Project 2: Memory Simulator – Comparing Different Page Replacement Policies

Kejvi Cupa and Param Chokshi

University of South Florida

Department of Computer Science and Engineering

Introduction

The goal of the project is to compare 3 different page replacement policies, which are FIFO (First In First Out), LRU (Least Recently Used), and Segmented-FIFO. The OS (Operating System) implements a page replacement policy to evict a page loaded into the memory from the disk when a new page (not loaded in the memory) is to be loaded from the disk. The procedure of removing and adding pages is called swapping. The algorithm used to decide which page to evict makes a significant contribution in increasing the performance (running time) of running processes. To compare the algorithms a simulator was created using C/C++ to simulate the process of reading memory traces. The simulator implemented all the aforementioned algorithms on the same memory trace, and the results were used to compare the performance of each algorithm in different scenarios. The simulator used some default data structures (specifically deque)provided by the C/C++ standard libraries to implement FIFO. For implementing LRU, a FIFO data structure was used along with the condition of updating the structure every time an event occurs in which the page was already loaded in the memory. Every time this event would occur, the page would be shifted to the end (last one to be removed), and thus removing the least used one every time a page is to be evicted. Segmented-FIFO is implemented by using two FIFO deques as the primary and the secondary buffers. Every page removed from the primary buffer is inserted into the secondary buffer. Here if an event occurs in which the page is not loaded in the primary buffer, a page fault (or a miss) is not observed if the page is available in the secondary buffer, but is instead transferred from the secondary to the primary buffer.

Methods

To compare the algorithms two actual traces (bzip.trace and sixpack.trace) are used. Since real memory traces are very huge in size, the working set chosen here will be in the range of 10,000-1,000,000 memory events. The different memory sizes used to compare were of the order 1, 2, 4 …2^n . Each algorithm was tested with the same parameters with both traces. Here the parameters selected helps in finding the memory sizes for which the memory fits the working set or is smaller than the actual requirement (leads to a higher number of page faults), or the point at which increasing the memory size doesn't affect page hits or misses.

Parameters used:

Events: 10000 and 1000000

Algorithms : FIFO, LRU, VMS(20%), VMS (40%), VMS(60%)

Memory sizes: 1, 2, 4 …2^{10}

Using these results generated through these values, we will evaluate "Page Faults vs Memory Size" line charts to make deductions. We will also evaluate if the ideal memory size is logical or not given today's technological advancement.

Results

Table 1: Page faults for the first 10000 events in sixpack.trace

# of frames	fifo	lru	vms (20)	vms (30)	vms (40)	vms (60)
1	8377	8377	8377	8377	8377	8377
2	6045	5894	6045	6045	6045	5834
4	4700	4304	4700	4451	4451	4346
8	3461	3047	3317	3203	3124	3083
16	2455	2101	2225	2210	2157	2123
32	1780	1570	1639	1616	1605	1579
64	1398	1326	1350	1344	1337	1330
128	1177	1136	1146	1140	1138	1133
256	1102	1079	1087	1084	1084	1082
512	865	828	849	842	833	827
1024	804	804	804	804	804	804
2048	804	804	804	804	804	804

Table 2: Page faults for the first 10000 events in sixpack.trace

# of Frames	fifo	lru	vms (20)	vms (30)	vms (40)	vms (60)
1	792379	792379	792379	792379	792379	792379
2	529237	483161	529237	529237	529237	483161
4	351810	282620	351810	306866	306866	288955
8	230168	176496	202958	189222	182379	178933
16	140083	108682	117537	114938	111733	109479
32	85283	67747	72107	70663	69542	68288
64	48301	41186	42977	42042	41881	41635
128	27778	21090	23802	23163	22725	21685
256	15440	11240	12324	11798	11581	11371
512	8089	5823	6374	6095	5997	5855
1024	5492	4468	4645	4540	4514	4472
2048	4314	3951	4003	3981	3973	3959
4192	3890	3890	3890	3890	3890	3890

