

Computer Science 360 –Operating Systems Summer 2021

Assignment #4

Due: Tuesday, August 3rd, 11:55 pm via push to your gitlab.csc repository

Programming Platform

For this assignment **your code must work on the Virtualbox/Vagrant configuration you provisioned for yourself in assignment #0**. You may already have access to your own Unix system (e.g., Ubuntu, Debian, macOS with MacPorts, etc.) yet I recommend you work as much as possible while with your CSC360 virtual machine. Bugs in systems programming tend to be platform-specific and something that works perfectly at home may end up crashing on a different computer-language library configuration. (We cannot give marks for submissions of which it is said “It worked on Visual Studio!”)

Individual work

This assignment is to be completed by each individual student (i.e., no group work). Naturally you will want to discuss aspects of the problem with fellow students, and such discussion is encouraged. However, sharing of code is strictly forbidden. If you are still unsure about what is permitted or have other questions regarding academic integrity, please direct them as soon as possible to the instructor. (Code-similarity tools will be run on submitted work.) Any fragments of code found on the web and used in your solution **must be properly cited** where it is used (i.e., citation in the form of a comment giving the source of code).

Goals of this assignment

- Complete a C program that simulates three page-replacement schemes that may be used by virtual-memory systems.

Goal: Implement FIFO, LFU, and the Clock page-replacement schemes

This goal requires you to simulate the actions of a virtual-memory subsystem. Specifically you are to take a sequence of logical memory operations generated by a target process and “convert” them into physical addresses. In reality, however, you will not be running a target process but instead using a memory trace generated for you already (using an application suite from Intel called pin) and stored into a text file. You are provided with a code skeleton named `virtmem.c` which currently reads in memory operations contained in a specified trace file and calls the `resolve_address()` function for each operation. This skeleton code also finds a free frame but does not replace pages when the simulated memory is full.

Here are the first few lines from a memory trace generated by a “Hello, World!” program:

```
I: 0x7feee195f090
I: 0x7feee195f093
W: 0x7ffe23dd2e88
I: 0x7feee195fea0
W: 0x7ffe23dd2e80
I: 0x7feee195fea1
I: 0x7feee195fea4
W: 0x7ffe23dd2e78
I: 0x7feee195fea6
```

Each line starts with a leading “I”, “W”, or “R” (standing for “instruction read”, “memory write”, or “memory read” respectively). This single character (and a trailing colon) is followed on the line by a virtual-memory address in hexadecimal. The address must be converted into a physical address, and it is this conversion step – and much that is needed to make it happen by way of data structures and algorithms – that your code will simulate.

The skeleton code and a few trace files generated by pin will be pushed into your `gitlab.csc.uvic.ca` repository within the `a4/` directory. (The trace files are given to you in a ZIP archive named `traces.zip`.)

Your programming tasks within `virtmem.c` are:

- to modify `resolve_address()`;
- to modify the `struct page_table_entry` data structure (if necessary), modify `startup()` and `teardown()` functions (if necessary),
- add functions (if necessary)
- to implement a simulation the FIFO, LFU and CLOCK page-replacement algorithms.

Note that `resolve_address()` takes two parameters (the logical address and whether the operation on the address is a read or write) and returns one value (the physical address).

- The simulator is invoked on the command line when running *virtmem*.
- The algorithm to be used is indicated by a command-line argument (`--replace=fifo`, `--replace=lfu` or `--replace=clock`).
- The size of a simulated physical frame is indicated at the command line (`--framesize=8` specifies physical frames of size 256 bytes each, i.e., 2^8). Note that the frame size is by this forced to be a power-of-two exponent.
- The size of the simulated physical memory is indicated at the command line (`--numframes=256` specifies a simulated memory where there are 256 frames). There is no requirement that the number of frames must be a power-of-two.
- The file containing the memory trace is indicated at the command line (`--file=hello_out.txt`). These traces are provided to you. (If you are interested in traces from other programs, perhaps the course instructor could make those available, although this would require a exchange of ideas via RocketChat.)
- A progress-bar is enabled via the `--progress` command-line argument.

