浙江大学

操作系统实验报告

课程名称	:	操作系统
姓　　名	:	沈一芃
学　　院	:	计算机学院
专　　业	:	计算机科学与技术
学　　号	:	3220101827
指导教师	:	李环 柳晴

2024 年　12　月　16　日

Lab 5 Report

实验名称：RV64 缺页异常处理与 fork 机制

联系方式：1669335639@qq.com 18310211513

1. 实验目的与要求

1.1. 实验目的

- 通过 **vm_area_struct** 数据结构实现对进程**多区域**虚拟内存的管理
- 在 Lab4 实现用户态程序的基础上，添加缺页异常处理 **page fault handler**
- 为进程加入 **fork** 机制，能够支持通过 **fork** 创建新的用户态进程

1.2. 实验要求

- 实现进程的多区域虚拟内存管理
- 正确处理三种缺页异常
- 实现fork机制，创建新的用户态进程

2. 实验环境

本机CPU	本机操作系统	实验环境	实验操作系统	配置环境
Mac Silicon M3	MacOS Sequoia 15.0.1	Docker 4.33.0	Ubuntu 24.10	Qemu-system-riscv64 9.0.2

3. 实验过程

3.1. 实验环境搭建

- 整个实验建立在lab4的基础上。从仓库克隆实验5代码到本地，跟随实验流程搭建文件结构如下：

```
├── arch
│   └── riscv
│       ├── include
│       │   ├── mm.h
│       │   ├── proc.h
│       │   ├── clock.h
│       │   ├── syscall.h   # need to modify; add do_fork
```

```
│       │   ├── defs.h
│       │   └── sbi.h
│       ├── kernel
│       │   ├── vm.c
│       │   ├── mm.c
│       │   ├── proc.c      # need to modify; don't allocate/copy/map the program here
│       │   ├── syscall.c   # need to accomplish; implement do_fork
│       │   ├── sbi.c
│       │   ├── trap.c      # need to accomplish; enable and handle page fault
│       │   ├── clock.c
│       │   ├── head.S
│       │   ├── entry.S
│       │   ├── Makefile
│       │   └── vmlinux.lds
│       └── Makefile
├── include
│   ├── elf.h
│   ├── printk.h
│   ├── stddef.h
│   ├── stdint.h
│   ├── stdlib.h
│   └── string.h
├── init
│   ├── main.c
│   ├── Makefile
│   └── test.c
├── lib
│   ├── Makefile
│   └── printk.c
├── user
│   ├── Makefile          # need to modify; allow PFH and FORK TEST
│   ├── main.c            # need to modify; directly pull from storage and replace
getpid.c
│   ├── link.lds
│   ├── printf.c
│   ├── start.S
│   ├── stddef.h
│   ├── stdio.h
│   ├── syscall.h
│   └── uapp.S
└── Makefile
```

- 修改user的makefile文件，加入测试项目

```
TEST        = PFH1
CFLAG       = ... -D$(TEST)
```

3.2. 缺页异常处理

3.2.0. 总括

- 本次实验的第一部分是处理缺页异常。所谓缺页异常，就是在开启虚拟内存后遇到一条正确映射或者权限错误的指令/地址。在lab4中，我们在创建进程时将elf文件加载到了对应的位置，开辟空间，复制内容，进行映射，保证了所有的指令和地址在程序被调度前都已全部加载到内存中并进行了映射。这种操作方式在更复杂的系统中会出现问题：1. 内存的大小是有限的，不一定能将所有进程都全部加载进内存。2. 所有的进程不一定在一开始就全部被加载。
- 我们采取了两种方式来解决上述问题：1. 使用`vm_area_struct`管理虚拟地址，记录虚拟地址要映射的物理区域的相关信息，如具体位置、读写权限等。2. 使用`demand_mapping`方法，仅当需要时才将对应页加载至磁盘并映射。
- 综上所述，lab5这一部分要做的，就是修改lab4中一开始就加载/开辟/映射，而是一开始先建立vma记录映射等关系，方便后续映射；之后开始运行，当调度到某一个程序，在某一条指令发生缺页时，再将该页加载至内存并映射。

