Operating Systems

Practical 2 Report

Virtual Memory Simulation

a1904344 Xiaoqing Zhao

a1903270 Feinan Guo

a1893592 Dang Quy Duong

Introduction

Page replacement algorithms are critical in virtual memory management, as they determine which memory pages should be replaced when new ones need to be loaded into physical memory. When the system runs out of available memory, the operating system must choose a page to replace, which can impact performance if not handled effectively. An optimal page replacement algorithm reduces page faults—instances where a required page is not in memory and must be fetched from the disk. High page fault rates can significantly degrade performance due to slow disk access times.
Efficient algorithms like **Least Recently Used (LRU)** reduce page faults by retaining frequently accessed pages. However, tracking page usage can introduce additional overhead. **Random Replacement (Rand)**, while simple, often performs poorly as it may randomly replace frequently used pages. **Clock**, a practical compromise between complexity and performance, uses a circular queue with reference bits to approximate LRU behaviour, providing reasonable performance with lower overhead. Each algorithm's efficiency depends on how well it minimizes page faults while balancing system complexity.

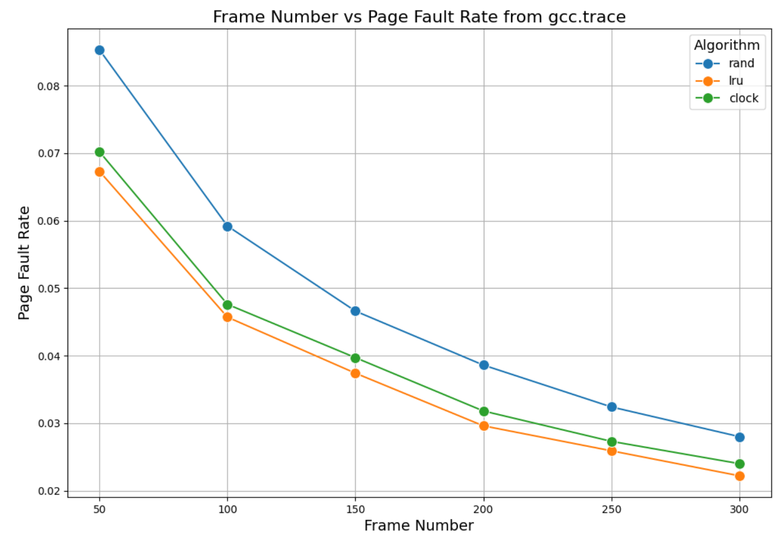
Methods

A virtual memory simulator was developed during the previous phase of the assignment to evaluate different page replacement algorithms. This simulator generates essential performance metrics, including page fault rates, disk reads, and disk writes, for various memory management scenarios. By simulating the behaviour of these algorithms under different conditions, we can compare their effectiveness in handling memory access.
To automate the simulation process, a script was developed to run the simulator across a range of frame numbers, from 1 to 300, for each of the provided trace files. These traces represent typical memory access patterns in common workloads. The script executed the simulator for each trace file and recorded the key metrics (page fault rate, disk reads, disk writes) for further analysis. The resulting raw data was used to evaluate and compare the performance of the different algorithms.
The trace files provided for the simulation were not selected by us but were predetermined as part of the assignment. Each trace represents common memory access patterns that occur in typical workloads, such as those from processes like gcc, swim, bzip, and sixpack. These traces help simulate realistic operating conditions for evaluating the page replacement algorithms, offering insights into how they perform under different types of memory access patterns.
The key metrics used to assess the performance of the page replacement algorithms were predefined as disk reads, disk writes, and page faults. These metrics are critical in understanding how efficiently each algorithm manages memory. Disk reads and writes indicate how frequently pages are swapped between memory and disk, while page faults represent instances where the required memory page is not found in the available frames. The analysis of these metrics allows for a comprehensive evaluation of the efficiency of each algorithm.

