CSC3150-Instruction-A3

Introduction

This assignment uses xv6, a simple and Unix-like teaching operating system, as the platform to guide you in implementing the ***mmap*** and ***munmp*** system calls. These two are used to share memory among processes and to map files into process address spaces. Generally speaking, this assignment focuses on ***memory-mapped files.*** A mechanism supporting memory-mapped files can handle files as if they are a portion of the program's memory. This is achieved by mapping a file to a segment of the virtual memory space (Reminder: Each process has its own virtual address space). Such mapping between a file and memory space is achieved using the 'mmap()' system call, and the mapping is removed using the 'munmap()' system call. We provide a virtual machine image where everything is configured and set. The image is available on Blackboard.

Submission

- **Due on: 23:59, 15 Nov 2023**
- **Plagiarism is strictly forbidden.** Please note that TAs may ask you to explain the meaning of your program to ensure that the codes are indeed written by yourself. Please also note that we would check whether your program is too similar to your fellow students' code and solutions available on the internet using plagiarism detectors.
- Late submission: A late submission **within 15 minutes** will not induce any penalty on your grades. But **00:16 am-1:00 am: Reduced by 10%; 1:01 am-2:00 am: Reduced by 20%; 2:01 am-3:00 am: Reduced by 30% and so on.** (e.g., Li Hua submitted a perfect attempt at 2:10 a.m. He will get (100+10 (bonus))*0.7=77p

Format guide

The project structure is illustrated below. You can also use `ls` command to check if your structure is fine. Structure mismatch would cause grade deduction.

For this assignment, you don't need a specific folder for the bonus part. The source folder should contain four files: ***proc.c, proc.h, sysfile.c, trap.c***

```
main@ubuntu:~/Desktop/Assignment_3_120010001$ ls
Report.pdf  source/

```

```
(One directory and one pdf.)
```

```
main@ubuntu:~/Desktop/Assignment_3_120010001/source$ ls
proc.c proc.h sysfile.c trap.c

(three .c files and one .h file)
```

Please compress all files in the file structure root folder into a single zip file and **name it using your student ID as the code shown below and above, for example, Assignment_3_120010001.zip.** The report should be submitted in the format of **pdf**, together with your source code. Format mismatch would cause grade deduction. Here is the sample step for compressing your code.

```
main@ubuntu:~/Desktop$
 zip -q -r Assignment_3_120010001.zip Assignment_3_120010001

main@ubuntu:~/Desktop$ ls
Assignment_3_120010001                Assignment_3_120010001.zip
```

Instruction Guideline

We limit your implementation within ***proc.c, proc.h, sysfile.c, trap.c*** four files, starting with "TODO:" comments. The entry (where you may start learning) of the test program is the main function in ***mmaptest.c*** under the 'xv6-labs-2022/user' directory.

Sections with (*) are introduction sections. These sections **introduce** tools and functions that will help you understand what this system is about and how the system works with these components. You might need to use it for this assignment. Do **NOT CHANGE** them except the TODO parts.

1. For the introduction sections, please figure out how functions work and how to use them.
2. Be sure you have a basic idea of the content before starting your assignment. We believe that those would be enough for handling this assignment.
3. (option) For students who are interested in the xv6 system and want to learn more about it, you are welcome to read "xv6-book" to get more details.
    a. https://pdos.csail.mit.edu/6.828/2022/xv6/book-riscv-rev3.pdf

Sections **without** (*) are TODO sections. In these sections, the logic of how this component/ function should work is a detailed list. You should implement functions in given places.

1. However, no sample code will be shown here. You need to figure out the implementation based on the logic and APIs provided in the introduction sections.

Arguments fetching*

<xv6-book> chapter 4.3

```
void argint(int, int*);
int argstr(int, char*, int);
void argaddr(int, uint64 *);
int argfd(int n, int *pfd, struct file **pf);
```

The kernel functions argint, argaddr, and argfd retrieve the n ' th system call argument from the trap frame as an integer, pointer, or file descriptor. They all call argraw to retrieve the appropriate saved user register (kernel/syscall.c:34).

