Measuring Performance of Page Replacement Algorithms

| Nicolas Hempe Ardila  Department of Computer Science & Engineering  University of South Florida  Tampa, FL, USA  nicolashempe@mail.usf.edu | Spencer Turkel  Department of Computer Science & Engineering  University of South Florida  Tampa, FL, USA  sturkel@mail.usf.edu |  |
| --- | --- | --- |

# ABSTRACT

In this project we will simulate the performance of page replacement algorithms in C++. These include First-In, First-Out (FIFO), Least Recently Used (LRU), Least Frequently Used (LFU), Most Frequently Used (MFU) and Optimal. Our program will compare all page replacement algorithms and measure their performance to determine which algorithms perform better or worse in multiple cases.

# INTRODUCTION

This project was designed to simulate a system that takes in a memory trace and apply the page replacement algorithms to print the steps and the page faults of the input. The simulator will show the action taken by the virtual memory to evaluate the memory trace and track which pages have been loaded into memory. The program will take an input from a data file, then the simulation will read a page at a time and looks to see if that page has already been loaded into memory. If the page has not been loaded, then we record the page and print a page miss. If the page has been loaded, we print a page hit. Using this data, we can evaluate the performance of each of the five algorithms used. Once the memory trace is entered, the algorithms take the trace and show the steps it takes to evaluate the pages. Then we can see the performance of the algorithms with the given trace.

# Methodology

In our project we measured the performance of five, page replacement algorithms.

## First-in, First-out (FIFO)

This algorithm holds the page table in a queue, and the input trace using a vector. First, we check if the page table is not full and if the current input has not been loaded into memory, then we load it into the empty frame. The algorithm runs a function on the current page and creates a current state of the page table. The algorithm searches through the page table and the current input to check if the page table has been loaded into memory. If we get a hit, we increment the hit counter and continue to the next page. If we don’t get a hit, we use the algorithm to remove the top page in the page table from the queue and load the current input into the queue then increment the miss counter.

## Least Recently Used (LRU)

For this algorithm we stored the current state in a set and run the first page algorithm. We check if the page table is not full and the page has not been loaded to memory and then add the page to empty frame. We then check the current input across the page table to see if the current input has already been loaded into memory. If we get a hit we increment the hit counter, but in this algorithm, we must keep track of the page table and access the least recently used frame. If we don’t get a hit, we will search the page table and compare for the least recently used with the current input and replace the page with the current input and increment the miss counter.

## Least Frequently Used (LFU)

In this algorithm we measure how frequently a page is used using a frequency where all pages are initialized to 0. We first check if the page table is not full and the current input is not in memory, then we load the page into the empty frame. When we run the algorithm, we must keep track of the frequency of the pages. Then we check if the current input that we want to add to virtual memory has already been loaded. If the current input is in the memory we get a hit and the page table remains and we increase the frequency of the current input page by one and increment the hit counter. If we don’t get a hit, we replace the page with the lowest frequency if one or more pages have the same frequency we apply another algorithm to replace the first page used in the page table with the current input and then reset the frequency of the page that was replaced and increment the miss counter.

## Most Frequently Used (MFU)

In this algorithm we must keep track of the frequency of page use. In the beginning we check if the page table is not full and the current input is not in memory then we load the page into the free frame. We first initialize the frequency to 0. We first check to see if the current input has been loaded into memory. If the current input is in main memory the page remains and we increment the hit counter and increase the page frequency. If we get a miss, then we replace the page with the highest frequency with the current input and reset the frequency of the page and increment the miss counter.

## Optimal (OPT)

This algorithm uses the index of the pages in the memory trace to decide which pages to replace. First, we check to see if the page table is not full and if the current input has not been loaded into memory, then we load that page into the empty frame. Then the algorithm searches if the current input is in memory. If it is a hit, then the page table remains, and we increase the hit counter. If we get a miss, we check the index of the other pages in the memory trace and replace the page that is not going to be used sooner than the other pages in the table then increment the miss counter.

# Results

Trivially, each algorithm has a worst case scenario where the input string has no repeated page references. Conversely, each algorithm performs optimally when the string may be considered to be the repetition of a cycle of unique page references such that the cycle has a length less than or equal to the number of entries in the page table.

