Ahmed Ghoneim
USF ID: U91248135

Project 2 Report

Introduction:

The operations of paging and page replacement algorithms play a vital role in enabling the Operating System (OS) to retrieve and process data as per the user's requirements. Initially, the system employs paging, a method used to store data, enabling the OS to fetch the requested information or processes from secondary to primary memory. When a page fault occurs, i.e., the requested page isn't in the main memory, and there isn't sufficient space to allocate the requested page, the system triggers a page replacement algorithm. This algorithm determines which existing page should be replaced. There are various such algorithms, including First-In-First-Out (FIFO) and Least-Recently-Used (LRU). These mechanisms are built into this C++ program, along with structs that represent trace/page entries for every vector element, holding data read from the trace file. The user is required to provide the trace file's name, number of frames, selected algorithm, and desired execution mode upon running the program. After execution, the program will output the number of events, disk reads, and disk writes that occurred.

Methods:

We conducted several tests to gain insights into the workings of these page replacement algorithms. By varying the number of frames and using different trace files, we can understand how changes in the system's allocated memory influence the algorithms' outcomes. The frame numbers tested ranged from 2^0 (1) to 2^9 (512), which helps us understand the number of unique page addresses in each trace file and the best-suited page replacement algorithm for different situations. For each algorithm, the hit ratio was calculated and plotted against the number of frames used. The hit ratio calculation involved the division of cache misses by the total number of requests (hits and misses), subtracted by 1.

Hit Ratio = 1 - (Number of cache misses / Total number of requests)

In our tests, "Read" represents the number of cache misses, counting the page faults that occurred for a given number of frames and trace count. The total number of requests corresponds to the total traces read and processed by the algorithm. The "Write" variable, an iterator, keeps track of the number of page replacements or ejections happening within the program. Evaluating these variables helps us determine the most effective page replacement algorithm.

Results:

Bzip.trace - FIFO:

Table 1 summarizes the results obtained from running various tests using the FIFO page replacement algorithm and the "bzip.trace" file. The values of 'Read,' 'Write,' and the computed hit ratio for each test are reported.

Table 1 reveals an interesting pattern: As the number of frames increases, the hit ratio shoots up, leading to a drop in the read and write count. Starting from a hit ratio of 37.02% with a single frame, there's a quick increase that starts to flatten when the number of frames hits 16, where the hit ratio is around 99.62%. After this point, adding more frames causes a minor change in the hit ratio and read/write count, while increasing the program's runtime cost.

When the number of frames was set to 512, 'Read' fell to 317, and 'Write' remained at 0, indicating 317 unique page addresses in the 'bzip.trace' file. Since there were more than enough frames, no replacements were needed, leading to a significant increase in runtime.

$$Hit\ Ratio\ -\ 1 - (Num\ of\ cache\ misses / Tot\ num\ of\ requests)$$

nFrames	Read	Write	Hit Ratio
1 (2^0)	629737	118475	0.370263
2 (2^1)	228838	54321	0.771162
4 (2^2)	128601	38644	0.871399
8 (2^3)	47828	18797	0.952172
16 (2^4)	3820	1335	0.99618
32 (2^5)	2497	851	0.997503
64 (2^6)	1467	514	0.998533
128 (2^7)	891	305	0.999109
256 (2^8)	511	125	0.999489
512 (2^9)	317	0	0.999683

Table 1: Data collected on FIFO and 'bzip.trace'

LRU:

Table 2 offers insights into the performance of the Least Recently Used (LRU) algorithm. These metrics were obtained from the same set of experiments previously conducted on the FIFO algorithm. The 'Read,' 'Write,' and 'Hit Ratio' values vary based on the user-defined nFrames.

nFrames	**Read**	**Write**	**Hit Ratio**
1 (2^0)	629737	118475	0.370263
2 (2^1)	154429	44024	0.845571
4 (2^2)	92770	35650	0.90723

8 (2^3)	30691	11092	0.969309
16 (2^4)	3344	1069	0.996656
32 (2^5)	2133	702	0.997867
64 (2^6)	1264	420	0.998736
128 (2^7)	771	224	0.999229
256 (2^8)	397	48	0.999603
512 (2^9)	317	0	0.999683

Table 2: Data collected on LRU and ‘bzip.trace’

Referencing Table 2, the LRU algorithm's behavior closely mirrors that of the FIFO algorithm. Starting at 37.02% with a single frame, the hit rate ascends exponentially to 99.67% with 16 frames, a rate slightly above the FIFO algorithm. However, like FIFO, the values of 'Read,' 'Write,' and the hit ratio exhibit negligible variations with the augmentation of available frames. When the frame count is 1, the 'Read' and 'Write' values are identical for both algorithms. However, the LRU values are significantly smaller than those of FIFO with larger frame counts. This discrepancy allows the LRU hit ratio to slightly surpass FIFO. This variance is attributable to the LRU algorithm's selection process for ejection: the algorithm continues to retain recently utilized pages, discarding the least recently used frame.

