EJ Lilagan, Kyrstn Hall

Dr. Xiaoyu Zhang

CS433 – Operating Systems

23 November 2022

Programming Assignment 5

Submitted Files:

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

How to Compile/Run the Program:

To compile: make
Execute: ./prog5 (page size) (memory size)

Features Implemented (via. pagetable)

The pagetable will consist of two files (pagetable.cpp & pagetable.h). It will consist of two classes (PageEntry & PageTable.) The PageEntry class will consist of three variables including frame_num, valid, and dirty (which will not be used in the assignment.)

- frame_num is an integer variable that will be used as a placeholder for a given page.
- valid is a bool variable to represent if the page is in the physical memory.

In contrast, the PageTable class will include a vector of page entries within the page table for implementing the constructor, destructor, and the operator [].

- Constructor: Loop through each row of the page table (page entry) to insert the following variables from the PageEntry class by using the push_back().
- Destructor: If the page table is not empty, pop until the page table is empty.
- PageEmtry& operator[]: Obtain a page from the page table vector.

Features Implemented (via. replacement)

The replacement will consist of two files (replacement.h & replacement.cpp). This will consist of the page_table object from PageTable class, member variables from the print func (references, page_faults, page_replacement), frame_count for not exceeding the total frames, and the total_frames and total_pages. In particular, the supporting functions will consist of the constructor, destructor, access_page, touch_page, load_page, replace_page, getPageEntry, and print_statistics.

Firstly, the functions that will not require much implementation would be both the constructor and destructor, as not much will be handled in replacement but in the page table.

Secondly, the access_page will simulate through each page entry and select the following conditions. Returns true (is a page fault) if not valid or involves frame_count and total_frames. Otherwise, return false when the page is valid. This will increment the number of references for each page used.

- Condition 1 checks for the page to be valid ⇒ calls touch page and return false.
- Condition 2 checks for the page to be invalid and the number of frames to be less than the total number of frames ⇒ calls load_page and returns true.
- Condition 3 checks for the page to be invalid and the number of frames to be greater than or equal to the total number of frames ⇒ calls replace_page and returns true.

Thirdly, touch_page will not require implementation as replacement is not needed, so it returns.

Fourthly, load_page will set valid to true (assuming it's 0), assign the newly inserted frame number as a tally of frames in memory (including the frame_count++), and tally the page_faults as an invalid page.

Fifthly, replace_page will play as the victim page or a page to be replaced in the page table and will be set as 0 to permanently replace it.

Sixthly, getPageEntry will just return the entry from the page table.

Lastly, print_statistics will return the updated references, page faults, and page replacements that are recorded from the algorithm simulations.

Features Implemented (via. fifo)

FIFO will keep track of the longest unused page that has been present in the main memory. This will be achieved once it has been replaced from the front of the array to update the existing pages. For instance, in the program, fifo_replacement.cpp will inherit replacement_cpp for overriding the updates made within the page table (load_page, replace_page).

- load_page will work the same as replacement.cpp but will push the page number onto the page table.
- replace_page will track an invalid page with no free frames available to update as follows
    - Set the front of the vector to false to indicate that it will be replaced (by valid).

    - Erase the front element by calling the begin() to get index 0 since <vector> does not have a function designed to pop the front element.
    - Once erased, set the page entry to true as it is available.
    - Push the newest page onto the vector and proceed to increment both fault and replacement to indicate that the page has been successfully replaced.

Choice of Data Structure: Vector used for the FIFO algorithm. Present in the load_page and replace_page functions.

- push_back() → insert an element onto the back of the vector.
- front() → direct the vector to point to the front or access the first element.
- begin() → returns the index from the front of the vector.
- erase() → erase all elements from the page of the vector.

Features Implemented (via. lifo)

LIFO will keep track of the longest unused page that has been present in the main memory. This will be achieved once it has been replaced from the rear of the list to update the existing pages. For instance, lifo_replacement.cpp will inherit replacement.cpp for overriding the updates within the page table (load_page, replace_page).

