LEAST FREQUENTLY USED (LFU) CACHE REPLACEMENT ALGORITHEM

Introduction

In modern computer systems, cache memory plays a crucial role in improving performance by storing frequently accessed data closer to the processor. However, cache memory has a limited size, and it is not possible to store all requested pages at the same time. When the cache becomes full and a new page needs to be loaded, the system must decide which existing page should be removed. This decision is handled by a page replacement algorithm.

The Least Frequently Used (LFU) page replacement algorithm is based on the principle that pages accessed fewer times in the past are less likely to be accessed again in the future. Therefore, LFU removes the page with the lowest access frequency when replacement is required.

This program simulates the LFU algorithm using Python and provides a clear view of cache behavior through page hits, page faults, and hit ratio**.**

Objective of the Algorithm

The main objectives of this LFU simulation are:

- To implement the LFU page replacement strategy.
- To track how often each page is accessed.
- To calculate performance metrics such as page hits, page faults, and hit ratio.
- To help understand how LFU manages cache memory efficiently.

Data Structures Used

The algorithm uses the following data structures:

1. **Cache List**
    - Stores the pages currently present in the cache.
    - The size of this list never exceeds the cache size.

2. **Frequency List**
    - Maintains the number of times each page in the cache has been accessed.
    - Each frequency value corresponds to a page at the same index in the cache list.
3. **Counters**
    - page_hits: Counts successful page accesses.
    - page_faults: Counts unsuccessful page accesses.

Working Principle of LFU Algorithm

The LFU algorithm works by monitoring how frequently each page is accessed:

When a page is accessed:

- If the page is already present in the cache, it is a page hit, and its frequency count is increased.
- If the page is not present, it is a page fault.
- If the cache has free space, the page is inserted.
- If the cache is full, the page with the lowest frequency count is removed, and the new page is inserted.

This approach ensures that pages which are rarely used are replaced first.

Algorithm

Step 1: Initialization

- Start the program.
- Initialize an empty cache list to store pages.
- Initialize an empty frequency list to store access counts.
- Read the cache size from the user.
- Read the page reference string (sequence of page accesses).

Step 2: Initialize Counters

- Set page_hits = 0.
- Set page_faults = 0.

Step 3: Process Each Page

For each page in the reference string, perform the following steps:

Step 4: Page Hit Check

- Check whether the current page exists in the cache.
- If the page is found:
    - Identify its index in the cache.
    - Increment the corresponding frequency value.
    - Increment the page hit counter.
    - Mark the operation as a **HIT**.

Step 5: Page Miss Handling

- If the page is not found in the cache:
    - Increment the page fault counter.
    - Mark the operation as a **MISS / PAGE FAULT**.

Step 6: Cache Space Check

- If the cache is not full:
    - Insert the new page into the cache.
    - Assign a frequency value of 1.
- If the cache is full:
    - Identify the minimum frequency value.
    - Locate the page corresponding to this minimum frequency.
    - Remove the least frequently used page.
    - Insert the new page into the cache.
    - Assign a frequency value of 1.

Step 7: Display Current Status

- Display the current contents of the cache.
- Display the frequency values for each cached page.
- This step helps visualize how the cache changes over time.

Step 8: Final Performance Calculation

After all pages have been processed:

- Calculate the total number of page requests.
- Compute the **hit ratio** using the formula:

$$\text{Hit Ratio} = \frac{\text{Page Hits}}{\text{Total Page Requests}}$$

Step 9: Display Summary

- Display total page hits.
- Display total page faults.
- Display the calculated hit ratio.

Step 10: Stop

- End the program execution.

Performance Metrics Explained

1. **Page Hits**
    - Occur when a requested page is already present in the cache.
    - Higher page hits indicate better cache performance.
2. **Page Faults**
    - Occur when a requested page is not present in the cache.
    - Page faults are costly as data must be fetched from slower memory.
3. **Hit Ratio**
    - Indicates the efficiency of the cache system.
    - A higher hit ratio means improved system performance.

Advantages of LFU Algorithm

- Effectively retains frequently accessed pages.
- Reduces unnecessary replacements.
- Suitable for workloads with repeated access patterns.
- Simple concept and easy to implement for small cache sizes.

Limitations of LFU Algorithm

- Requires additional memory to store frequency counts.
- Older pages with high frequency may remain even if no longer needed.
- Not ideal for workloads with rapidly changing access patterns.

Conclusion

The Least Frequently Used (LFU) cache replacement algorithm is an efficient memory management technique that prioritizes pages based on their usage frequency. By replacing the least frequently accessed pages, LFU minimizes page faults and improves cache utilization. This simulation provides a clear understanding of cache behavior and demonstrates how frequency-based replacement strategies contribute to system performance optimization.