Sixpack.trace

FIFO:

Table 3 portrays the data collected from testing the FIFO algorithm with the 'sixpack.trace' file, using various frame counts to ascertain 'Read' and 'Write' values and calculate the hit ratio.

nFrames	**Read**	**Write**	**Hit Ratio**
1	792379	159542	0.207621
2	529237	136484	0.470763
4	351810	94710	0.64819
8	230168	57121	0.769832
16	140083	31314	0.859917
32	85283	18805	0.914717
64	48301	11936	0.951699

128	27778	8346	0.972222
256	15440	5426	0.98456
512	8089	3353	0.991911

Table 3: Data collected on FIFO and 'sixpack.trace'

The hit ratio depicted in Table 3 climbs slowly and steadily compared to the FIFO results for 'bzip.trace'. With one frame, the hit ratio is initialized at a significantly lower value of 20.76%. As the frame count elevates, so does the hit ratio, showing diminishing returns past 256 frames, at a hit ratio of 98.46%. The subsequent test with 512 frames shows a hit ratio of 99.19%, signifying the hit ratio's plateauing effect. Within this series of tests, there isn't a single instance where the 'Write' value is zero, and 'Read' equals the number of page addresses in the file. However, upon closer examination, these values can be attained with 4096 frames, albeit at a significant runtime cost. The static number of page addresses in the 'sixpack.trace' file is 3890. Therefore, this trace file leans more heavily on the page replacement algorithm than the 'bzip.trace' file.

LRU:

Table 4 encapsulates data from the same set of experiments previously conducted on the FIFO algorithm but applied to the LRU algorithm. The 'Read,' 'Write,' and calculated hit ratio values are presented based on the user-specified nFrames, showcasing the algorithm's performance with the 'sixpack.trace' file.

nFrames	**Read**	**Write**	**Hit Ratio**
1	792379	159542	0.207621
2	483161	135857	0.516839
4	282620	71260	0.71738
8	176496	32717	0.823504
16	108682	19342	0.891318
32	67747	13730	0.932253
64	41186	9672	0.958814
128	21090	6526	0.97891
256	11240	4092	0.98876
512	5823	2444	0.994177

Ahmed Ghoneim
USF ID: U91248135

Table 4: Data collected on LRU and 'sixpack.trace'

We also conducted tests on the Least Recently Used (LRU) algorithm using the same frame counts as in the FIFO tests, with the 'sixpack.trace' file. The results were captured in Table 4, which, like before, provides the values for the 'Read' and 'Write' variables based on the frame counts used, as well as the computed hit ratio.

LRU Hit Ratio Vs. Number of Frames for 'sixpack.trace':

Analyzing Table 4, we can see that the LRU algorithm's behavior is similar to that of the FIFO algorithm. The initial hit rate with one frame is 20.76%, which gradually escalates to 98.89% with 256 frames. While the increase plateaus with increasing frame counts, the LRU hit rates are slightly higher than those of FIFO. Although the 'Read' and 'Write' values are equal for both FIFO and LRU when the frame count is 1, the LRU values are notably smaller than those of FIFO at larger frame counts. This factor allows the LRU hit ratio to be marginally higher than FIFO's. These outcomes are analogous to those observed with the 'bzip.trace' file. Considering these comparisons and findings, we recommend using the LRU page replacement algorithm for this virtual memory simulation.

Conclusion:

The choice of the most suitable page replacement algorithm for a particular trace file is crucial for enhancing the program's performance and minimizing runtime costs. Through our simulation, we've acquired an in-depth understanding of two key page replacement algorithms, FIFO and LRU. We conducted tests with each algorithm using ten different frame counts and two separate trace files.

These tests revealed that the assigned memory size significantly impacts memory performance, regardless of the algorithm used. As the number of frames increases, the 'Read' and 'Write' counts decrease, thereby increasing the hit rate of memory access.

Our tests spanned from 2^0 (1) to 2^9 (512) frame counts, allowing us to study the process results under three different scenarios: insufficient, adequate, and surplus physical memory.

Ahmed Ghoneim
USF ID: U91248135

When physical memory was insufficient, we observed higher 'Read' and 'Write' values, indicating more page faults and ejections. This circumstance resulted in a lower hit rate. Conversely, surplus physical memory led to reduced 'Read' and 'Write' values, with the number of page faults equaling the page addresses present in the trace file, and no page ejections. This situation resulted in a higher hit rate. However, having the right amount of physical memory caused the values to stabilize, indicating a balance between performance and cost.

Our findings indicate that the LRU algorithm tends to produce slightly higher hit rates than the FIFO algorithm. Additionally, the optimal number of frames to achieve a plateaued hit rate varies between different trace files, indicating that the required memory size depends on the specific trace file used.

In summary, our tests helped us understand each algorithm's behavior and effectiveness, allowing us to determine the best algorithm for different situations and memory traces.