# Assignment Report: Implementation of Memory Mapping in xv6

**Filbert Hamijoyo - 122040012**

## 1 Introduction

In this assignment, I implemented memory mapping functionalities in the xv6 operating system, specifically focusing on the `mmap` and `munmap` system calls. The primary objective was to enhance xv6's memory management capabilities by enabling processes to map files or devices directly into their address space. This facilitates efficient file I/O operations and inter-process communication by leveraging the operating system's memory management unit.

My work involved modifying the kernel to handle memory mapping requests, ensuring proper synchronization and protection mechanisms, and implementing robust error handling. Additionally, I used a given comprehensive test program, `mmaptest.c`, to validate the correctness and reliability of the implemented features. The project required a deep understanding of low-level memory management, system calls, and process synchronization within the xv6 environment.

## 2 Design

The design of the memory mapping functionality in xv6 encompasses several key components:

### 2.1 System Calls Implementation

#### 2.1.1 `sys_mmap`

The `sys_mmap` system call facilitates the mapping of files or devices into a process's address space. The implementation involves:

- **Argument Validation**: Ensuring that the size and offset are valid and properly aligned.
- **Address Allocation**: Selecting an appropriate virtual address if none is specified, ensuring page alignment.
- **VMA Slot Allocation**: Allocating a Virtual Memory Area (VMA) slot to track the mapping details.
- **Page Mapping**: Allocating physical memory pages and mapping them into the process's page table with the specified protections and flags.
- **File Duplication**: Handling file reference counts appropriately.

**Code Implementation:**

```c
uint64 sys_mmap(void) {
    uint64 addr, size, offset;
    int protection, flags, fd;
    struct file *f = NULL;
```

```
    struct proc *p = myproc();

    // Fetch arguments
    argaddr(0, &addr);
    argaddr(1, &size);
    argint(2, &protection);
    argint(3, &flags);
    if (argfd(4, &fd, &f) < 0) {
        return -1;
    }
    argaddr(5, &offset);

    // Validate size and alignment
    if (size <= 0 || offset % PGSIZE != 0) {
        return -1;
    }

    // Choose address if unspecified
    if (addr == 0) {
        addr = PGROUNDUP(p->sz);
    } else if (addr % PGSIZE != 0) {
        return -1; // Ensure page alignment
    }

    // Map file offset and check for writable MAP_SHARED
    if ((protection & PROT_WRITE) && (flags & MAP_SHARED) && !f->
       writable) {
        return -1;
    }

    // Find a free VMA slot
    struct vma *vma = NULL;
    for (int i = 0; i < VMASIZE; i++) {
        if (!p->vma[i].valid) {
            vma = &p->vma[i];
            break;
        }
    }
    if (!vma) {
        return -1;
    }

    // Set VMA fields
    vma->start = addr;
    vma->length = size;
    vma->prot = protection;
    vma->flags = flags;
    vma->offset = offset;
    vma->f = f;
    vma->valid = 1;

    if (f) filedup(f);

    // Expand process size if necessary
    if (addr + size > p->sz) p->sz = addr + size;

```

```c
    // Page permissions
    int perm = PTE_U;
    if (protection & PROT_READ) perm |= PTE_R;
    if (protection & PROT_WRITE) perm |= PTE_W;
    if (protection & PROT_EXEC) perm |= PTE_X;

    // Map pages with file contents, handling MAP_PRIVATE and MAP_SHARED
    begin_op();
    for (uint64 va = addr; va < addr + size; va += PGSIZE) {
        uint64 pa = (uint64)kalloc();
        if (pa == 0) {
            end_op();
            return -1;
        }
        memset((void *)pa, 0, PGSIZE);

        // Copy content from file starting at the specified offset
        if (flags & (MAP_SHARED | MAP_PRIVATE)) {
            ilock(f->ip);
            int n = readi(f->ip, 0, pa, offset + (va - addr), PGSIZE);
            iunlock(f->ip);

            if (n < 0) {
                end_op();
                kfree((void *)pa);
                return -1;
            }
        }

        if (mappages(p->pagetable, va, PGSIZE, pa, perm) < 0) {
            end_op();
            kfree((void *)pa);
            return -1;
        }
    }
    end_op();

    return addr;
}
```

Listing 1: Implementation of `sys_mmap`

**Explanation:**

The `sys_mmap` function begins by retrieving and validating the arguments provided by the user. It ensures that the requested size and offset are appropriate and aligned to page boundaries. If no address is specified, it automatically selects the next available address space. It then locates a free VMA slot to store the mapping details.

