CSCI-442 - Project 5 - A Simple Memory Management Simulator

1) Introduction

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 Isengard although feel free to develop on your local machine!

2) Deliverables

2.1) Deliverable 1

- All unit tests in the starter code pass when running `make test`.

2.2.) Deliverable 2

- All functionality from Deliverable 1
- The replacement strategy can be specified using the `--strategy` flag (Section 7, Section 10).
- The maximum number of frames a process may use is configurable using the `--max-frames` flag (Section 6, Section 10).
- All required information is present in the output as specified in Section 8.
- Details about individual memory accesses are correctly displayed when the `--verbose` flag is set (Section 8).
- Pass all tests `./test-my-work.sh`.

Examples of the correct output for this deliverable are provided with the starter code.

3) Grading

Grading for this project will be based on:

- Passing unit tests
- Output matching provided examples
- Output matching "hidden" grader test cases

It is vital that you follow all output requirements as mentioned in Section 8.

- Given that you have been provided with example outputs and a script to help you verify your program's outputs, no tolerance will be made for programs that do not abide by these output formatting rules and examples.

Make sure all debugging and other non-required print statements have been commented out before submitting your deliverables.

4) Requirements and Reference

- You are **required** to use the starter code provided for this project.
- To compile your code, the grader should be able to `cd` into it and simply type `make`. Typing `make test` should run the unit tests, all of which should pass. The grader must be able to do this using the `Makefile` that is provided with the starter code.
- The grader must be able to execute your program by typing `./mem-sim` from the root of your repository.
- Do not modify the directory structure provided in the starter code.
- Your project must execute correctly on Isengard.
- As long as they are present on Isengard, you are encouraged to use any C++ standard libraries you may find useful, such as `<bitset>`, `<map>`, `<queue>`, etc.
- 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.

5) Submission Checklist

Please make sure that you have done all of the following *prior* to submission:

1. Your code compiles on Isengard.
    - Run `make clean && make`. Your program should compile and "Successfully Compiled!" should print to the screen.
2. Your program should be able be invoked from the root of your repository.
    - Run `./mem-sim` from the root of your repository.
3. Your program passes all of its unit tests.
    - Run `make test` from the root of your repository.
4. Your implementation of the simulation logic and replacement strategies is finished.
    - Run `./test-my-work.sh` from the root of your repository.
    - Your solution should be identical to the provided solutions.
5. All the files required for your project have been committed and pushed to your GitHub repository.
6. `./make-submission` runs without error
7. The `zip` file made by `./make-submission` is uploaded to Gradescope correctly.

6) Getting Started

You have been provided with starter code that has some basic functionality implemented and a set of unit tests to help you with implementing the project. The starter code implements command line flag parsing and simulation file parsing, in addition to functionality from data structures that were required to implement these features. However you will still need to complete the implementation of many of the included data structures.

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.

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 how ever you see fit.

The starter code has already implemented the flag parsing functionality, and within the `Simulation` class there exists an implementation of a `print_summary` function that should be used once you have populated the `Simulation` class with the correct variables and functions.

6.1) Where to Start?

It is recommended that you start the project by implementing the functionality for the various classes that have been provided for you. You are able to check your work on your implementations using the provided unit testing functionality, discussed in the following section.

Many of these data structures are dependent on each other. For example, think about the relationship between virtual addresses and physical addresses, or pages, page tables, and processes. Thinking about these things, perhaps drawing a diagram to see how they all fit together, will help you better understand how to implement the project. This will also help you better understand how all these pieces need to fit together for your operating system to perform memory management.

While the command line flag parsing functionality has been implemented for you, you should take a look at the `FlagOptions` struct that stores information retrieved from command line input. This struct is passed into the `Simulation` class via its constructor, and the values contained in it should be used for various aspects of your simulation. For example, the `FlagOptions` struct contains variables that let you know if you should be printing the verbose output (Section 8), what the maximum number of frames for a process should be (Section 6), or what replacement strategy you should be using (Section 7).

6.2) Unit Tests

The starter code contains a number of unit tests to help you implement the various data structures in the project. To run the tests, run the following from within your repository:

`make test`

Most of them will fail until you implement the corresponding functionality. You can run only certain tests by executing the `make test` command with a `TEST_FILTER` option:

`make test TEST_FILER="Test Case Pattern"`

For example, to run only the `Process` class's test cases, you would type:

`make test TEST_FILTER="Process*"`

To run a specific test, say the `TotalSize` test from the `Process` test cases, you would type:

`make test TEST_FILTER="Process.TotalSize"`

6.3) Output Testing

The starter code also has example outputs and a script that you can run to verify your solution with the provided outputs. The example outputs themselves are located under `tests/`, and the verification script is named `test-my-work.sh`.

