1 Memory Management

This week, the designing of several memory page replacement algorithms and comparing their effectiveness against certain memory access patterns will be explored. As with the first part of the project, **please** take a look at the new code provided and ask questions if certain things do not make sense. The majority of the code has been completed for you, though it is still recommended to understand how the pieces fit together.

1.1 Setting up the environment

Please see the section on how to setup the environment in the handout for part 1 of the project, as most of the same instructions apply, specifically the sourcing of the `source.me` file and, if it has not been done, compiling the linked list library.

1.2 Compiling the memory request class

Once the environment is set up, the memory request class can be compiled. The memory request class is needed so that there can be a way to provide the algorithms with a memory request and a way to keep track of whether or not a memory request was a hit or a miss. Also this class helps with the book keeping of the data in the pages for data logging. To compile the memory request class, run the commands:

```
$ cd page_req/
$ make
```

1.3 Compiling the Memory Management Unit

Now that the memory request class is compiled, the memory management unit(or MMU) class can be worked on. The memory management unit class is in the folder `isu_mmu` and it is an emulation of how memory is handled at the hardware level. Here in the MMU is where the implementations of the page replacement algorithms will reside. The FIFO replacement algorithm has been implemented as an example. Note that there are some TODOs in the code that are in the functions `isu_mmu_page_fetch`, `isu_mmu_page_rep_lru`, and `isu_mmu_page_rep_clock`. These are for the students to implement, and in the case of `isu_mmu_page_fetch` there are some basic instructions on what is expected. For the other page replacement algorithms, follow the FIFO example on the general flow of what the algorithms should do. Once the changes are made, to compile the MMU run the commands:

```
$ make
```

This will generate an object file that will be used in the compilation of the test program.

1.4 Compiling the test program

With the MMU and memory request classes compiled, the MMU can now be tested. A testing program `mem_test.c`, in the root folder of the project, is provided. To compile this program, run the command:

```
$ make
```

This would compile the previously mentioned pieces and the `mem_test` program. Run the program without arguments to see what arguments it takes.

1.5 Task for this lab

The task for this lab is for the groups to implement the different page replacement algorithms, specifically Least Recently Used, and the Clock algorithms. Using the test program, multiple page replacement algorithms can be tested with multiple memory access patterns. The expected outputs from each page replacement algorithm with each memory access pattern will eventually be given in the `answers` folder for comparison. Comment in your lab report the outputs of each algorithm for each memory access pattern and compare them against the other algorithms. Are those the expected outputs/behavior? Which seems to perform the best over all the memory access patterns?

1.6 Going further

After the page replacement algorithms have been implemented, it is recommended that the students play around with the size of L1 cache(or if desired, L2 cache, and RAM as well) and observe the difference in total time it took to handle all memory requests, and the difference in hit rate. Comment on what was experimented with and what observations were made in regards to the effects of changing the cache sizes.

1.7 Extra credit

For extra credit, you can implement your own page replacement algorithm. Again, feel free to get creative, and follow the provided examples to get started. One option would be to implement the Not Recently Used algorithm. Each new algorithm implemented will be an extra 5% with a maximum of 25% extra credit.

1.8 License