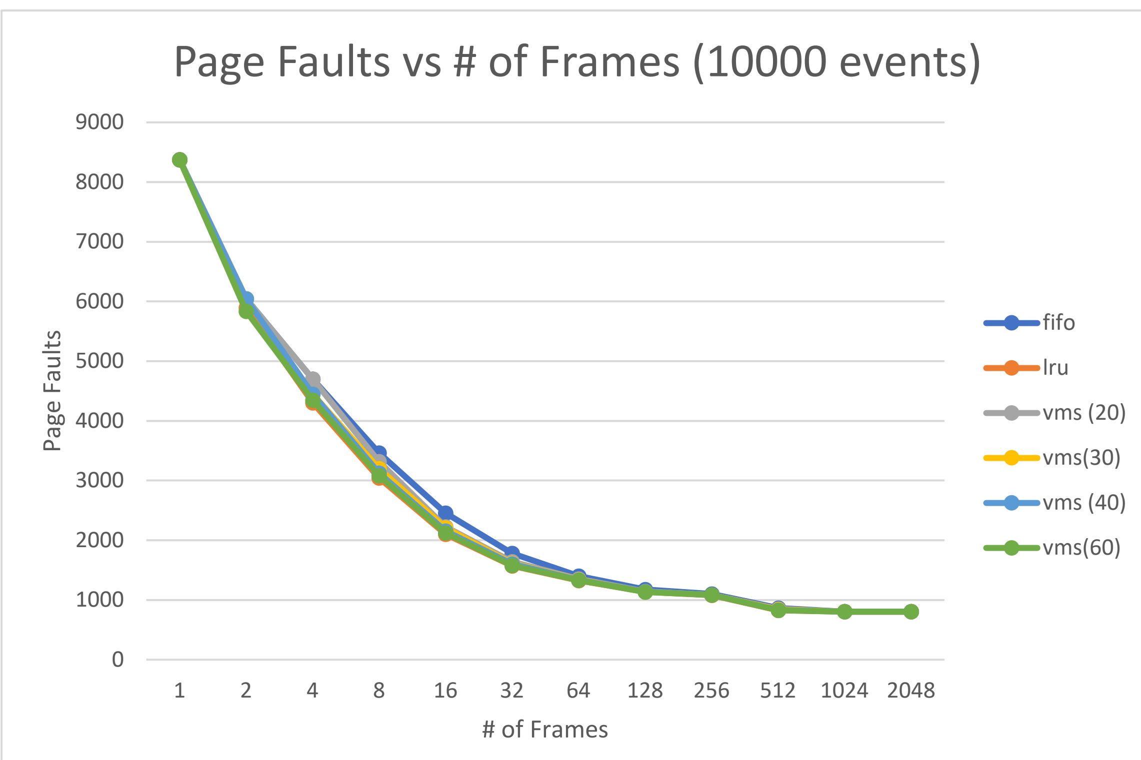

Figure 1 : Line chart corresponding to table 1

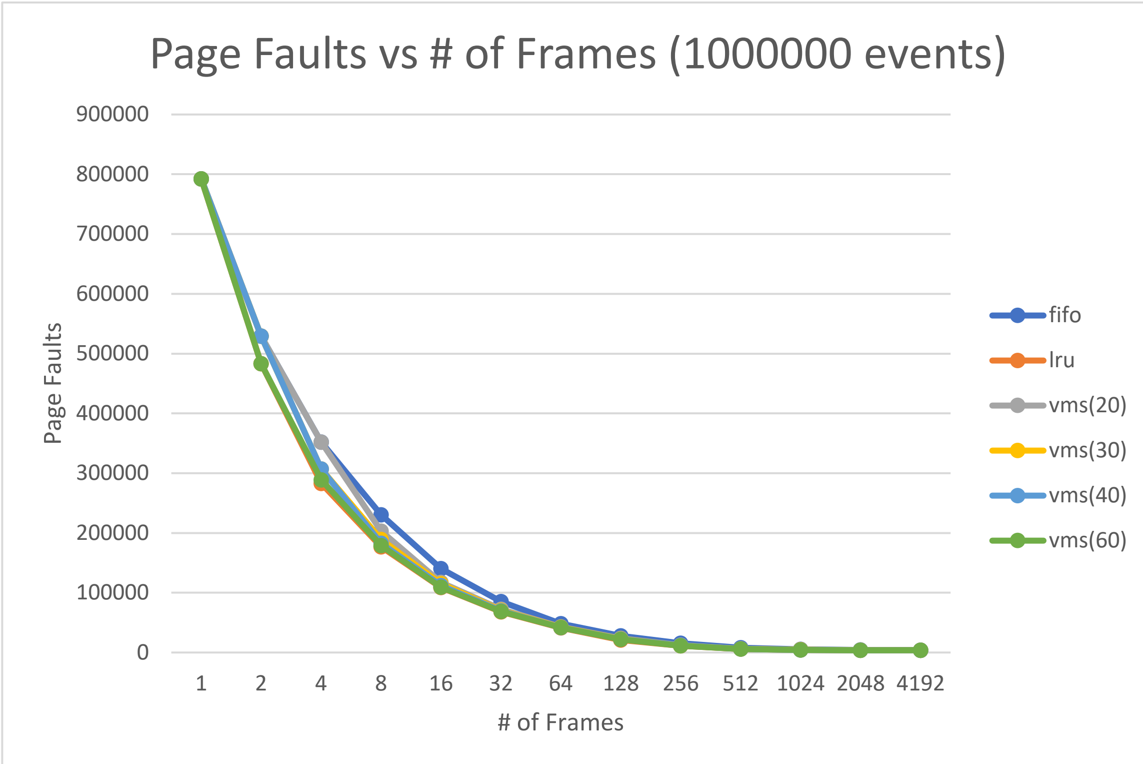

Figure 2: Line chart corresponding to table 2

Table 3: Page faults for the first 10000 events in bzip.trace

# of frames	fifo	lru	vms (20)	vms (30)	vms (40)	vms (60)
1	5275	5275	5275	5275	5275	5275
2	2399	1978	2399	2399	2399	1978
4	1348	1130	1348	1166	1166	1141
8	875	801	816	816	803	799
16	682	639	652	653	647	642
32	569	542	549	545	545	543
64	487	460	475	469	467	462
128	403	388	393	395	392	393
256	306	288	293	288	288	287
512	283	283	283	283	283	283
1024	283	283	283	283	283	283

Table 4: Page faults for the first 1000000 events in bzip.trace

# of frames	fifo	lru	vms (20)	vms (30)	vms (40)	vms (60)
1	629737	629737	629737	629737	629737	629737
2	228838	154429	228838	228838	228838	154429
4	128601	92770	128601	96509	96509	92954
8	47828	30691	44238	44006	38290	33997
16	3820	3344	3490	3445	3431	3394
32	2497	2133	2356	2322	2285	2210
64	1467	1264	1391	1356	1308	1274
128	891	771	818	785	780	776
256	511	397	449	426	409	402
512	317	317	317	317	317	317
1024	317	317	317	317	317	317
2048	317	317	317	317	317	317
4.19E+03	317	317	317	317	317	317

