CSCI 333
Fall 2019

Project 02: Virtual Memory Management

Due: Friday, Nov 15th 11:59pm
Total Points: 100

In this project you will be *simulating* page management systems discussed in class using the C programming language. The simulator will be given a sequence of page allocations and memory references by processes and must keep each process's page-table up-to-date. If/when the physical memory becomes full then a page currently resident in memory will need to be selected to be paged out so the new page can be placed into memory. Since the project is **simulating** a page management system it does **not** require any actual memory for the pages or copy of data into and out of actual memory. Instead the simulator will simply print out what action is being performed and update its internal page-table. The selected page replacement algorithms will be implemented in this project are:

1) FIFO
2) SC (Second-Chance)
3) LRU

The details of these algorithms are available in the text book and slides, sometimes even including implementation suggestions. In all cases the frames will be allocated globally with no concern for a minimum number of frames per process, a page pool, or anything like that.

Command Line Arguments

The code for handling all of the command line arguments is already completed for you. It is in the main.c file in the `main` function. When not given any arguments, the program outputs basic information on its usage:

```
usage: pager input_file [FIFO|SC|LRU]
```

where the square brackets indicate select-one arguments. Informative errors are shown whenever the arguments are not correct and the program exits with an error code (not 0).

- `input_file` – the file name with memory information described in the next section
- `FIFO|SC|LRU` – page replacement algorithm to simulate, must choose one

Input File Format

The code for reading the data from the input file is already provided for you in the file_data.c file in a set of functions. These functions are exclusively used in the `main` function and do not need to be used by you since it is already complete. This section is here mainly for your reference on how to create example input files to test with. An example input file is included in the repo called 'example-input.txt'.

The input file contains information about the memory on the system along with a series of page allocations and memory references. The first line in the file contains 4 integers in base-10 separated by whitespace:

- Logical memory size in number of pages
- Physical memory size in number of frames
- Page/frame size in bits (for 4KiB pages this will be 12)
- Total number of processes

After the first line there are 2 different lines which are distinguished by the first character on the line – either 'a' for a page allocation or 'r' for a memory reference. After the first character there are 3 values as:

- Allocation of page:
  - PID (base-10 integer)
  - Page Number (base-10 integer)
  - access allowed (1 or more characters: 'r' for read, 'w' for write, 'x' for execute)
- Reference of memory:
  - PID (base-10 integer)
  - logical address memory (hexadecimal integer)
  - access requested (single character: 'r' for read, 'w' for write, 'x' for execute)

The PID will always be in the range 0 to total number of processes given on the first line. The range includes the 0 but not the total number of processes (just like indices normally work).

The values may be separated by any amount of whitespace. Each line may have leading/trailing whitespace as well. Any line that is just whitespace or no characters at all is ignored and skipped. If there are any issues reading the file an error is shown and the program exits with an error code.

Output

The program will output what is happening whenever a page is assigned to a frame:

```
Page 0 of process 0 was paged into frame 3
Page 1 of process 5 was paged into frame 4
...
Page 3 of process 4 is selected to be paged out of frame 5
It has not been modified so it will be discarded
Page 2 of process 6 was paged into frame 5
...
Page 1 of process 5 is selected to be paged out of frame 2
It has been modified so it will be written to the swap space
Page 7 of process 2 was paged into frame 2
```

In the simple case when there are free frames it is simply mentioned the page of a process was brought into a particular frame. In the situation when there are no free frames then the victim page/frame is mentioned along with a line about whether that page will be written to disk or not (based on if it has been modified since it was paged in).

Besides paging events the output must include info about invalid memory accesses:

```
Process 0 attempted to access page 0 which has not been allocated
...
Process 1 attempted to read from page 2 but that page can only be executed
...
Process 5 attempted to write to page 1 but that page can only be read or executed
...
Process 2 attempted to execute page 16 but that page can only be read
```

The first example above is if the accessed page has not been allocated at all for the process. The last three examples are if the process attempted to access a page but is not allowed to perform the specified action on that page.

After all memory operations have completed the program will output a divider (40 − symbols) followed by the page fault rate, the total number of page faults, the total number of page faults that caused a frame to be evicted and discarded, and the total number of page faults that caused a frame to be evicted and written to disk. The display should look like the following:

```
----------------------------------------
Page Fault Rate: 0.000025
Total Page Faults: 100
Total Page Faults Evicting and Discarding a Frame: 3
Total Page Faults Evicting and Writing a Frame: 1
```

Complete example outputs are included in the repository for the provided example input.

Design Hints and Suggested Methodology

The code for argument parsing, file reading, and running the high-level part of the simulator is already included in the files provided. Additionally, skeletons for each of the functions that MUST be filled in are also provided.

