CSCI 447 - Operating Systems, Spring 2024
Homework 4 : Inverted Page Table, Demand Paging
Due Date: Check Canvas

~~Collaboration~~ Policy

Your homework submissions must be yours. You may chat with other students on a high-level about topics and concepts, but you cannot share, disseminate, co-author, or even view, other students' code. If any of this is unclear, please ask for further clarification.

1 Free Response Questions

Upload a .pdf document to Canvas for the *Homework 4 - questions* submission.

1.1 Q1 : 5 points

For an average page fault service time of 5 milliseconds, and a memory access time of 300 nanoseconds, what is the maximum access(s) out of 2,000 attempts that can cause a page fault so that the effective access time is at most 6.6 microseconds? Your answer should be an integer value. Please show your work.

1.2 Q2 : 5 points

Consider the following page reference strings (requests for pages):

```
1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6
```

How many page faults occur when LRU, FIFO, and Optimal replacement are used, when physical memory is made up of 1, 3, 5, and 7 frames?

1.3 Q3 : 5 points

Assume a demand-paging system where CPU utilization is 20%, Paging disk is utilized 97.7% of the time, and other I/O devices are utilized 5% of the time. For each of the following, indicate whether it will improve CPU utilization. If your answer is "it depends," then be sure to explain the multiple scenarios. In all cases, be sure to explain your answers:

- Install a faster CPU
- Install a bigger paging disk
- Increase the degree of multiprogramming
- Decrease the degree of multiprogramming
- Install more main memory
- Install a faster hard disk or multiple controllers with multiple hard disks
- Add prepaging to the page-fetch algorithms
- Increase the page size

2 Coding Task : 2 points for setup, 68 points for the program

Your task is to implement an inverted page table, shared by all processes, in a demand paging system, and compare the performance of two commonly used page replacement algorithms with a random page replacement strategy. The aim is to compare the performance of the two strategies. The requests for memory accesses and notification of process termination will come from a simulation program that you are being given (*MemSim.o*, see Lab 4).

2.1 Setup

1. Create and checkout a new branch on your gitlab for this assignment. Name the branch *homework4*. **When creating a new branch, ALWAYS do that from the master branch.**
2. In the *homework4* branch, create a new directory, *homework4*. All work for this homework assignment should be done in the *homework4* branch and *homework4* directory.
3. Download the *homework4Files.tgz* file from the course website, and save it to the *homework4* directory. Unzip and untar.

2.2 Provided Software

The following software is provided for your use in the assignment:

- *MemSim.h* header file, defining the `Simulate()` function, provided by the simulator.
- *PageSim.h* header file, defining the functions that you need to provide in your *PageSim.c*.
- *MemSim.o* the object file for the simulator which has been compiled on the CS computers.
- *PageSim.c* A demonstration program which implements the functions defined in *PageSim.h* and has a `main()` function which invokes the `Simulate()` function. You can compile and run this program with the provided header files and one of the static library files. Note that the hash table used in this demonstration is not the page table that you need to implement. It is merely for demonstration purposes.

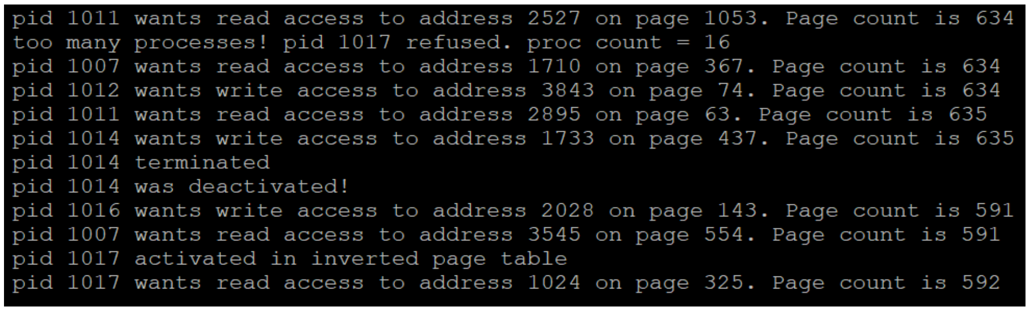
Any one pid randomly generates a set of addresses (for either reading or writing). A single pid might need a single address during the simulation (unlikely), or a large count of addresses.

