Operating System CSC3150 Project 4

Yuqi Ma

yuqima1@link.cuhk.edu.cn

Motivation

- Gain Deep-Dive **Kernel** Experience
- Implement Essential **system calls** in xv6 operating system
- **Modified** from MIT6.S081 lab. DO NOT TRY TO COPY
- A Strong Portfolio Piece for Your Resume

My Comment

- It takes some time to **understand**
- Tens of lines of code to **implement**

Submission Guide

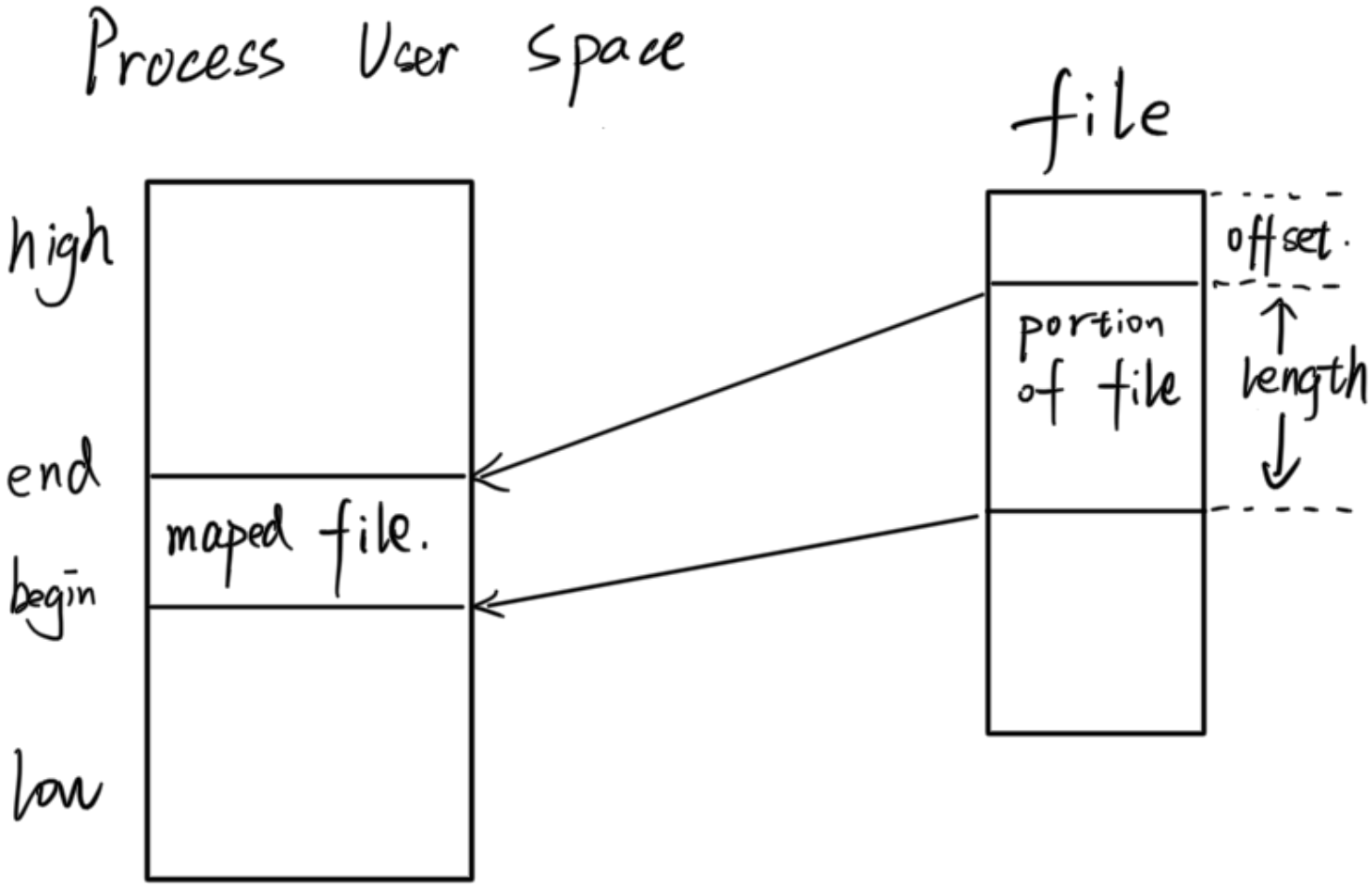
- **Submit a zip to Blackboard**.
- Zip structure: one report pdf and one source folder. The source folder only includes 4 source C files.
- A script (gen_submission.sh) for you to generate the zip file.

