CS 361 – Computer Systems – Fall 2014
Homework Assignment 3
Simulation of a Memory Paging System

Due: **Sunday October 19th. Electronic copy due at 9:00 a.m., optional paper copy may be delivered to the TA.**

Overall Assignment

For this assignment, you are to write a simulation to analyze a virtual memory paging system given a certain number of memory accesses and a certain amount of available memory.

Command Line Arguments

The command line for this program will be as follows:

vmpager inputDataFile [#memoryAccesses frameTableSize . . .]

- inputDataFile is a data file to be read and interpreted as page requests. See below for details.
- #memoryAccesses is the number of memory accesses to process. If not provided, or if specified as zero, then your program should read and process the entire input data file.
- frameTableSize is the number of frames available to the virtual memory manager. Default = 256.
- You may have additional parameters that follow the above parameters, particularly if you want to explore optional enhancements. This is up to you, but must be well-documented in your readme file.

Data File Interpretation

1. The data file will be interpreted as an array of structs, defined as follows:

```
typedef struct {
    uint8_t pid;
    uint8_t page;
} MemoryAccess;
```

2. To access the file you will first need to open( ) the file, and then use fstat( ) to determine its size. You can then use mmap( ) to memory map the file to a MemoryAccess pointer, which effectively gives you an array of MemoryAccesses. The upper limit on the size of the array can be determined by dividing the size of the file by the size of a MemoryAccess (using integer math.)
3. A set of possible data files will be provided, having certain characteristics. Alternatively you may want to test with small files, possibly with known pre-determined contents that you have previously generated with a separate program.

Page Table(s):

In a real system, each process would have its own page table, which would need to be stored when the process was not running, and restored when the process recommenced. For this simulation we can simplify things by assuming that there will always be exactly 256 running processes, and that each process will have exactly 256 pages in their page table. That will let us simulate all the page tables as a single two-

dimensional array, but we will still need to keep track of several pieces of information for each page table entry:

```
typedef struct {
     int frame; // -1 if not currently loaded
     int pageHits;
     int pageMisses;
} PageTableEntry;
```

So now the page tables for all of the processes can be simulated as a single two-dimensional array of PageTableEntrys:

```
PageTableEntry pageTables[ PAGETABLESIZE ][ NPROCESSES ];
```

where PAGETABLESIZE and NPROCESSES are both set at 256 for this assignment.

Frame Table

The frame table corresponds to physical memory, and keeps track of which page (if any) is currently loaded into each frame of memory. For this simulation we will need to keep track of three pieces of information for each frame, the process ID that is using that frame (or 0 if vacant), the process's page number, (or 0), and a bool to indicate if the frame is really vacant (since 0, 0 would be valid pids and page numbers):

```
typedef struct {
     uint8_t pid;
     uint8_t page;
     bool vacant;
} FrameTableEntry;
```

The frame table will be simulated by an array of FrameTableEntrys, where the size of the frame table is given as a command line argument to the simulation (or 256 if not given.)

Simulation Algorithm(s):

The basic algorithm for the simulation is to loop through the memory accesses in the input data file, determine whether each one is a page hit or a page miss, and then after all memory accesses have been simulated, report the total number of page hits and page misses. Determining whether a memory access is a page hit or miss will involve looking at the page table, and possibly modifying the page table and/or frame table, depending on the specific algorithm being simulated, as follows:

Page Hits: If the page table entry corresponding to a particular (pid, page) memory access shows a non-negative frame number, then the page is already loaded, and a page hit has occurred. Increment the global page hit counter and the page hit counter in the page table entry.

Page Misses: If the page table entry corresponding to a particular (pid, page) memory access shows a negative frame number, then the page miss has occurred. Increment the two page miss counters as above, but now it is also necessary to "load" the missing page, by updating the frame table and page table.

1. First select a frame in the frame table (see below) to hold the new page, and record that frame number in the page table entry corresponding to the page miss.

