CS433: Assignment 5 Report

Rusty Dillard

Submitted files:

- fifo_replacement.cpp
- fifo_replacement.h
- lifo_replacement.cpp
- lifo_replacement.h
- lru_replacement.cpp
- lru_replacement.h
- main.cpp
- pagetable.cpp
- pagetable.h
- replacement.cpp
- replacement.h

How to compile and run:

To compile:

$make main

To run:

$main 1024 32 (both numbers *must* be powers of 2)

1. Overview of the Program:

When a computer's memory is full, the operating system must choose which pages of memory to evict to make room for new data. This is known as the page replacement problem, and there are many different algorithms for choosing which pages to evict. Three common algorithms are Least Recently Used (LRU), First-In-First-Out (FIFO), and Last-In-First-Out (LIFO) and I've implemented them in the above listed files.

The LRU algorithm evicts the page that has not been accessed for the longest time. The idea behind LRU is that pages that have not been accessed recently are less likely to be accessed in the future. LRU can be implemented in several ways. One common approach is to maintain a queue of all the pages in memory, with the least recently used page at the front of the queue and the most recently used page at the back. When a page needs to be evicted, the algorithm evicts the page at the front of the queue. When a page is accessed, the algorithm moves it to the back of the queue. This ensures that the least recently used page is always at the front of the queue and will be evicted first.

The advantage of LRU is that it tends to perform well for most workloads. It is particularly effective for workloads with a high degree of temporal locality, meaning that recently accessed pages are likely to be accessed again in the near future. However, implementing LRU can be computationally expensive, especially in systems with a large amount of memory. Maintaining a

queue of all the pages in memory can be time-consuming, and updating the queue every time a page is accessed can be even more so.

The FIFO algorithm evicts the page that has been in memory the longest. The idea behind FIFO is that pages that have been in memory for a long time are less likely to be needed in the future. FIFO is easy to implement, requiring only a queue of pages in memory. When a page needs to be evicted, the algorithm evicts the page at the front of the queue. When a new page is added to memory, it is added to the back of the queue. Unlike LRU, FIFO does not consider how recently a page has been accessed.

The advantage of FIFO is that it is simple and easy to implement. It does not require any complicated bookkeeping, and it performs well for workloads with a low degree of temporal locality. However, FIFO can perform poorly for workloads with a high degree of temporal locality. Pages that are frequently accessed will be evicted just as readily as pages that are rarely accessed. Additionally, FIFO does not consider the size of the pages being evicted. A large page that has been in memory for a long time will be evicted just as readily as a small page that has only been in memory for a short time.

The LIFO algorithm evicts the page that has been in memory the shortest. The idea behind LIFO is that pages that have been in memory for a short time are less likely to be needed in the future. LIFO is implemented using a stack of pages in memory. When a page needs to be evicted, the algorithm evicts the page at the top of the stack. When a new page is added to memory, it is pushed onto the top of the stack. Like FIFO, LIFO does not consider how recently a page has been accessed.

The advantage of LIFO is that it is simple and easy to implement. It does not require any complicated bookkeeping, and it performs well for workloads with a low degree of temporal locality. However, like FIFO, LIFO can perform poorly for workloads with a high degree of temporal locality. Pages that are frequently accessed will be evicted just as readily as pages that are rarely accessed. Additionally, like FIFO, LIFO does not consider the size of the pages being evicted. A large page that has only been in memory for a short time will be evicted just as readily as a small page that has been in memory for a long time.

2. **PageTable:**

This code defines the constructor and destructor for the PageTable class. In the constructor, a PageTable object is initialized with a given number of pages. This is done by creating a vector of PageEntry objects with size given by the num_pages parameter. In the destructor, there is nothing to do as the vector will be automatically destroyed when the PageTable object goes out of scope since the standard library cleans up vectors automatically.

