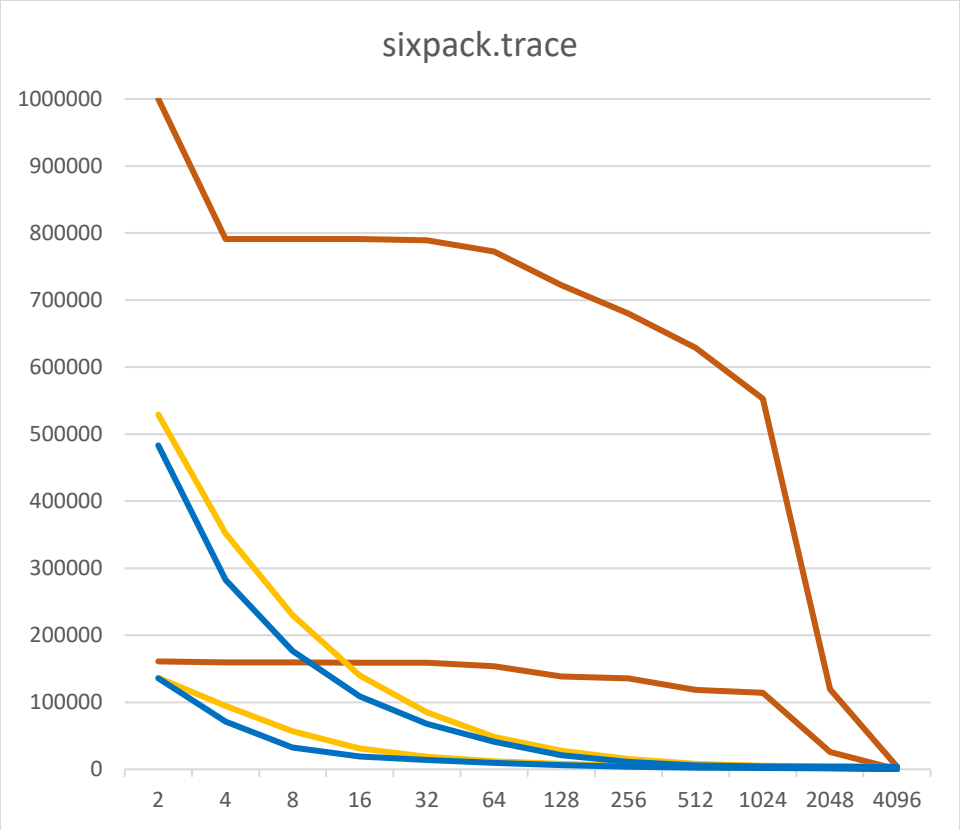
Behavior of other traces:



Bzip is very read heavy and needs only ~300 pages to completely fit in memory. LRU and FIFO here have very similar performance.



Swim requires ~2500 different frames. Random performed relatively well. It may be speculated that this trace has less time locality than the other traces.



Once again, a similar trend with sixpack. Here, random performs especially poorly likely due to high time locality that FIFO and LRU can take advantage of. Random also is very insensitive to more available memory, being almost flat between 4 and 1024 frames. Sixpack took the most memory to fully satisfy requiring nearly 4000 different pages.