CSC3150 Assignment 3 Report

Student ID: 121090684

Name: Yang Yuzhe

1 Introduction

1.1 Overview

When the xv6 operating system faces an `mmap` request, it operates in a structured manner to map a file into a process's address space. Initially `sys_mmap()` is invoked, the primary task of `sys_mmap()` is to locate an unused region within the process's address space suitable for mapping the file. Once this region is identified, a VMA structure is created and added to the process's table of mapped regions. This VMA includes details like the address, length, permissions, and a pointer to the `struct file` for the file being mapped. Crucially, `sys_mmap()` increases the file's reference count to ensure that the file structure remains available even after the file is closed. This operation essentially creates a bridge between the file and the process's memory, allowing the file to be accessed and manipulated as if it were part of the program's own memory space. When a process then tries to access this mapped memory, it may cause a page fault if the physical memory hasn't been allocated yet. The xv6 system handles this by allocating a physical page, reading the necessary data from the file onto this page, and then mapping it into the user address space. When the file is no longer needed, `sys_munmap()` is called to unmap the memory. If certain flags are set, it also performs file write-back operations. This mechanism allows direct user-mode access to files or devices, significantly enhancing I/O performance without the need for kernel-user mode data copying.

1.2 Accomplishment

- `mmap` a file into a process's address space.
- `munmap` a file from a process's address space.
- PageFault Handle
- `fork`

1.3 Difficulties

1.3.1 Unfamiliar with xv6 system

Being unfamiliar with the xv6 system posed a significant challenge. It required dedicating time to thoroughly read and understand the xv6 source code to grasp its implementation and inner workings. This step was crucial to effectively develop and debug the system calls needed for the assignment.

1.3.2 Environment

The vitural machine provided by the course is too slow for VS code IntelliSense and development. So I chose to use my own MacBook for development. Another challenge was using MacBook with an ARM architecture, which is incompatible with running xv6 on qemu directly. Therefore, I downloaded risv64 toolchain by `x86` architecture of `brew`, which is translated by `Rosetta2`. And then, I could compile xv6 on my MacBook with modified `Makefile`: `TOOLPREFIX = /opt/riscv-gnu-toolchain/bin/riscv64-unknown-elf-`

1.3.3 Debug

For a better development, I used `gdb` to debug the xv6 system, which is also downloaded by `x86 brew`. after I setup `launch.json` and `tasks.json` (in Appendix) in VS code, I could debug xv6 with breakpoints on my MacBook.

