Assignment 3 Report

1 Introduction [3']

Overview: In assignment 3, we are asked to implement mmap and pagefault handle program.

1. Each process has its own virtural memory, which can be used to store the data in the running process. In real world, data are stored in the disk. However, CPU can only access data in the memory. Therefore, we need to move the data from disk into memory. When moving a data from disk to memory, we cannot simply move it into the memory. We also have to preallcoate some space for the new data, and mark which process needs that data. That's where mmap is designed for. When the system encounters a mmap request, it first allocate some space for the data, and mark which process will use the data.
2. By implementing this assignment, I have learned more about the OS, as well as how mmap in the OS works. Moreover, I get to know more about the OS kernel functions.
3. When implementing the mmap, the main challenge I encountered is the difficulty in reading the codes. At first, I do not know where I can start reading all these complicated codes, and I also don't know what I should do for the task. However, after discussing with some of my friends and refering to some books and codes, I finally understand what shall I do in this assignment. I think the main difficulty I met is the toughness in reading all these codes and understand the requirements of the tasks. However, after the implementing, I found that actually understanding is the most difficult part. Meanwhile, lazy allocation in xv6 also makes some implementation a bit hard.

2 Implementation [5']

In this section, I will detailly explain my implementation.

2.1 VMA

This is the implementation of VMA structure. First, mapped marks whether the virtural memory space has been used. Addr records the physical space the VMA corresponds to. Len records the length of the VMA area. Prot is the privilege(read/write). Flag is the type of the mapping. If it is shared, then the modification will be written back to the main memory. Offset is used to know the actual area the VMA maps to. F is the file descriptor. MAPCNT records how many maps are mapped to the page.

2.2 mmap

For mmap process, we first need to call mmap() function. After that, we can fetch argument. According to the fetch result, we can get the parameter. We then cope with the results according to the valid result of argument. If argument valid, we can then preallocate some VMA space for the data. Then, after refering to the information in the vma structure, we can increment the reference count (mapcnt) and return the virtual address.

2.3 PageFault

For pagefault, we first need to use rstval() to get the virtual address of where pagefault happens. Then, we just need to make sure whether the access is crossing-boundary. Then, if not crossing-

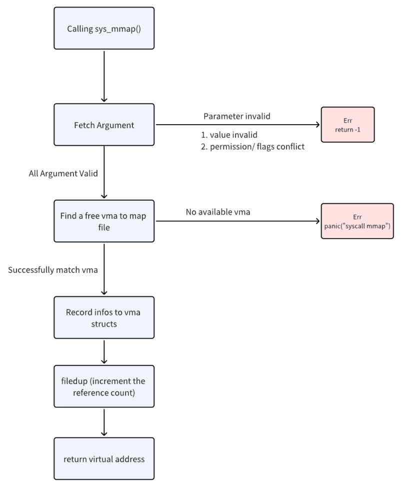
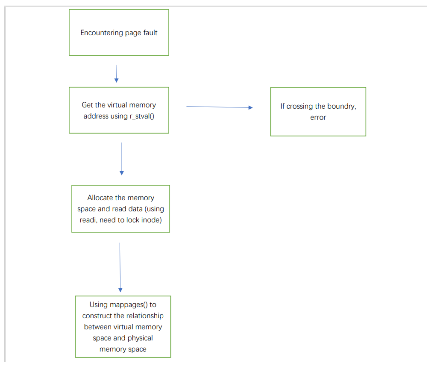


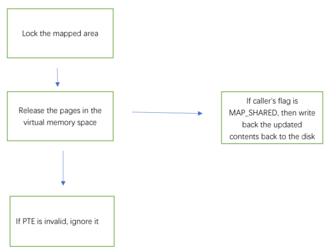
Figure 1:

boundary, we can get where the virtual memory actually maps to. Then, we can allocate some space for the incoming data, and get the data from the disk. We can use readi() to get the data(remember to lock the process!). At last, we can use mappages to get the relationship between virtual memory and the physical memory.



Encountering page fault
Get the virtual memory address using r_stval()
If crossing the boundry, error
Allocate the memory space and read data (using readi, need to lock inode)
Using mappages() to construct the relationship between virtual memory space and physical memory space

Figure 2:



Lock the mapped area
Release the pages in the virtual memory space
If caller's flag is MAP_SHARED, then write back the updated contents back to the disk
If PTE is invalid, ignore it

Figure 3:

2.4 munmap

munmap is the reverse process of mmap. In short, it releases the virtual memory space according to the addr. First, it locks the mapped area. Then, it releases the pages mapped in the virtual memory area. If the flag of the caller is MAPSHARED, we should write the contents in memory back to the disk. Moreover, if PTE is invalid, we should ignore it.

2.5 Bonus

For bonus part, we need to further implement fork(). When forking the father process's id and the blank space in the virtual memory area, we also need to copy the father process's mapping area information. My implementaion ensures that the child has the same mapped regions as the parent. For exit part, I inserted some codes to make sure that the area that the file maps will not be used.

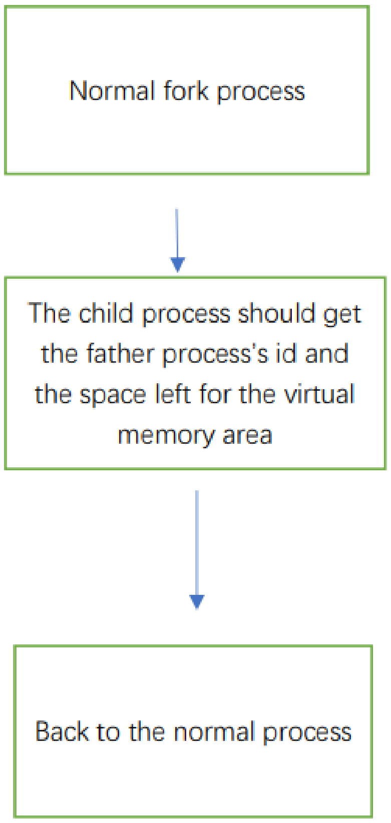

Normal fork process
The child process should get the father process's id and the space left for the virtual memory area
Back to the normal process

Figure 4:

3 Test [2']

In this section, I will explain which part of my implementation helped me pass the test.

3.1 mmap f

My implementation of vma structure and mmap function(Including page fault handling part).

3.2 mmap private

The flags record in the vma structure and mmap function(Including page fault handling part).

3.3 mmap read-only

The prot record in the vma structure and mmap function(Including page fault handling part).

3.4 mmap read/write

The prot record in the vma structure and mmap function(Including page fault handling part).

3.5 mmap dirty

mmap and mummap function(Including page fault handling part).

3.6 mmap two files

mmap and mummap function(Including page fault handling part).