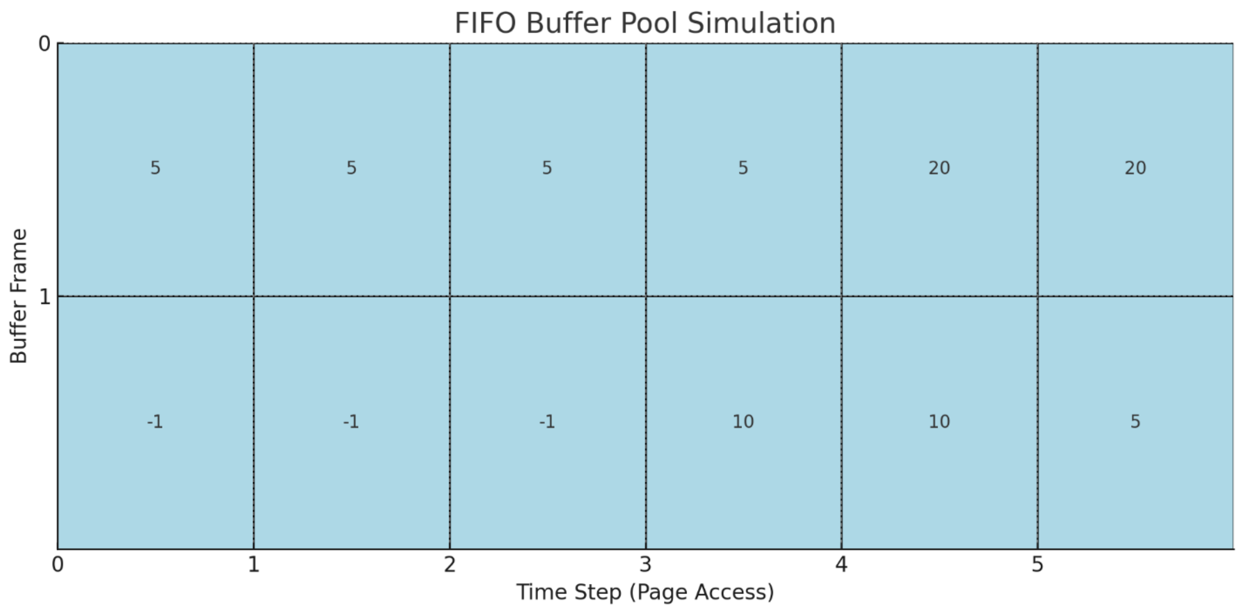
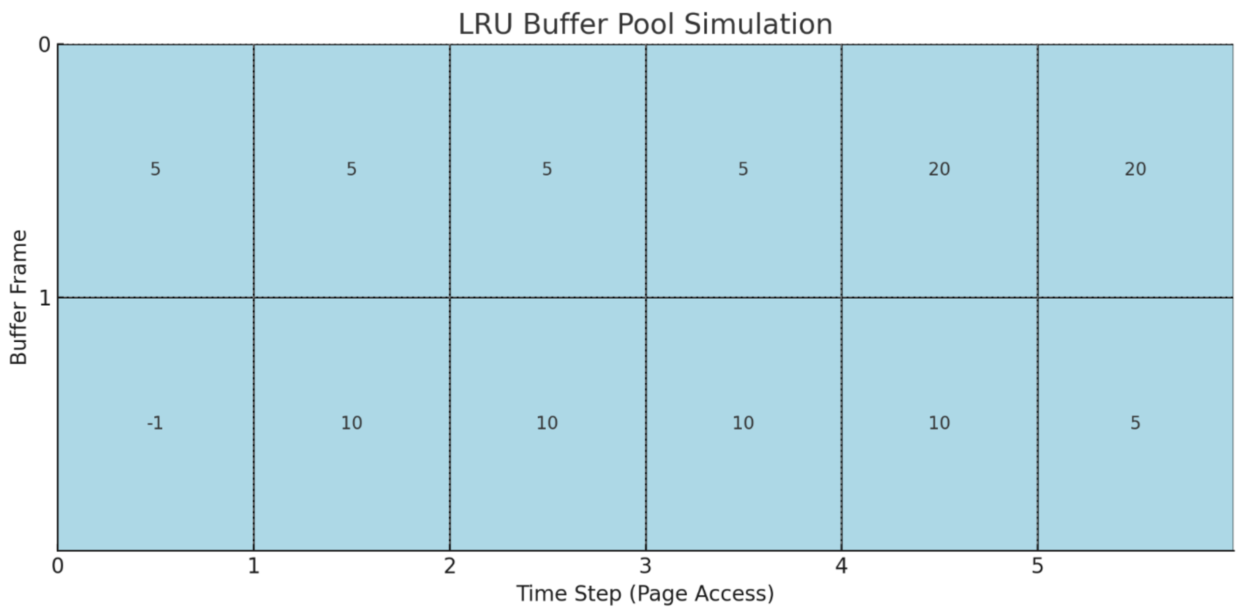