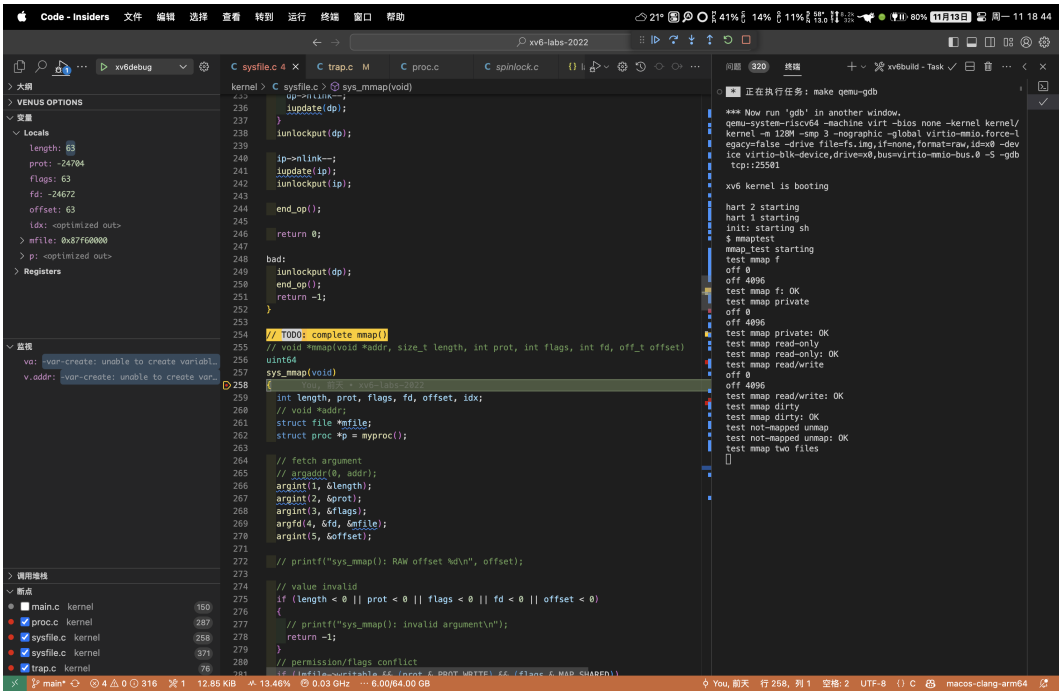

Figure 1: Debug xv6 on MacBook

2 Implementation

2.1 VMA

- `uint64 addr`: Starting virtual address of the mapped region.
- `int length`: Length of the mapped region.
- `int prot`: Permissions flags indicating the protection level of the mapped region.
- `int flags`: Additional flags specifying attributes of the mapping.
- `int fd`: File descriptor representing the mapped file.
- `int offset`: Starting offset within the file.
- `struct file *mfile`: Pointer to the corresponding file structure. It helps in managing the reference count of the file, ensuring it remains accessible even after closing the file descriptor.
- `struct inode *ip`: Pointer to the corresponding inode (index node) of the mapped file. This is crucial for file operations and maintaining file metadata.

2.2 mmap

The `sys_mmap` function in xv6 is designed to handle memory mapping requests. It begins by fetching arguments: `length`, `prot`, `flags`, `fd`, and `offset`. These are parameters essential for defining the memory mapping, such as the size of the mapping, protection flags, file descriptor, and the offset in the file.

The function first checks if the input values are valid, returning an error (-1) if any value is invalid. It then checks for conflicts between the requested permissions and the flags set for the file mapping. If a conflict is detected, such as requesting write permissions on a non-writable file, it returns -1.

Upon passing these checks, the function searches for a free VMA within the process's address space. This is done by iterating through the process's VMA array (`p->vma`) and looking for an entry with a length of zero, indicating it's unused.

Once a free VMA is found, the function sets its properties based on the parameters: the address (set to the size of the process's memory space, `p->sz`), length, protection flags, file descriptor, and offset. It also duplicates the file structure (`filedup`) to increment its reference count, ensuring the file remains available as long as it's mapped.

The process's memory size is then increased by the rounded-up `length` of the mapping to account for the new mapping. Finally, the function returns the start address of the mapped area, indicating successful mapping. If no free VMA is found, it returns -1, signaling failure to map.

Overall, `sys_mmap` effectively sets up a memory mapping in the process's virtual address space, linking it to a specific file or device, based on the given parameters and the process's current state.

The flowchart for `sys_mmap()` is shown in Figure 2.

2.3 PageFault

This code segment handles page faults in user-space processes within the xv6 operating system. Upon detecting a page fault caused by either loading (13) or storing (15) operations, it retrieves the virtual address associated with the fault and allocates a chunk of memory.

The code then searches the process's VMA to find a matching entry for the faulted virtual address. If a match is found, it allocates a page of memory for the VMA, maps the corresponding file to that memory page, and updates process attributes.

If no matching VMA is found, indicating an invalid memory access, an error message is printed, and appropriate error handling is performed.

Finally, regardless of success or failure, the code performs a final check to ensure the process hasn't been terminated. If the process has been killed, it exits. This code primarily manages the dynamic allocation of memory and handles page mapping during page faults in user processes.

The flowchart for `Page Fault Handling` is shown in Figure 3.

2.4 munmap

The function is designed to implement the memory unmap operation (`munmap`) in user space. It begins by obtaining a pointer to the current process's structure and retrieving the virtual address and length parameters from the system call. After ensuring the validity of the parameters, the function iterates through the process's VMA array, identifying the VMA that includes the specified virtual address and recording its index.

Subsequently, the function handles cases where the VMA is writable and a shared mapping exists. If the VMA is writable and shared, modifications are written back to the corresponding file. Following this, the function calculates the number of pages and calls the `uvmunmap` function to perform the actual unmapping, freeing the associated resources.

The function then updates the VMA information, adjusting the length, address, offset, and closing the file if necessary. Finally, the function returns 0 to indicate successful completion of the unmapping operation. In essence, this function locates the region to be unmapped based on VMAs, performs operations according to VMA attributes, and completes the unmapping of user space memory.

The flowchart for `sys_mmap()` is shown in Figure 4.

