Project 2

Memory Management Simulator

Project Assigned: November 1, 2018
Intermediate Deliverable Due: November 14, 2018 @ 11:59pm
Final Deliverable Due: November 28, 2018 @ 11:59pm

For this project, you will implement a simulation of an operating system's memory manager. The simulation will read files representing the process images of various processes, then replay a set of virtual address references to those processes using one of two different replacement strategies. The output will be various statistics, including the number of memory accesses, page faults, and free frames remaining in the system.

This project must be implemented in C++, AND it must execute correctly on the Linux machines in the Alamode lab (BB136) that runs an Arch Linux, or in a Ubuntu OS.

1 Simulation Properties

Your program will simulate memory management for a hypothetical computer system with the following attributes:

1. Pages and frames are both 64 bytes in size.
2. Main memory consists of 512 frames for a total of 32 kilobytes of storage.
3. Addresses are 16 bits long, with the ten most-significant bits representing the page or frame and the six least-significant bits representing the offset.
4. The maximum number of frames allocated to a process is static. Processes may be allocated frames until either reaching this limit or the system runs out of free frames to allocate.
5. All frames in main memory are available for use by user processes; the OS does not occupy any memory (unlike in a real computer).
6. Page tables do not occupy main memory, and reading from a page table does not constitute a memory access (unlike in a real computer).
7. No translation look-aside buffer exists (so you don't need to simulate one).
8. Processes exist for the entire duration of the simulation; if you've done the last memory access for a given process as specified in the file, it continues to occupy its current frames for the remainder of the simulation.
9. Segfaults (memory access faults) are fatal and should cause the simulation to exit immediately.

2 Simulation File Format

The simulation file specifies both the set of processes that are currently active in the system and the sequence of virtual addresses that should be accessed. Its format is as follows:

```
num_processes
process_id process_file     // The process ID and corresponding process image file
process_id process_file     // The process ID and corresponding process image file

process_id virtual_address   // PID of process and the address being accessed
process_id virtual_address   // PID of process and the address being accessed
process_id virtual_address   // PID of process and the address being accessed
process_id virtual_address   // PID of process and the address being accessed
```

Here's an example. Note that the comments won't be in the actual files.

```
2                 // 2 processes active in the system
10 process_1.txt  // Process with PID 10 and file containing its process image
42 process_2.txt  // Process with PID 42 and file containing its process image

10 0010000110011001    // Process 10 accesses address 0010000110011001
10 0010000110011010    // Process 10 accesses address 0010000110011010
10 0010000110011011    // Process 10 accesses address 0010000110011011
42 0110000110011001    // Process 42 accesses address 0110000110011001
42 0100000110011010    // Process 42 accesses address 0100000110011010
10 0010000110011001    // Process 10 accesses address 0010000110011001
...                    // Keep reading until EOF
```

The first line specifies the number of processes active in the system. You can use this value to control how many subsequent values you interpret as processes.

Each process has both a process ID and a file that contains the data that should be used as its process image. The file should be assumed to be in binary format, though you can read each byte into a `char` array.

The starter code contains an example simulation file, as well as a few dummy process images under the `inputs/` directory.

3 Command-Line Flags

Your simulation must support invocation in the format specified below, including the following command-line flags:

```
./mem-sim [flags] simulation_file.txt

-v, --verbose
  Output information about every memory access.

-s, --strategy (FIFO | LRU)
  The replacement strategy to use. One of FIFO or LRU.

-f, --max-frames <positive integer>
  The maximum number of frames a process may be allocated.

-h --help
  Display a help message about these flags and exit
```

You may (but are not required to) use the `getopt_long(3)` method for parsing these flags. As always, check the `man` page for additional information.

`--verbose`

The output required for the `--verbose` flag is described in the next section, along with the output expected when it's not set.

`--strategy (FIFO | LRU)`

This flag determines the replacement strategy that your simulation must use when either a process has been allocated the maximum number of frames (determined by `--max-frames`) or the system has no free frames available. If the flag is not provided, it should default to FIFO.

`--max-frames <positive integer>`

This flag requires a positive integer argument and specifies the maximum number of frames that can be allocated to a single process, assuming the system still has free frames available. If a process already has this number of frames, or the system has no more free frames to spare, you must replace one of the process' other pages using the replacement strategy specified by `--strategy` to bring in a new page. You will also need to vary this parameter to correctly demonstrate Belady's anomaly. If the flag is not provided, it should default to 10.

`--help`

The `--help` flag must cause your program to print out instructions for how to run your program and the flags it accepts and then immediately exit.