Since each algorithm gives the same results at the extremes, it was difficult to create/discover patterns of reference strings that characterize the algorithms. Research was not very revealing on this matter, though perhaps the authors were not searching for the right keywords. The following results will then be related to our attempts to find meaningful tendencies of variance between the algorithms with inputs that lie between the extrema.

After testing out our algorithms with different inputs and page table sizes, we found out that in most cases First-in, First-out (FIFO) and Least Frequently Used (LFU) performed the worst compared to the other algorithms while OPT always performed the best.

FIFO and LRU share a worst-case scenario that when there are cycles of N page references, where N > the number of frames, there will be no hits.

The following output shows the performance of the page replacement algorithms with a 12-page memory trace with 4 frames. (Figure 1) In this example the worst performing algorithm was FIFO with 10 page faults while Optimal only had 6 page faults.

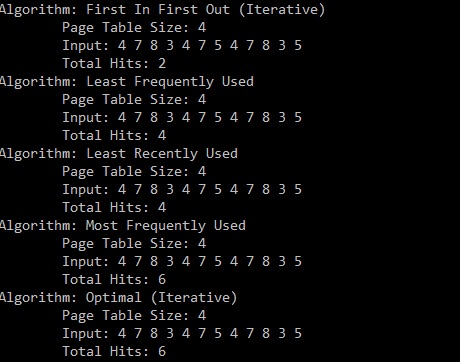


Figure : Page Replacement Algorithms with input 4 7 8 3 4 7 5 4 7 8 3 5.

When a page reference string is palindromic, the algorithm with the worst performance every time was LFU (Figure 2). Another case where LFU suffers is when a page is referenced many times and then not at all for a period afterwards, since that page will take up the frame without use for that period. In fact, although this may be due to operator error, we could not find any patterns of page reference strings where LFU performed closest to OPT.

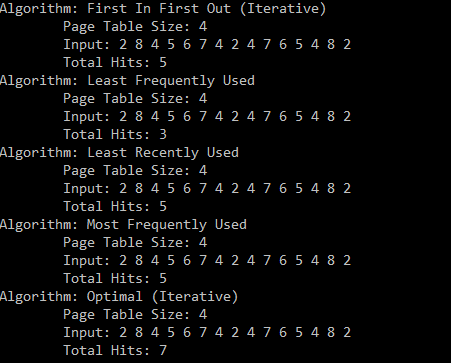


Figure : Page Replacement Algorithm with input 2 8 4 5 6 7 4 2 4 7 6 5 4 8 2. LFU with 12 page faults.

The following (Figure 3) shows the output for the First-in, First-out algorithm showing the page table size, input, total hits, and a step-by-step trace.

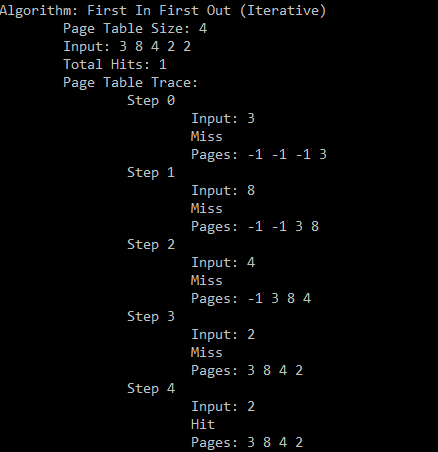


Figure : Example output for the FIFO algorithm.

# Conclusion

In this project we designed a program to simulate the performance of five page replacement algorithms and comparing their results. The algorithms we used were First-in, First-out (FIFO), Least Recently Used (LRU), Least Frequently Used (LFU), Most Frequently Used (MFU), and Optimal. We observed, given non-extreme inputs, the worst performing algorithms were FIFO and LFU, and the best performing algorithm was Optimal. The performance of the Optimal algorithm was expected, since it relies on knowledge of the entire input, which the other algorithms do not. Further work could examine why FIFO and LFU seemed to perform worst.

We attempted to create patterns in our inputs to identify situations where certain non-optimal algorithms outperform others, but these patterns were difficult to construct and verify. Further work could be done to identify more patterns.

Overall, the program we constructed was useful for visualization and experimentation, but it still requires a mathematical mind to recognize meaningful characterizations of the algorithms.