Key Prerequisite Concepts

- The xv6 Teaching Operating System
- File Descriptors
- Page Tables & Virtual Memory

Introduction- What is mmap()?

- Establish a mapping between a process's user space and a file (or device).
- Modification will affect both user space and kernel space.



Introduction-Why mmap()?

Efficient Large File Handling

- Enables random access to file data as if it were in memory.
- Eliminates the need for explicit read()/write() system calls for each access.
- The kernel handles loading (page-in) and writing back (page-out) data on demand.

Introduction-How to use it?

```
1 // Defined in user.h
2 void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

- addr: the start address of the mapped region; 0 if the user leave it to kernel to then the kernel chooses the page-aligned address (always 0 in this assignment).
- length: how many bytes of the file to map.
- prot: Protection flags (e.g., PROT_READ, PROT_WRITE). Determines how the process can access the memory (**cannot conflict with the open mode of the file**).
- flags: Controls visibility of changes (MAP_SHARED vs. MAP_PRIVATE). This is crucial for data persistence and inter-process communication.
- fd: the file descriptor
- offset: the starting offset in the file

Introduction-Example usage

```
char *p = mmap(0, PGSIZE*2, PROT_READ, MAP_PRIVATE, fd, 0);
```

- 0: kernel will choose the page-aligned address for the user
- **PROT_READ** indicates that the mapped memory should be read-only, i.e., modification is not allowed.
- **MAP_PRIVATE** indicates that if the process modifies the mapped memory, the modification should **not** be written back to the file (of course, due to **PROT_READ**, updates are prohibited in this case).

Introduction-How to implement?

- To ensure the function of mmap, you should complete the **VMA** (Virtual Memory Area) struct definition in proc.h and the function `**sys_mmap**` in sysfile.c

Introduction-What is VMA?

- Virtual Memory Area: A kernel data structure that acts as a "mapping descriptor".

- Each VMA represents a contiguous region of virtual memory that **has the same permissions** and is backed by **the same kind of object**. The operating system needs to keep track of these mappings, including **where they start**, **how large** they are, what **permissions** they have, and what **file** or device they're associated with.

Introduction- What is VMA- Trace the code

kernel/proc.h