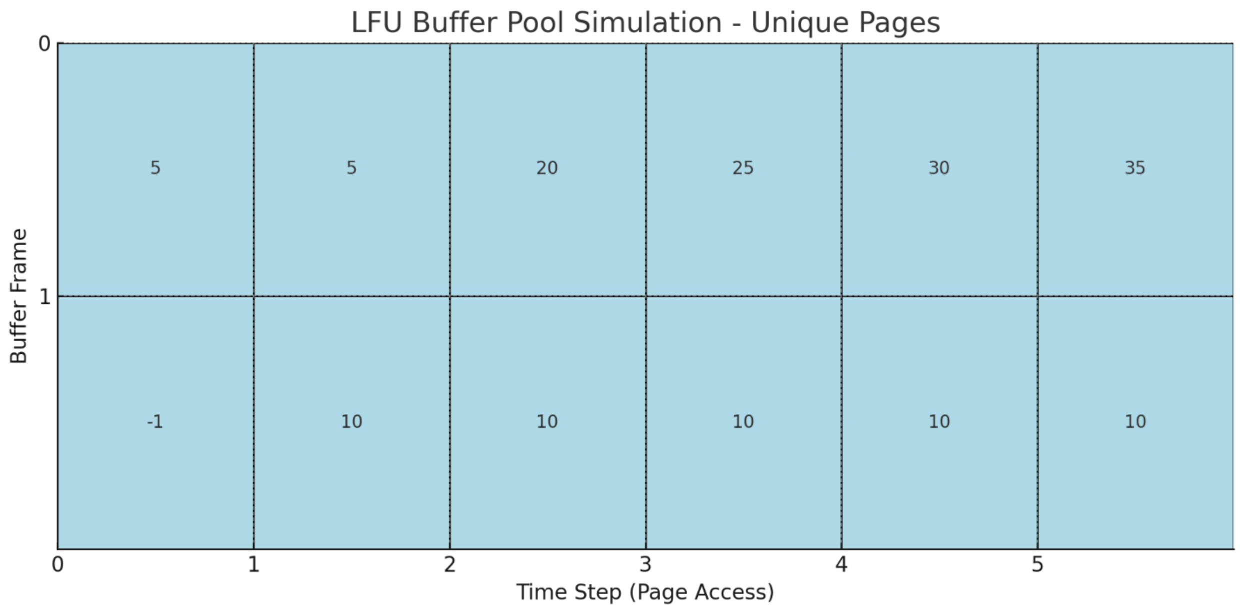
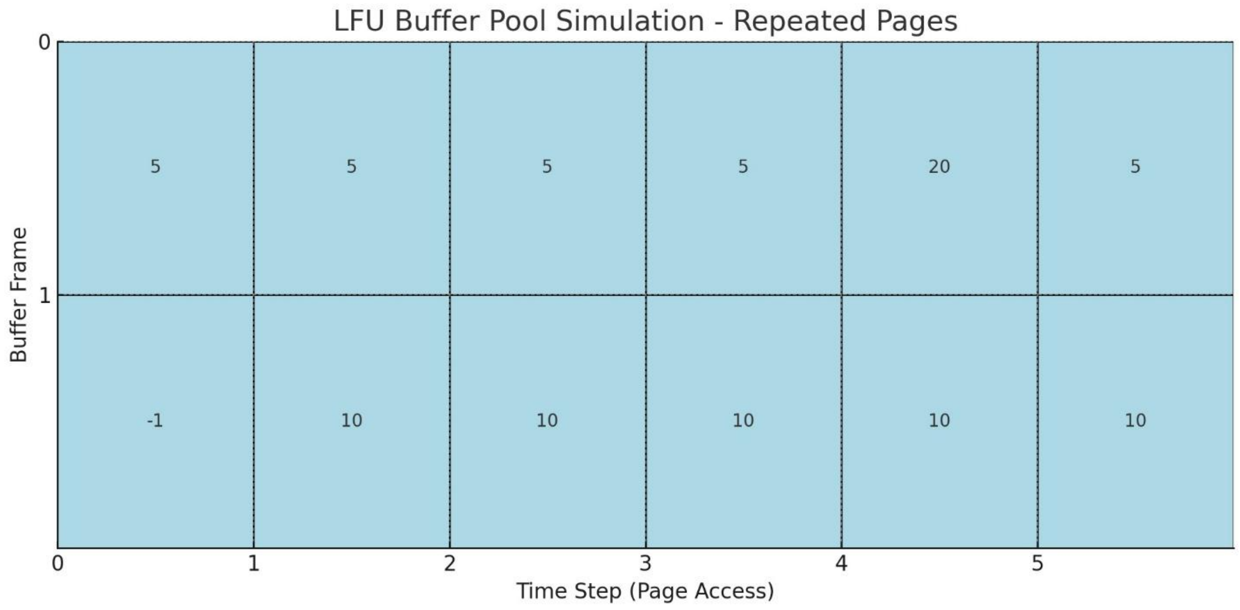

