

# CS3500: Operating Systems

## Lab-7: Implementing Memory-Mapped Files with mmap and munmap in xv6

October 14, 2025

### 1 Introduction

UNIX systems provide `mmap` and `munmap` system calls that enable applications to exercise precise control over process address space management. These system calls facilitate memory sharing between processes, enable mapping of files directly into process memory regions, and serve as building blocks for sophisticated page fault mechanisms like garbage collection schemes covered in our lectures. In this lab assignment, you will extend the xv6 operating system by implementing `mmap` and `munmap` capabilities, with particular emphasis on file memory mapping.

#### 1.1 Before You Start Coding:

- Study Chapter 4 of the xv6 book covering page table mechanisms and virtual memory architecture
- Examine the following xv6 source files:
  - `kernel/vm.c` — houses virtual memory subsystem code
  - `kernel/proc.c` — handles process lifecycle operations
  - `kernel/trap.c` — manages interrupt handling and page fault processing
  - `kernel/file.c` — provides file system abstraction
- Reference the RISC-V privileged architecture specification for comprehensive technical details

### 2 Lab Setup

Begin this lab by obtaining the xv6 source code and switching to the designated branch:

```
$ git checkout c14b6396f8c70f5e22a454510c2a3d0b35757a32
$ make clean
```

- Move the provided `mmaptest.c` file (available on Moodle) into the `xv6-riscv/user` folder to ensure it can be compiled for testing in this lab.
- Configure your docker environment to use the updated repository directory
- Should you encounter any complications, examine existing system call implementations or utility functions from the base version and adapt them according to your specific requirements

**Important:** If Task-1 has not been completed (i.e., `_mmaptest` has not been added to `UPROGS` and the `mmap`/`munmap` system calls have not been declared or defined), then invoking `make qemu` will lead to compilation errors.

## 3 System Call Overview

### 3.1 mmap System Call

The function signature for `mmap` is as follows:

```
void *mmap(void *addr, size_t len, int prot, int flags,
           int fd, off_t offset);
```

While `mmap` supports numerous invocation modes, this lab focuses exclusively on a subset of features pertinent to file memory mapping. You may make the following assumptions:

- `addr` will consistently be zero, indicating that the kernel should determine the virtual address for file mapping
- `mmap` yields the selected address, or `0xffffffffffffffffffff` upon failure
- `len` specifies the byte count for mapping (this value may differ from the actual file size)
- `prot` specifies memory access permissions including read, write, and execute capabilities (assume `prot` contains `PROT_READ` or `PROT_WRITE` or both)
- `flags` takes one of two values: `MAP_SHARED` (changes must be synchronized to the file) or `MAP_PRIVATE` (changes remain local to the process)
- `fd` represents the file descriptor for the target file
- `offset` remains zero (indicating mapping begins at file start)

### 3.2 munmap System Call

The function signature for `munmap` is as follows:

```
int munmap(void *addr, size_t len);
```

The `munmap` function must eliminate memory mappings within the specified address range. Should the process have modified memory mapped with `MAP_SHARED`, these changes must be synchronized to the file before unmapping. While `munmap` calls may affect only partial regions of an existing mapping, you may assume they will target either the beginning, end, or entirety of a region (without creating gaps in the middle).

## 4 Implementation Requirements

### 4.1 Lazy Page Table Population

Your solution must populate page tables on-demand through page fault responses. Specifically, `mmap` operations should neither allocate physical memory nor perform file reading. Instead, implement these operations within page fault handling routines in (or invoked by) `usertrap`, similar to the copy-on-write lab. This lazy approach ensures rapid `mmap` operations for large files and enables mapping files that exceed available physical memory.

Processes mapping identical `MAP_SHARED` files need **not** share the same physical pages.

### 4.2 Process Exit Handling

Upon process termination, any changes made to `MAP_SHARED` regions must be synchronized to their corresponding files, simulating explicit `munmap` invocations.

## 5 Tasks

### 5.1 Task 1: Basic System Call Setup

1. Insert `_mmapttest` into `UPROGS` within the Makefile
2. Incorporate `mmap` and `munmap` system calls to enable `user/mmapttest.c` compilation
3. Initially, configure both `mmap` and `munmap` to return error codes
4. Constants like `PROT_READ` are predefined in `kernel/fcntl.h`
5. Execute `mmapttest`, expecting failure at the initial `mmap` invocation

### 5.2 Task 2: VMA (Virtual Memory Area) Management

Maintain records of process-specific `mmap` operations:

1. Create a data structure representing the VMA (virtual memory area). This structure must store address, size, access permissions, file information, and other attributes for each memory-mapped region
2. Given xv6's lack of dynamic kernel memory allocation, implement a fixed-size VMA array with allocation from this array as required. Sixteen entries should suffice
3. Integrate the VMA data structure into the process structure within `proc.h`

### 5.3 Task 3: Implement mmap

Develop the `mmap` functionality:

1. Locate an available address space region within the process for file mapping
2. Insert a VMA entry into the process's mapping table
3. Configure the VMA to reference a `struct file` pointer for the mapped file
4. Ensure `mmap` increments the file's reference counter to prevent premature deallocation upon file closure (reference: `filedup`)
5. Execute `mmapttest`: the initial `mmap` operation should complete successfully, but subsequent memory access will trigger page faults and terminate `mmapttest`

### 5.4 Task 4: Page Fault Handling

Implement page fault response for memory-mapped regions:

1. Load 4096 bytes from the corresponding file into the allocated page
2. Establish the mapping in user address space
3. Utilize `readi` for file reading, providing the appropriate offset parameter (ensure proper inode locking/unlocking for `readi`)
4. Configure page permissions accurately
5. Execute `mmapttest`: execution should progress to the initial `munmap` operation

## 5.5 Task 5: Implement munmap

Develop the `munmap` functionality:

1. Identify the VMA associated with the target address range and remove the designated pages (utilize `uvmunmap`)
2. When `munmap` eliminates all pages from a mapping, reduce the reference count for the associated `struct file`
3. For modified pages in `MAP_SHARED` mappings, synchronize changes to the file before unmapping. Consult `filewrite` for implementation guidance
4. Optimally, your solution would synchronize only `MAP_SHARED` pages that were actually modified. The RISC-V PTE dirty bit (D) indicates page modifications. However, since `mmaptest` doesn't verify dirty bit usage, you may synchronize pages without examining D bits

## 5.6 Task 6: Process Exit Handling

Update `exit` to remove all process memory mappings as though `munmap` were invoked. Execute `mmaptest`, which will ensure that all tests up to “test mmap two files” should succeed (if implemented correctly), though “test fork” may still fail.

## 5.7 Task 7: Fork Handling

Modify `fork` to replicate parent memory mappings in the child process:

1. Remember to increment reference counts for each VMA’s `struct file`
2. During child process page fault handling, allocating new physical pages rather than sharing with the parent is acceptable.
3. Execute `mmaptest`: All tests should now pass.

## 5.8 Task 8: Implement Shared Mmap Support

Develop the shared mmap functionality:

1. Implement a shared page registry with locking and reference counting to track shared physical pages.
2. Extend `sys_mmap()` to correctly handle `MAP_SHARED`, incrementing reference counts for reused mappings and creating new shared entries when needed.
3. Modify the kernel page fault handler to lazily allocate shared physical pages, read file content once, and map shared pages on faults within shared VMAs, while preserving original handling for private VMAs.
4. Manage shared mapping reference counts properly during `fork()`, `munmap()`, and process exit to avoid premature page deallocation.
5. Execute `mmaptest` to confirm that all basic mmap tests continue to pass to ensure backward compatibility.
6. Execute `mmapsharedtest`, which tests for:
  - mmap syscall existence and error handling
  - basic shared mmap functionality
  - shared physical page behavior across forks and concurrent process writes

## 5.9 Task 9: Final Testing

Execute `usertests -q` to verify overall system functionality remains intact.

## 6 Expected Output

Upon completion, your implementation should produce output similar to this:

```
$ mmaptest
test basic mmap
test basic mmap: 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 lazy access
test lazy access: OK
test mmap two files
test mmap two files: OK
test fork
test fork: OK
test munmap prevents access
usertrap(): unexpected scause 0xd pid=7
    sepc=0x924 stval=0xc0001000
usertrap(): unexpected scause 0xd pid=8
    sepc=0x9ac stval=0xc0000000
test munmap prevents access: OK
test writes to read-only mapped memory
usertrap(): unexpected scause 0xf pid=9
    sepc=0xaf4 stval=0xc0000000
test writes to read-only mapped memory: OK
mmaptest: all tests succeeded
$ mmapsharedtest
mmapsharedtest start
test mmap syscall exists
mmap syscall error handled
mmap syscall: ok
test basic map_shared
basic map_shared: ok
test shared physical pages
parent mapped 0x0000000000006000 ok
child mapped 0x0000000000007000
child: shared page success
parent: shared page success
shared_task: passed
mmapsharedtest done
$ usertests -q
usertests starting
test copyin: OK
test copyout: OK
...
usertrap(): unexpected scause 0xf pid=6642
    sepc=0x43ea stval=0x3f012000
OK
test lazy_copy: OK
ALL TESTS PASSED
```