Proc*

```
// Defined in proc.h
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
```

```
25 };
26
27 // Defined in proc.c
28 // Return the current struct proc *, or zero if none.
29 struct proc* myproc(void)
```

Pages*

<xv6-book> chapter 3

```
1  // Defined in riscv.h
2  typedef uint64 pte_t;
3  typedef uint64 *pagetable_t; // 512 PTEs
4
5  #define PGSIZE 4096 // bytes per page
6  #define PGSHIFT 12  // bits of offset within a page
7
8  #define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
9  #define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
10
11 #define PTE_V (1L << 0) // valid
12 #define PTE_R (1L << 1)
13 #define PTE_W (1L << 2)
14 #define PTE_X (1L << 3)
15 #define PTE_U (1L << 4) // user can access
16
17
18 // one beyond the highest possible virtual address.
19 // MAXVA is actually one bit less than the max allowed by
20 // Sv39, to avoid having to sign-extend virtual addresses
21 // that have the high bit set.
22 #define MAXVA (1L << (9 + 9 + 9 + 12 - 1))
```

Ports & Flags*

```
1 // Defined in fcntl.h
2 #define PROT_NONE       0x0
3 #define PROT_READ       0x1
4 #define PROT_WRITE      0x2
5 #define PROT_EXEC       0x4
```

```
6
7 #define MAP_SHARED      0x01
8 #define MAP_PRIVATE     0x02
```

Traps

```
1  void usertrap(void)
2  {
3  ...
4    /// TODO: manage pagefault
5    else if(r_scause() == 13 || r_scause() == 15){
6        ...
7    }
8  ...
9  }
10
11 // Supervisor Trap Cause
12 static inline uint64
13 r_scause()
14 {
15   uint64 x;
16   asm volatile("csrr %0, scause" : "=r" (x) );
17   return x;
18 }
19
20 // Supervisor Trap Value
21 static inline uint64
22 r_stval()
23 {
24   uint64 x;
25   asm volatile("csrr %0, stval" : "=r" (x) );
26   return x;
27 }
```

Usertrap handles an interrupt, exception, or system call from user space. It calls r_scause() to get the exception code. r_stval() provides trap value. (especially refers to address when there is a pagefault in this assignment)

In this assignment, you are asked to manage **PageFault** of files, i.e. **Load page fault** and **Store page fault**.

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2–4	*Reserved*
1	5	Supervisor timer interrupt
1	6–8	*Reserved*
1	9	Supervisor external interrupt
1	10–15	*Reserved*
1	≥16	*Designated for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10–11	*Reserved*
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–23	*Reserved*
0	24–31	*Designated for custom use*
0	32–47	*Reserved*
0	48–63	*Designated for custom use*
0	≥64	*Reserved*

Table 4.2: Supervisor cause register (`scause`) values after trap. Synchronous exception priorities are given by Table 3.7.

File*

```
1  // Defined in file.h
2  struct file {
3    enum { FD_NONE, FD_PIPE, FD_INODE, FD_DEVICE } type;
4    int ref; // reference count
5    char readable;
6    char writable;
7    struct pipe *pipe; // FD_PIPE
8    struct inode *ip;  // FD_INODE and FD_DEVICE
9    uint off;          // FD_INODE
10   short major;       // FD_DEVICE
11 };
```

```
12
13 // in-memory copy of an inode
14 struct inode {
15   uint dev;           // Device number
16   uint inum;          // Inode number
17   int ref;            // Reference count
18   struct sleeplock lock; // protects everything below here
19   int valid;          // inode has been read from disk?
20
21   short type;         // copy of disk inode
22   short major;
23   short minor;
24   short nlink;
25   uint size;
26   uint addrs[NDIRECT+1];
27 };
28
29 // Write to file f.
30 // addr is a user virtual address.
31 int filewrite(struct file *f, uint64 addr, int n);
32
33 // Increment ref count for file f.
34 struct file* filedup(struct file*);
35
36 // Close file f.  (Decrement ref count, close when reaches 0.)
37 void fileclose(struct file*);
```

