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 proceses and to map files to 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, 13 November, 2024**
- **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 submit a perfect attemp of assignment3 on 2:10 am. He will get (100+10 (bonus)) * 0.7 = 77 points for his assignment3.)
- You should submit a zip file to the **Blackboard**. The zip file structure is as follows.

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 extra credit 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_xxxxxxxxx.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_xxxxxxxxx.zip Assignment_3_xxxxxxxxx
main@ubuntu:~/Desktop$ ls
Assignment_3_xxxxxxxxx Assignment_3_xxxxxxxxx.zip
```

Tips on interactions between host and virtual machine

Here are some useful tips for you to interact between the host machine and the virtual machine. If you are familiar with it and "**Format guide**", you can ignore this section.

*In the terminal, you should not include "<" and ">". Here, they are just to present a custom **string variable**.*

1. **Copy the assignment folder to your virtual machine.** You can copy the folder in the VSCode or use the scp command below.
   $\newline$
   In the **host** machine:
   $\newline$

   ```
   cd <your_host_path_to_project_zip>
   scp -P 2200 ./csc3150-project3.zip csc3150@127.0.0.1:~
   ```

   *If you **have spaces** in the path, use the double quote to include your path, e.g.* `cd "your host path"`.

2. **Unzip the assignment folder in your virtual machine.**
   $\newline$
   In the **virtual** machine:

   ```
   unzip ~/csc3150-project3.zip ~/
   chmod -R +x ~/csc3150-project3
   ```

   Then, you can browse the assignment folder.

   After finishing the project, you should wrap your file following the format instructions. We prepare a script for you to generate the submission zip. This optional script is just for your convenience to wrap the files. You can wrap your file in your own way, only ensuring that you follow the format.

3. Suppose that you have already copied your Report.pdf to the virtual machine (like the way you copy the assignment zip from the host machine to the virtual machine).

   In the **virtual** machine:

   ```
   cd ~/csc3150-project3
   bash gen_submission.sh
   ```

   `gen_submission.sh` script will ask for your student id and path of your `Report.pdf`.

   Then you can find your submission folder under `~/csc3150`
   `project3/submission/Assignment_3_<your_student_id>.zip`

4. You can use the following command to copy the submission zip to your **host** machine.

In the **host** machine:

```
scp -P 2200 csc3150@127.0.0.1:~/csc3150-
project3/submission/Assignment_3_<your_student_id>.zip
<your_host_machine_folder_path>
```

Then you will get the submission zip in `your_host_machine_folder_path` . **Don't forget to submit your zip file to the BlackBoard**.

Instrction Guideline

We limit your implementation within ***proc.c, proc.h, sysfile.c, trap.c*** four files, where there are some missing code sections starting with "**TODO**" comments. The entry (where you may start learning) of the test program is the main function in ***mmaptest.c*** under the 'csc3150-project3/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 some of the functions** when implementing the **TODO** parts.

You are **ONLY** allowed to modify the **TODO** parts in these four files! And we will grade your project **ONLY** based on the implementation of the **TODO** parts. Any other modification will be considered invalid.

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. (optional) 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 listed in detail. You should implement functions in the 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 nth 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*

```
// Define 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

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

Pages*

<xv6-book> chapter3

```
// Defined in riscv.h
typedef uint64 pte_t;
typedef uint64 *pagetable_t; // 512 PTEs

#endif // __ASSEMBLER__

#define PGSIZE 4096 // bytes per page
#define PGSHIFT 12  // bits of offset within a page

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

#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
```

```
// one beyond the highest possible virtual address.
// MAXVA is actually one bit less than the max allowed by
// Sv39, to avoid having to sign-extend virtual addresses
// that have the high bit set.
#define MAXVA (1L << (9 + 9 + 9 + 12 - 1))
```

Prots & Flags*

```
// Defined in fcntl.h
#define PROT_NONE 0x0
#define PROT_READ 0x1
#define PROT_WRITE 0x2
#define PROT_EXEC 0x4

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

(TODO) Traps

```
// trap.c
void usertrap(void)
{
    ...
    // TODO: manage pagefault
    else if(r_scause() == 13 || r_scause() == 15){
    ...
    }
    ...

