# Assignment Report: Title

In this report, when citing a reference, such as the xv6-book, feel free to use footnotes[1].

Your report should follow the template with the following section structure.

**No page limitation**

## 1 Introduction [3']

When facing an mmap request, the system would first check for permissions and flags conflicts. Then, it searches for free space in the process's address space and sets up a new VMA with the requested attributes. If the mapping is shared, it duplicates the file and maps it to the process's memory. When handling a page fault, the system finds the corresponding VMA and maps the file to the process's memory.

In this assignment, I have implemented the mmap() and munmap() calls in the xv6 operating system. This involved setting up a new VMA, checking for conflicts, mapping the file to the process's memory, and handling page faults by mapping the file to the process's memory. Moreover, I managed to work mmap() when fork() is involved;

There are several difficulties I have encountered during the whole process:

1. Though inspired by filewrite() function, I found that the function provided by xv6 itself can only start from places where index=0.
2. When implementing the fork process, my computer refused to directly pass the values of vma[i] in p to vma[i] in np. And I do not know why the function does not work until I modify the exit() function.

## 2 Implementation [5']

### 2.1 VMA

The **VMA** structure consists of the following parameters:

1. Original parameters in mmap():*addr, length, prot, flags, offest*;
2. The file we are going to read;
3. *used* to mark whether this vma is used;
4. *mapped* to mark whether there is a page fault that happened and handled in this vma (will be used in bonus).

### 2.2 mmap

I first fetch all the arguments, proc, and the file and check their validity. Then I try to find an empty vma to map . Once an empty vma is found, I will find a free space, and store all these arguments into the corresponding structs. Here I do a little bit of modification to the vma[i].prot. The original prot posted in is the permission of mmap, its relationship with the **PTE** permission is that it equals with right shifting **PTE** for one digit. I also add **PTE_U** and **PTE_V**. Finally, I returned the virtual address I got for it.

[1]Reference: https://pdos.csail.mit.edu/6.828/2005/readings/pdp11-40.pdf

### 2.3 PageFault

When a page fault occurs, I will get the corresponding virtual address, then search the vma where the address lies in. If the matching vma is found, the pagefault is handled by allocating a new physical page (mem), mapping it to the faulting virtual address using mappages(), and then mapping the corresponding file data to the allocated physical page using mapfile().

### 2.4 munmap

The function first retrieves the virtual address and length from the arguments. Then, it searches for the corresponding vma in the process's vma array. If the vma is found and the address is valid, it checks if the vma has the MAP_SHARED flag set. If it does, it calls my_filewrite() to write the modified data back to the file. After that, it unmaps the specified range of virtual memory using uvmunmap().

#### 2.4.1 About my_filewrite()

The function's implement is mostly base on the function filewrite() provided by the system. By changing the off from f.off to an argument off. The original one always start from the index=0, while we need a particular start place.

### 2.5 Bonus

Fork is easy, for every vma we use, I will filedup() its corresponding file, copy the whole vma's value to np. When exit, we need to leave all the mapped areas in case for memory leak.

## 3 Test [2']

Briefly discuss which part of your implementation helped you pass each test. (Several sentences for each subsection should suffice.)

### 3.1 mmap f

### 3.2 mmap private

### 3.3 mmap read-only

### 3.4 mmap read/write

### 3.5 mmap dirty

### 3.6 mmap two files

All the above used mmap() and munmap(), for **3.1, 3.2, 3.4, 3.6**, the implementation of pagefault is also used