Struct "file" "inode" is presented for your information.

filewrite() will be invoked to write back when the memory map is over. i.e. Calling munmap or Calling exit of process. Similarly to fileclose().

filedup() will be invoked when there is an increment of accessing file. (mmap(), fork())

```
1 // Defined in fs.c
2
3 // Read data from inode.
4 // Caller must hold ip->lock.
5 // If user_dst==1, then dst is a user virtual address;
6 // otherwise, dst is a kernel address.
7 int readi(struct inode *ip, int user_dst, uint64 dst, uint off, uint n);
8
9 // Write data to inode.
10 // Caller must hold ip->lock.
11 // If user_src==1, then src is a user virtual address;
12 // otherwise, src is a kernel address.
```

```
// Returns the number of bytes successfully written.
// If the return value is less than the requested n,
// there was an error of some kind.
int writei(struct inode *ip, int user_src, uint64 src, uint off, uint n);

// Lock the given inode.
// Reads the inode from disk if necessary.
void ilock(struct inode *ip);

// Unlock the given inode.
void iunlock(struct inode *ip);
```

Function that you need to use when handling page fault, pay attention to how readi() works and figure out the parameter you should send to readi().

If you have no idea what readi() is doing, think about read() or memcpy(), which deal with pointers and address.

Similarly as writei()

ilock() and iunlock() are locks of inode, which are used to ensure consistency of the memory.

Hint

You may take a look at **sys_open()** to know how inode, file, and locks work.

(TODO) VMA Struct

```
// we already define size of VMA array for you
#define VMASIZE 16

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

Explanation

The VMA (Virtual Memory Area) struct is used to manage and track the memory regions that are mapped into the address space of a process. 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. This is what the vma struct is used for.

Implementation

- Keep track of what mmap has mapped for each process.
- Define a structure corresponding to the VMA (virtual memory area), recording the address, length, permissions, file, etc. for a virtual memory range created by mmap.
- Since the xv6 kernel doesn't have a memory allocator in the kernel, it's OK to declare a fixed-size array of VMAs and allocate from that array as needed. A size of 16 should be sufficient. (I already define VMASIZE for you)

Hint

Take a look at what parameter will be sent into mmap().

The VMA should contain a pointer to a struct file for the file being mapped;

If you would like to use more variables in VMA for further implementation, feel free to use them. There is not only one correct answer.

(TODO) mmap()