}

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

Usertrap handles an interrupt, exception, or system call from user space. It calls r_scause() to get the exception code. In this assignment, you are asked to handle the PageFault exception.

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.

Hint:

- `r_stval()` provides trap value. (i.e. the address causing the exception)
- The swapping mechanism is not supported in the xv6 system. If the physical memory is filled, you are expected to kill the process. (You shall learn to use `kalloc()` and `setkilled()` functions)
- If there is spare space in physical memory, map one page of the file with the corresponding vma. (`mapfile()` and `mappages()`)

```
// file.c
// read a page of file to address mem
// The off parameter in the mapfile and readi represents the offset
// from the start of the file where the read operation should begin.
void mapfile(struct file * f, char * mem, int offset){
    // printf("off %d\n", offset);
    ilock(f->ip);
    readi(f->ip, 0, (uint64) mem, offset, PGSIZE);
    iunlock(f->ip);
}

// vm.c
// Create PTEs for virtual addresses starting at va that refer to
```

```
// physical addresses starting at pa. va and size might not
// be page-aligned. Returns 0 on success, -1 if walk() couldn't
// allocate a needed page-table page.
int mappages(pagetable_t pagetable, uint64 va, uint64 size, uint64 pa, int
perm)
{
    uint64 a, last;
    pte_t *pte;
    if(size == 0)
        panic("mappages: size");
    a = PGROUNDDOWN(va);
    last = PGROUNDDOWN(va + size - 1);
    for(;;){
        if((pte = walk(pagetable, a, 1)) == 0)
        return -1;
        if(*pte & PTE_V)
            panic("mappages: remap");
        *pte = PA2PTE(pa) | perm | PTE_V;
        if(a == last)
            break;
        a += PGSIZE;
        pa += PGSIZE;
    }
    return 0;
}
```

File*

```
// Defined in file.h
struct file {
    enum { FD_NONE, FD_PIPE, FD_INODE, FD_DEVICE } type;
    int ref; // reference count
    char readable;
    char writable;
    struct pipe *pipe; // FD_PIPE
    struct inode *ip; // FD_INODE and FD_DEVICE
    uint off; // FD_INODE
    short major; // FD_DEVICE
};

// in-memory copy of an inode
struct inode {
    uint dev; // Device number
    uint inum; // Inode number
    int ref; // Reference count
    struct sleeplock lock; // protects everything below here
    int valid; // inode has been read from disk?

    short type; // copy of disk inode
    short major;
    short minor;
    short nlink;
    uint size;
    uint addrs[NDIRECT+1];
};
```

```
// Write to file f.
// addr is a user virtual address.
int filewrite(struct file *f, uintaddr, int n);

// Increment ref count for file f.
struct file* filedup(struct file*);

// Close file f. (Decrement ref count, close when reaches 0.)
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()`)

```
// Defined in fs.c
// Read data from inode.
// Caller must hold ip->lock.
// If user_dst==1, then dst is a user virtual address;
// otherwise, dst is a kernel address.
int readi(struct inode *ip, int user_dst, uint64 dst, uint off, uint n);

// Write data to inode.
// Caller must hold ip->lock.
// If user_src==1, then src is a user virtual address;
// 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.

imilarly 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()