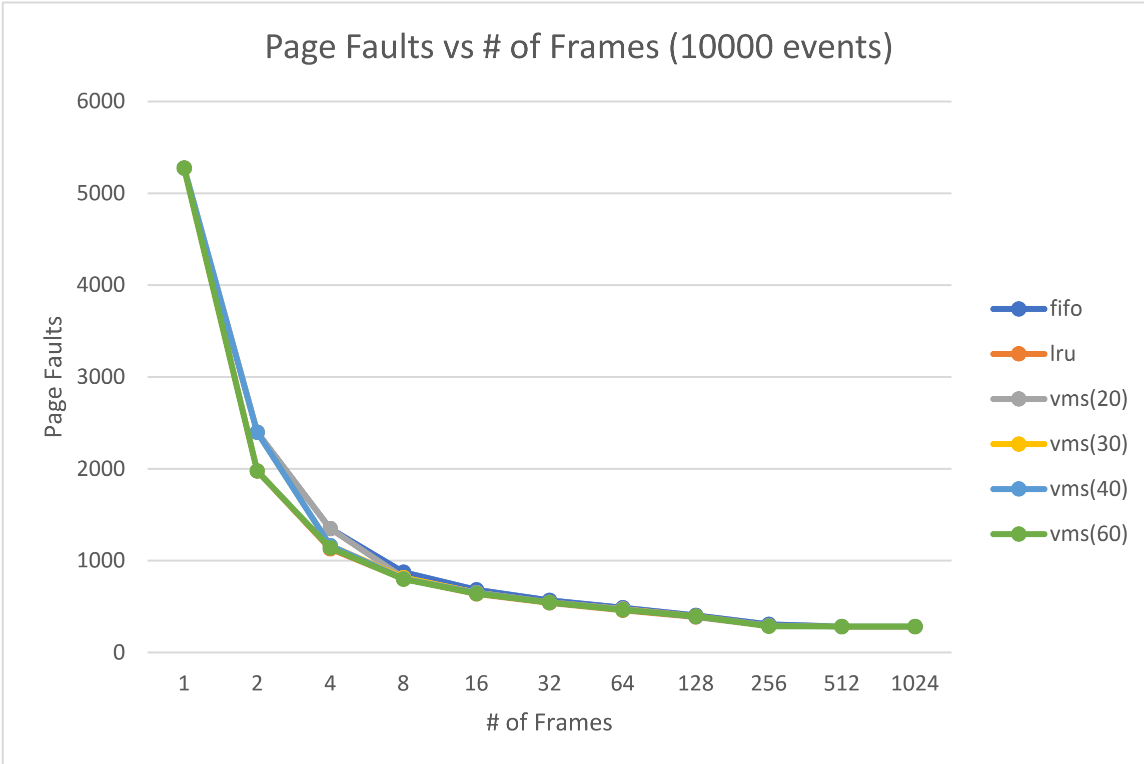

Figure 3: Line chart corresponding to table 3

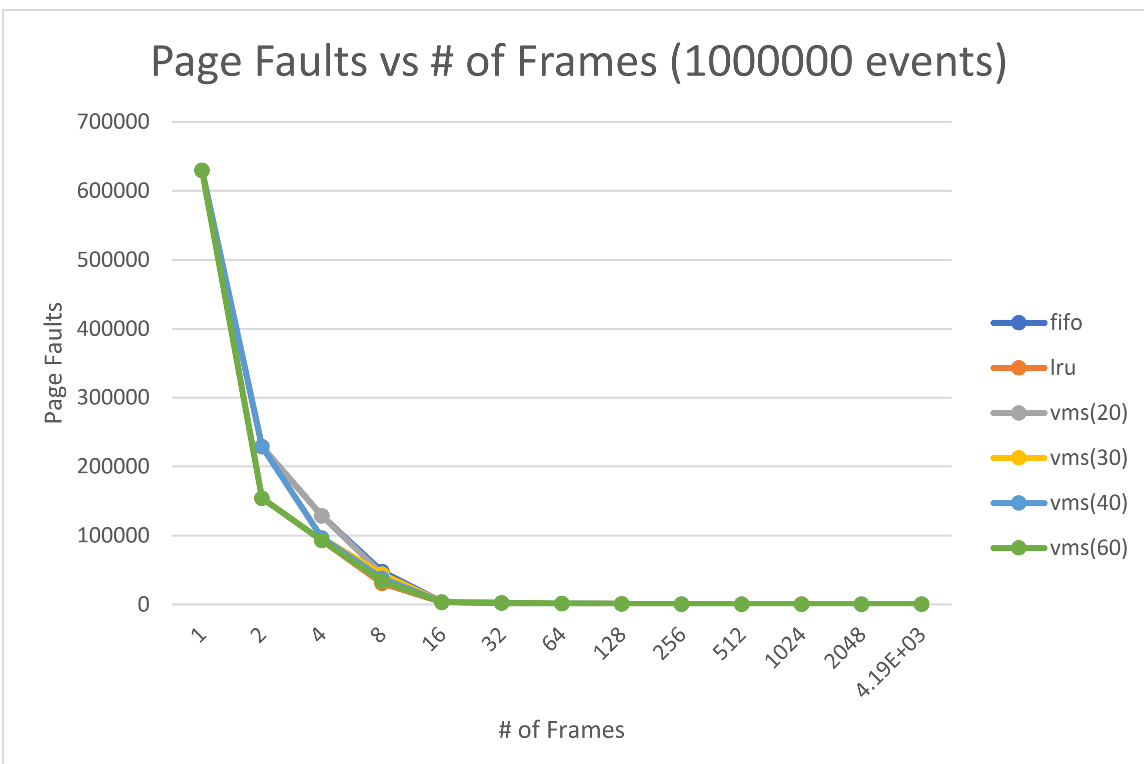

Figure 4: Line chart corresponding to table 4

Here, the main strategy in evaluating performance is to look at the number of page faults or page misses during each test. Lower the number better the performance. Another metric is page hit which is simply the number of events – the number of page misses, here the higher the number better the performance. Looking at the tables and the figures, in both cases LRU seems to be more efficient than other algorithms. Each algorithm reaches a point where increasing the memory doesn't significantly improve performance. And in most cases LRU reaches the point earlier (for lesser memory size) than other algorithms. For example. in table 1 it is apparent that the performance doesn't improve significantly once the memory size is 512 frames, i.e. 512*4 KB. Similar patterns can be seen across all the tables. This indicates that the size of the memory fits the working set of the trace.

Generally increasing the memory events increases the number of misses and the size for which the memory fits the working set. For sixpack.trace, we can see the difference in the value for which the memory fits the working set in the tables with different working sets (10000 and 1000000). For 1000000 events that value increases from 512*4 KB (2 MB) to somewhere between 2048*4 KB(8 MB) to 4192*4 KB (16 MB). While the same is not observed in bzip.trace. Bzip.trace shows the same value 512*4 KB even for 1000000 events. This indicates that Bzip.trace has a smaller overall working set than sixpack.trace. Or in other words, bzip.trace loads a lesser number of different pages than sixpack.trace. Thus, for bzip.trace the memory requirements come down to only 2 MB of the physical memory regardless of the number of events. While for sixpack.trace that value can increase from 16 MB to more if more events were considered. These days, computers are generally equipped with a physical memory of size 4 GB – 16 GB (and even more). Processes will run ideally until their working set (or ideal size of allocated physical) is less than the actual memory. The larger the working set, the more inefficiently it runs.