For each page to be mapped, it allocates physical memory, initializes it, and copies the relevant file content if necessary. The pages are then mapped into the process's page table with the specified protections. Proper error handling ensures that any failures during the mapping process result in a clean exit.

#### 2.1.2 sys_munmap

The sys_munmap system call handles the unmapping of previously mapped memory regions. The key steps include:

- **Argument Retrieval**: Obtaining the address and length of the region to unmap.
- **VMA Identification**: Locating the corresponding VMA slot that overlaps with the specified region.
- **Page Unmapping**: Removing the page mappings from the process's page table and freeing the associated physical memory.
- **VMA Slot Management**: Adjusting or splitting the VMA slot as necessary to reflect the unmapped region.
- **Dirty Page Handling**: For MAP_SHARED mappings, ensuring that any modified pages are written back to the file.

**Code Implementation:**

```c
uint64
sys_munmap(void) {
    struct proc *p = myproc();
    uint64 addr, length;

    // Retrieve arguments
    argaddr(0, &addr);
    argaddr(1, &length);

    // Validate length
    if (length <= 0) {
        return -1;
    }

    // Round up length to nearest page boundary
    uint64 unmap_limit = PGROUNDUP(addr + length);

    // Iterate through VMAs to locate overlapping regions
    for (int i = 0; i < VMASIZE; i++) {
        struct vma *vma = &p->vma[i];
        if (vma->valid && !(unmap_limit <= vma->start || addr >= vma->
            start + vma->length)) {

            uint64 start_unmap = addr > vma->start ? addr : vma->start
                ;
            uint64 end_unmap = unmap_limit < (vma->start + vma->length
                ) ? unmap_limit : (vma->start + vma->length);

            // If VMA is MAP_SHARED, write back modified pages to the
                file
            if (vma->f && (vma->flags & MAP_SHARED)) {
                begin_op();
                for (uint64 va = start_unmap; va < end_unmap; va +=
                    PGSIZE) {
                    pte_t *pte = walk(p->pagetable, va, 0);
                    if (pte && (*pte & PTE_V) && (*pte & PTE_R)) {
                        uint64 pa = PTE2PA(*pte);
                        char *kv = (char *)pa;

```

```
                        // Write back page and clear dirty bit
                        ilock(vma->f->ip);
                        int n = writei(vma->f->ip, 0, (uint64)kv, vma->offset + (va - vma->start), PGSIZE);
                        if (n != PGSIZE) {
                            printf("sys_munmap: writei failed for va=0x%lx\n", va);
                        }
                        *pte &= ~PTE_R;  // Clear dirty bit
                        iunlock(vma->f->ip);
                    }
                }
                end_op();
            }

            // Free pages within this region
            int npages = (end_unmap - start_unmap) / PGSIZE;
            uvmunmap(p->pagetable, start_unmap, npages, 1);

            // Adjust or split the VMA as necessary
            if (start_unmap == vma->start && end_unmap == vma->start + vma->length) {
                vma->valid = 0;
                if (vma->f) {
                    fileclose(vma->f);
                    vma->f = 0;
                }
            } else if (start_unmap == vma->start) {
                vma->start = end_unmap;
                vma->offset += end_unmap - vma->start;
            } else if (end_unmap == vma->start + vma->length) {
                vma->length = start_unmap - vma->start;
            } else {
                int j;
                for (j = 0; j < VMASIZE; j++) {
                    if (!p->vma[j].valid) break;
                }
                if (j == VMASIZE) {
                    printf("sys_munmap: No free VMA slots available\n");
                    return -1;
                }

                p->vma[j] = *vma;
                p->vma[j].start = end_unmap;
                p->vma[j].length -= (end_unmap - vma->start);
                p->vma[j].offset += end_unmap - vma->start;
                vma->length = start_unmap - vma->start;
            }
        }
    }

    return 0;
}
```

Listing 2: Implementation of `sys_munmap`

**Explanation:**

The `sys_munmap` function begins by retrieving the address and length of the memory region to unmap. It then rounds up the length to ensure page alignment. The function iterates through the process's VMAs to find overlapping regions with the specified address range.

For `MAP_SHARED` mappings, it writes back any modified (dirty) pages to the underlying file before unmapping. It then proceeds to remove the page mappings from the process's page table and frees the associated physical memory. If the unmapping spans the entire VMA, the VMA slot is marked as invalid. Otherwise, the VMA is adjusted or split to accurately reflect the remaining mapped regions.