- load_page will similarly work the same as FIFO algorithm since it is inserting new pages.
- replace_page will work opposed to FIFO by accessing the last page, which is as follows
    - Set the back of the vector to false to indicate that it will be replaced (by valid).
    - Erase the rear element by pop_back().
    - Once erased, set the page entry to true as it is available.
    - Push the newest element onto the vector and proceed to increment both fault and replacement to indicate that the page has been successfully replaced.

Choice of Data Structure: Vector used for the LIFO algorithm as shown in the load_page and replace_page functions.

- push_back() → insert an element onto the back of the vector.
- back() → direct the vector to point to the back or access the last element.
- pop_back() → delete an element from the back of the vector.

Features Implemented (via. lru)

LRU will keep track of the least recently used page in the main memory. This will be achieved once it has been replaced from the rear of the array to update the existing pages. For instance, lifo_replacement.cpp will inherit replacement.cpp for overriding the updates within the page table (touch_page, load_page, replace_page).

- touch_page will first erase the LRU page from the list, re insert it in the front, and add it to the map
- load_page will add the new page in the front of the list, and add it to the map
- replace_page will work which is as follows

    - Pop the back of the list as this is the LRU page
    - Erase it from the map
    - Set its corresponding page to false (not valid)
    - Push the new page into front of the list
    - Add the new page in the map
    - Set its corresponding page to true (valid) as it was replaced
    - Incrementing page fault and page replacement variables

Choice of Data Structure: Doubly linked list and hashmap, using list and unordered map from C++ STL library

- push_back() → insert an element onto the back of the list.
- push_front() → insert an element onto the front of the list.
- begin() → direct the list to point to the front or access the first element.
- back() → direct the list to point to the back or access the last element.
- pop_back() → delete an element from the back of the list.
- erase() → removes element from list

Features Implemented (via. main)

Main has 2 tests:

- Test 1 - FIFO with small_refs.txt
    - Pushes values from small_refs.txt into a vector
    - Calls access page to start simulation of FIFO
    - Displays logical address, page number, frame number, and whether the page is at fault or not, for each number in small_refs.txt
- Test 2 - FIFO, LIFO, LRU with large_refs
    - Pushes values from large_refs.txt into a vector
    - Calls access page to start simulation for all 4 implementations
    - Utilized chrono library to calculate the elapsed time of the simulation

Elapsed Time Results with large_refs.txt

FIFO	LIFO	LRU
0.217745 seconds	0.0727945 seconds	0.325477 seconds

**Both tests also display the number of references, page faults, and page replacements during the simulation

Lessons Learned/Re-learned:

The biggest takeaway we have gotten from this assignment is choosing the right data structure, especially for implementing LRU. LRU required us to search through a list, delete it, and insert it back into a list. At first, we thought a vector would be the best approach, as it is simple to

implement. However, the elapsed time would range from 20-40 seconds, and it would take more than 10 minutes to run on gradescope. According to the professor, the best approach would be using a doubly linked list as deleting only requires points to change, but vectors require a lot of copying and are expensive. Additionally, my partner and I learned how to use hashmaps in CS 311 and thought it would be an ideal approach rather than using a loop, as the average complexity for searching in a hashmap is O(1). Furthermore, hashmap has a constant search and access time to efficiently achieve the runtime to not exceed over a second, even if the tradeoff for the hashmap is the amount of memory it uses up. We were able to get our time to <1 from this.

Sources

- Algorithms in Page Replacement Algorithm
- Continuation of Page Replacement Algorithm
- FIFO algorithm
- LIFO algorithm
- LRU ideas to implement
- LRU Cache Implementation - GeeksforGeeks
- LRU Implementation 2
- Doubly Linked List Example
- Unordered_map Library ←→ Hash Library
- List Library
- Vector Library