## 7 Implementation Hints

- Begin by establishing system call infrastructure and fundamental framework components
- Leverage existing virtual memory management routines as implementation templates
- Exercise meticulous care with file reference counting to prevent resource leaks
- The lazy allocation approach is fundamental to achieving optimal performance
- Adopt an incremental testing strategy - establish basic functionality before implementing advanced features
- Employ `printf` debugging techniques to trace program execution flow
- Analyze how current system calls implement comparable functionality

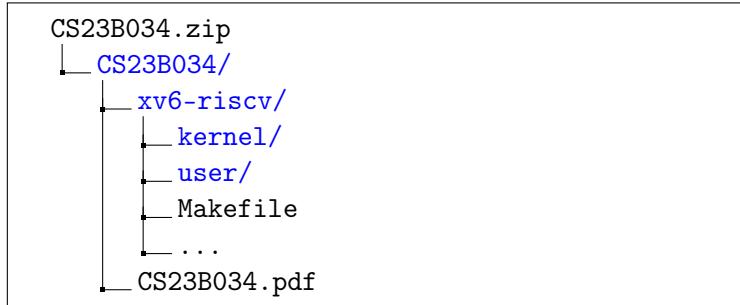
## 8 Submission

Establish a directory named using your roll number in uppercase format (e.g., CS23B034). This directory must include:

- The complete `xv6-riscv` directory containing all code modifications.
- A PDF report named using your roll number in uppercase format (e.g., CS23B034.pdf).

### 8.1 Directory Structure

Your final zip archive structure must conform to this layout:



Within your report, provide concise explanations of the implementation logic and rationale for each task's code. Particularly, for **each task**, create separate sections that summarize:

- **Code Modifications Made:** Describe the key changes or additions you implemented.
- **Testcases Passed:** List the testcases that successfully passed.
- **Errors During the Process:** Document any compilation or runtime errors encountered, along with how you addressed them.

## Appendix A: Task 1 Complete Solution

**Note for Students:** This appendix provides the complete solution for Task 1 to ensure that all students can proceed with the remaining tasks even if they encounter difficulties with the basic system call setup. Without completing Task 1, `make qemu` will fail with compilation errors, preventing further progress.

Task 1 establishes the basic infrastructure for `mmap` and `munmap` system calls in xv6. These system calls initially return error codes but allow `user/_mmaptest.c` to compile and execute successfully.

### File Modifications Required

The following files need to be modified in the specified order:

#### 1. Makefile

**Location:** Root directory of xv6

Find the UPROGS section and add `$U/_mmaptest` to the list:

```
UPROGS=\n    $U/_cat\\n\n    $U/_echo\\n\n    $U/_forktest\\n\n    ...\\n\n    $U/_mmaptest\\n
```

#### 2. kernel/syscall.h

Add system call numbers at the end of the file:

```
#define SYS_mmap 57\n#define SYS_munmap 58
```

**Important:** Use the next available numbers after the highest existing system call number in your version.

#### 3. user/user.h

Add function prototypes for user programs:

```
void* mmap(void *addr, int length, int prot, int flags, int fd, int offset);\nint munmap(void *addr, int length);
```

#### 4. user/usys.pl

Add entries for the new system calls at the end:

```
entry("mmap");\nentry("munmap");
```

#### 5. kernel/syscall.c

Add external function declarations with other `extern` declarations:

```
extern uint64 sys_mmap(void);\nextern uint64 sys_munmap(void);
```

Add entries to the `syscalls` array:

```

static uint64 (*syscalls[])() = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
...
[SYS_mmap]    sys_mmap,
[SYS_munmap]  sys_munmap,
};
```

## 6. kernel/sysfile.c

Add the following function implementations at the end of the file:

```

uint64
sys_mmap(void)
{
    return -1;
}

uint64
sys_munmap(void)
{
    return -1;
}
```

## 7. kernel/fcntl.h

Verify these constants are defined (add if missing):

```

#define PROT_NONE 0x0
#define PROT_READ 0x1
#define PROT_WRITE 0x2
#define PROT_EXEC 0x4
#define MAP_SHARED 0x01
#define MAP_PRIVATE 0x02
```

## Testing the Implementation

After making all modifications:

1. Build xv6:

```
$ make clean
$ make qemu
```

2. Test the setup:

```
$ mmaptest
```

## Expected Behavior

The `mmaptest` should start and fail at the first `mmap` call with output similar to:

```

mmap_test starting
test basic mmap
mmaptest failure: mmap (1), pid=3
```

**This failure is expected and correct for Task 1.** The system calls are returning -1 as intended.