EECE.4810/EECE.5730: Operating Systems

Spring 2020

Programming Project 4

Due ~~11:59 PM, Friday, 5/8/20~~ 12:00 PM, Saturday, 5/9/20

1. Introduction

In this project, you will implement a paged memory manager. Your program will allocate an array to represent memory, then handle a series of requests from threads representing different processes. Each "process" will have its own virtual memory space—and therefore its own page table—while your program will also track the list of free frames and handle page replacement using the clock algorithm discussed in class.

This assignment is worth a total of 135 points (+ 15 extra credit points). **This assignment will take the place of your final exam, and will therefore count for 15% of your final grade.** The grading rubric in Section 4 applies to students in both EECE.4810 and EECE.5730.

Spend a significant amount of time on program design before you write a single line of code. Start by reading the entire specification, from the introduction to the last hint, and take notes or identify key sections. Then, plan how the pieces of your program will fit together.

2. Project Submission and Deliverables

Your submission must meet the following requirements:

- Your solution may be written in C or C++.
- Your code must run on the Linux machines in Ball 410, but you may write it elsewhere.
- You must submit a .zip file via Blackboard containing the following files:
    - All source/header files you write. You should have at least three such files:
        - A source file containing your `main()` function and global variables
        - A header/source file pair for any structure/class definition(s) and functions
            - My solution uses 2 pairs of files—one for thread-specific data and functions and one for memory management data and functions)
            - Globals declared in your main file should be declared `extern` in the source files only if necessary—if no code in the file accesses a global variable, you don't need the `extern` declaration.
    - A makefile that can be used to build your project
    - A README file that contains group member names, a brief description of the contents of your submission, and directions for compiling and running your code.
- You may work in a group of 2 students on this assignment. If you work in a group:
    - Contact Dr. Geiger if your group has changed.
    - As noted above, your README file should contain all group member names.

3. Specification

General overview: As noted above, your program will use multiple threads to model processes accessing memory. The program should contain a different structure or object to represent each major subsystem in a paged memory manager.

Your program will model the following key operations to handle memory accesses:

- Translating virtual addresses to physical addresses
- Determining whether a page is resident in memory
- Identifying a free frame to use for a page that is not resident in memory
- Identifying a page to evict from memory if no free frames are available

More details on the necessary data types and operations can be found in Section 6: Hints.

Input: Your program reads input from two types of sources—the command line and files:

Command line arguments: Your main executable should take the following command line arguments:

- The name of an input text file containing information about the processes and memory to be simulated
    - As noted in previous projects, a "text file" does not require a .txt extension
- The name of a text file that will contain your program's output
- The seed for the random number generator (RNG), if you want your program to repeat the same pseudo-random values. If no seed is specified, your program should approximate "true" randomness by seeding the generator using the current system time.
    - See http://cplusplus.com/reference/cstdlib/srand/ for more details on RNG seeding
    - I use random numbers to fill main memory as needed; seeding the random number generator allows me to duplicate memory contents in multiple runs

For example, if your executable name is `proj4`, the following command runs the program with `infile1.txt` as the input file, `outfile1.txt` as the output file, and a random number generator seed of 1: `./proj4 infile1.txt outfile1.txt 1`

Main input file: The main input file contains the following information:

- Total main memory size, in bytes
    - In high-level languages, the `char` data type is always 1 byte, so your main memory array should be an array of type `char`.
- The page size, in bytes
- Number of processes to be simulated (in your program, 1 thread = 1 "process")
- One configuration file name per "process." Each thread is responsible for reading its own configuration file.

3. Specification (continued)

Input (continued):

Per-thread input file: Each thread will read from a file that contains the following information:

- Total virtual memory size for that thread, in bytes
- A series of memory accesses, specified as pseudo-instructions in the format `<rw> r<#> <addr>`, where:
  - `<rw>` is a character, `'R'` or `'W'`, representing the access type (read or write, respectively)
  - `<#>` is the number of a single register, always preceded by the letter `'r'`
    - $0 \leq$ `<#>` ≤ 31, so each thread should maintain a set of 32 integer registers
    - Your memory array should be an array of bytes, so you'll have to build an integer from individual bytes on a read and break down each integer into its component bytes on a write
    - Data should be stored in memory in big-endian format
  - `<addr>` represents a virtual address in the address space of the "process"
    - Your program must translate that virtual address to a physical address to read or write the appropriate data

Output: This program should print all output to the text file named in the command line arguments. Each line of output should include the number of the "process" that printed it, as well as information about the operation performed.