FIFO Buffer Pool Simulation
Buffer Frame
0
1
5 5 5 5 20 20
-1 -1 -1 10 10 5
0 1 2 3 4 5
Time Step (Page Access)


LRU Buffer Pool Simulation
Buffer Frame
0
1
5 5 5 5 20 20
-1 10 10 10 10 5
0 1 2 3 4 5
Time Step (Page Access)





In this analysis, we observe how buffer pool algorithms manage memory using three heuristics—FIFO, LRU, and LFU. Each algorithm was tested with specific inputs using a visual simulation akin to what Jeliot would produce, offering clear insights into the behavior of buffer frames during page access. These simulations help explain the core mechanics behind each heuristic and provide the basis to assess their efficiency in real-world scenarios.

For the FIFO (First-In, First-Out) algorithm, the simulation reveals a simple mechanism: the buffer replaces the oldest page whenever a new one needs to be loaded, and all buffer slots are full. With only two available buffers and six pages (5, 5, 5, 10, 20, 5), FIFO initially handles the first few pages without issue. However, when new pages such as 10 and 20 are introduced, previously loaded pages like 5 are removed despite being accessed frequently afterward. As a result, the final appearance of 5 causes a page fault, even though it had been loaded multiple times before. This behavior highlights FIFO's major weakness: it disregards frequency and recency of access, treating all pages the same once loaded. This leads to inefficiency, especially in cases where a frequently accessed page is removed prematurely (Marsic, 2012).

Contrastingly, the LRU (Least Recently Used) heuristic offers a more nuanced approach by prioritizing recently accessed pages. When running LRU with inputs 5, 10, 5, 10, 20, and 5, the system maintains a stack-like structure to keep track of page usage order. Initially, the buffer loads 5 and 10. When 5 appears again, it is simply moved to the top of the usage stack, ensuring it won't be evicted soon. As new pages arrive, the least recently accessed one—whichever is at the bottom of the stacks, is removed. This mechanism ensures that if a page like 5 is reused often, it remains in the buffer, resulting in fewer page faults and improved buffer efficiency. The LRU algorithm thrives in environments where temporal locality is present—that is, pages recently accessed are likely to be accessed again soon. This makes it a superior choice for programs with predictable access patterns, such as nested loops or frequent variable reuse (Marsic, 2012).

The LFU (Least Frequently Used) algorithm offers a different perspective by maintaining a count of how often each page has been accessed. In the first LFU test (with repeated pages: 5, 10, 5, 10, 20, 5), the buffer efficiently retains pages that are used more frequently. As 5 and 10 are reused, their access counts increase, giving them a better chance of staying in memory. When a new page like 20 arrives and must replace another, LFU targets the one with the lowest frequency count. This allows the buffer to adapt intelligently over time to actual usage trends, giving preference to high-traffic pages. The simulation reflects this behavior as 5, which appears three times, is prioritized over 20 and similar less-used pages. This approach is especially useful in databases or systems where certain items are accessed disproportionately more than others.

However, LFU's strength becomes its weakness in scenarios with no repeated values. The second LFU test (5, 10, 20, 25, 30, 35) shows how LFU degenerates when all pages have equal usage frequency—each used once. In such cases, the algorithm fails to distinguish between more and less important pages and effectively behaves like FIFO. Every new page causes a replacement, with little long-term memory retained, resulting in frequent page faults. This outcome underlines a critical limitation of LFU: it assumes the presence of access patterns over time. Without those patterns, its complexity does not yield better performance compared to simpler algorithms.

These simulations clarify the contextual effectiveness of each algorithm. FIFO works best in sequential access scenarios where older pages are unlikely to be reused. LRU, which accounts for recent usage, adapts well to localized access patterns and loops, retaining pages that matter in the short term. LFU is ideal for long-running processes with skewed access distributions, where high-frequency pages dominate and must be preserved. Yet, in random or one-off access environments, it may not perform better than FIFO or LRU.

In summary, while all three buffer replacement algorithms have their use cases, LRU and LFU generally offer better efficiency in environments with some predictability in page access. FIFO's simplicity makes it easy to implement, but its lack of adaptability limits its effectiveness.

Visualization of these heuristics, as shown in the simulation outputs, brings to light their practical strengths and limitations, helping system designers make informed decisions when optimizing memory usage.

References:

Marsic, I. (2012, September 10). *Software Engineering*. Rutgers: The State University of New Jersey. Retrieved from https://my.uopeople.edu/pluginfile.php/57436/mod_book/chapter/46513/CS4403MarsicTextbook.pdf

Wordcount: 756