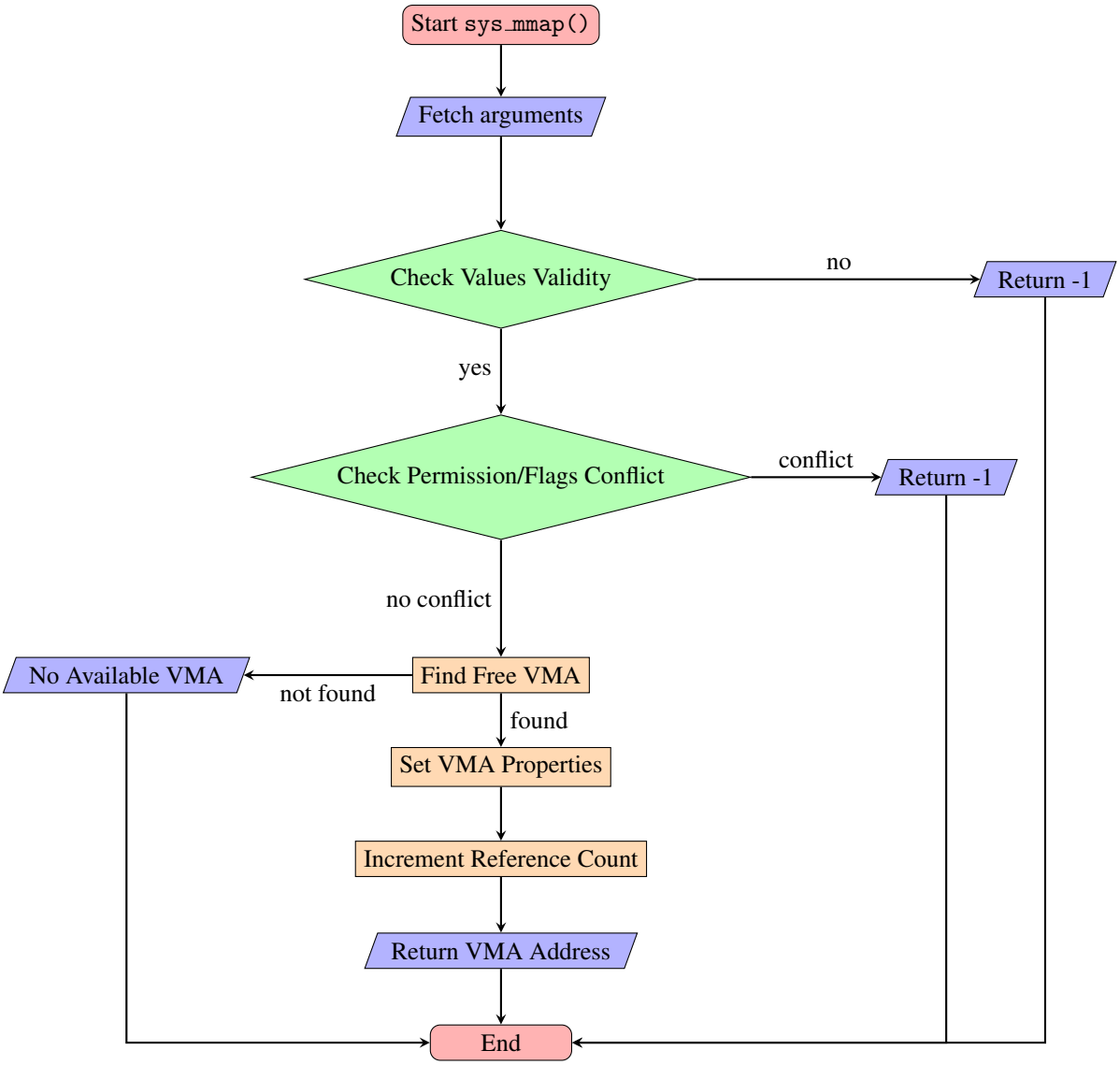

Figure 2: Flowchart for `sys_mmap()`

2.5 fork

The `fork` function creates a new process by duplicating the parent process into a child process. The child process is an exact copy of the parent, with its own memory space, registers, file descriptors, and other resources.

The existed part of this function has allocated a new process and copy its page table and user memory. In order to implement this function, we need to copy the VMA from parent process to child process.

The implementation of copy VMA is similar to `sys_mmap()` function when we asssign all the needed value to the new VMA. Additionally, we need to check whether the VMA is valid. If it is valid, we need also copy its file resources and increment the reference count.

3 Test

The screenshot of program result has shown in Figure 5.