3. **PageEntry:**

This is a class definition for PageEntry, which has three member variables: frame_num of type int, valid of type bool, and dirty of type bool. Frame_num is an integer that represents the number of the frame in which the page is currently stored. Valid is a boolean that indicates whether the page is currently in memory or not. If valid is true, it means that the page is in memory and its contents

can be accessed directly. If valid is false, it means that the page is not currently in memory and needs to be loaded from disk. Dirty is another boolean that indicates whether the page has been modified since it was last loaded from disk. If dirty is true, it means that the page has been modified and its contents need to be written back to disk before it can be evicted from memory. If dirty is false, it means that the page has not been modified and its contents can simply be discarded when it is evicted from memory.

4. **Replacement:**

This code defines a class called "Replacement" that simulates a page replacement algorithm. The constructor initializes the member variables, including the page table. The access_page function simulates a single page access and returns true if it's a page fault. The function also calls other functions, such as touch_page, load_page, and replace_page depending on the status of the page being accessed. The print_statistics function prints out the number of page references, page faults, and page replacements that occurred during the simulation.

5. **FIFO_Replacement:**

In this code, I implement the First In First Out (FIFO) page replacement algorithm. The FIFOReplacement class inherits from the Replacement class and overrides two functions, load_page and replace_page, which are called when an invalid page is accessed and there are free frames available, and when an invalid page is accessed and there are no free frames available, respectively. In the FIFOReplacement constructor, two additional member variables are initialized: frame_page_index, which keeps track of which pages are stored in which frames, and fifo_frame, which keeps track of the index of the frame that was added first. The load_page function updates the frame_page_index and page_table data structures to add a new page to a frame. It sets the new page to the next available frame, marks it as valid, and reduces the number of available free frames. The replace_page function updates the frame_page_index and page_table data structures to replace an old page with a new one. It sets the old page to invalid, adds the new page to the frame_page_index table, sets it to the next frame to be replaced, and marks it as valid. If the FIFO frame index reaches the end, it is set back to the beginning. The FIFOReplacement destructor is also implemented, but it does not contain any code.

6. **LIFO_Replacement:**

In this code, I implement the definition for the LIFOReplacement class that is inherited from the Replacement class. It includes a constructor that initializes the last page to 0 and a destructor that does nothing. It also has two member functions: load_page() and replace_page(). The load_page() function updates the page table by setting the new page to the next available frame, marking it as valid, setting the last page to the new page, and decrementing the amount of available free frames. The replace_page() function updates the page table by setting the old page to invalid, setting the new page to the frame that the last in frame had, marking it as valid, and setting the last page to the new page, then finally returns 0.

7. **LRU_Replacement:**

This code contains my implementation of an LRU page replacement algorithm. The LRUReplacement class is inherited from a Replacement base class and has a constructor and destructor. The load_page and replace_page methods are implemented, as well as a touch_page method that is called when an existing page is accessed. In load_page, if there are free frames available, the page is added to the front of the list and the page table is updated with the new frame number and validity. In replace_page, if there are no free frames, the least recently used page is removed from the back of the list, and the new page is added to the front of the list. The page table is updated with the new frame number, validity, and dirty bit. In touch_page, when an existing page is accessed, it is moved to the front of the list and the page table's dirty bit is set to true. The page table is stored in a map, and the page list is stored as a deque. The map is used to quickly find the position of a page in the list.

8. What I Learned:

When writing the code for this assignment, I learned that the page replacement algorithms can be fairly simple to implement. I know that there would be a lot more to these algorithms, though, if I were to be implementing the actual page replacement algorithms. I would need to allocate all of the memory accordingly and actually evict pages from it, etc. This definitely gives me a good idea of how to get the bare bones of the page replacement algorithms started.

9. Conclusion:

In summary, LRU, FIFO, and LIFO are all page replacement algorithms with different strengths and weaknesses. LRU is generally considered the most effective algorithm for most workloads, but it can be computationally expensive to implement. FIFO and LIFO are simpler and easier to implement, but they may perform poorly for workloads with a high degree of temporal locality. Ultimately, the best choice of algorithm will depend on the specific requirements of the system and the characteristics of the workloads it is expected to handle. A hybrid approach that combines elements of multiple algorithms may also be effective in some cases.