Coming back to our discussion of the algorithms, in almost all the cases LRU performs the best, FIFO the worst, and segmented FIFO lies in between. Here LRU performs better, as in the traces a certain number of pages are used repeatedly during a certain number of events. For example, accessing an array in different ways in a program keeps on accessing the same set of pages multiple times. Thus, LRU performs better since it keeps the recently used pages and removes the least recently used page decreasing the number of misses. While for FIFO, if some pages of the array were accessed earlier than other operations on the array, there would be a lot more page misses than LRU. The Segmented-FIFO works closely to LRU if the percentage of the secondary buffer is closer to 100 % and more closely to FIFO if the percentage is closer to 0 %. Since both the traces here favors LRU, Segmented-FIFO for 60 % works better than Segmented-FIFO for 40%, 30%, and 20%.

Conclusions:

From the experiments performed here, we can say that LRU works better overall. Here an argument can be made that this might be only true for the traces used in the experiment. But as seen in the previous section, LRU is favored because of the fact it deals better with repeatedly used pages. This happens a lot when accessing various data structures. Thus, in general, LRU will work better most of the time. Also, it is a common programming practice to declare and initialize data structures earlier than the actual usage (accessing) of the data structure. This leads to more page misses in FIFO. Also, FIFO doesn't work well if we are constantly switching multiple data structures for the same reasons as above. Here Segmented-FIFO provides a solution between FIFO and LRU. Since we don't know a lot about the memory traces beforehand, Segmented-FIFO can provide a better probability of working better with traces that might favor either FIFO or LRU. Another important outcome of the experiment was the realization that after a certain point increasing memory size doesn't increase performance. This point is achieved when the memory fits the working set, i.e. it is closer to the size of different pages accessed by the memory trace.

(Approximate time spent together working on the experiment: 25 hours)