2.3 The Paging System

Your paged memory system must use demand paging and a single inverted page table for all processes. The characteristics of the system are as follows:

- Address size : 32 bits
- Page size : 4 Kbytes
- Memory available for page frames : 8 Mbytes
- Maximum number of processes : 64

From that information you can determine the number of bits of an address that constitute the offset, and the total number of frames allowed. Most importantly, note the maximum number of processes that are permitted. Thus you need a tally of the unique pids that are “running,” and not permit new pids to read or write to the page table if there are already 64 processes with entries in the page table. In Figure 1 – where the max number of processes is set to 16 instead of 64 – note that pid 1017 attempts either a read or write to the page table, but the process count is already 16, so pid 1017 is refused. Only after pid 1014 terminates, do you eventually see pid 1017 get activated, and get read access. Also note that `pid 1014 terminated` is the echo of what *MemSim* generated, which occurs BEFORE the inverted page table is cleaned up, and `pid 1014 deactivated!` is the output of *PageSim.c* AFTER the inverted page table is cleaned up. And, the term `activated` refers to the event when a <pID,page> is inserted into the inverted page table. You don't have to use this exact terminology in your program, but it is recommended that your code prints status messages to help you assert correct behavior.



```
pid 1011 wants read access to address 2527 on page 1053. Page count is 634
too many processes! pid 1017 refused. proc count = 16
pid 1007 wants read access to address 1710 on page 367. Page count is 634
pid 1012 wants write access to address 3843 on page 74. Page count is 634
pid 1011 wants read access to address 2895 on page 63. Page count is 635
pid 1014 wants write access to address 1733 on page 437. Page count is 635
pid 1014 terminated
pid 1014 was deactivated!
pid 1016 wants write access to address 2028 on page 143. Page count is 591
pid 1007 wants read access to address 3545 on page 554. Page count is 591
pid 1017 activated in inverted page table
pid 1017 wants read access to address 1024 on page 325. Page count is 592
```

Figure 1: Sample run-time output (not all lines shown)

2.4 Page Replacement Strategies

Please implement and compare the performance of the below page replacement strategies. Refer to sections 10.4.4, and 10.4.5 in the textbook, 10th edition, and the lecture slides, and lab 4, for complete details. **You may not import nor rely on nor use any type of external code (a library, a function, an executable, or otherwise) that implements any sort of eviction algorithm. You are coding from SCRATCH.**

1. **Least Recently Used, Enhanced Second-Chance Algorithm (LRU)** : This is a variation of the "clock" approximation of LRU, for which both the reference bit and the modify ("dirty") bit of each frame are considered. The best page to replace is in the frame with both bits 0 that is closest (inspecting in the direction of higher index values of the page table) to the most recent eviction location. If no such frame exists, the order of preference is (from most preferred to least preferred):

   - Reference bit = 1, modify bit = 0
   - Reference bit = 0, modify bit = 1
   - Reference bit = 1, modify bit = 1

   Once the replacement page is chosen according to the above criteria, the reference bit is reset to 0 for all frames (inclusive) between the frame used for the previous page replacement and frame selected as the next to be evicted.

2. **Least Frequently Used (LFU)** : This algorithm requires a reference count for each page. When the page is first paged-in, the reference count is set to 0. Each time the page is referenced, its reference count is incremented. The reference counts are aged by dividing them by 2 at regular intervals. For example after every 1,000 page reference requests to the entire memory system, all pages should have their reference counts halved.

3. **Random** : This approach selects, at random, a frame.

