Assignment Report: Title

In this report, when citing a reference, such as the xv6-book, feel free to use footnotes[1].

Your report should follow the template with the following section structure.

No page limitation

1 Introduction [3']

The overall strategy for implementing this assignment is called "lazy map". That is, only when there is a page fault, we start to use the allocation.

When the **‘mmap‘** function call is used, we only find the **‘VMA‘**. When the code start to access one position in the mapping, the ‘**usertrap‘** will be called, and inside the **‘usertrap‘**, we really start allocation memory.

As for **‘munmap‘,** we will write back no matter is there any dirty bits or not.

In the following session we will introduce how we design **‘VMA‘**, **‘mmap‘**, **‘usertrap‘**, **‘munmap‘** sequetially.

2 Implementation [5']

2.1 VMA

we design the vma like this:

```
struct vma {
  struct file *mapped_file;
  int mapped_length;
  uint64 mapped_starting_address;
  int memory_flags;
  int mapped_flags;
  int max_page_length;
  int offset;
  int used_length;
};
```

Although there are some filed that is not used in this project yet such as mapped_flags and max_page_length, but other fields are useful.

- We use the mapped_file track the file we are maping into
- We use the mapped_sttarting_address to check the virtual_address of the starting mapping address
- We use the used length to track where is the mapping no

2.2 mmap

After we design the VMA structure, we need to combine VMA together with the memory allocation.

[1]Reference: https://pdos.csail.mit.edu/6.828/2005/readings/pdp11-40.pdf

2.2.1 Overview

We take the argument:

```
  argint(0, &virtual_or_physical);
  argint(1, &length);
  argint(2, &memory_flags);
  argint(3, &sharing_flags);
  argint(4, &fd);
  argint(5, &offset);
};
```

- We check if arguments we have is legal
- We find empty vma
- We set empty vma
- We return the starting address of mapping recorded in the vma

There are several key helper function:

- vma_searching
- get_file_from_fd
- memory_flag_generator
- set_vma

2.2.2 vma_searching

We iterate through all the available vma in the vma array. When we find a vma that has not been used, then we use it directly. For simplification, I don't want to map multiple file to the same VMA space.

Interestingly, in XV6 system, we find that the initialized structure will be set to zero instead of null. Grasp of functions is here:

```
int vma_searching(int bytes_number, struct proc *p) {
    for (int i = 0; i < VMASIZE; i++) {
        if (p->vma[i].mapped_file == 0 && p->vma[i].mapped_length == 0) {
            return i; // Empty VMA slot found.
        }
    }
    return -1; // No empty VMA slot found.
}
```

Notice that, kernel uses proc→ofile to maintained the file structure, we need to lock this process, since in the followin steps we filedup this file.

2.2.3 memory$_f lags_g enerator$

This function is used to translate map_flags and shared_flags to the corrsponding memory flags. The mapping reason can be checked in fcntl.h and vm.c and so on. In this file we simply do the following mapping:

```
int memory_flag_generator(int map_flags, int shared_flags){
  int page_flags = PTE_U|PTE_V;
  if (map_flags&0x1){
    page_flags = page_flags|PTE_R;
  }
  if (map_flags&0x2){
    page_flags = page_flags|PTE_W;
```

```
  }
  if (map_flags&0x4){
    page_flags = page_flags|PTE_X;
  }

  return page_flags;
}
```

Since, all the mapped region will be used in the userspace, we set it to be valid and user mode.

2.2.4 set_vma

We transfer the argument we take and the memory flags we translate to an empty_vam so that we can use it in the future interrupt:

```
void set_vma(
              int vma_number,
              int mapped_length,
              int memory_flags,
              int shared_flags,
              int offset,
              struct file *mapped_file,
              struct proc *p) {
    acquire(&p->lock); // Assuming xv6 style lock for the process table.
    p->vma[vma_number].mapped_file = mapped_file;
    p->vma[vma_number].mapped_length = mapped_length;
    p->vma[vma_number].memory_flags = memory_flag_generator(memory_flags, shared_flags);
    p->vma[vma_number].mapped_flags = shared_flags;
    p->vma[vma_number].mapped_starting_address = (uint64)MAXVA-2*PGSIZE - (vma_number + 1) * 8 *
    p->vma[vma_number].offset = offset;
    p->vma[vma_number].used_length = 0;
    release(&p->lock);
}
```

Notice that we set the starting address like this is because, the space right below trapframe is empty, and trapoline and trapframe each take one page 1. We set the max size of the VMA to be 8*PGSIZE, that is because, it is already really large

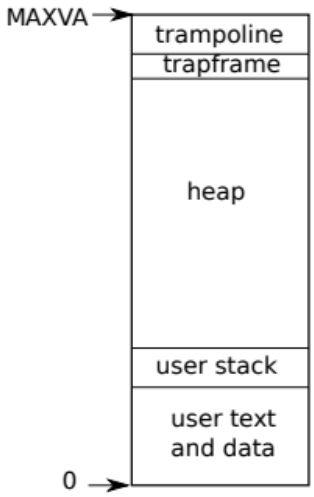

Figure 1: This memory layout shows that there are much space left for heap

2.3 PageFault

In usertrap will be called when user trying to access some memory that does not set as valid yet or trying to write unwritable space or unreadable space and so on...