```
1 // Defined in user.h
2 void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)
3
4 // TODO: kernel mmap executed in sysfile.c
5 uint64
6 sys_mmap(void)
7 {
8 }
```

- Arguments explanation: In the mmaptest.c, we call '***char *p = mmap(0, PGSIZE*2, PROT_READ, MAP_PRIVATE, fd, 0);***'. This call asks the kernel to map the content of file '***fd***' into the address space. The first '***0***' argument indicates that the kernel should choose the virtual address (In this homework, you can assume that '***addr***' will always be zero). The second argument '***length***' indicates how many bytes to map. The third argument '***PROT_READ***' indicates that the mapped memory should be read-only, i.e., modification is not allowed. The fourth argument '***MAP_PRIVATE***' indicates that if the process modifies the mapped memory, the modification should not be written back to the file nor shared with other processes mapping the same file (of course, due to ***PROT_READ***, updates are prohibited in this case). The fifth argument is the file description of the file to be mapped. The last argument '***offset***' is the starting offset in the file. The return value indicates whether mmap succeeds or not.
- sys_xxx() function is the kernel's implementation of the xxx() system call. In the xv6 operating system, system calls are prefixed with sys_ to distinguish them from other functions and to indicate that they are system calls. The kernel functions argint, argaddr, and argfd retrieve the n ' th system call argument from the trap frame as an integer, pointer, or a file descriptor. See the **Arguments fetching** section.
- Implementation of mmap: Find an unused region in the process's address space in which to map the file, and add a VMA to the process's table of mapped regions. The VMA should contain a pointer to a struct file for the file being mapped; mmap should increase the file's reference count so that the structure doesn't disappear when the file is closed (hint: see filedup).
- Run mmaptest after mmap() implemented: the first mmap should succeed, but the first access to the mmap-ed memory will cause a page fault and kill mmaptest.
  - Before mmap() implemented

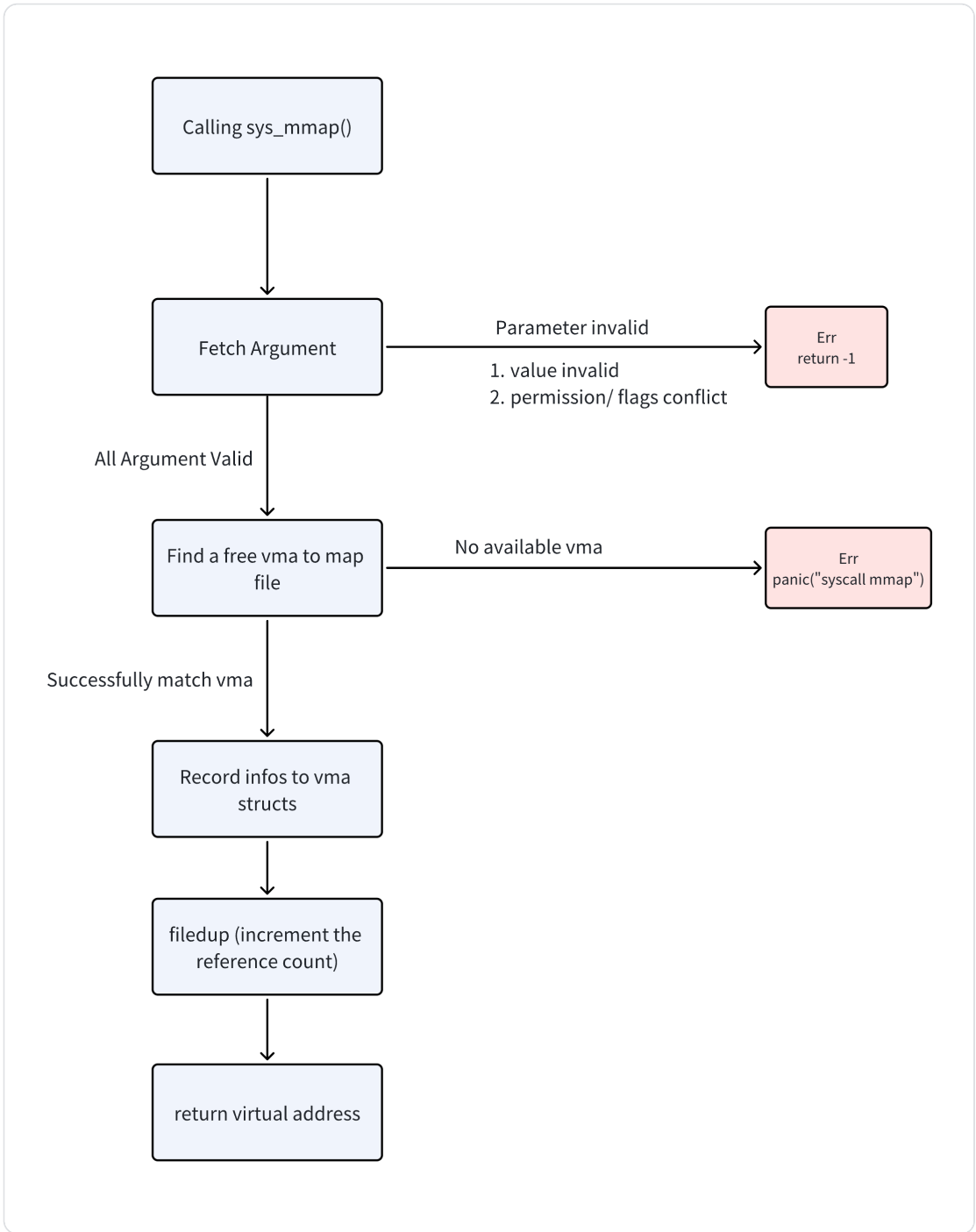
```
$ mmaptest
mmap_test starting
test mmap f
mismatch at 0, wanted 'A', got 0x1
mmaptest: mmap_test failed: v1 mismatch (1), pid=4
```

  - Page fault occurs after mmap() implemented (work correctly)