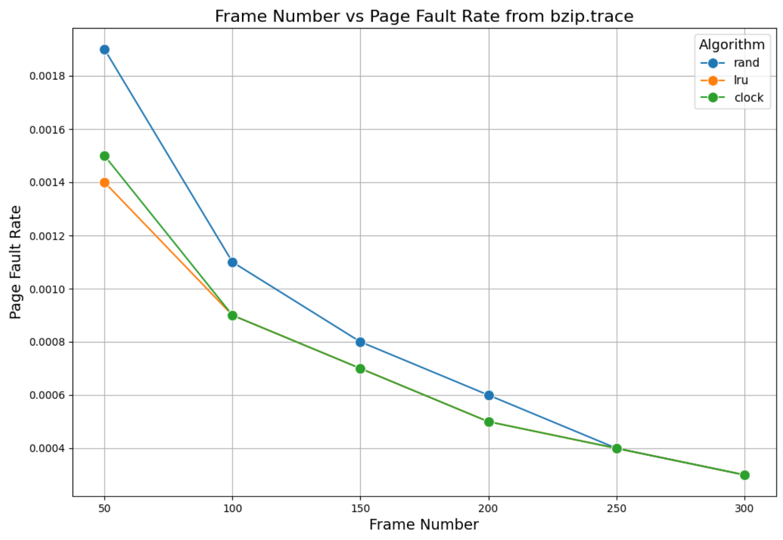
Results

For the **bzip.trace,** the rand algorithm starts with a high page fault rate of 0.0018 at 50 frames, which drops to 0.0007 at 150 frames and 0.0004 at 300 frames. However, LRU and Clock perform significantly better. LRU begins at 0.0014 and falls to 0.0008 by 150 frames and 0.0002 at 300 frames, while Clock starts at

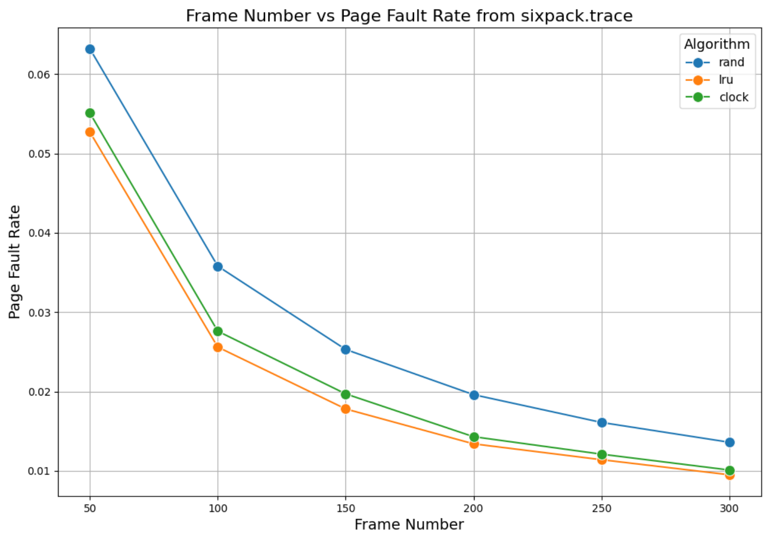
0.0013, decreases to 0.0009, and reaches 0.0003 by 300 frames. LRU's tracking of recently used pages enables more efficient replacements, making it more reliable, especially with fewer frames.



