Page Replacement Algorithms

Summary: Implement a program that simulates maintaining a page table using three different page replacement algorithms.

1 Background

In this assignment you will write a program that simulates the contents of page table as memory requests are made. This starts with loading a reference string (a sequence of page numbers) which represents the pages being requested by a specific program. Whenever the program requests a page not already in a frame (i.e., not in physical memory), we say that a *page fault* occurs as the system must load the piece of data into a frame. Typically, there are many more pages (virtual memory) than frames (physical memory), meaning not every page can being loaded physical memory at the same time. At some point, the program will have a page fault but all frames are being used. This requires us to select a particular page for *eviction* to make space for a new page, and the selection is performed according to some page replacement algorithm.

This document is separated into four sections: Background, Requirements, Data File, and Submission. You have almost finished reading the Background section already. In Requirements, we will discuss what is expected of you in this homework. In Data File, we discuss the format of the simulation data. Lastly, Submission discusses how your source code should be submitted on Canvas.

2 Requirements [36 points]

Our goal is to write a program that simulates a page table. The page table is effectively an array of page entries, where each entry includes a frame number and some other metadata. The metadata includes a so-called valid/invalid bit, which indicates if a particular page is loaded into physical memory (i.e., it exists in some frame). Whenever a program tries to access a particular page (see Simulator.c which loads and uses a specific reference string), it informs the page table. The page table's job is to make sure pages are loaded when needed. Initially all pages have their valid bit set to zero to represent the fact that nothing is loaded. When a page is accessed, it is brought into memory. For our limited simulation, bringing something into memory only means setting it's valid bit to one to represent that it is loaded (we won't actually do anything). When there are unused frames, the page table will select the first unused frame to use rather than use a page replacement algorithm. A page replacement algorithm is used to select which of the existing pages should be evicted from memory (i.e., have it's valid bit set to zero) when all frames are in use, so that a new page can be made to use the frame that the evicted page previously occupied. Your program must support three page replacement algorithms: First-in First-Out (FIFO), Least Recently Used (LRU), Most Frequently Used (MFU). Tie breaking for LRU and MFU should always select the lowest page number (e.g., for MFU, if both page 1 and 2 have the same frequency, replace page 1). Header files are provided.

Your job is implement a page table and these three page replacement algorithms. Requirements:

1. **DataLoader.c:** A file containing a set of functions to load a data file. (This is not a class.) [4 points total]
    - *struct test_scenario* load_test_data(char* filename):* Loads the data file and returns a struct containing its information.
2. **PageTable.c:** A file containing a set of functions to simulate a page table. [32 points total]
    - struct page_table_entry: Stores general information about a page table entry. [4 points]

  - Must use exactly one unsigned int variable to store both the dirty bit and the valid/invalid bit. The right most bit will be the valid/invalid bit. The second bit from the right will be the dirty bit. For our simulation, the dirty bit should always be 0.
- struct page_table: Stores general information about a page table object. [2 points]
- *struct page_table* page_table_create(int page_count, int frame_count, enum replacement_algorithm algorithm, int verbose):* Initializes the page table. [4 points]
  - Verbose parameter is for your debugging only - during grading we always set it to 0.
- *void page_table_destroy(struct page_table** pt):* Shuts down the page table. [2 points]
- *void page_table_access_page(struct page_table *pt, int page):* Simulates the program accessing a particular page.[14 points]
  - Supports putting storing page into first free frame if available. [4 points]
  - Support FIFO page replacement. [4 points]
  - Support LRU page replacement. [4 points]
  - Support MFU page replacement. [4 points]
- *void page_table_display(struct page_table* pt):* Displays page table replacement algorithm, number of page faults, and the current contents of the page table. [2 points]
- *void page_table_display_contents(struct page_table *pt):* Displays the current contents of the page table. [4 points]

You may add other helper functions as needed. The output for the sample data file is given below:

```
====Page Table====
Mode: FIFO
Page Faults: 5
page frame | dirty valid
   0     1 |     0     1
   1     1 |     0     0
   2     2 |     0     1
   3     0 |     0     1

====Page Table====
Mode: LRU
Page Faults: 5
page frame | dirty valid
   0     2 |     0     1
   1     1 |     0     1
   2     2 |     0     0
   3     0 |     0     1

====Page Table====
Mode: MFU
Page Faults: 5
page frame | dirty valid
   0     1 |     0     1
   1     1 |     0     0
   2     2 |     0     1
   3     0 |     0     1
```

2.1 Verbose Output

This should not be displayed, we only give it as a suggestion for how to do debugging. For the purposes of troubleshoting, we implemented a more complete output for the program when the verbose option is enabled (see below). This output shows the various pieces of data that each page entry contains.

```
====Page Table====
Mode: FIFO
Page Faults: 5
page frame | dirty valid | order  last freq
   0     1 |     0     1 |     8     7    2
   1     1 |     0     0 |     3     6    2
   2     2 |     0     1 |     4     3    1
   3     0 |     0     1 |     5     5    2

====Page Table====
Mode: LRU
Page Faults: 5
page frame | dirty valid | order  last freq
   0     2 |     0     1 |     8     7    2
   1     1 |     0     1 |     3     6    2
   2     2 |     0     0 |     4     3    1
   3     0 |     0     1 |     5     5    2

====Page Table====
Mode: MFU
Page Faults: 5
page frame | dirty valid | order  last freq
   0     1 |     0     1 |     8     7    2
   1     1 |     0     0 |     3     6    2
   2     2 |     0     1 |     4     3    1
   3     0 |     0     1 |     5     5    2
```

3 Data File

Your program is required to read in a text file which contains a description of a *reference string* that will be simulated. The first line gives the number of pages for the system, the second gives the number of frames, and the third line lists the number of page requests (aka entries) in the reference string. After that, reference string entries are listed in sequence from 0 to the number indicated. All numbers are positive integers. A sample data file is attached to this assignment (not the one we will grade with) and an annotated version is shown below. You may assume that a reference string contains at most 512 entries.

```
4;number of pages. see canvas for the version of this file you need to support
3;number of frames
7;number of entries in reference string
0;0th page in reference string
1;1th page in reference string
2;...
3
3
1
0;6th page in reference string
```

4 Submission

The submission for this assignment has one part: a source code submission. The file should be attached to the homework submission link on Canvas.

Source Code: Please name your main class as "LastNameDataLoader.c", and "LastNamePageTable.c" (e.g. "AcunaDataLoader.c", and "AcunaPageTable.c").