Assignment Report: Memory Mapping in xv6

Chaoren WANG - 122090513

1 Introduction [2']

This assignment requires implementing the 'mmap' and 'munmap' system calls within 'xv6', a Unix-like educational operating system. The goal of this project is to support memory-mapped file operations, enabling processes to access files directly within their memory space. The 'mmap' call maps a file segment into a process's virtual memory, allowing efficient memory sharing and inter-process communication, while 'munmap' removes this mapping.

The project's scope was limited to modifying four files: 'proc.c', 'proc.h', 'sysfile.c', and 'trap.c'.

2 Design [5']

To implement 'mmap' and 'munmap', I extended 'xv6' by adding virtual memory area (VMA) management, page fault handling, and cleanup operations across the specified four files.

2.1 Virtual Memory Area (VMA) Management

In 'proc.h', I implemented the 'vma' structure to manage memory regions mapped into each process's address space. Each 'vma' entry contains fields such as 'addr', 'length', 'prot', 'flags', 'file', 'offset', and 'used'.

Figure 1 showed the 'vma' structure, illustrating the purpose of each field in managing memory-mapped regions.

```
struct vma {
  uint64 addr;                // Starting virtual address
  int length;                 // Length of mapping
  int prot;                   // Protection bits
  int flags;                  // Mapping flags
  struct file *file;          // Mapped file / device
  int offset;                 // Offset into file
  int used;                   // Whether this VMA is used
};
```

Figure 1: Structure of 'vma' in 'proc.h' file

2.2 'mmap' System Call

The 'sys_mmap' function in 'sysfile.c' maps a file into a process's memory. It starts by retrieving essential arguments (virtual address, length, protection level, flags, file descriptor, and offset). Then, it validates them to ensure they are compatible with both file permissions and the mapping requirements specified by the system. After validation, the function searches for an available slot in the Virtual Memory Area (VMA) to allocate the memory segment. Finally, it fills the VMA structure with detailed information and increments the file's reference count, establishing the mapped memory region within the process's address space.

2.3 'munmap' System Call

The 'sys_munmap' function in 'sysfile.c' handles the removal of mapped memory regions by checking the VMA associated with the specified address range. If the memory region was mapped with 'MAP_SHARED' and has been modified, it writes any changes back to the file using 'writei'. It then reduces the file's reference count, closes the file if needed, and unmaps the pages from memory.

2.4 Page Fault Handling

Page faults are handled in 'trap.c'. When a page fault occurs within an 'mmap' region, the faulting address is identified through 'r_stval'. A new physical memory page is allocated via 'kalloc', initialized to zero, and mapped to the faulting virtual address. If the page is backed by a file, the relevant data is loaded from the file into the allocated page. Finally, the page table entry is configured according to the specified protection settings ('PROT_READ', 'PROT_WRITE', 'PROT_EXEC').

2.5 Page Alignment

In 'xv6', virtual addresses for 'mmap' regions must be page-aligned to ensure efficient memory management, this helps CPU to load entire page without break. The 'PGROUNDUP' and 'PGROUNDDOWN' macros are used to align addresses to the nearest page boundaries, which is important for accurate mapping and unmapping of pages.

2.6 Fork Handle (Extra Credit)

To support 'mmap' in 'fork', the child process copys the parent's 'vma' regions, with each file's reference count incremented to maintain access. In the child's page fault handler, new physical pages are allocated for faults in 'mmap' regions to ensure data separation. Also, the 'exit' function in 'proc.c' is modified to add a VMA clear part, unmap the pages from page table and close the file.

3 Environment and Execution [2']

3.1 Environment

The project was developed within an 'xv6' virtual machine provided by the course ('CSC3150_a3_xv6.qcow2'), as showed in Figure 2.

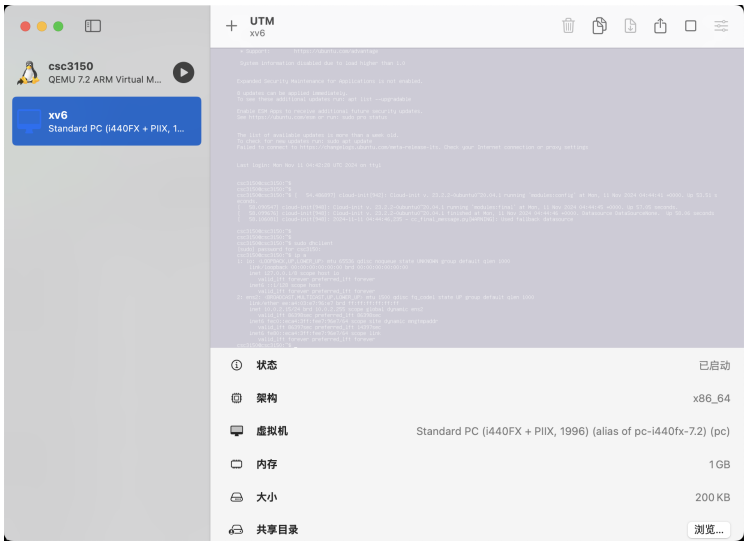

Figure 2: Running virtual machine 'xv6' in UTM

3.2 Execution

To compile and run the system, the ‘make qemu’ command was executed, followed by testing with the ‘mmaptest’ program. The test suite included cases such as ‘mmap f’, ‘mmap private’, ‘mmap read-only’, ‘mmap read/write’, and ‘mmap dirty’. The ‘mmaptest’ results, shown in Figure 3, confirm that all tests and bonus tests passed successfully, validating the functionality of the implemented system calls.

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ mmaptest
mmap_test starting
test mmap f
test mmap f: OK
test mmap private
test mmap private: OK
test mmap read-only
test mmap read-only: OK
test mmap read/write
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
test mmap two files: OK
test mmap offset
test mmap offset: OK
test mmap half page
test mmap half page: OK
mmap_test: ALL OK
fork_test starting
fork_test OK
mmaptest: all tests succeeded
$
```

Figure 3: ‘mmaptest’ successful execution results

4 Conclusion [2’]

This assignment provided hands-on experience into virtual memory management for operating system. By implementing ‘mmap’ and ‘munmap’ calls in ‘xv6’, I gained a deeper understanding of the memory management, particularly in address alignment, and page fault handling logic.

The extra credit task of managing ‘mmap’ regions in ‘fork’ extended my understanding of process isolation and file reference counting.

Overall, this assignment enhanced my knowledge of operating system concepts, which are fundamental for understanding how modern operating systems manage memory and support inter-process communication.