COMP SCI 3004/7064 Operating Systems Practical 2

Virtual Memory Simulation

Prac2 Group 12
Gia Bao Hoang - a1814824
Nghia Hoang - a1814303
Hoang Nam Trinh - a1807377

1. Introduction

In computer systems, efficient memory management is crucial, with page replacement at its core. As processes run, not all data can fit in physical memory. When data is missing, causing a page fault, the system must decide which page to replace. This decision is guided by the page replacement algorithm. The right algorithm can significantly boost performance, while the wrong choice can lead to thrashing—where the system excessively swaps pages, slowing processes.
Various algorithms aim to optimize this decision. This study evaluates four algorithms: RAND, FIFO, LRU, and Clock, using a simulator. We seek to understand each traced program's memory needs, determine the best algorithm for constrained memory, and see if one consistently excels. Our goal is to deepen our understanding of memory trace intricacies and page replacement strategies.

2. Methods

2.1 Simulator Description:

The core tool utilized in this study is `memsim.c`, a C program crafted to simulate the performance of various page replacement algorithms pivotal for virtual memory management. To deploy `memsim.c`, it first undergoes compilation, post which it is executed from the terminal. The execution requires four specific inputs:

1. `*inputfile*`: This represents the memory traces we intend to analyze.
2. `*numberframes*`: This parameter signifies the space of memory sizes.
3. `*replacementmode*`: This encompasses the four algorithms under scrutiny - rand (random), lru (least recently used), fifo (first in first out), and clock.
4. `*debugmode*` (quiet/debug): A mode to assist in debugging the simulator.

2.2 Memory Traces Explanation:

The study revolves around four distinct memory traces: swim, bzip, gcc, and sixpack. These traces are authentic recordings of programs in execution, sourced from the renowned SPEC benchmarks. Each trace encapsulates one million memory accesses, specifically extracted from the commencement of each program. Structurally, every trace is a sequence of lines, with each line delineating a hexadecimal memory address followed by either an 'R' (indicating a read) or a 'W' (indicating a write).

2.3 Experimental Design:

To ensure a holistic and in-depth understanding of the performance of the replacement algorithms, our experimental design is meticulously structured. Here's a detailed breakdown:

2.3.1 Memory Size Range:

Each memory trace is tested across all replacement algorithms, starting from a single memory frame and expanding until disk read rates near zero. The exact memory range varies per trace, ensuring adaptability and efficiency insights.

2.3.2 Data Collection:

We measure the Disk Read Rate, defined as the ratio of disk reads to events. Disk reads denote the frequency of data retrieval from the disk to memory. This metric is visualized graphically for a side-by-side algorithm comparison and anomaly detection.

2.3.3 Graphical Representation:

For each memory trace, we produce graphs with:

- X-Axis: Number of frames.
- Y-Axis: Either $1-y$ or $1-\log(y)$, where y is the Disk Read Rate

The $1-y$ metric showcases the hit rate, indicating memory management efficiency. Using $1-\log(y)$ aids in:

- Comparative Analysis: Enhancing clarity when contrasting replacement algorithms.
- Rate of Change: Highlighting how the hit rate varies with increasing frames, offering insights into an algorithm's adaptability.

2.4 Rationale Behind Experiments:

Our experimental design aims to deeply understand replacement algorithm performance across varied memory conditions. By analyzing from limited to ample memory, we assess each algorithm's adaptability and areas of strength or weakness. This approach also facilitates a relative comparison of their behaviors in different scenarios. We employ graphs to simplify complex data, enabling clearer trend recognition, comparative analysis, and anomaly detection. This visual representation ensures insights are accessible to both experts and beginners. In essence, our design is a blend of rigorous data collection and insightful analysis, geared to influence future memory management strategies.

3. Results

In the results, before analyzing individual graphs, we define the optimal number of frames. This is the point where increasing memory size no longer leads to significant improvements in the hit rate. Beyond this, adding frames offers diminishing returns on hit rate enhancement.