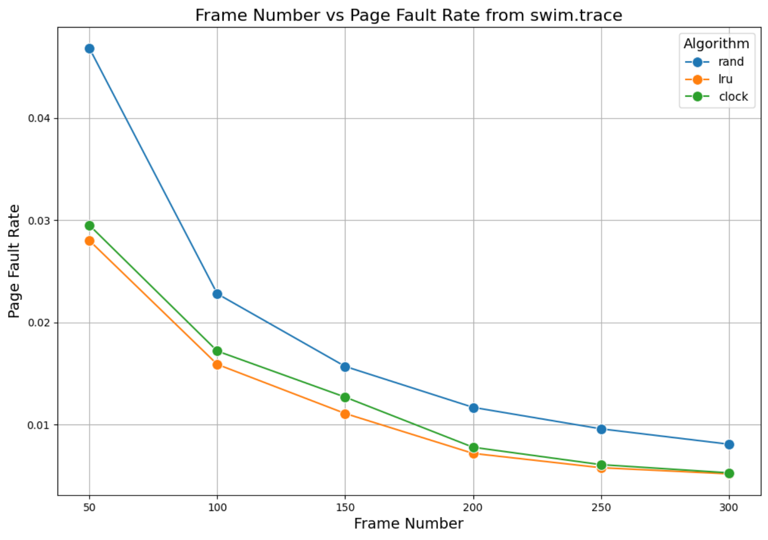
In the **gcc.trace**, (picture below) the differences between the algorithms become more pronounced. The rand algorithm again shows the highest page fault rate, especially with fewer frames, indicating that its random nature leads to suboptimal page replacements when memory is constrained. LRU and Clock, on the other hand, both see a steady decline in page fault rates as the number of frames increases. LRU consistently performs better than Clock, suggesting that the gcc trace benefits more from the precise tracking of page usage provided by LRU. The performance gap between rand and the other two algorithms is especially evident in this trace, highlighting the inefficiency of random page replacement strategies in more complex or memory-intensive workloads. As memory availability increases, all algorithms improve, but LRU remains the best choice for minimizing page faults.



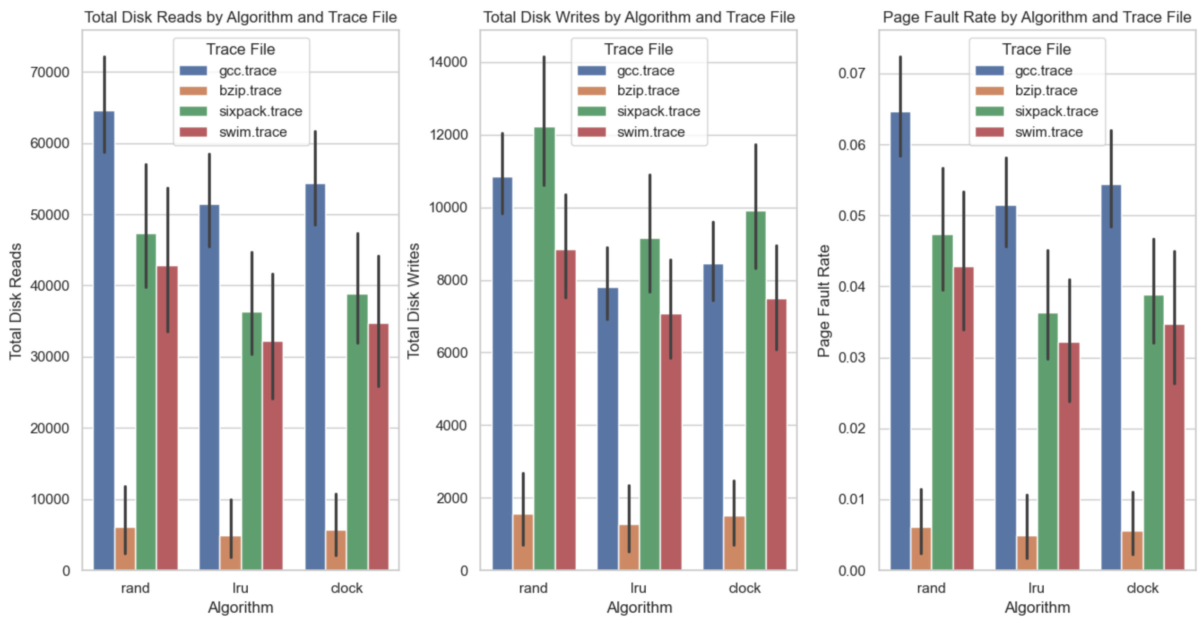
For the **sixpack.trace** (picture below), the overall trend remains similar, with rand displaying the highest page fault rates across the board. As the number of frames increases, both LRU and Clock significantly reduce their page fault rates. However, in this trace, the performance of LRU and Clock is nearly identical, with LRU maintaining a slight advantage. This indicates that for certain types of workloads, Clock can approximate the performance of LRU without the need for the more complex tracking mechanisms. Nonetheless, rand continues to perform poorly, further solidifying its reputation as the least efficient algorithm among the three in terms of reducing page faults.