Every address (virtual and physical) and value read from/written to memory should be printed as a 32-bit hexadecimal value, with leading zeroes used when necessary.

Your program should print at least one line of output for each key operation mentioned above, as well as for the actual memory access itself. In most cases, the output will be generated after the specified operation is complete.

- *For each memory access:* reprint the pseudo-instruction from the input file before starting the access, then print the final result (the data read or written) when done. Note: output lines from some—possibly all—of the steps below will print in between these two lines.

  Examples from two separate accesses:

```
P0 OPERATION: R r3 0x00000003
...                    Detailed steps in performing operation—don't actually print dots!
P0: r3 = 0x7351ff4a (mem at virtual addr 0x00000003)

P1 OPERATION: R r1 0x00000020
...
P1: r1 = 0x70e93ea1 (mem at virtual addr 0x00000020)
```

- *Translating virtual addresses to physical addresses:* print the virtual address and the physical address to which it is translated.

  Examples from two separate translations:

  ```
  P0: translated VA 0x00000003 to PA 0x00000003

  P1: translated VA 0x00000020 to PA 0x00000040
  ```

- *Determining whether a page is resident in memory:* print a message indicating whether the requested page has a valid translation. If the page is not resident in memory, print another line after a free frame is found, indicating what the new translation is.

  Valid translation example:

  ```
  P0: valid translation from page 0 to frame 0
  ```

  Invalid translation example:

  ```
  P1: page 1 not resident in memory
  ...                    Output related to finding free frame—don't actually print dots!
  P1: new translation from page 1 to frame 2
  ```

- *Identifying a free frame to use for a page that is not resident in memory:* if a free frame is available, print its frame number.

  Example:

  ```
  P1: using free frame 2        This line printed between the two lines shown above for P1
  ```

- *Identifying a page to evict from memory if no free frames are available:* print the process and page number of the page being evicted.
  - Page replacement is done globally, meaning there's one page replacement "clock" for the entire system. Since multiple processes use the same page numbers, the list tracking pages in use must identify pages by both their process and page number.

  Example:

  ```
  P1: evicting process 0, page 0
  ```

4. Grading Rubric

Your assignment will be graded according to the rubric below; partial credit may be given if you successfully complete part of a given section of the project. Sections include:

- Correctly processing main input file: **10 points**
- Starting and joining threads correctly: **10 points**
- Having each thread correctly process its input file: **10 points**
- Reading/writing memory correctly: **10 points**
    - You'll get credit for this part regardless of whether your program translates addresses correctly—I'm looking to see if, given an address and a read/write operation, it can get the correct data from memory or write data to it correctly.
- Translating addresses correctly: **15 points**
    - This part is only about accessing a page table entry and reading a frame number.
- Maintaining page table correctly: **15 points**
    - This part is about actually storing translations properly, as well as the appropriate status bits (valid, reference, dirty)
- Correctly determining if a page is resident in memory: **10 points**
- Identifying a free frame for a page that is not resident in memory: **15 points**
- Maintaining the free frame list: **15 points**
    - This part is about ensuring all frames are initially marked as free, each frame is removed from the free list when it's used in translation, and all frames for a given process are returned to the free list when the process ends.
- Evicting a page if no free frames are available: **15 points**
- Maintaining the page replacement clock: **15 points**
    - This part is about ensuring pages are added to the page replacement clock when they're brought into memory and properly removed from this list either when they're evicted or when the associated process ends.
    - **Change as of 5/7:** Since I (still) don't have my page replacement code working, the combined 30 points from these two sections will be graded as follows:
        - **15 points:** Does your page replacement code look like it should work?
        - **15 points (extra credit):** Does your page replacement code actually work?
- All synchronization: **10 points**
    - Getting full credit here will require that you synchronize accesses to shared data to allow as much overlap as possible—you're going to need multiple locks.
    - For example, if one thread is reading from your memory array, a second thread should be able to access the free frame list.

5. Test Cases

Test cases for this assignment come in the form of files posted on Blackboard. There are a couple of main input files (generally called `infileX.txt`, where `X` is a number), a few thread input files (`threadY.txt`, where `Y` is a number), and a couple of output files (`outfileZ.txt`—you can guess what `Z` is). *(I plan to add more once I finish ironing out the last couple of bugs in my solution—at the time I'm posting this, it doesn't quite handle page eviction correctly.)*

The Blackboard assignment lists the command line used to generate each output file.