3.1 Swim

In the analysis of the Swim trace, as illustrated in Figure 3.1, we observe a memory frame range extending from 0 to 800 frames. The hit rate graph reveals an exponential decay. Notably, LRU and Clock algorithms closely align in their performance, with LRU slightly edging ahead. In contrast, FIFO lags, particularly evident between the 20 to 90 frame range. Optimal hit rates for LRU, Clock, and FIFO are achieved between 80 to 90 frames, whereas Rand peaks around 120 frames. The 1-log(y) graph further accentuates LRU's superior adaptability, but an intriguing trend emerges across all algorithms: a dip around the 100-frame mark, followed by a steady ascent, a pattern even the Random algorithm adheres to. It's worth noting that the Random algorithm's performance is particularly erratic up to the 200-frame mark, post which it stabilizes considerably.

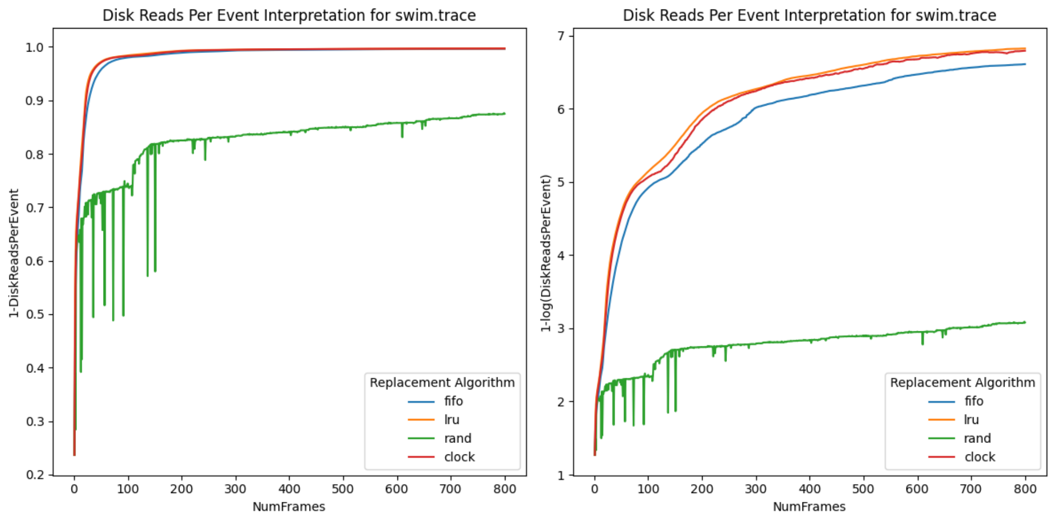

Figure 3.1: Swim Analysis: Hit Rate (left) and Log-Transformed Hit Rate (right) vs. Number of Frames

3.2 Bzip

In the Bzip trace depicted in Figure 3.2, spanning 0 to 320 frames, the hit rate graph showcases an exponential decay. By the 10-frame juncture, LRU momentarily edges ahead of Clock and FIFO, but soon all three converge near their peak efficiencies. While LRU, Clock, and FIFO reach their optimal efficiency around 15 frames, Random peaks much later, near 300 frames. Additional frames offer diminishing returns beyond these points. The log-transformed graph, with its distinct jaggedness, suggests sporadic memory access patterns, possibly due to fewer data points. LRU slightly outperforms Clock, but FIFO's adaptability is striking. Initially trailing, FIFO's rate of change aligns with LRU around 140 frames and remains competitive with both LRU and Clock nearing 300 frames. This highlights FIFO's potential, especially given its straightforward design. Notably, the Random algorithm exhibits erratic behavior, with two significant spikes around 220 and 270 frames.