2. If the selected frame was not vacant, then a "victim page" must be "removed" from the frame table to make room for the new page. Do this by changing the frame field in the page table entry for the victim page to -1.

3. Finally mark the frame as containing the new page.

Your program should examine the following paging algorithms, which differ in the way that frames are chosen:

1. Infinite Memory. Or at least enough that no page ever needs to be paged out, which is the same thing for our purposes. This generates the minimum number of possible page faults, because once a page is loaded the first time, it never needs to be paged out and loaded again.

   For this algorithm the Frame table is not used. Simply initialize all frame numbers in the page table to -1, and then change them to +1 the first time that they are accessed. The first time that a particular process accesses a particular page, it counts as a page miss, and any additional accesses to the same page by the same process count as page hits.

2. FIFO. For this simple algorithm, the frame table is accessed as a circular queue, taking the next frame in the circle as the next victim whenever a page fault occurs. Initially all of the frames should be marked as vacant. When "loading" a page into a frame, it will be necessary to see if there is already a page in that frame, and if there is, "clearing" it out by marking the corresponding entry in the page table as invalid (-1).

3. Note: With a little bit of work, you should be able to collect statistics for both FIFO and infinite memory in a single run of the simulation. This will most likely require adding some extra fields to the PageTableEntry struct defined above.

Required Output

- All programs should print your name and ACCC account ID as a minimum when they first start.
- Beyond that, this program should report its command line arguments and the total number of page hits and page misses recorded. (It would be nice to examine the individual page hit and miss counts stored in the page table, but that would probably require plotting or graphing to visualize.)
- In addition to a program printout, a short memo / report shall be written with the results of the algorithm analysis. What you want to look at is the ***additional*** page faults required for a given input data file as a result of limiting the page table size.
- Your report should contain a plot showing the additional page faults required by finite memory (e.g. results for FIFO - results for infinite memory) as a function of different frame table sizes. Note that any frame table size larger than 65536 or the total number of memory accesses, whichever is smaller, is effectively infinite for the purposes of this simulation.

Other Details:

- The TA must be able to compile your program (on an AWS machine) by typing "make vmpager". Provide a makefile if necessary. As always, you are free to develop wherever you wish, but the TA will be grading the programs based on their performance on the AWS machines.

What to Hand In:

1. Your code, **including a makefile if needed, and a readme file,** should be handed in electronically using Blackboard.

2. The purpose of the readme file is to make it as easy as possible for the grader to understand your program. If the readme file is too terse, then (s)he can't understand your code; If it is overly verbose, then it is extra work to read the readme file. It is up to you to provide the most effective level of documentation.
3. The readme file must specifically provide direction on how to run the program, i.e. what command line arguments are required and whether or not input needs to be redirected. It must also specifically document any optional information or command line arguments your program takes, as well as any differences between the printed and electronic version of your program.
4. If there are problems that you know your program cannot handle, it is best to document them in the readme file, rather than have the TA wonder what is wrong with your program.
5. **For this assignment you must also hand in the results of your analysis in the form of a short memo / report, as described above.**
6. A printed copy of your program, along with any supporting documents you wish to provide, (such as hand-drawn sketches or diagrams) may be delivered to the TA, and must match exactly the electronic submissions.
7. Make sure that your **name and your ACCC account name** appear at the beginning of each of your files. Your program should also print this information when it runs.

Optional Enhancements:

- Explore additional paging algorithms, i.e. additional approaches for selecting victim frames when there are no free frames left. Unfortunately the text book used for this course does not cover victim frame selection, so you will need to consult another source if you wish to pursue this enhancement. Note that some of these algorithms care whether memory accesses are for reading or for writing, which then requires modifying the MemoryAccess struct to include a field indicating that.
- Can you express the results of your simulation in a (simple) formula? I.e. for a given number of processes, memory sizes, etc, can you predict the page hit ratios and program timings?
- Other enhancements that you may think of - Check with the TA for approval.