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

// TODO: kernel mmap executed in sysfile.c
uint64
sys_mmap(void)
{

}
```

- 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 then 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 whethermmap 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.

- 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.

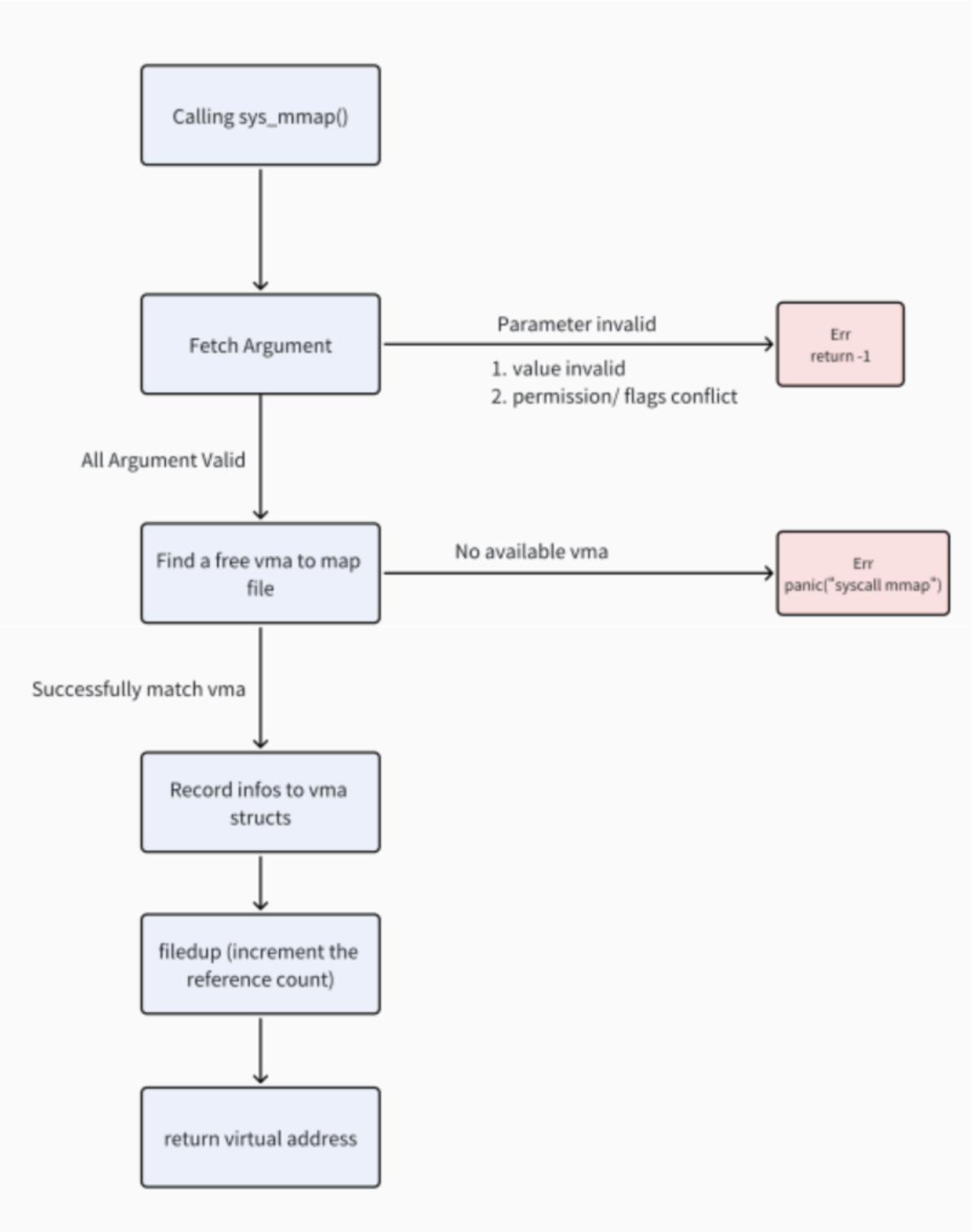
  - Before `mmap()` implemented

    ```
    $ 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
    $
    ```

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.

```
// TODO: complete munmap()
uint64
sys_munmap(void)
{
}

//defined in vm.c
void uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free);
```

(TODO) Page Alignment

This is a reminder to raise your awareness that all the virtual addresses in your kernel implementation should be page-aligned! It's very important to keep this rule in real implementation. That is to say, wrap the addresses with `PGROUNDUP` or `PGROUNDOWN` under different situations. You have to figure out which to use.

(EXTRA CREDITS) 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' + extra credits

You can test the correctness of your code using the following commands under `~/csc3150-project3` 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
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
```

function	points
mmap f	13p
mmap private	5p
mmap read-only	5p
mmap read/write	5p
mmap dirty	5p
mmap two files	5p
not-mapped unmap	12p
mmap offset	5p
mmap half page	15p
Compile Success	20p
fork_test (extra credit)	

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.**

LaTex Editor

For your convenience, you might use Overleaf, an online LaTex Editor.

1. Create a new blank project.
2. Click the following highlight bottom and upload the template we provide.
3. Click Recompile and you will see your report in PDF format.