The **swim.trace** presents a similar scenario, where the rand algorithm starts with an extremely high page fault rate when the number of frames is limited. Although the fault rate decreases as more frames are added,



it never reaches the efficiency of LRU or Clock. LRU again shows the lowest page fault rate, with Clock closely following. In this trace, both LRU and Clock manage to keep the page fault rate low, even with fewer frames, which demonstrates their effectiveness in handling memory-limited environments. As the number of frames increases, the performance gap between LRU and Clock narrows, suggesting that for larger memory capacities, either algorithm could be a suitable choice, though LRU remains slightly more efficient.



The provided image contains three bar charts that compare the performance of three-page replacement algorithms—Random (rand), Least Recently Used (LRU), and Clock—across four different memory traces: gcc, bzip, sixpack, and swim. These charts highlight the total disk reads, total disk writes, and page fault rates for each algorithm under different trace file conditions.

The first chart illustrates total disk reads for each algorithm. It is evident that the rand algorithm consistently produces the highest number of disk reads across all trace files, particularly in the gcc.trace. This high number of reads indicates that the rand algorithm is inefficient in managing memory, frequently causing page faults that require data to be loaded from the disk. In contrast, both LRU and Clock algorithms perform significantly better, with LRU slightly outperforming Clock in most cases. The bzip.trace results in the fewest disk reads, especially for LRU, showcasing its ability to minimize page faults and manage memory efficiently.

The second chart focuses on total disk writes. Similar to the disk reads, rand exhibits the highest number of disk writes, particularly in the gcc and sixpack traces. A high number of disk writes indicates that the algorithm frequently replaces pages that are still in active use, leading to inefficiencies. LRU demonstrates the lowest number of disk writes across all traces, indicating that it is the most efficient algorithm in reducing unnecessary disk operations. Clock follows closely behind, although it generally has a slightly higher number of disk writes compared to LRU.

The third chart compares the page fault rate for each algorithm. Rand consistently shows the highest page fault rates across all trace files, again highlighting its inefficiency. LRU outperforms both rand and Clock, maintaining the lowest page fault rates in all traces. Clock performs reasonably well but tends to have slightly higher page fault rates than LRU, particularly in the gcc and sixpack traces. The bzip.trace results in the lowest page fault rates across all algorithms, further demonstrating that certain workloads can benefit significantly from more sophisticated algorithms like LRU and Clock.

Conclusion

The results confirm that LRU consistently outperforms both Rand and Clock in minimizing disk reads, writes, and page faults across all traces. This makes LRU the most efficient page replacement algorithm, particularly in memory-intensive environments. However, Clock offers a reasonable trade-off between performance and overhead, approximating LRU's effectiveness in some scenarios with simpler implementation requirements. For workloads where system overhead is a concern, Clock may be more favourable due to its lower complexity, despite LRU's better overall performance.
Rand on the other hand, proves inefficient, especially under complex workloads like gcc andsixpack, where its random page replacement strategy leads to frequent page faults and unnecessary disk operations. Nonetheless, specific scenarios where memory access patterns are less demanding (such as bzip.trace) show that Rand can perform adequately with sufficient frame numbers. Future work could explore optimizing Clock further, potentially integrating elements of LRU to improve its performance without significantly increasing complexity.