Advanced Operating Systems
Report

Point total: 20
Point values are unique to each project assignment.

Note: One page (approximately 600 words). Discuss any non-trivial routines implemented to complete the task and high level motivations behind your design decisions. Discussions on approaches taken to debug your solution on your own are also appreciated.

Part 1 Report - 10pts
Part 2 Report - 10pts

Breakdown of the 10pts:

1. Approach and methods - 5pts total. 0pts if what you wrote about in your report does not reflect the code you submitted. You must provided analysis or reasoning for why a change or code segment was required to get full points.
2. Calculations - 3pts testing results and any extra test cases you wrote yourself
3. Discussion - 2pts observations from test results, why do you think your solution worked/ didn't work, key takeaways.

Solution:

Part A - Pagefault Handler

Approach and methods -

I handle the page fault by writing pagefault handler function to correctly handle all possible types of page faults. First, After getting the current process, I use the rcr2() function to get the address of the page that caused the fault. And then, the code walks through the linked list of memory-mapped regions and checks if the faulting address falls within the bounds of any of the regions. The code would find a valid memory-mapped region and then check if the page fault was caused by a write operation. Moreover, the permission bits needed for the mappages() call are determined based on the protection bits of the mmap region. If the mmap region does not have write permission, then the page table entry is marked with only user permission. Otherwise, it is marked with both user and write permission. After that, the mappages() function is used to map the page to the faulting virtual address in the current process's page table. I check if the region type of the page is MAP-FILE or not. And then the code would need to check the status of its file descriptor for the file and clear the dirty bit of the page table entry when the region type is MAP-FILE. Otherwise, the code would kill the flag indicating that the process has to be killed and cleaned up because it cannot continue its execution after the page fault.

Calculations - I passed all the tests. I designed a test for this function to test whether reading unmapped memory regions can cause page faults.

Discussion -

The function is designed to handle page faults that occur when a user process attempts to access a memory page that is not currently in physical memory. The code includes several debugging statements that print information about the faulting process and address as well as an error message, being helpful in diagnosing and debugging issues with the operating system. Moreover, in my extra test function, if the test code attempts to access a memory address that is not valid, it can possibly trap into the page fault function to avoid crash of the systems. On the other hand, it is possible to fail to release resources and decrease system performance if we can't handle page faults well. Hence, we should carefully implement the page fault handler and ensure that it correctly handles all possible types of page faults.

Part B - Anonymous Mappings

Approach and methods -

The mmap function is used to map a region of memory into the address space of a process. According to the instruction of this assignment, I modify the original code and add conditions to check whether the fd argument is valid. If it is valid and the flag is MAP-FILE, the program would allocate a duplicate file descriptor using fdalloc(), and then call filedup() to increment the open count of the file and set it for the new memory region. On the other hand, in the munmap() function, I have to close a duplicated file descriptor using fileclose() when memory regions are removed. Moreover, I implemented msync() as the kernel function to check whether any regions have been allocated to the process. Through iterating over all the memory mapped regions of the process, it checks if the address is marked as dirty in the page table. If it is marked as dirty when a page fault occurs, it means the memory region has been modified and needs to be written back to the file on disk. In addition, These changes involved pagefault handler function having to set page table flags and check mmap region type.

Calculations -

I passed all the tests. By modifying test 7 as test 9, I set a 1MB as file size for this stress test to test whether my codes handle errors well.

Discussion -

content In msync() code, it performs a linear search through the mmap regions to find matching regions. However, for a large number of mmap regions, I think this way could become less efficient. To avoid a large number of mmap regions, I checked whether the number of memory regions exceeds the maximum number of memory regions in the mmap function. From these testing codes, I observed that trying to write the read-only page could cause page fault. Moreover, I understand that the mapped memory region allows direct access to the file content after the file is memory mapped to the process's address space using the mmap() function. Without calling msync(), the changes made to the memory mapped region may not be written back to the file on disk. On the contrary, with calling msync(), any changes made to the memory-mapped content will be reflected in the original file, keeping consistency between the memory and file data. In the future, if I have more time, I would implement optimization in msync() code such as using binary search to quickly find the correct region.