## 2.2 Page Fault Handling

To support memory-mapped regions, the page fault handler in `trap.c` was extended to manage faults arising from these mappings. The enhancements include:

- **VMA Lookup**: Determining if the faulting address lies within a valid VMA.
- **Copy-On-Write (COW)**: Implementing COW semantics for `MAP_PRIVATE` mappings to ensure that modifications do not affect the underlying file.
- **Page Allocation and Mapping**: Allocating new physical pages as needed and updating the page tables with appropriate permissions.
- **Error Handling**: Ensuring that any invalid accesses result in process termination to maintain system stability.

**Code Implementation:**

```c
else if (r_scause() == 13 || r_scause() == 15) { // Page fault handling
        uint64 va = r_stval(); // faulting address
        uint64 fault_page_start = PGROUNDDOWN(va);

        struct vma *vma = NULL;
        for (int i = 0; i < VMASIZE; i++) {
            if (p->vma[i].valid && va >= p->vma[i].start && va < p->vma[i].start + p->vma[i].length) {
                vma = &p->vma[i];
                break;
            }
        }

        if (vma == NULL) {
            printf("usertrap: no VMA found for address 0x%lx, pid=%d\n", va, p->pid);
            setkilled(p);
            goto err;
        }

        bool is_write = (r_scause() == 15); // Write page fault

        // Check if page is mapped
        pte_t *pte = walk(p->pagetable, fault_page_start, 0);
        if (pte == 0 || (*pte & PTE_V) == 0) {
            // Page not yet mapped, proceed with allocation
            goto allocate_page;
        }

```

```
        // Handle Copy-On-Write for MAP_PRIVATE
        if (is_write && (vma->flags & MAP_PRIVATE)) {
            uint64 pa = (*pte) >> 10;
            pa = pa << 12;

            char *old_mem = (char*)pa;

            char *new_mem = kalloc();
            if (new_mem == 0) {
                printf("usertrap: kalloc failed for address 0x%lx, pid
                    =%d\n", va, p->pid);
                setkilled(p);
                goto err;
            }

            // Copy the contents to the new page
            memmove(new_mem, old_mem, PGSIZE);

            uint64 new_pa = (uint64)new_mem;

            // Update the PTE to point to the new physical page with
                write permissions
            *pte = (new_pa >> 12) << 10 | ((*pte) & 0x3FF) | PTE_W;

            sfence_vma(); // Flush TLB
        } else {
        allocate_page: {
            char *mem_kva = kalloc();
            if (mem_kva == 0) {
                printf("usertrap: kalloc failed for address 0x%lx, pid
                    =%d\n", va, p->pid);
                setkilled(p);
                goto err;
            }

            memset(mem_kva, 0, PGSIZE);

            if (vma->f != NULL) {
                uint64 file_offset = vma->offset + (fault_page_start -
                     vma->start);
                mapfile(vma->f, mem_kva, file_offset);
            }

            // Determine page permissions based on VMA protection
                flags
            int perm = PTE_U | PTE_V;
            if (vma->prot & PROT_READ)
                perm |= PTE_R;
            if (vma->prot & PROT_WRITE && !(vma->flags & MAP_PRIVATE))
                 {
                perm |= PTE_W; // Allow write only if not MAP_PRIVATE
            }
            if (vma->prot & PROT_EXEC)
                perm |= PTE_X;

            uint64 pa = (uint64)mem_kva;
```

```c
            if (mappages(p->pagetable, fault_page_start, PGSIZE, pa, perm) < 0) {
                printf("usertrap: mappages failed for address 0x%lx, pid=%d\n", va, p->pid);
                kfree(mem_kva);
                setkilled(p);
                goto err;
            }
            sfence_vma();
        }
        }
    } else {
        printf("usertrap(): unexpected scause 0x%lx pid=%d\n", r_scause(), p->pid);
        printf("            sepc=0x%lx stval=0x%lx\n", r_sepc(), r_stval());
        setkilled(p);
    }

    if (killed(p))
        exit(-1);
    if (which_dev == 2)
        yield();
    usertrapret();

err:
    if (killed(p))
        exit(-1);
    usertrapret();
}
```

Listing 3: Page Fault Handling in `trap.c`

**Explanation:**

The enhanced page fault handler begins by identifying the faulting virtual address and locating the corresponding VMA. If no VMA is found for the address, the process is terminated to maintain system integrity. For write faults (`MAP_PRIVATE`), the handler implements Copy-On-Write (COW) by allocating a new physical page, copying the contents from the original page, and updating the page table entry to point to the new page with write permissions.