3.2.1. 定义VMA数据结构与处理函数

- vma的数据结构和相关宏定义如下：

```
#define VM_ANON 0x1
#define VM_READ 0x2
#define VM_WRITE 0x4
#define VM_EXEC 0x8
struct vm_area_struct {
    struct mm_struct *vm_mm;     // 所属的 mm_struct
    uint64_t vm_start;           // VMA 对应的用户态虚拟地址的开始
    uint64_t vm_end;             // VMA 对应的用户态虚拟地址的结束
    struct vm_area_struct *vm_next, *vm_prev;    // 链表指针
    uint64_t vm_flags;           // VMA 对应的 flags
    // struct file *vm_file;     // 对应的文件（目前还没实现，而且我们只有一个 uapp 所以暂不需要）
    uint64_t vm_pgoff;           // 对应文件时这块 VMA 起始地址对应的文件内容相对文件起始位置的偏移量
    uint64_t vm_filesz;          // 对应的文件内容的长度
};

struct mm_struct {
    struct vm_area_struct *mmap;
};
```

- 相应的，在`init_task`中，我们要更改过去的加载逻辑，并用`uint64_t do_mmap()`函数初始化vma区域
    - 这里对于vma的处理，是申请一个全新的页。显然，这样十分浪费内存。可以在buddy system中自行修改，添加一个小块内存分配的接口

```
uint64_t do_mmap(struct mm_struct *mm, uint64_t addr, uint64_t len, uint64_t vm_pgoff, uint64_t vm_filesz, uint64_t flags){
    struct vm_area_struct *new_vas = alloc_page();
```

```
    memset(new_vas, 0, PGSIZE);
    // for (uint64_t i = 0; i < PGSIZE; i++)   *((char*)new_vas + i) = 0;
    new_vas->vm_mm = mm;
    new_vas->vm_start = addr;
    new_vas->vm_end = addr + len;
    new_vas->vm_pgoff = vm_pgoff;
    new_vas->vm_filesz = vm_filesz;
    new_vas->vm_flags = flags;
    new_vas->vm_prev = NULL;
    new_vas->vm_next = mm->mmap;
    mm->mmap = new_vas;

    return new_vas->vm_start;
}
```

- vma以链表的形式串在一起，用`mm_struct`表示。每个task中包含一个`mm_struct`。故而，我们还需要一个函数`struct vm_area_struct *find_vma()`搜索整个链表，找到对应的vma。

```
struct vm_area_struct *find_vma(struct mm_struct *mm, uint64_t addr){
    struct vm_area_struct *tmp_vas = mm->mmap;
    while(tmp_vas != NULL){
        if (tmp_vas->vm_start <= addr && addr <= VM_END)
            return tmp_vas;
        else
            tmp_vas = tmp_vas->vm_next;
    }
    return NULL;
}
```

3.2.2. 修改`task_init`实现demand mapping

- 在`task_init`中，初始化task的pgd后，不再调用旧`load_program`读elf后分配空间/复制内容/建立映射，不再分配空间给用户栈并建立映射。应该调用新的`load_program`读elf后建立vma，并再为用户栈创建一块vma。
    - 本次实验中两块vma的区别是，用户栈的vma具有`anonymous`匿名属性。

```
void task_init() {
   ...
   for (int i = 1; i < nr_tasks; i++){
       ... // after initializing pgd
       load_program(task[i]);
       do_mmap(&task[i]->mm, USER_END - PGSIZE, PGSIZE, -1, -1, VM_ANON | VM_READ |
VM_WRITE);
   }
}
void load_program(struct task_struct *task) {
    Elf64_Ehdr *ehdr = (Elf64_Ehdr *)_sramdisk;
    Elf64_Phdr *phdrs = (Elf64_Phdr *)(_sramdisk + ehdr->e_phoff);
```

```
    for (int i = 0; i < ehdr->e_phnum; ++i) {
        Elf64_Phdr *phdr = phdrs + i;
        if (phdr->p_type == PT_LOAD) {
            printk("entry: %lx, memsz: %lx, filesz: %lx\n", ehdr->e_entry, phdr-
>p_memsz, phdr->p_filesz);
            do_mmap(&task->mm, ehdr->e_entry, phdr->p_memsz, phdr->p_offset, phdr-
>p_filesz, VM_EXEC | VM_READ | VM_WRITE);
        }
    }
    task->thread.sepc = ehdr->e_entry;
}
```

