Matthew Kelleher
December 7, 2022
CMPSC 473

Project 3 Report

DESIGN OF SCHEDULER:

Date Structures:

- **MEM_MANAGER** holds all variables and structures related to the memory manager. Stores the policy type, a pointer to virtual memory, the virtual memory size, how big a page is, and a pointer to the page table structure.

- **PAGE_TABLE** holds all variables and structures related to the memory managers' page table. Set up like a linked list. Stores the total size, the number of current pages, and a pointer to the head and tail of the page table

- **PAGE** holds variables related to a page inside of the page table. Set up like a node in a linked list. Stores a pointer to the address of the page and the next page, the page's virtual page number, physical page number, reference bit, modified bit, and if it has been marked for a third chance.

Methods:

- **new_mem_manager** called at the start of the program to initialize a new mem_manager structure. The policy, pointer to virtual memory, virtual memory size, number of frames, and page size are passed as parameters and set to their respective variables in a new mem_manager structure. A new empty page table is created and attached to the mem_manager. Next, mprotect() is called on the virtual memory to set it to have no access, so first access to any part of memory will cause a SIGSEGV signal. Finally, a SIGSEGV handler is created and attached to the handler method to catch any SIGSEGV signals from mprotect().

- **handler** automatically called when there is a SIGSEGV signal on the virtual memory. First using the siginfo_t pointer the address of the fault is determined. From this, we find the virtual page number, the base address of the page, and the offset of the address from the base address. Next, we check if the page is already in the page table using find(), If it is not, we then determine if the page table is full. If it is, we call pop() to evict a page according to the replacement policy, keep track of the evicted page's physical number, if it needs to be written back,

and set it to have no read/write access. We then create a new page with the information determined and set its reference bit as 1. If the page was already in the page table, we set the reference bit as 1 and unmark it as on its third chance. From here we now determine if a read or write access cause the SIGSEGV signal. If it is a write we give the page read and write permissions, set the modified bit as 1, and determine the fault type. It is fault type 1 if it is not a present page in the page table, fault type 2 if it is a write to a read only page (ref_bit = 1, mod_bit = 0), and fault type 4 if it has read and write access but the reference bit is 0 (ref_bit = 0, mod_bit = 1). If it is a read, then we give the page read-only permissions and determine the fault type. It is fault type 0 if it is not a present page in the page table and fault type 3 if it already has read permissions, but the reference bit was 0. The mm_logger is then called with the determined virtual page number, fault type, evicted page number, write-back, and physical address. Finally, if it is a new page, it is added to the page table using push().

- **push** called to add a page to the end of the page table. Takes in a pointer to a page structure to add to the end of the page table. If the page table is empty, it sets it as the head and tail, if not it is set as the tail and the old tail points to the new page. Finally, the page table number of pages variable is increased.

- **pop** called to evict a page from the page table according to the replacement policy to make room for a new page. Returns NULL if the page table is empty. If the policy is first-in-first-out, the first page element in the page table is removed, the next node becomes the new head, the page table number of pages variable is decreased, and the removed page is returned. If the policy is third chance, then we loop through the page table till we find a page who ran out of chances. This is determined by first checking the reference bit, if it is 1 then it is set to 0 and the page loses all read and/or write access so the bit can then be reset later, and we go to the next page in the page table. If it is 0, we checked the modified bit or if it has been marked as out of chances. If the modified bit is 1, we mark the page as on its third chance and go to the next page, if the modified bit is 0 or the page is marked as on its third chance then the page is removed from the page table, the next node becomes the new head, the page table number of pages variable is decreased, and the removed page is returned.

- **find** called to find a page in the page table. Takes in a virtual page number of the page you want to find as a parameter. Iterates through the page table checking if any page has a matching virtual page number. If a page does it is returned, if not and we go through the whole table then NULL is returned to indicate the page is not in the page table.

ISSUES NOT IMPLEMENTED:

All APIs have been implemented, passing all given test cases and all personal test cases.

DISTRIBUTION OF THE WORKLOAD WITHIN THE TEAM:

The project was done entirely by myself, all functions were created and implemented on my own.

THINGS LEARNED:

- How to manage memory using a page table
- How to implement first-in-first-out and third-chance page replacement policies
- How to use mprotect() to set access flags to cause SIGSEGV signal
- How to implement sigaction() to catch SIGSEGV signals from mprotect()
- How to access information from siginfo_t and ucontext_t