Marshall Lowe (mhl5178)
Christopher Kurcz (cjk6056)
CMPSC 473
12/7/2022

Project 3 Design Report

mm_init() Design:

The purpose of the function is to initialize everything for our manager. So, all that happens in this function is that we initialize all of our global variables that store things like the number of frames for the manager, the pointer for the start of virtual memory, the policy we have to follow for our manager, and more. Then, we set our sigsegv_handler to handle any segfaults that occur, and set the protection on the virtual memory to none so that a fault occurs on any access.

sigsegv_handler() for FIFO policy Design:

In the case of the FIFO policy, the sigsegv_handler's job is to catch access to pages of memory that are not currently in physical memory, or pages that may possibly require a permission change. We used C's sigaction functionality to register this handler to catch the SIGSEGV signal. The use of sigaction allows us to use the parameters that we supplied in our handler's definition to retrieve vital information about the fault.

When the signal triggers our handler, the first step is to use the siginfo_t structure to retrieve what address the fault occurred in. From this address, we were able to use division to figure out the page number of the page that the fault occurred in, as well as what the offset is inside the page. We then check our queue to see if the page that caused the fault is already in physical memory. If it is, we know that we do not have to evict a page from our physical memory (or our queue), and can get its physical address from the struct field of the already existing page's frame number multiplied by the page size and added to the offset. If we do not have a reference to the faulted page number, we initialize a structure for that page and begin the process of checking if we have to evict a page from physical memory.

Since we keep track of the amount of pages in physical memory, as well as the number of available frames, we can quickly tell if the frames are full or not. If the frames are not full, we do no evictions and simply add the page's data structure to our queue. If the frames are full, we evict from the head of the queue as that is the oldest page and add the new page to the tail. We then are able to retrieve the physical address by checking what frame the old page lied in multiplied by page size and adding the offset to that. Furthermore, we also are able to retrieve the page number of the evicted page, as well as whether or not a writeback is needed by checking the data structure of our evicted page. To ensure that a reference to the evicted page is caught in the future, we mprotect that page's memory with PROT_NONE.

The next step is to determine the type of fault that occurred. Using the ucontext_t structure, we are able to extract the base type of the fault from the REG_ERR general register. This information tells us if the fault was due to a read or a write. We then check our internal data structures such as the queue and the referenced page to determine the specific fault type that occurred. We then utilize this specific fault type to determine the type of protections that we have to give the new page so that it faults correctly on next access. Right before

sigesgv_handler returns, it calls the mm_logger function with all the information that it collected to log the data to the output file.

sigsegv_handler() for THIRD policy Design:

In the case of the third chance policy, our sigse_handler works almost entirely the same as with the FIFO policy, except for a few exceptions. Firstly, our queue is set up to be a circular link list, so that the last element in the list points to the head, and there is a pointer to the next page up for eviction called the “hand”. Also, pages store a referenced and modified bit, along with a boolean keeping track of whether or not they have taken their third chance to keep from being evicted.

However, the main difference is in how we choose what page to evict when the queue is full and a new page is trying to be added. Starting from the page pointed to by the hand of the queue, we keep going through all the pages in the queue and follow the third chance policy to figure out which page will get evicted. First, if the current page has a reference bit of 1, we set it to 0 and then set the protection of that page to none, so that any future references to that page will reset the bit to 1. Second, if the referenced bit is 0 and the modified bit is 1, we check if the page has taken a third chance or not. If it has, then it gets evicted, otherwise we set the thirdChanceTaken boolean to true and move to the next page in the queue. Third, if a page has a referenced and modified bit of 0, we evict the page.

The other difference is when we check for what type of fault occurred. Using the ucontext_t struct to figure out if it was a read or a write. If it was a write, and the referenced page has permission to write, we know that it is a type 4 error. If it was a read and the page has permission to read or write, then we know that it is a type 3 error.

Distribution of Work:

Marshall created most of the queue and page structs with their functionalities, and implemented the FIFO policy. Chris updated the queue and page structs to also be able to handle the THIRD policy, as well as implementing the THIRD policy. However, both of us together made the design decisions for the entire project, and we also both helped each other fix bugs and make minor changes to the code each of us wrote.