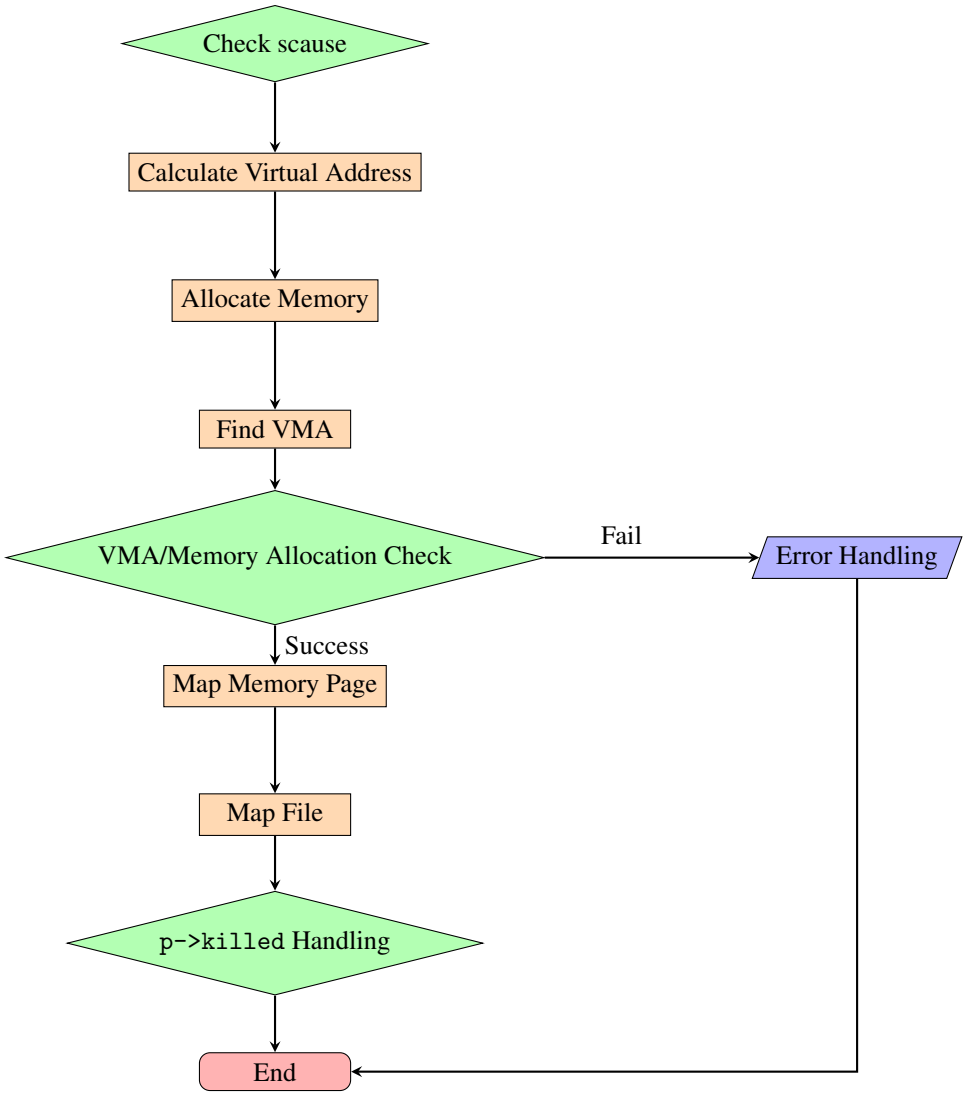

Figure 3: Flowchart for Page Fault Handling

3.1 mmap f

The implementation of the file-backed memory mapping in `mmap f` was key. By correctly setting up the VMA structures and ensuring that the file contents were lazily loaded into memory upon access, the system was able to handle page faults effectively. This allowed the mapped file contents to be accessed as if they were part of the process's memory.

3.2 mmap private

For `mmap private`, the implementation ensured that changes to the mapped memory did not affect the underlying file or other mappings of the same file. This was achieved by handling the `MAP_PRIVATE` flag appropriately, which involved creating copies of the pages on write operations, thereby maintaining the privacy of the modifications.