2.3.1 overview

- We obtain the virtual address that the user space want to access using this r_staval()
- Check the VMA array is there any vma contain this address using function find_trap_vma, if yes that means we need to do further mapping
- If there is some mapping information, use that information to read file and write it to the address using memory_mapping
- If there is no vma fited, we might need to copy the parents vma using lazy_copy(in fork test)
- If the pagefault still there, we then goto err

2.3.2 find_trap_vma

We iterate though all vma to find if there any mapping corresponding to the va area:

```
void *find_trap_vma(uint64 question_address, struct proc* p){
  for (int i = 0; i < VMASIZE; i++) {
      if ((question_address>=p->vma[i].mapped_starting_address) &&
         (p->vma[i].mapped_starting_address+8*PGSIZE> question_address)) {
          return &p->vma[i]; //corresponding vma found
      }
  }
  return 0;
}
```

2.3.3 memory_mapping

We use this function to actually do the mapping. Since everytime, we only ma one page, we do not need to use a for loop. And the key function we used here is

- kalloc: allocate a physical memory
- mappages: update pagetable, correlate physical page and virtual page
- mapfile: read file to physical address
  So here is the function:

```
int memory_mapping(struct vma *trap_vma, struct proc *p) {
    struct file *f = trap_vma->mapped_file;
    acquire(&p->lock);
    char *mem = kalloc(); // Allocates one page of physical memory.
    if (mem == 0) {
        release(&p->lock);
        printf("we have out of memory!\n");
        return -1; // Out of memory.
    }
    memset(mem, 0, PGSIZE);
    // Read the content from file into the memory page with the given offset.
    if (f->ip == 0) {
        // The file is not open, handle accordingly
        kfree(mem);
        release(&p->lock);
        printf("File is not open.\n");
        return -3; // Indicate that the file is not open.
    }
```

```
    mapfile(f, mem, trap_vma->used_length+trap_vma->offset);
    if (mappages(p->pagetable, trap_vma->mapped_starting_address+trap_vma->used_length, PGSIZE,
        kfree(mem);
        release(&p->lock);
        printf("we fail to map the page");
        return -3; // Failed to map page.
    }
    trap_vma->used_length += PGSIZE;
    release(&p->lock);
    return 0;
}
```

2.4 lazy_copy

will be discussed in fork test

2.5 munmap

Basic idea is no matter how, just write back.

- find unmapp vma
- writeback
- change page_table: uvmunmap
- check is the vma used length is zero
- if so close file

```
uint64
sys_munmap(void)
{
  uint64 va;
  int length;
  argaddr(0, &va);
  argint(1, &length);
  struct proc *p = myproc();
  struct vma *unmap_vma = find_unmap_vma(va, p);

  // we need to find the corresponding vma first
  if (unmap_vma == 0){
    printf("there is no mapping vma");
    return -1;
  }

  // we need to write it back no matter how
  // write back the page first
  for (int i =0; i<length/PGSIZE; i++){
    filewrite(unmap_vma->mapped_file, va+PGSIZE*i, PGSIZE);
    // free the page and decrease the used page
  }
  // we need to do the unmap
  uvmunmap(p->pagetable,va,length/PGSIZE,1);

  // we then need to change the vma correspodningly
  unmap_vma->mapped_length -= length;
  // if mapped length is already zero or less, we need to clear the VMA
  // and decrease the file reference
  if (unmap_vma->mapped_length <= 0){
    fileclose(unmap_vma->mapped_file);
    unmap_vma->mapped_file=0;
```

```
    unmap_vma->mapped_length=0;
  }
  return 0;
}
```

2.6 Bonus

we should handle this section in fork(), but not allowed. Therefore, I handle this in for using lazy coppy When in fork test, child should check parent's vma and copy their vma, and call the memory_mapping again:

```
int lazy_copy(struct vma *parents_vma, struct proc *child_p){
  acquire(&child_p->lock);
  struct vma *child_vma = vma_searching_trap(child_p);
  child_vma->mapped_file = parents_vma->mapped_file;
  child_vma->memory_flags = parents_vma->memory_flags;
  child_vma->mapped_flags = parents_vma->mapped_flags;
  child_vma->mapped_length = parents_vma->mapped_length;
  child_vma->mapped_starting_address = parents_vma->mapped_starting_address;
  child_vma->max_page_length = parents_vma->max_page_length;
  child_vma->offset = parents_vma->offset;
  child_vma->used_length = 0;
  filedup(child_vma->mapped_file);
  release(&child_p->lock);
  return memory_mapping(child_vma, child_p);
}
```

3 Test [2']

Briefly discuss which part of your implementation helped you pass each test. (Several sentences for each subsection should suffice.)

3.1 mmap f

Returm a virtual address and use mappage

3.2 mmap private

Checking the flags

3.3 mmap read-only

Checking the flags

3.4 mmap read/write

Implementing munmap

3.5 mmap dirty

Implementing munmap

3.6 mmap two files

It just work out