```
#define VMASIZE 16

// TODO: complete struct of VMA
struct vma {
};
```

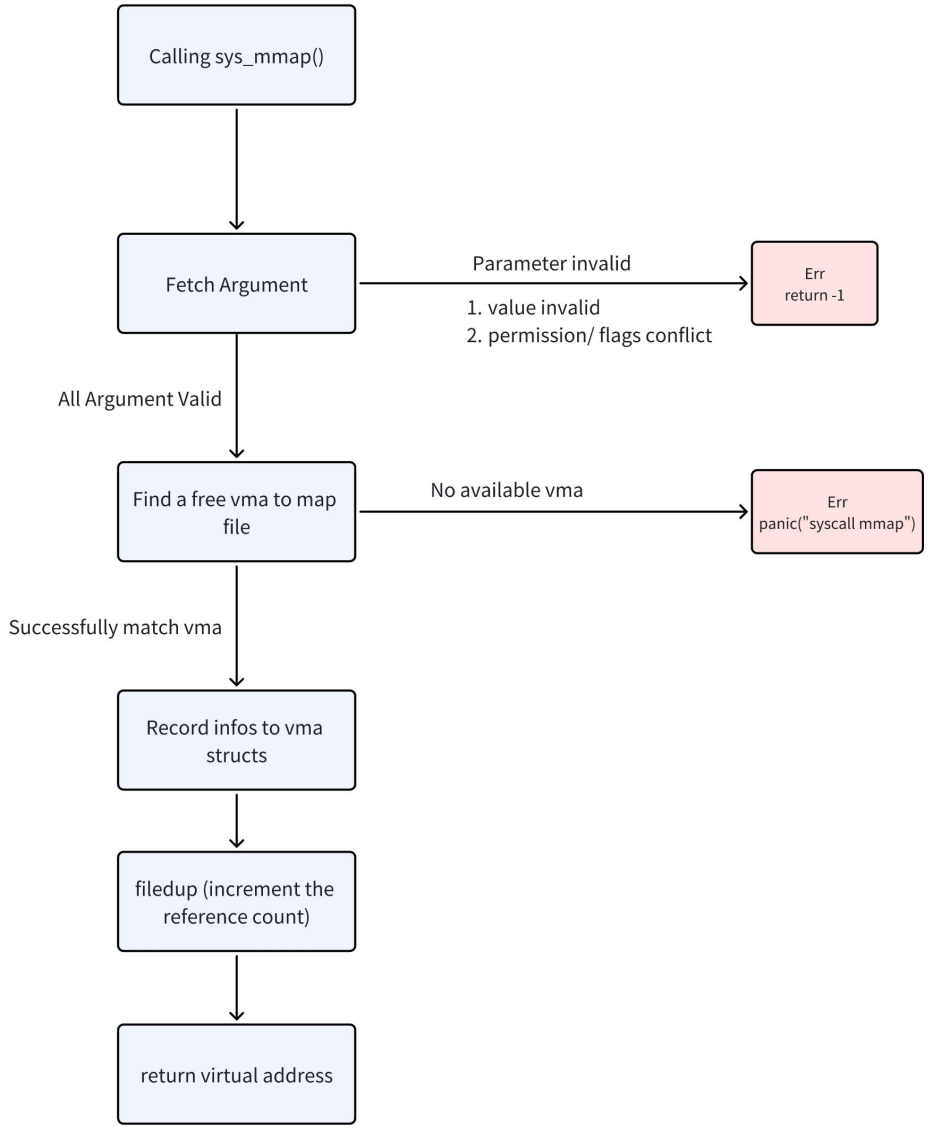
```
// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID

  // wait_lock must be held when using this:
  struct proc *parent;         // Parent process

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  struct vma vma[VMASIZE];     // virtual mem area
};
```

Implementation-VMA

- Design Your VMA Struct
- Think about the parameters of mmap() – what information needs to be persisted?
- Define a structure to record the **address**, **length**, **permissions**, **file**, etc. for a virtual memory range created by mmap.
- Look at the parameters of mmap() to see what should be recorded in the VMA.
- You may add other fields to VMA struct if you later find that they are useful.

Implementation-sys_mmap

1. Fetch the arguments and check if they are valid from the user call.
2. Search for an available VMA in the current process and record the map information.
3. Increase the file duplicate.



Implementation-sys_mmap

Q: How to fetch the arguments? There is no argument passed by calling sys_mmap?

A: User-space arguments are stored in the process's trapframe.

The kernel uses helper functions (argint, argaddr, argfd) to safely retrieve them.

These functions internally rely on argraw, which reads from the saved user registers (a0-a5).

kernel/syscall.c

```
static uint64
argraw(int n)
{
  struct proc *p = myproc();
  switch (n) {
  case 0:
    return p->trapframe->a0;
  case 1:
    return p->trapframe->a1;
  case 2:
    return p->trapframe->a2;
  case 3:
    return p->trapframe->a3;
  case 4:
    return p->trapframe->a4;
  case 5:
    return p->trapframe->a5;
  }
  panic("argraw");
  return -1;
}
```

Implementation-sys_mmap

Q: How to fetch the arguments from the trapframe?

A: The kernel uses helper functions (argint, argaddr, argfd) to safely retrieve them.

e.g.

int var;

argint(1, &var);

kernel/syscall.c

```c
// Retrieve an argument as a pointer.
// Doesn't check for legality, since
// copyin/copyout will do that.
void
argaddr(int n, uint64 *ip)
{
  *ip = argraw(n);
}
```

```c
// Fetch the nth 32-bit system call argument.
void
argint(int n, int *ip)
{
  *ip = argraw(n);
}
```

Implementation-sys_mmap

Q: What's the possible `prot` and `flags` arguments of mmap? How to pass them?

A: check whether the combination of arguments is valid. Arguments are passed in 'FlagA | FlagB' form.

kernel/fcntl.h