3.3 mmap read-only

In `mmap read-only`, the challenge was to enforce read-only permissions on the mapped memory. This was done by setting the appropriate permissions in the page table entries. Any attempts to

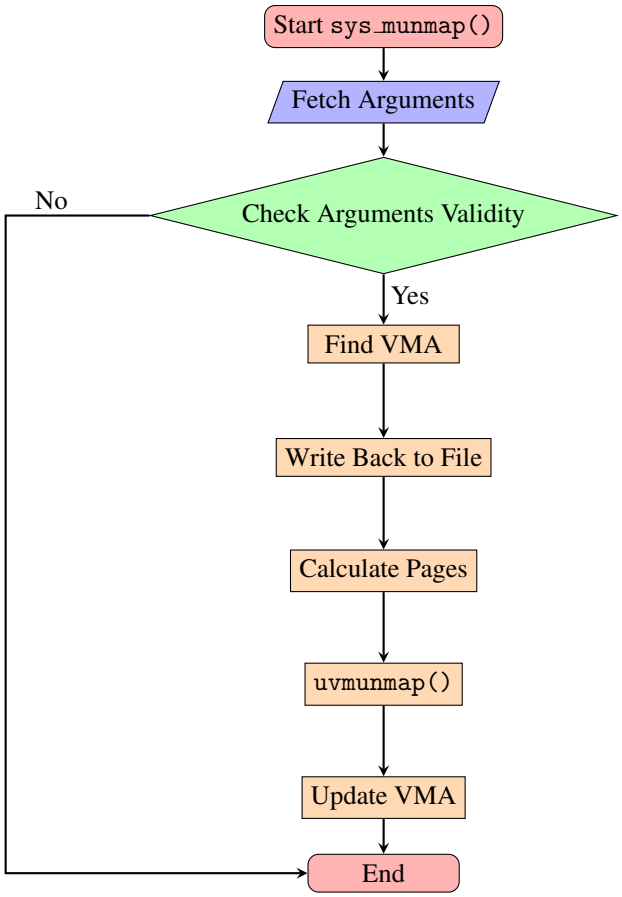

Figure 4: Flowchart for `sys_munmap()`

write to these pages resulted in page faults, which were handled by not allowing modifications, thus maintaining the read-only integrity of the mapped area.

3.4 mmap read/write

For `mmap read/write`, the key was to allow both reading and writing operations on the mapped memory. This involved setting up page table entries with both read and write permissions and ensuring that any modifications were correctly handled, either reflected back to the file (for `MAP_SHARED`) or kept private (for `MAP_PRIVATE`).

3.5 mmap dirty

The `mmap dirty` test required handling pages that had been modified (marked as dirty). The implementation ensured that these dirty pages were appropriately written back to the file when necessary, especially during `munmap` or process exit, thus preserving the changes made to the mapped memory.

3.6 mmap two files

Successfully passing the `mmap two files` test involved correctly handling multiple file mappings simultaneously. This required the implementation to manage multiple VMA structures and ensure that each mapping was independent and interacted correctly with its respective file, even when multiple files were mapped into the process's address space.

A VS code GDB Setup

```
~/L/M/3/D/C/C/C/c/C/xv6-labs-2022 make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel
 -m 128M -smp 3 -nographic -global virtio-mmio.force-legacy=false
-drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-dev
ice,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ mmaptest
mmap_test starting
test mmap f
off 0
off 4096
test mmap f: OK
test mmap private
off 0
off 4096
test mmap private: OK
test mmap read-only
test mmap read-only: OK
test mmap read/write
off 0
off 4096
test mmap read/write: OK
test mmap dirty
test mmap dirty: OK
test not-mapped unmap
test not-mapped unmap: OK
test mmap two files
off 0
off 0
test mmap two files: OK
mmap_test: ALL OK
fork_test starting
off 4096
off 0
off 0
off 0
off 4096
fork_test OK
mmaptest: all tests succeeded
$
```

Figure 5: Program Result

```
{
  "name": "xv6debug",
  "type": "cppdbg",
  "request": "launch",
  "program": "${workspaceFolder}/kernel/kernel",
  "stopAtEntry": true,
  "cwd": "${workspaceFolder}",
  "miDebuggerServerAddress": "127.0.0.1:25501",
  "miDebuggerPath": "/usr/local/bin/gdb",
  "MIMode": "gdb",
  "preLaunchTask": "xv6build"
},
```

Figure 6: `launch.json`

```
"tasks": [
    {
        "label": "xv6build",
        "type": "shell",
        "isBackground": true,
        "command": "make qemu-gdb",
        "problemMatcher": [
            {
                "pattern": [
                    {
                        "regexp": ".",
                        "file": 1,
                        "location": 2,
                        "message": 3
                    }
                ],
                "background": {
                    "beginsPattern": ".*Now run 'gdb' in another window.",
                    "endsPattern": "."
                }
            }
        ],
        "group": "build"
    },
```

Figure 7: `tasks.json`