3.2.3. 捕获并处理缺页异常

- 当程序运行时会发生缺页异常，我们在`trap_handler`中根据`scause`完成对应的接口，并在接口中调用`do_page_fault`处理缺页异常即可。缺页异常的处理逻辑为：
  - 读取`stval`寄存器中存下的异常地址
  - 查找该地址属于vma的哪一块。若不存在于vma中，则是一个错误地址，无法处理。
  - 比较该地址的权限与缺页异常类型，不允许一条没有vma执行权限的缺页指令加载并映射。
  - 对本实验，通过查看anonymous属性，确定缺页属于用户栈还是程序部分。
    - 若用户栈缺页，直接分配空间并映射即可。
    - 若程序缺页，则需要分类讨论程序缺页位置，保证内容对齐。

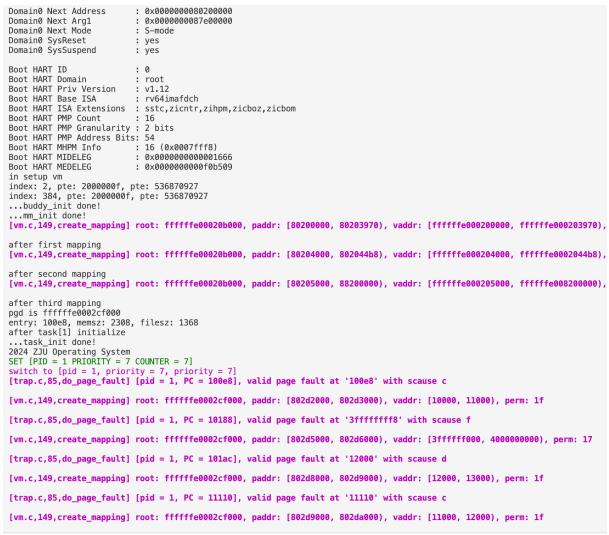
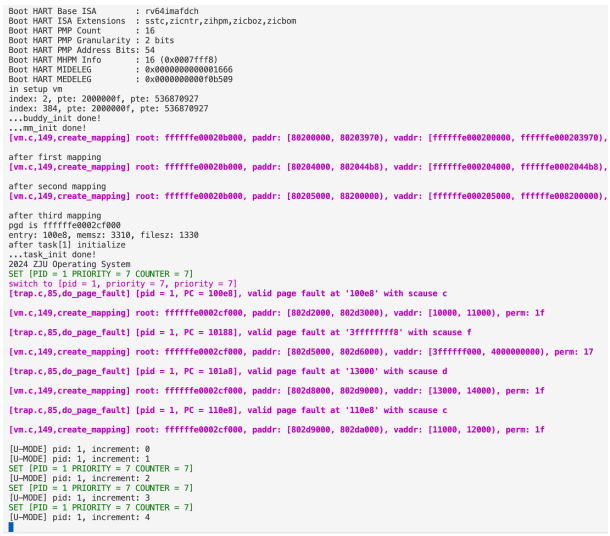
> 1. 此处保证对齐的原因在lab4中已经阐述过。vma中的程序开始地址vm_start（也就是elf中的e_entry）并不一定是4kb页对齐的。若直接把从vm_start开始的4kb页复制到新分配的4kb页中可能导致一条指令被截断在两页中。
> 2. 从e_filesz到e_memsz之间还有一段空闲空间。这段空间虽然置0，但也需要分配空间并进行映射。
> 3. 故而当deman mapping加载一页至内存并映射时，需要考虑缺页地址的页下界/页上界和vm_start /vm_end /vm_start+vm_filesz的关系。要实现的效果是：内存中分配一页，将vm_start开始到vm_end某一整页的内容对齐的加载到分配页上，建立映射。
> 4. ⚠️ 这里的分配存在一个问题。我们将在讨论与心得中展开分析。

```
void do_page_fault(struct pt_regs *regs, uint64_t scause) {
    uint64_t stval = csr_read(stval);
    struct vm_area_struct* tmp_vma = find_vma(&(current->mm), stval);
    if (tmp_vma == NULL){
        Err("Unexpected virtual address: %lx\n", stval);
    } else {
        if (regs->a0 == 0xc && !(tmp_vma->vm_flags & VM_EXEC)){
            Err("VMA doesn't support EXEC\n");
        }else {
            Log("[pid = %d, PC = %lx], valid page fault at '%lx' with scause %lx\n",
current->pid, regs->sepc, stval, scause);
```

```
            if (tmp_vma->vm_flags & VM_ANON){
                uint64_t *tmp_page = alloc_page();
                memset(tmp_page, 0, PGSIZE);
                // for (uint64_t i = 0; i < PGSIZE; i++)   *((char*)tmp_page + i) =
0;
                create_mapping(current->pgd, PGROUNDDOWN(stval), (uint64_t)tmp_page -
PA2VA_OFFSET, PGSIZE, 0x17);
            } else {

                uint64_t *paddr = alloc_page();
                memset(paddr, 0, PGSIZE);
                uint64_t VUP = PGROUNDUP(stval);
                uint64_t VDOWN = PGROUNDDOWN(stval);
                struct vm_area_struct *vma = tmp_vma;