```
$ mmaptest
mmap_test starting
test mmap f
Now, after mmap, we get a page fault
usertrap(): unexpected scause 0x000000000000000d pid=6
            sepc=0x0000000000000076 stval=0x0000003fffffc000
$
```

Sample of page fault

Progress Chart



(TODO) PageFault Handle

<xv6-book> chapter 4.5, 4.6

- Add code to cause a page-fault in a mmap-ed region to allocate a page of physical memory.
- Find corresponding valid vma by fault address.
- Read 4096 bytes of the relevant file onto that page, and map it into the user address space.
- Read the file with readi, which takes an offset argument at which to read in the file (but you will have to lock/unlock the inode passed to readi).
- Set the permissions correctly on the page. Run mmaptest; it should get to the first munmap.
- See Section **Trap**

(TODO) munmap()

- Implement munmap:
  - find the VMA for the address range and unmap the specified pages (hint: use uvmunmap).
  - If munmap removes all pages of a previous mmap, it should decrease the reference count of the corresponding struct file.
  - If an unmapped page has been modified and the file is mapped MAP_SHARED, write the page back to the file. Look at **filewrite** for inspiration.
  - Ideally your implementation would only write back MAP_SHARED pages that the program actually modified. The dirty bit (D) in the RISC-V PTE indicates whether a page has been written. However, mmaptest does not check that non-dirty pages are not written back; thus, you can get away with writing pages back without looking at D bits.

```
1 // TODO: complete munmap()
2 uint64
3 sys_munmap(void)
4 {
5 }
6
7 //defined in vm.c
8 void uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free);
```

(BONUS) Fork Handle

- In your Assignment 1, you should already know that fork() creates a sub process with the same info. Therefore, you should handle how mmap() works when fork() is invoked.
- Ensure that the child has the same mapped regions as the parent. Don't forget to increment the reference count for a VMA's struct file. In the page fault handler of the child, it is OK to allocate a new physical page instead of sharing a page with the parent. The latter would be cooler, but it would require more implementation work.

Grading Rules

Program part 90' + bonus 10'

You can test the correctness of your code using the following commands under '~/xv6-labs-2022' directory.

```
make qemu
mmaptest
```

'make qemu' turns on the xv6 system, and you will see your terminal starting with '$'. You can execute 'ls' command to see the files including 'mmaptest'. 'mmaptest' command executes the executable file mmaptest to test your programs. You are expected to have the following outputs

```
$ mmaptest
mmap_test starting
test mmap f
off 0
off 4096
test mmap f: OK
test mmap private
off 0
off 4096
test mmap private: OK
test mmap read-only
test mmap read-only: OK
test mmap read/write
off 0
off 4096
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
```

```
22 off 0
23 off 0
24 test mmap two files: OK
25 mmap_test: ALL OK
26 fork_test starting
27 off 4096
28 off 0
29 off 0
30 off 0
31 off 4096
32 fork_test OK
33 mmaptest: all tests succeeded
```

mmap f	13p
not-mapped unmap	12p
mmap private	5p
mmap read-only	5p
mmap read/write	5p
mmap dirty	5p
mmap two files	5p
Compile Success	40p
fork_test (bonus)	10p

Report part 10'

You shall strictly follow **the provided latex template** for the report, where we have emphasized important parts and respective grading details. **Reports based on other templates will not be graded**.