6. Hints

The following points are based, to some degree, on my implementation, so remember you're (mostly) welcome to implement your solution any way you want.

I wrote my first hint in the introduction, but it bears repeating: **spend a significant amount of time on program design before you write a single line of code.** The grading rubric spells out many key pieces to your program, but you should plan how those pieces fit together.

Some other hints to consider *(with more potentially to be added)*:

Structures/classes: I found the following structures (or classes, if you're using C++) to be useful in this program:

- A single page table entry (PTE), which can be used to build each process's page table
- A single node in the free frame list, which tracks physical frames not currently in use
    - While there are many ways to track free frames, I simply built a linked list with one node per available frame.
- A single node in the page replacement "clock," which tracks all currently resident pages that can be considered for eviction
    - Your program should use global page replacement—one "clock" for all processes.
    - You'll therefore need some way for each node to refer to page table entries for all processes, such as a pointer to the corresponding PTE or an array index or indexes, depending on how you store your PTEs.
- Two monitors
    - One contains everything related to memory management
        - The (hopefully) obvious global structures: memory array, free frame list, page replacement clock, as well as some other global information
        - The page tables for all processes, making it easier for code in one thread to access a PTE for another.
        - Having one monitor doesn't mean having one lock for all of its parts—you want to allow as much interleaving as possible.
    - The other monitor is for the output file, since every thread prints to it. You could get some odd output if you don't ensure only one thread prints to the file at a time.

6. Hints (continued)

Functions: I implemented the following functions; you may, of course, use others:

- A single thread function that represents each "process."
    - This function mostly sets up everything that needs to be done on a per-thread basis, then reads every memory access from the input file and calls a function that does most of the work, namely …
- Memory access: given an operation (read/write), process number and virtual address, perform that operation and return the result (or store it through a pointer argument)
    - This function is the starting point—it may invoke one or more of the other functions, which may in turn invoke one or more other functions, and so on …
- Translate: given a process number and virtual address, return the corresponding physical address
    - If you want to have this function do the work of updating the page table, you may also want to pass in the operation, so it can set the dirty bit on a write.
- Get a free frame: return the number of the first available free frame
- Add a free frame: add a frame to the free list
    - This function can be used both in the initial setup (when all frames are free) and when a process ends (so all its frames return to the free list)
- Add a node to the page replacement "clock": every time there's a new page "brought in to memory," it needs to be added as a candidate for eviction
- Evict a node: choose a candidate for eviction and remove that node from the list
    - You could write a helper function to remove the node, or you could simply overwrite the "removed" node with the information of the page that replaces it.

Other notes: Assorted points I found tricky or just thought you'd find helpful:

- As noted earlier, data should be stored in big-endian format in your memory array
    - Remember, the array itself is an array of bytes, but each thread contains a set of integer (32-bit, or 4-byte) values.
    - So, you'll have to combine bytes together on a read, or split a register value apart on a write. The endianness determines the order in which you do that.
- Each time I "bring a new frame into memory", I fill it with random values. If you do this, using the same RNG seed for multiple program runs ensures memory holds the same values each time. Debugging's hard enough without memory changing every time.
    - It's also useful to take one program run and print the contents of each frame—frame number, physical address, and byte in each array location—so you can compare what's "actually" in physical memory to what your program sees.

6. Hints (continued)

Other notes (continued):

- It's very easy to forget about certain "cleanup" tasks before a thread exits, namely:
    - Freeing all of its frames (and therefore returning them to the free frame list)
    - Removing all associated nodes from the page replacement "clock"
    - Deallocating anything you allocated inside the thread function itself
- Synchronization is very important. It's nice to have multiple locks to allow lots of interleaving, but the more locks you have, the trickier the program is to debug.
    - I spent a while chasing down a bug in which my program deadlocked because I'd locked access to one shared structure, then encountered a condition in which I needed to call a function that attempted to obtain the same lock. There's nothing wrong with doing that … if you unlock the lock before calling the second function.

There are probably more hints to be added, based on one of two things (that I can currently think of) happening:

1. *At least one of you asking a question that makes me realize I need to clarify something about the assignment.*
2. *Me finally figuring out this horrible page eviction bug. Have I mentioned my solution doesn't quite work at the time I'm posting this spec? It's about 99% of the way there … I think. Please don't let that fact discourage you.* ☺