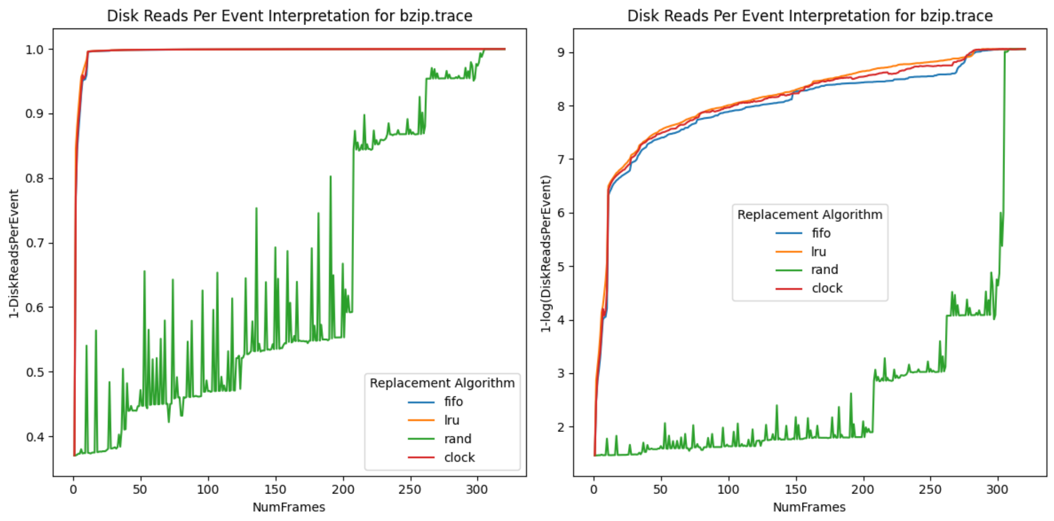

Figure 3.2: Bzip Analysis: Hit Rate (left) and Log-Transformed Hit Rate (right) vs. Number of Frames

3.3 Gcc

In the Gcc trace illustrated by Figure 3.3, spanning 0 to 800 frames, the hit rate graph displays an exponential trajectory. LRU marginally outpaces Clock, with FIFO not far behind. This rapid rise in hit rate eventually levels off, reflecting consistent algorithmic responses to this trace. Regarding peak efficiency, while most algorithms optimize between 60 to 70 frames, a gradual improvement persists up to 800 frames. However, the incremental nature of these gains implies that allocating memory beyond the initial optimal mark might not yield substantial benefits for the Gcc trace. The log-transformed perspective further accentuates LRU's swift adaptability to Gcc's memory patterns. A shared characteristic among FIFO, LRU, and Clock is the dual peaks observed around the 180 and 380 frame intervals, suggesting specific memory access patterns within the Gcc trace. The Random algorithm, though erratic, fluctuates within a more restricted range, highlighting its inherent unpredictability.

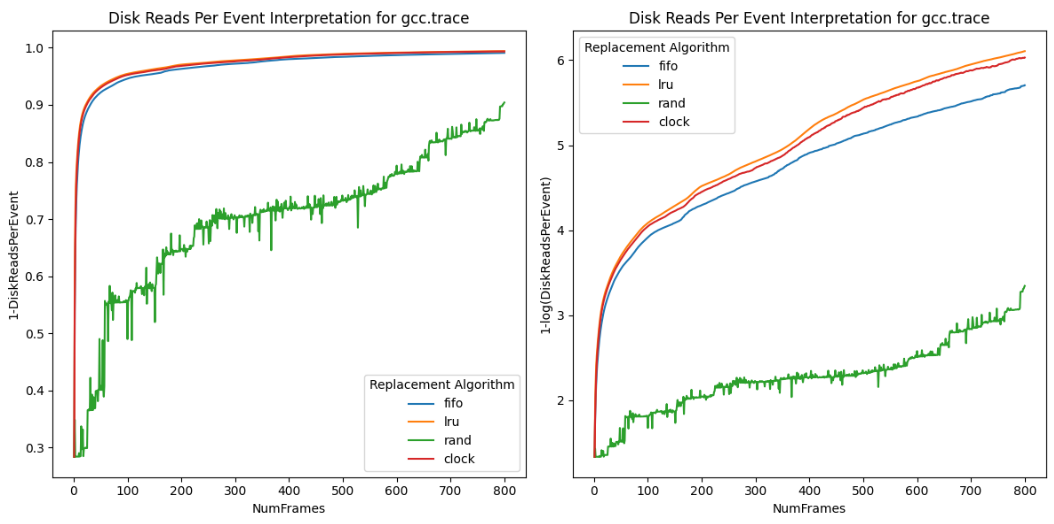

Figure 3.3: Gcc Analysis: Hit Rate (left) and Log-Transformed Hit Rate (right) vs. Number of Frames

3.4 Sixpack

In the Sixpack trace, represented in Figure 3.4 and spanning 0 to 800 frames, the hit rate graph follows an exponential curve. LRU slightly outperforms Clock, with FIFO a tad behind. Notably, both LRU and Clock exhibit a minor dip around the 60-frame threshold, coinciding with FIFO's performance at its optimal point. Excluding the Random algorithm, the majority achieve their best efficiency between 70 to 80 frames. However, a gentle increase continues up to 800 frames, indicating potential advantages, though modest, of extended memory allocation for the Sixpack trace. The log-transformed view reveals that LRU and Clock's rate of change closely align, both marked by periodic bumps, hinting at intermittent memory access shifts in the Sixpack trace. In contrast, FIFO's adaptability lags in this perspective. The Random algorithm, while unpredictable, maintains a consistent noise-like fluctuation throughout.