There are 4 functions you must implement regardless of the page replacement algorithm. All of these functions are called by the `main` function in response to specific actions listed in the input file. Their skeletons are in pager.c along with comments about what they should do:

- `alloc_page` – called to request a page be allocated for a process. It is given the PID and page number to be allocated, along with the access allowed for that page (some combination of `READ`, `WRITE`, and `EXECUTE` flags). This does not bring the page into memory or print any information out.
- `check_log_addr` – called to check a logical address is valid for a given PID and access (one of `READ`, `WRITE`, or `EXECUTE` constants). If the page for that logical address is truly invalid (not allocated to the process at all) or if the access requested is incompatible with the page's allowed access, then it prints out information messages and returns the constant `INVALID_PAGE`. Otherwise it returns either `VALID_PAGE` or `PAGE_FAULT` if it is resident in memory or not. In either case it updates the `REFERENCED` and `DIRTY` flags as appropriate along with any other values that need to be update for each reference.
- `claim_frame` – called to claim a specified frame by a specific page of a process. If the frame is occupied by another page this will have to page out the data (either discarding or writing) along with printing out informational messages. It also updates the frame and page tables as appropriate along with a final message about paging into the frame.
- `print_summary` – prints the summary information at the very end of the simulation.

All of these functions take a pointer to a `pager_data` structure which is defined in pager.h. The structure includes all the necessary fields for the frame and page tables. However, you may need to add additional fields for the various page replacement algorithms. It also uses the `page_table_entry` and `frame` structures. Once again these have a minimal number of fields, you may need to add additional ones. If you add any fields that need to be initialized to a specific value (non-0) then you will need to add some lines to the `pager_data_init` function. If you malloc any additional data then you will need to add some code to the `pager_data_dealloc` function. Note that `page_table_entry` is a special C type called a bit-field struct. You can access the individual set of bits as you would fields from any struct and the bit-shifts and masking are done for you. But only integers. This means any additional data you add to `page_table_entry` must be integral and fit in the remaining 12 bits.

Besides the 4 functions above you will also need to define a function to select victim frames for use page replacement algorithm. The `main` function will automatically select the correct function based on the command-line arguments. Thus there are actually 3 versions of this function, each one prefixed with one of `fifo_`, `sc_`, or `lru_`. Their skeletons and descriptions are in the files fifo.c, sc.c, and lru.c respectively.

- `select_victim_frame` – this is called when there is a page fault reported by `check_log_addr` to for the page replacement algorithm to select a victim frame. It returns the selected victim frame.

As a shortcut please do assume that no page is ever deallocated. This assumption will allow several shortcuts in the various algorithms. For example, FIFO and SC have no need to explicitly make a queue since the sequence is always in order.

It is highly recommended that you get FIFO working completely first with ***identical*** output to the example. The other simulators (especially LRU) are more difficult but there are descriptions in the book. That will tell you if most of the functions are working and only one function will need to be written from scratch for each (although the other ones may need some minor updating as well).

Grading

If the program cannot be compiled and run, it will result in a score of zero. No collaboration is permitted on this assignment. ***See me if you need help***. Grading breakdown is as follows:

Documentation, Readability, Structure, and Efficiency - 10 points
Code must be well documented. Documentation should be clear and descriptive. Code should be well structured with minimal redundant code. I should be able to tell what it's doing by reading it. This is in addition to comments – the code itself should be "self-documenting" with descriptive variable and function names (but not overly verbose). Code must be as efficient as possible without sacrificing the readability/structure.

Pager Functionality – 60 points
For each of FIFO, LRU, and SC algorithms the points are broken down as follows:

- 10 pts – `alloc_page` – updates the pager information, doesn't print anything
- 20 pts – `check_log_addr` – translates and validates the logical address
- 20 pts – `claim_frame` – a page claims a frame, evicting a possibly resident page
- 10 pts – `print_summary` – prints out the summary information, including divider

Page Replacement Algorithm Functionality – 3x10 points
For each of FIFO, SC, and LRU algorithms the function to select victims is needed.

Project Check-In

To ensure you are making progress throughout the 2.5 weeks, on Friday November 8 I will have a "check in" where I will check your progress and with the rest of the time spent working on the assignment itself, giving you a chance to ask questions and get assistance. If you fail to show sufficient progress, ***there will be an automatic 10% reduction in the final grade for the project***. Sufficient progress would be something equivalent to:

- Completed `alloc_page`, `print_summary`, and `check_log_addr`. Most of the work for these can be done without virtual memory knowledge. At this point, examples with more frames than pages work. You should also have begun `claim_frame`.
  - Once you finish the functions above plus `fifo_select_victim_frame`, FIFO should be completely working and testable.