4 Required Output

Regardless of the flags a user passes, your simulation must always output the following information:

- The total number of memory accesses
- The total number of page faults
- The number of free frames remaining
- For each process:
  - Total number of memory accesses
  - Total number of page faults
  - The percent of memory accesses that caused a page fault
  - The resident set size of the process at the end of the simulation

If the `-v` or `--verbose` flag is set, your program must also output information about each memory reference. The required information is as follows:

- The ID of the process making the memory reference
- The virtual address being accessed
- Whether the memory access resulted in a page fault or not
- The physical address corresponding to the virtual address
- The process' current resident set size (RSS)

5 Replacement Strategies

Your memory management simulation must support two different page-replacement strategies: FIFO and LRU. Which strategy to use should be provided as a command-line flag, as follows:

- First In, First Out `(--strategy FIFO)`
- Least Recently Used `(--strategy LRU)`

If the flag is not provided, your simulation should default to using FIFO.

Both of these strategies should be implemented as they are described in your textbook. While LRU is not feasible to implement in real operating systems, your simulation has no such problem. You are free to keep track of whatever data you need to implement the two required strategies, regardless of how feasible the collection of that data would be in a real OS.

6 Getting Started

To save you some time for final exams, you'll be provided with starter code for this project. You may opt not to use it, but if you choose not to, you must implement a comparable set of unit tests that can be run using `make test` and ensure they all pass. In addition, the starter code has been tested in Linux machines only. If you want to use the starter code in other OS, it is your responsibility to modify it and make sure you run the code appropriately.

The starter code already has a `makefile` that builds everything under the `src/` directory, placing temporary files in a `bin` directory and the program itself (named `mem-sim` by default) in the root of the repository. It also includes a `make test` target that will automatically build all `_tests.cpp` files placed anywhere under the `src/` directory.

It has numerous classes declared that attempt to model the various concepts in memory management you'll need. Most are located in subdirectories of the `src/` directory. Your first task should be to skim through these files to get a handle on what is provided for you.

You should start by implementing the lowest-level classes, since the implementation of the other classes typically depend on these. Both `PhysicalAddress` and `VirtualAddress` are good places to start.

All methods that are declared in a header file have a stub implementation in their corresponding `.cpp` files. Most of these functions have unit tests already written for them, and you will be required to implement the function stubs such that all the tests pass. You are free to add additional methods and unit tests however you see fit.

To run the tests, run the following from within your repository:

```
make test
```

Most of them will fail until you implement the corresponding functionality.

7 Requirements and Reference

- Use good design. Do not code monolithic functions. You should avoid coding practices that make for fragile, rigid and redundant code.
- Use good formatting skills. A well formatted project will not only be easier to work in and debug, but it will also make for a happier grader.
- You can develop your project anywhere you want, but it must execute correctly on the machines in the Alamode lab (BB136) or in a Ubuntu OS.
- Your final submission must contain a `README` file with the following information:
  - Your name.

  - A list of all the files in your submission and what each does.
  - Any unusual/interesting features in your programs.
  - Approximate number of hours you spent on the project.
  - A couple paragraphs that explain what Belady's anomaly is and how to use your example input file to demonstrate its effects. Be sure to:
    * Define Belady's anomaly.
    * Give the command line invocations that should be used to demonstrate the anomaly with your program
    * Attempt to explain why the anomaly occurs

- To compile your code, the grader should be able to `cd` into it, and simply type `make`. Typing `make test` should run your unit tests, all of which should pass.

8 Deliverables

You are required to submit each deliverable by 11:59 on the due date. Any requirements not mentioned in an intermediate deliverable are due as part of the final deliverable.

All deliverables must be submitted to Canvas. Two entries have been created: P2-D1 and P2-Final.

8.1 Intermediate Deliverable: Due November 14, 2018 @ 11:59pm

You must submit a version of your code where:

- All unit tests in the starter code pass when running `make test`
- Your program reads a simulation file in the required format and prints:
  1. The size of each specified process in bytes
  2. Each of the virtual addresses the file contains

Zip or tar all your files into a single file with the name:

```
CanvasUserName-D1.zip (or .tar)
```

Replace "CanvasUserName" with your real username. Submit this single file to Canvas.

8.2 Final Deliverable: Due November 28, 2017 @ 11:59pm

You must submit a completed version of your program where:

- The replacement strategy can be specified using the `--strategy` flag (FIFO or LRU).
- The maximum number of frames a process may use is configurable using the `--max-frames` flag.
- All required information is present in your output.

- Details about individual memory accesses are correctly displayed when the `--verbose` flag is set.
- An input file that clearly demonstrates Belady's anomoly when using FIFO is present.
- The README contains all requested information.

Zip or tar all your files into a single file with the name:

```
CanvasUserName-Final.zip (or .tar)
```

Replace "CanvasUserName" with your real username. Submit this single file to Canvas.