```
#ifdef LAB_MMAP
#define PROT_NONE       0x0
#define PROT_READ       0x1
#define PROT_WRITE      0x2
#define PROT_EXEC       0x4

#define MAP_SHARED      0x01
#define MAP_PRIVATE     0x02
#endif
```

- prot (Protection Flags): Define how the memory can be accessed (e.g., PROT_READ, PROT_WRITE).
- flags (Mapping Type): The most critical for implementation. Determines the visibility and persistence of modifications (MAP_SHARED vs. MAP_PRIVATE).
- You MUST check these flags in your implementation, especially during munmap.

Implementation-sys_mmap

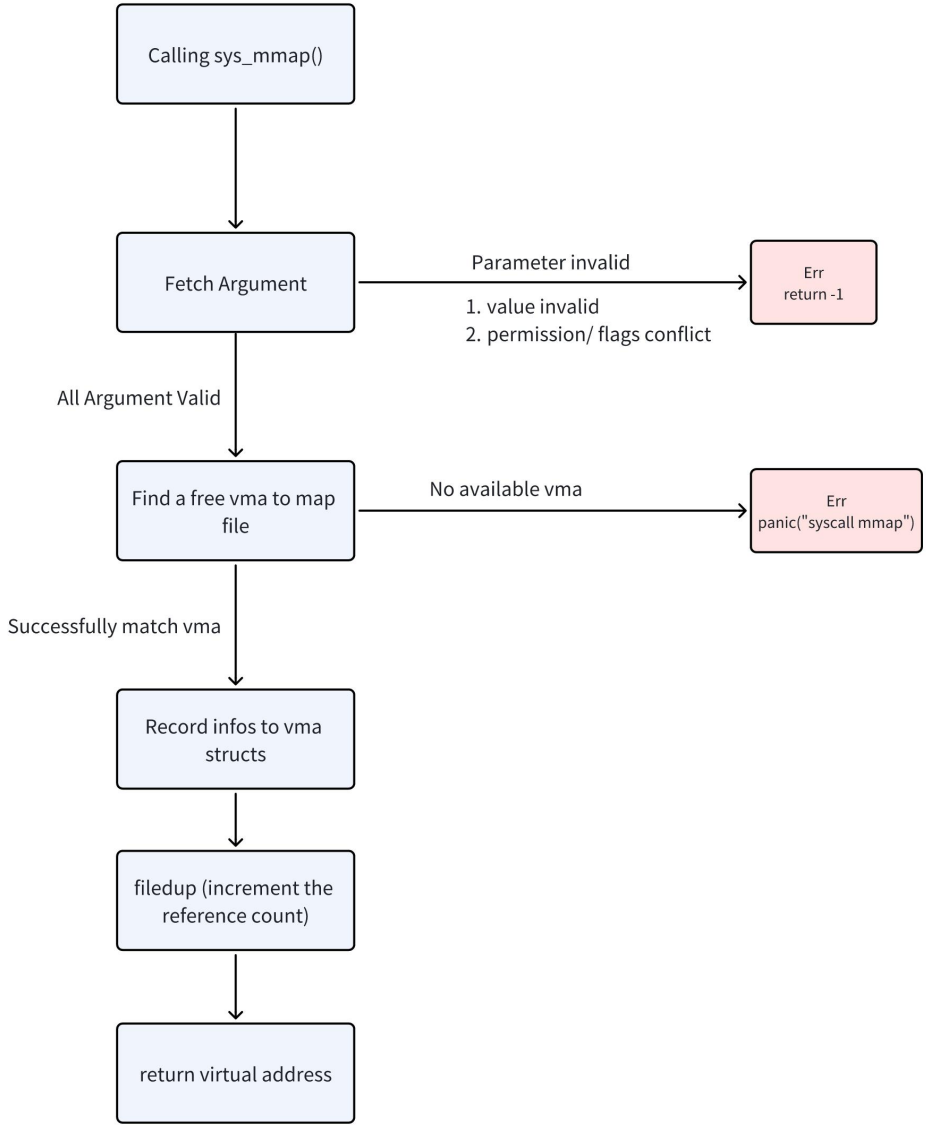
- Q: What is the return address of the mmap?
- A: The chosen virtual address. A standard approach is to use the process's current size (p->sz) as the start address, then update p->sz += length to grow the process's address space.

Implementation-sys_mmap

Finish? No file content in the memory!

We only recorded the mapping intention in the VMA.

No physical page is allocated, and no page table entry (PTE) is created yet.



Introduction-page fault handle

- When the application accesses this memory, the kernel loads the actual data through a ***page fault*** exception.
- This deferred allocation strategy is known as **Lazy Loading**.
- It defers the costly work of I/O and memory allocation until the moment it is absolutely necessary.
- This access triggers a page fault, transferring control to the kernel.

Implementation-page fault handle

kernel/trap.c

