Assignment Report: Memory Mapping in xv6

Meng Ziqi - 120020153

1 Introduction [2']

This assignment focuses on implementing memory-mapped files using the `mmap()` and `munmap()` system calls within the `xv6` operating system. Additionally, it required handling page faults and enhancing process management functions ('fork', 'freeproc', and 'exit') to support memory mapping. Memory-mapped files allow a file to be mapped directly into a process's memory space, enabling efficient file access by treating file content as if it were part of the program's memory. This assignment also provided a chance to explore memory sharing and lifecycle management of mapped memory across processes.

2 Design [5']

2.1 Virtual Memory Area (VMA) Structure

The `vma` structure was designed to manage regions of mapped memory. Each `vma` represents a contiguous memory region with attributes such as starting address (`vastart`), size (`sz`), protection flags (`prot`), and mapping flags (`flags`). The file pointer (`f`) and offset allow each `vma` to track the specific file and file offset corresponding to the mapped memory.

2.2 mmap() System Call

The `sys_mmap` function implements the `mmap()` system call. This function fetches parameters like `addr`, `sz`, `prot`, `flags`, `fd`, and `offset` using `argaddr` and `argint`, validates them, and ensures the requested memory size aligns to a page boundary. It then finds an available `vma` entry in the process structure, initializes it with the provided parameters, and duplicates the file descriptor for the file being mapped. Finally, the function returns the starting virtual address of the newly mapped region.

2.3 munmap() System Call

The `sys_munmap` function handles unmapping a previously mapped memory region. It fetches the base address and length of the region to be unmapped and iterates through the process's VMAs to find the relevant region. For regions mapped with `MAP_SHARED`, any modified pages are written back to the file. `uvmunmap` is used to unmap pages from the page table, and the VMA is updated or split as needed. The file descriptor is closed if the entire region is unmapped.

2.4 Page Fault Handling in usertrap

In `usertrap`, page faults are managed by lazily allocating physical memory pages. When a fault occurs, the function identifies the faulting address and searches for the relevant VMA. If found, it allocates a page, reads the file contents into it, and maps it to the process's virtual address space with the appropriate permissions. This method supports efficient file access and resource management by loading pages on demand, thus optimizing memory usage.

2.5 Fork Handle for Memory-Mapped Regions

To ensure memory-mapped regions are correctly shared with child processes, I modified the `fork` function. Upon process creation, the child inherits the parent's VMAs, and the file reference count for each mapped file is incremented to keep track of shared mappings. Instead of sharing physical pages, the page fault handler in the child process will allocate new pages as needed, providing copy-on-write-like behavior without actual shared memory pages, which simplifies the implementation and avoids complex synchronization.

2.6 Resource Cleanup in freeproc and exit

In the `freeproc` and `exit` functions, I enhanced the cleanup logic to correctly release memory-mapped regions upon process exit, while avoiding a complete reimplementation of these functions. In `freeproc`, the modifications included iterating through each valid `vma` entry to unmap pages and invalidate the entries, ensuring all mapped regions are freed when a process exits. In `exit`, I added code to close file descriptors associated with `vma` entries, as memory-mapped regions require explicit file closure to accurately track file usage. These updates ensure that resources are handled efficiently, preventing memory leaks or dangling pointers.

3 Environment and Execution [2']

3.1 Environment

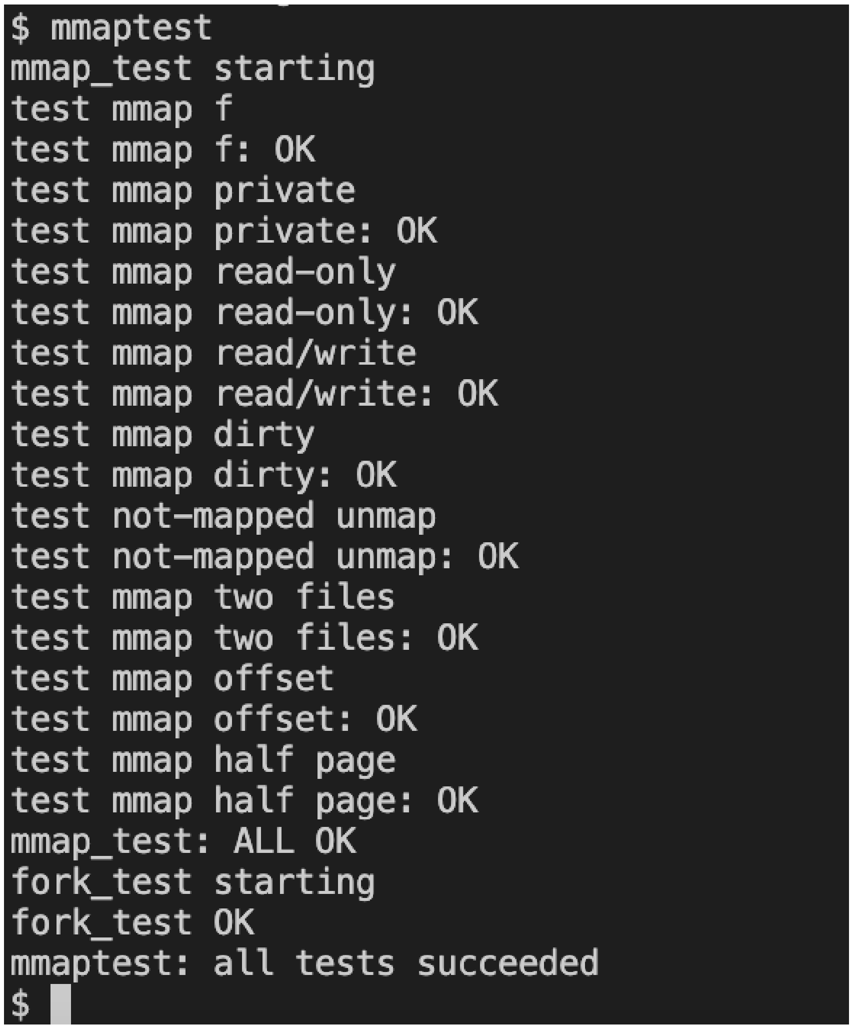
The project was developed and tested in the `xv6` development environment provided, which runs in a QEMU virtual machine. All code modifications were limited to the specified files: `proc.c`, `proc.h`, `sysfile.c`, and `trap.c`.

3.2 Execution and Testing

The project was compiled and run using the `make qemu` command, which starts `xv6`. The functionality was verified using `mmaptest`, which tested the `mmap` and `munmap` operations under various scenarios. Additional tests verified memory sharing and cleanup in `fork`, `exit`, and `freeproc` functions to ensure that child processes correctly inherit mappings and resources are freed on process termination. All test cases passed, confirming the correctness of the implementation.

4 Conclusion [2']

This assignment deepened my understanding of memory management within operating systems, particularly with memory-mapped files, lazy page allocation, and resource lifecycle management. Implementing `mmap` and `munmap` in `xv6` provided insight into memory sharing across processes, while modifications to `fork`, `usertrap`, and cleanup functions introduced me to process-level resource management. I gained valuable experience with page fault handling and learned how to efficiently manage memory and file resources in a simplified OS environment.



```
$ 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 1: testing results