Project 3: Virtual Memory Manager

Benjamin Zydney & Yifan Lu
School of Electrical Engineering and Computer Science, Pennsylvania State University
CMPSC 473: Operating Systems Design & Construction
December 7, 2022

PennState College of Engineering | ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Design of the Manager + Implementation Choices / Restrictions:

Our memory manager was implemented in two disjoint but similar parts: the FIFO page replacer and the third-chance page replacer.

For the FIFO page replacer, we first defined data structures for each page (`page_info`) and defined a `queue` data structure, with pointers to the `head` and `tail`, as well as the `size` and `max_size` of the queue. We then wrote basic initialization functions `new_page`, `new_node`, and `new_queue`, which initialized a page, node, and queue, respectively. `enqueue` and `dequeue` were implemented to add and remove pages from a queue, and a `get_page` function was implemented to find whether or not a page was present in a queue.

We then wrote a `handler_fifo` function to handle `SIGSEGV` signals. In this function, we first calculate the address that caused the `SIGSEGV`. Then, we retrieved the error, and determined the necessary protection needed for the given fault. We then checked if the page was in the queue, and if not, appropriately evicted and protected pages from the queue. The added page was then protected, and `mm_logger` was called to log the action.

For the third-chance page replacer, we again defined a data structure for each page (`third_page_info`), with additional storage for the `R` and `M` bits. However, instead of using a regular queue, we used a `circular_queue` data structure, since that was what was recommended in the textbook. Similar initialization functions `new_third_page`, `new_circular_node`, and `new_circular_queue` were defined. `enqueue_circular` and `remove_page_circular` were defined to add and remove pages from a circular queue, and a `get_circular_page` function was implemented to find whether or not a page was present in a circular queue.

We then wrote a `handler_third_page` function to handle `SIGSEGV` signals. In this function, we first calculate the address that caused the `SIGSEGV`. Then, we retrieved the error, and determined the necessary protection needed for the given fault. We then checked if the page was in the queue, and if not, appropriately evicted and protected pages from the queue. The added page was then protected, and `mm_logger` was called to log the action.

Unimplemented Issues / Problems with Implementation:

After testing the program with the sample inputs and with some custom test cases that we created, which included multiple corner cases, we were unable to find any unimplemented issues. However, there were a few problems that we encountered in the process of designing the project, but have been fixed since then.

We found that this project, like project 2, was a good bit easier than project 1, so our main problem was learning the third-chance algorithm required to complete this project.

The only other non-trivial issue we had was with adapting our implementation to work on MacOS. When writing the program, we first implemented FIFO on Ben's Windows laptop, and then tested it on the computers in W135. Everything seemed to work correctly, but when we ran it on Yifan's MacBook, there were multiple syntax errors with the `ucontext.h` library. Changing the header declaration to `sys/ucontext.h` fixed some of the issues, but there were still problems with the register macros. After doing some digging on the issue on Stack Overflow and asking one of the TAs, we learned that MacOS has different names for the register macros than Linux and Windows. We were able to fix the issue for MacOS by changing the names of the registers to the correct versions, but this caused the program to not work properly on the W135 computers. Yifan kept a backup of the working version on MacOS to run tests on, but the Linux version was kept on GitHub.

Things Learned:

Coming into this project, we were well-versed with the first-in-first-out replacement policy, but were rather unfamiliar with the third-chance replacement policy. Even though we had watched the lectures and looked over all of the slides, we did not find it to be as intuitive to understand or to implement as the allocation policies in project 2, since it was not covered in class. The project document was fairly helpful in explaining the basic concept, but we found that the explanation of algorithm in the textbook (Operating System Concepts, 10th edition, Silberschatz et al.) was much more clear. Additionally, we learned how to use the `ucontext.h` library and the `mprotect()` and `sigaction()` functions within `mman.h` and `signal.h`, which we used extensively in this project to properly manage segmentation faults. After implementation, we feel that we have a strong knowledge of all of the libraries and replacement policies used, but are curious if our implementation could be made any more efficient, either in time or in space.

Distribution of Workload:

The workload for this project was divided fairly evenly. Most of the work was done either together in-person or over text, which ensured that only one of us was committing work at a time so that no work would be lost or overridden by a merge. When we ran into errors that we needed to debug, such as the ones mentioned above, we would both talk about the potential causes of the errors and possible solutions. We would then both try to fix the errors, and if either of the proposed fixes worked, we would commit that change.