For read faults or non-COW scenarios, the handler allocates a new physical page, initializes it, and maps it into the process's address space. Proper synchronization and error handling ensure that any issues during page allocation or mapping result in safe termination of the offending process.

## 2.3 Virtual Memory Areas (VMA)

A VMA structure was introduced to keep track of memory-mapped regions:

**Code Implementation:**

```c
struct vma {
    uint64 start;          // Starting virtual address of the VMA
    uint64 length;         // Length of the memory region in bytes
    int prot;              // Protection flags (e.g., PROT_READ, PROT_WRITE)
    int flags;             // Mapping flags (e.g., MAP_SHARED, MAP_PRIVATE)
```

```
    uint64 offset;          // Offset within the file
    struct file *f;         // Pointer to the mapped file
    int valid;              // Indicates if the VMA slot is in use
};
```

Listing 4: Definition of VMA Structure

**Explanation:**

The `struct vma` defines the properties of a virtual memory area, including its start address, length, protection and mapping flags, file offset, and associated file pointer. The `valid` flag indicates whether the VMA slot is currently in use. Each process maintains an array of such VMAs (`struct vma vma[VMASIZE];`), allowing multiple memory-mapped regions to coexist within a single process's address space.

# 3 Environment and Execution

## 3.1 Running Environment

The implementation was developed and tested using the following setup:

- **Virtual Machine**: UTM
- **Image**: `CSC3150_a3_xv6.qcow2`
- **Network Mode**: Emulated VLAN
- **Port Forwarding**: Host port 2200 mapped to guest port 22 for SSH access

## 3.2 Setup and Access

To set up and access the virtual machine, follow these steps:

1. **Login Credentials**:
   - `Username`: csc3150
   - `Password`: csc3150
2. **Assign IP Address**:
   - Execute `sudo dhclient` to obtain an IP address.
   - Verify the assignment using `ip a`.
3. **SSH Access**:
   - Connect via SSH using the command:

```
ssh -p 2200 csc3150@127.0.0.1
```

Listing 5: SSH Command

## 3.3 Compilation and Execution

The project was compiled and executed as follows:

1. Navigate to the xv6-labs-2022 directory:

```
cd xv6-labs-2022
```

Listing 6: Navigation Command

2. Compile and run xv6 using QEMU:

```
make clean
make qemu
```

Listing 7: Compilation Commands

3. Within the xv6 environment, execute the test program:

```
mmaptest
```

Listing 8: Execution Command

### 3.4 Execution Output

The successful execution of the test program is demonstrated in the screenshot below.

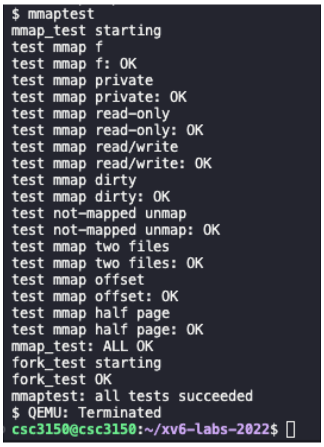


Figure 1: Output of `mmaptest` Execution

As shown in Figure 1, all memory mapping tests passed successfully, indicating the correct implementation of the `mmap` and `munmap` functionalities. The output confirms that memory regions were correctly mapped, accessed, and unmapped, with appropriate handling of shared and private mappings, as well as proper synchronization between parent and child processes during fork operations.

## 4 Conclusion

This assignment provided an in-depth exploration of memory management within operating systems, specifically focusing on the implementation of memory mapping in xv6. By developing and integrating the `mmap` and `munmap` system calls, I enhanced xv6's ability to handle file and device mappings, which are fundamental for efficient I/O operations and memory utilization.

Through this project, I gained valuable insights into low-level memory management, process synchronization, and kernel-user space interactions. The successful execution of the test program validated the robustness of the implemented features and reinforced my understanding of operating system principles. Additionally, addressing challenges such as Copy-On-Write and ensuring proper synchronization mechanisms deepened my proficiency in systems programming and kernel development. This experience has significantly strengthened my skills and prepared me for more complex tasks in operating system design and implementation.