```
// TODO: page fault handling
else if(r_scause() == 13 || r_scause() == 15) {
  // uint64 va = r_stval();
  printf("Now, after mmap, we get a page fault\n");
  goto err;
}
```

1. Locate the VMA using the fault address.
2. Read 4096 bytes (1 PAGESIZE) of the relevant file onto that page, map it into the user address space.
3. Set the permissions correctly on the page.

Implementation-page fault handle

1. Locate the VMA using the fault address. How?
   `r_stval` provides the address causing the trap. Use this address to find the VMA.

```
// Supervisor Trap Value
static inline uint64
r_stval()
{
  uint64 x;
  asm volatile("csrr %0, stval" : "=r" (x) );
  return x;
}
```

Implementation-page fault handle

2. Read 4096 bytes (1 PAGESIZE) of the relevant file onto that page, map it into the user address space.

Please trace the code of `mapfile`, `mappages` and `kalloc`.

Implementation- page fault handle

3. Set the permissions correctly on the page. According to the VMA flags and prot!

```
#define PTE_V (1L << 0) // valid
#define PTE_R (1L << 1)
#define PTE_W (1L << 2)
#define PTE_X (1L << 3)
#define PTE_U (1L << 4) // user can access
```

Introduction-munmap

- Unmaps the specified memory region, reversing the work of mmap().

- Two critical actions must be considered:

  - Writing back dirty pages for MAP_SHARED mappings.

  - Releasing resources (pages, file references).

Implementation-munmap

1. Find the VMA for the address range and unmap the specified pages. (use `uvmunmap`)
2. If munmap removes **all** pages of a previous mmap, it should decrease the reference count of the corresponding struct file.
3. If an unmapped page has been modified and the file is mapped MAP_SHARED, write the page back to the file. (use `filewrite`)

Introduction-page align

- What? Organize memory in such a way that each page of memory starts at a memory address that is a multiple of the PAGESIZE (4096 bytes in this project).

- Why? Efficient Memory Access! Memory access operations are often optimized when memory addresses are aligned with the PAGESIZE.

- The kernel manages memory in page-sized units. All virtual addresses and lengths for mappings must be page-aligned.

Implementation-page align

- How? Use these two macros!

```
#define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
```

- You have to figure out use which on under different situations.

Introduction-How to test?

- After `make qemu`, run `mmaptest`
- **Before your implementation of mmap**(), you should fail in the first test.
  Sample output:

```
$ mmaptest
mmap_test starting
test mmap f
mismatch at 0, wanted 'A', got 0x1
mmaptest: mmap_test failed: v1 mismatch (1), pid=4
```

Introduction-How to test?

- **After your implementation of mmap**(), **before your implementation of page fault handle**, you should encounter a page fault trap.
  Sample output:

```
$ mmaptest
mmap_test starting
test mmap f
Now, after mmap, we get a page fault
usertrap(): unexpected scause 0x000000000000000d pid=3
            sepc=0x0000000000000076 stval=0x0000000000007000
```

Introduction-How to test?

- **After your implementation of page fault handle**, you should pass a few tests.
  Sample output:

```
$ 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
mmaptest: mmap_test failed: file does not contain modifications, pid=4
```

Introduction-How to test?

- **Afte your implementation of munmap**, you should pass all tests except forktest.
  Sample output:

```
$ 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
usertrap(): unexpected scause 0x000000000000000d pid=4
            sepc=0x0000000000000076 stval=0x0000000000013000
fork_test failed
```

Final tips

- Remember to `make qemu` after any modification of xv6. You are modifying kernel!
- To quit the qemu, press `control` and then press `A`, release and then press `X`.
- Begin early! It’s hard for you to trace the code and understand the whole system.
- Good luck!