Tejas Marimuthu
Ayman Awal

Project 3 Report

mm_init()

In this function we are saving all the arguments in global variables that we created. We also pass the same variables into our vmm.c through our custom made function **getData()** for ease of access.

In this function we create a sigaction that is associated with our handler and we call a function sigaction to replace the segfault handler with our handler that we created. We also **mprotect** the virtual memory and set the protection to None so nothing can access it.

handler()

We acquire the fault address and fault operation through the **siginfo_t *si** and **void *ucontext**. With this information we are able to get the page address, offset and page number. We mprotect that page according to the fault operation. If it is a read fault, we mprotect with **PROT_READ**, else we mprotect with **PROT_WRITE**.

Next we check to see if the page exists or not in our page table with the **checkPageExistence(int page)** function. This will help us determine what type of **mm_logger** to use.

We have a current_permission variable which helps us to determine what mm_logger to use as well. If it is third chance we call getPermission(int page) to get the original permission of the existing page in the table. If it does not exist it returns -1.

We have a variable called evicted_page which determines if we evicted a page or not. If it -1 we did not evict it, otherwise we receive the page we evicted from **addPage(int page, int permission, void *address).**

After receiving the value for evicted_page, we now decide which mm_logger function to run depending on the permission, the page existence and previous permission. This will allow us to know what type of fault occurred. If we did evict a page, we set the mprotect that page to **PROT_NONE.**

Memory Manager functions

Struct Element: Used to store page information

getData(): Gets called in the interface.c file and accepts the same arguments as mm_init() function. Created for easy access to the variables in the vmm.c.

getLength(): Returns the number of pages in the table

deleteLastPage(): Deletes the tail node in the page table. Used in FIFO.

checkPageExistence(): Determines if the page already exists in the page table or not.

getFrame(): Takes page as an argument and returns the position of our page in our page table.

getWriteBack(): Returns write_back, a global variable. Tells us if we need to write back or not.

getModifiedBit(): Returns **m** bit that tells us if we need to write back or not.

getPermission(): Takes page as an argument and iterates through the table and returns the permission of that page.

evict(): We evict the first page that the clock algorithm comes across that has the reference bit and tcv both equal 0. We implemented an infinite while loop and returns when we find a page that satisfies this condition.

addPage(): Takes in page, permission and address. We initially use the checkPageExistence(int page) function to see if the page is in our page table already or not. If the output is 0, it means it does not exist in our table and we check if eviction is required or not. If it is, we call deleteLastPage() for FIFO. If it is Third Chance, we call the evict() function. If no eviction is required for FIFO, we increment the curr_frame.

Post eviction check, if it is FIFO we assign all the arguments to a struct element temp accordingly and make the newly added page the new head of our linked list. We add from the left and evict from the right for FIFO.

Post eviction check, if it is Third Chance and no pages were evicted, that means the table is not completely full. So we iterate through the table and check for the first available spot and insert the page there. Otherwise we replace the page that we got from our evict function stored in evicted_page and set all the new page values there.

If the table has become full after adding a page, we can begin the clock algorithm and set the global variable **position** variable to the head. The variable allows us to keep track of what position the clock is at.

If the page already exists in the table, we will reset the reference bit, denoted by **r,** to 1. If the modified bit, **m,** has the value 0, we will set it to the value of the **permission**. We do not change the value of the modified bit unless we are writing to a read-only page.

addFrame(): Creates a temporary element and adds it to the end of head pointing to NULL.

createList(): Creates an empty circular list for Third Chance. Called in the init function.