2.5 Software that you must write

You must write one or more C source files that implement the functions defined in *PageSim.h* and provide a `main()` function, which invokes the `Simulate()` function defined in *MemSim.h*. `Simulate()` will call two functions, specified in *PageSim.h*, that you must provide. Those two functions are:

- `int Access(int pid, int address, int write)` : Called when the process with process id `pid` wants to access memory at `address`. If `write` is nonzero, the process wants write access, otherwise, the process wants read access. If that address can be accessed, return 1, otherwise return 0.
- `void Terminate(pid)` : Called when the process with process id `pid` is terminating. The frames used by this process are now free for other use.

2.6 Invocation

Your *PageSim.c* file should have a `main` routine that accepts (requires) a single command line argument. The command line argument MUST be one of the following (this is lab 4):

- LRU
- LFU
- Random

A sample invocation would be the following : `./PageSim LRU`

2.7 Results

Your program should output the following, in this order, at the termination of a runs:

- The final state of the Inverted Page Table, row-by-row on the command line, in the format <`row,PID,p,dirty bit,reference bit`>. Please print rows in ascending order. This does not mean that your inverted page table as implemented must contain the dirty and reference bits; those can be stored elsewhere in your program.
- Number of page faults that require only reading of the page to be swapped in.
- Number of page faults requiring both writing of the replaced page and reading of the page swapped in.
- Total number of page faults (the sum of the above two).
- The total percentage of page faults that require both writing of the replaced page and reading of the page swapped in.

Note that when a process terminates, there may be MULTIPLE <pID,page> 2-tuples in the inverted page table, and you should set the valid bit to 0 for all of them. Also note that setting a valid bit to 0 does NOT count as a page fault. Only when something is evicted to make room for a new <pID,page> 2-tuple, does that count as a page fault.

2.8 Testing

Consider creating your own *MemSim.c* to generate edge case PID, memory request 2-tuples to test your implementation. For example, to generate repeatedly the same page request, or increase the count of pIDs that terminate, or ... these sort of setups will permit you to more easily test your implementation of your inverted page table. Note that you are NOT submitting to git your *MemSim.c*; instead a custom *MemSim.c* will be used to test your implementation of your inverted page table. Conversely, do not put into your copy of *MemSim.c* any logic for your inverted page table. The entirety of your "submission" code should be inside of *PageSim.c*.

2.9 Files to submit

Please push to the homework4 branch, into the homework4 directory, the following :

- The files *PageSim.c* and *PageSim.h*
- Any other custom .c file(s), if you needed to create and use a custom data structure(s)
- A *Makefile*, which if invoked will generate a *PageSim* executable, without warnings.
- A short report, 2 "pages" maximum, a plainText file, called *ReplacementAnalysis.txt*, that includes your findings on the page replacement strategy that you would recommend for use in the provided system. **Substantiate** your recommendation with your run-time results.
- Do NOT submit *MemSim.c* if you wrote one for testing purposes

3 Hints

- When testing your software, perform simulations with large enough iterations to guarantee that evictions occur, otherwise, you won't be able to test accurately if LRU, LFU, and Random eviction work correctly. For example, if physical memory can store x frames, then perform at least $1000x$ iterations of the simulation.
- After an eviction, the "brought in" frame should be placed into the location that has just been made "available."
- How you implement the inverted page table is up to you – two (of many possible) choices are a linked list or a static array. Think this through, because your choice of data structures will inform you how to best implement the replacement algorithms.
- Keep in mind that when a page is brought in for the first time (and an eviction did not occur), that is still considered a page fault.
- You are tasked with implementing an inverted page table. You are NOT implementing physical memory.

4 Rubric

Written Questions	15
Programming Task : setup (git), including branch	1
Programming Task : .o nor executable file(s) have NOT been pushed to your git branch	1
Programming Task : main routine correctly accepts command-line arguments	3
Programming Task : Access function written and compiles	5
Programming Task : Terminate function written and compiles	5
Programming Task : Makefile, without warnings	5
Programming Task : correctness of paging program, for each of LRU, LFU, and Random	40
Programming Task : quality of your code, including choice of data structures, efficiency, readability, commenting, etc.	5
Programming Task : completeness of your *ReplacementAnalysis.txt* file, including (brief) discussion, explaining your selection of the optimal replacement algorithm	5
Total	85 points