                void *page = paddr;
                uint64_t page_start;
                uint64_t file_start;
                uint64_t map_start = MAX(vma->vm_start, PGROUNDDOWN(stval));
                uint64_t map_end = MIN(vma->vm_start + vma->vm_filesz,
PGROUNDDOWN(stval)+PGSIZE);
                if (map_start < map_end) {
                    page_start = map_start - PGROUNDDOWN(stval);
                    file_start = vma->vm_pgoff + map_start - vma->vm_start;
                    memcpy(page + page_start, _sramdisk + file_start, map_end -
map_start);
                    create_mapping(current->pgd, PGROUNDDOWN(stval), (uint64_t)(page
- PA2VA_OFFSET), PGSIZE, 0x1f);
                } else {
                    if (vma->vm_end >= PGROUNDDOWN(stval))
                        create_mapping(current->pgd, PGROUNDDOWN(stval), (uint64_t)
(page - PA2VA_OFFSET), PGSIZE, 0x1f);
                    else
                        Err("absolute wrong addr");
                }
            }
            asm volatile ("sfence.vma zero, zero");
        }
    }
}
```

3.2.4 测试

- 至此，我们实现了缺页异常的处理。对两个TEST-PFH的测试结果如下

make run TEST=PFH1	make run TEST=PFH2
	

3.3. fork系统调用

3.3.0. 总括

- fork的目的，是复制一个进程，建立一个新进程。此处的复制，不仅仅是对程序内容以及进程本次的priority/counter的复制，也是对进程的执行状态的复制。对lab5，父进程调用`fork`函数，触发ecall系统调用；当fork出的子进程第一次被调度时，它要从父进程触发ecall后的状态开始运行。分析系统调用流程，即在`_trap`中完成对寄存器上下文的保存后进入`trap_handler`，子进程需要从`trap_handler`处理完系统调用后开始运行，即跳转回`_trap`处开始执行。
- 故而，fork的流程可以理解为：复制task中记录的进程信息（如priority），复制程序部分内容，复制上下文状态信息。
    - **其中，上下文状态其实就是子进程如何正确模拟父进程处理完ecall后回归U态继续执行。**
    - 梳理父进程返回流程：`do_fork` -> `trap_handler` -> `_trap` -> `U态 fork函数后`。而子进程初次被调度时的流程则是：`schedule` -> `switch_to` -> `__switch_to` -> `U态ra` -> (此处要返回至父进程ecall后继续运行的状态，即：) `_trap` -> `U态 fork函数后`。显然，这里经过了一个从U态进入S态又回到U态的过程。我们需要处理的是U态和S态的寄存器状态，U态和S态的栈空间，U态和S态的栈指针，CSR寄存器。
- 从实现上，我们又可以将四部分改写成三块内容：处理task结构，处理页表和vma，处理返回相关指针。其中对应关系较为复杂。task结构包扩进程信息；此外，