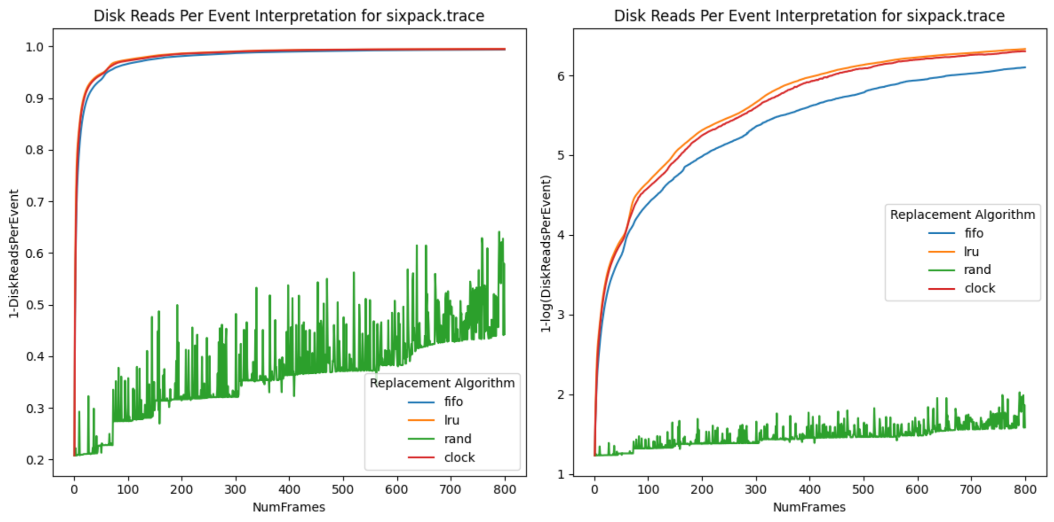

Figure 3.4: SixpackAnalysis: Hit Rate (left) and Log-Transformed Hit Rate (right) vs. Number of Frames

4. Conclusions

4.1 Summary of Key Findings:

Across the four application traces analyzed, Bzip stands out as the most straightforward, with all traces predominantly exhibiting an exponential decay pattern. Notably, LRU and Clock consistently display analogous patterns across all traces, with FIFO's trend also bearing resemblance, particularly in the Gcc trace.

4.2 Performance Analysis:

LRU consistently outperforms the other algorithms across all traces, closely followed by Clock and then FIFO. The marginal performance difference between LRU and Clock is intriguing, given that both algorithms pivot on the principle of replacing the least accessed frames. This underscores the efficacy of this approach. FIFO, despite its simplicity—operating on a first-in-first-out queue structure—delivers commendable performance. In fact, its rate of increase aligns with LRU and Clock for Bzip, suggesting FIFO's viability for simpler programs.

4.3 Insights on Algorithm Behavior:

The random replacement algorithm serves as a testament to inherent unpredictability. Instead of a stable 50% hit rate, its performance is capricious. However, its true value lies in benchmarking; any algorithm underperforming compared to random is deemed inefficient.

4.4 Implications for Real-world Applications:

For versatile and high-performing memory management, LRU and Clock emerge as top contenders. While LRU slightly edges out Clock, a deeper dive is required to ascertain specific scenarios where Clock might be preferable. Their shared underlying concept of replacing the least accessed memory frame underscores their effectiveness. On the other hand, FIFO's simplicity and effectiveness make it a viable choice for simpler programs.

4.5 Recommendations for Future Endeavors:

While LRU and Clock are recommended for their adaptability and performance, FIFO's simplicity makes it a compelling choice for less complex programs. Random algorithm, despite its erratic behavior, serves as a valuable benchmark. Any algorithm underperforming compared to random should be reconsidered. Further studies should delve into specific scenarios to determine the precise conditions under which Clock might be favored over LRU.