Some functionality is already provided for you by the skeleton code. For example, it handles the processing of command line arguments. It also handles the reading of trace files, and splitting virtual memory addresses into the page-number of offset components. (Recall that these latter details depend for correct operation upon the value provided as the frame size when running the simulator.)

As an example, `hello-trace.out` contains a textual representation of the memory operations generated by running the Unix `ls` command in my Vagrantshire. Assuming you have compiled the skeleton `virtmem.c` in your Vagrantshire, are running this from your current directory, and have the trace files unzipped in the current directory, the following will appear (without page-replacement) where the frame size is 2^{12} bytes in size, where there are 256 such physical frames, and the progress bar is printed. (The scheme below is specified as FIFO, but this is ignored in the skeleton code!)

```
./virtmem --file=hello-trace.out --framesize=12 \  
--numframes=256 --replace=fifo --progress
```

And here is the output:

```
Progress [.....] 100%  
Memory references: 211540  
Page faults: 138  
Swap ins: 138  
Swap outs: 0
```

It so happens that the trace above was “simulated to completion”. If we change the number of frames to a smaller number:

```
./virtmem --file=hello-trace.out --framesize=12 \  
--numframes=100 --replace=fifo --progress
```

then here the simulation *does not* run to completion:

```
Progress [.....] 52%  
Simulator error: cannot resolve address 0x55ae15c20008 at line 112007
```

And a word about “Swap outs” and “Swap in”:

- When a page is first loaded into a frame (i.e. within a fully-implemented operating systems), the contents of that page are either obtain from the file system (i.e. code for the program, or global variables with initial values also stored in the program’s binary) or the contents of the page are zeroed out (i.e. region of memory corresponding to the page is for uninitialized local and global data). If the physical frame corresponds to a page in which data is stored, then the contents of that page must be stored onto disk if the frame is chosen for replacement; and because we cannot reconstruct the page from the program’s binary, that frame must be stored in swap space. This action of saving a frame’s content onto disk is called a “**swap out**”.
- If there is a fault on a particular page, and if that page had previously been mapped to a physical frame, and if that frame had been swapped out, then servicing the page fault must include reading back into the memory the contents of the physical frame as it had been stored within the swap space. This action of loading a frame’s content into disk is called a “**swap in**”. *To simplify your logic, you can consider any load of a frame’s contents (including those for code or the first use of initialized global data) as a “swap in”.*

You must keep track of the number of swap ins and swap outs that would be required giving the simulations parameters and the addresses that make up the simulation.

The line number message indicates that the simulator attempted to resolve the memory access indicated by line 112007 within `hello-trace.out`, but could not do so. In this case the error results from there no longer existing any more free frames (i.e., the skeleton file does not implement page replacement!).

To sum up, you are to:

- Implement a FIFO page replacement scheme and update the appropriate global variables so `output_report()` works.
- Implement an LFU page replacement scheme and update the appropriate global variable, ditto comments about `output_report()`.
- Implement an Clock page replacement scheme and update the appropriate global variable, ditto comments about `output_report()`.
- Keep track of the number of swap-ins and swap-outs.
- Test your implementation with a variety of trace files, frame sizes and memory sizes.
- Make sure your operations work on 64-bit addresses (i.e., `long ints`). You'll get weird and hard-to-debug error messages if you depend too much upon 32-bit integers (i.e., `regular ints`).

What you must submit

- One C source-code file named `virtmem.c` containing your solution.
- One `README.md` describing (as clearly as possible) the data structures and algorithms you have used in your implementation, as well as the overall strategy of your solution.

Evaluation

During evaluation we will use the traces provided to you, but will also use traces that will not be shared beforehand.

Our grading scheme is relatively simple.

- “A” grade: An exceptional submission demonstrating creativity and initiative. The simulator runs without any problems.
- “B” grade: A submission completing the requirements of the assignment. The simulator runs without any problems.
- “C” grade: A submission completing most of the requirements of the assignment. The simulator runs with some problems.
- “D” grade: A serious attempt at completing requirements for the assignment. The simulator runs with quite a few problems.
- “F” grade: Either no submission given, or submission represents very little work.