To use the script, from the root of your repository, type these commands into your terminal of choice:

```
chmod +x test-my-work.sh
./test-my-work.sh
```

The `chmod +x` command only needs to be run once per computer.

The sections below discuss the more technical aspects of the project, so it is suggested that you read them carefully.

7) 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 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. The default maximum number of frames is 10, however the user may input a maximum frames value when executing the simulation (Section 10).
6. All frames in main memory are available for use by user processes; the OS does not occupy any memory (unlike a real computer).
7. Page tables do not occupy main memory, and reading from a page table does not constitute a memory access.
8. No translation look-aside buffer exists, so you do not need to simulate one.
9. 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.
10. Segmentation faults (memory access faults) are fatal and should cause the simulation to exit immediately.
    - There are two kinds of segfaults: invalid page segfaults, and invalid offset segfaults.
        - Invalid page segfaults occur when a process is trying to access a page that it does not have access to.
        - Invalid offset segfaults occur when a process is trying to access an offset that does not exist as valid data in a given frame.
11. If a process has not reached its maximum number of allocated frames, it should pick the first available frame.
12. The replacement strategies in the simulation are *local* replacement strategies. Once a process has reached its maximum number of allocated frames, it needs to pick one of its pages that is in main memory to replace.

8) 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 discussed in Section 10.

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 you need to implement the two required strategies, regardless of how feasible the collection of that data would be in a real OS.

9) Required Output

Examples of all outputs can be found within the starter code under `tests/`.

9.1) Not Implemented for you

`–verbose`

If `––verbose` or `-v` is specified, your simulation must 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).

Here is an example of what this should look like for one memory reference:

```
PID 10 @ 0000010011101111 [page: 19; offset: 47]
    -> PAGE FAULT
    -> physical address 0000000000101111 [frame: 0; offset: 47]
    -> RSS: 1
```

It is recommended that you take advantage of the << operator overloads written for the virtual and physical address classes when printing this information.

9.2) Implemented For You

This section is provided for your reference. All the logging and output functionality in this section has been written for you.

Unless the `––csv` or `--c` flag is input, your program should always output this information to the screen:

- 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.

Here is an example of how this should look:

```
Process  10:  ACCESSES: 30     FAULTS: 19     FAULT RATE: 63.33    RSS: 10
Process  42:  ACCESSES: 31     FAULTS: 29     FAULT RATE: 93.55    RSS: 10
Process  99:  ACCESSES: 53     FAULTS: 53     FAULT RATE: 100.00   RSS: 10

Total memory accesses:             114
Total page faults:                 101
Free frames remaining:             482
```

10) Simulation File Format

This section is provided as a reference. All the file input parsing has been written for you.

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
...                             // Keep reading until EOF
```

Here is 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 "process file" field is the filename of the process image. It is a filename that points to the location of the process image relative to the location of the `mem-sim` binary file that you run using `./mem-sim`.

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

11) Command-Line Flags

This section is provided as a reference. All the command line input parsing has been written for you.

Your program 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.

-i, --file-verbose,
    Print process size and virtual addresses when reading in file.

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

-c, –-csv

The output required for the `–-csv` flag is described in Section 8.

-v, –-verbose

The output required for the `–-verbose` flag is described in Section 8.

-s, –-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. A strategy must be supplied when using this flag. If this flag is not provided, your program should default to using FIFO.

-f, –-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. If the flag is not provided, it should default to 10.

-h, –help

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

12) Collaboration Policy

This is an **individual project**. All code you submit should be written by yourself. You should not share your code with others.

Please see the syllabus for the full collaboration policy.

> ***Warning***
>
> **Plagarism will be punished harshly!**

13) Access to Isengard

Remote access to Isengard is quite similar to ALAMODE, but the hostname is `isengard.mines.edu`.

For example, to `ssh` into the machine with your campus MultiPass login, use this command:

```
$ ssh username@isengard.mines.edu
```

Note: you need to be on the campus network or VPN for this to work. If you are working from home, use either the VPN or hop thru `imagine.mines.edu` first.