Computer Science 360 – In Operating Systems
Spring 2017

Assignment #3

Due: Monday, April 3rd, 11:30 pm by submission to conneX
(no late submissions accepted)

Programming Platform

For this assignment you must do your work using *linux.csc.uvic.ca*. You can remotely login to this machine anywhere on campus or from home by using *ssh*.

You may already have access to your own Unix system (e.g., Linux, macOS) yet we recommend you work *linux.csc.uvic.ca* rather than try to complete the assignment on your machine for later submission to conneX. Bugs in systems programming tend to be platform-specific, and something that works perfectly at home may end up crashing on a different hardware configuration. *Your code will be evaluated on* ***linux.csc.uvic.ca****, therefore you must ensure any work done on personal laptops or desktops also works correctly on that server.*

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 programs.)

Goals of this assignment

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

- Demonstrate your work to a course Teaching Assistant for credit, and explain your chosen data structures and algorithms that appear in the source code of your solutions.

Goal: Implement FIFO, CLOCK, and LRU 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: 0x7fecad0272d0
I: 0x7fecad0272d3
W: 0x7fffa4bba4d8
I: 0x7fecad02aa40
W: 0x7fffa4bba4d0
I: 0x7fecad02aa41
I: 0x7fecad02aa44
W: 0x7fffa4bba4c
```

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 trace files are in the CSC file system at */home/zastre/csc360/a3*. You should copy the C source files into your account, but be careful copying over the trace files as they can be large.

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, LRU 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=lru* 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, although you are free to construct your own traces. (Please see the last-page appendix to this assignment for more information on generating traces.)
- A progress-bar can be 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, *ls_out.txt* contains the memory operations generated by running the Unix ls command on a directory in my account. Assuming you have compiled the skeleton *virtmem.c* in your account and are running it from your current directory, here is a run (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=/home/zastre/csc360/a3/ls_out.txt --framesize=12 \
  --numframes=256 --replace=fifo --progress
```

And here is the output:

```
Progress [............................................................] 100%
Memory references: 563939
Page faults: 239
Swap ins: 239
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=/home/zastre/csc360/a3/ls_out.txt --framesize=12 \
  --numframes=100 --replace=fifo --progress
```

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

```
Progress [...........                                              ]  19%
Simulator error: cannot resolve address 0x7f4038987c44 at line 108820
```

The line number message indicates that the simulator attempted to resolve the memory access at line 108820 within *ls_out.txt*, but could not do so. In the case the error is the result of there being no 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 LRU 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()*.
- Test your implementation with a variety of trace files, frame sizes and memory sizes.
- You may want to introduce an additional scheme such as "lruclean" which goes further a uses the "dirty" bit to guide victim-frame selection.
- 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 "int"s).

What you must submit

- One C source-code file named *virtmem.c* containing your solution.

Evaluation

Given that there are a variety of possible solutions to this assignment, the teaching team will not evaluate submissions using a marking script. Students will instead demonstrate their work to our course marker. Sign-up sheets for demos will be provided a few days before the due-date; each demo will require about 15 minutes.

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.

Appendix

Using *pin*

Pin is a program known as a binary instrumentation tool. It was created by Intel and has been made available by them on their website:

https://software.intel.com/en-us/articles/pintool/

The idea behind such tools is that they permit systems programmers to get precise data on how programs behave at run time. *Precise* here means run-time statistics at the level of individual instructions (as opposed to run-time stats at the level of, say, methods or functions or even lines of code, as these would be provided by tools such as *gprof* or *gcov*).

I have provided a shell script named */home/zastre/csc360/a3/gentrace.sh* which invokes a pintool I have created that combines both an instruction trace and a data trace. (If you would like to see the actual C++ code in *csc360trace.cpp*, then please visit me during office hours.) You can use this script to generate your own traces, but be careful! The traces can be large.

For example, to run this script on a program you write called *foo.c*, then ensure *foo.c* is compiled into an executable named *foo* then type:

/home/zastre/csc360/a3/gentrace.sh ./foo

After temporarily instrument the binary, the program will run and then a text file named *foo_out.txt* will appear in your current directory.

(Unfortunately the *gentrace.sh* script only works with executables that take no arguments. I'll try to fix that over time.)