3.3.1. 处理task结构

- 这一部分复制进程信息和部分上下文状态信息，同时更新nr_task数量以及pid。

> 1. S态的寄存器就存在S态的栈空间内，而这片空间和task位于同一页，故而复制task的过程也就复制了S态栈，只需要知道S态sp即可恢复出S态寄存器。

2. CSR寄存器的U态的寄存器状态为系统调用前的状态。而我们在复制task时，也复制了CSR寄存器和U态寄存器的值，只不过这些值是父进程刚被调度时的值。其中的部分内容在运行至系统调用的过程中发生了变动。后续还需要修改。

3.3.2. 处理页表和vma

- 处理页表和vma只需要复制父进程的页表和vma即可。但此处，我们将复制页表和复制程序部分内容、复制U态栈合并。

1. 由于我们要完全复刻父进程的状态，所以我们要在此处将父进程已经加载进内存并映射好的程序/U态栈部分进行同样的加载/映射。通过vma，我们可以查看一共有多少可能需要加载和映射的页。再根据复制的页表，查看每一页是否进行了映射（valid位情况），即可确定这一页是否被加载和映射。
2. vma包含程序和U态栈两部分。故而此处解决了这两部分内容的复制。

3.3.3. 处理返回相关指针

- 此处要处理的，就只剩下部分CSR寄存器、U态寄存器以及S态栈指针了。
- 进一步分析，CSR寄存器和U态寄存器中，从调度到系统调用，发生了需要记录的变化的寄存器只有U态sp。与此同时，由于要模拟3.3.0中提到的父进程返回链，ra以及sepc这两个和返回相关的值也需要重新决定赋值。
- 这一部分的具体分析详见思考题。

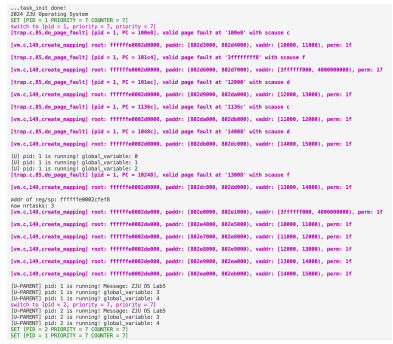
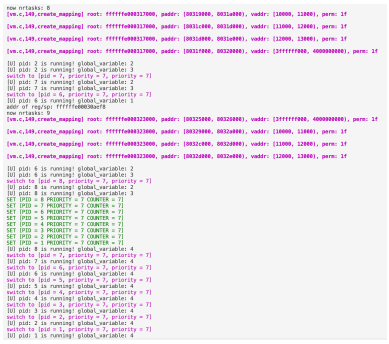
3.3.4. 完整do_fork与测试

- 完整的`do_fork`函数如下：

```
uint64_t do_fork(struct pt_regs *regs){
    printk("addr of reg/sp: %lx\n", regs);
    // Err("unfinished\n");
    nr_tasks++;
    printk("now nrtasks: %lu\n", nr_tasks);
    struct task_struct *_task = alloc_page();
    memcpy(_task, current, PGSIZE);
    task[nr_tasks-1] = _task;
    uint64_t *_pgd = alloc_page();
    memset(_pgd, 0, PGSIZE);
    _task->pid = nr_tasks - 1;
    _task->pgd = _pgd;
    _task->thread.ra = (uint64_t)__ret_from_fork;
    _task->mm.mmap = NULL;
    for(uint64_t i = 0; i < 512;i++){
        _pgd[i] = swapper_pg_dir[i];
    }
```

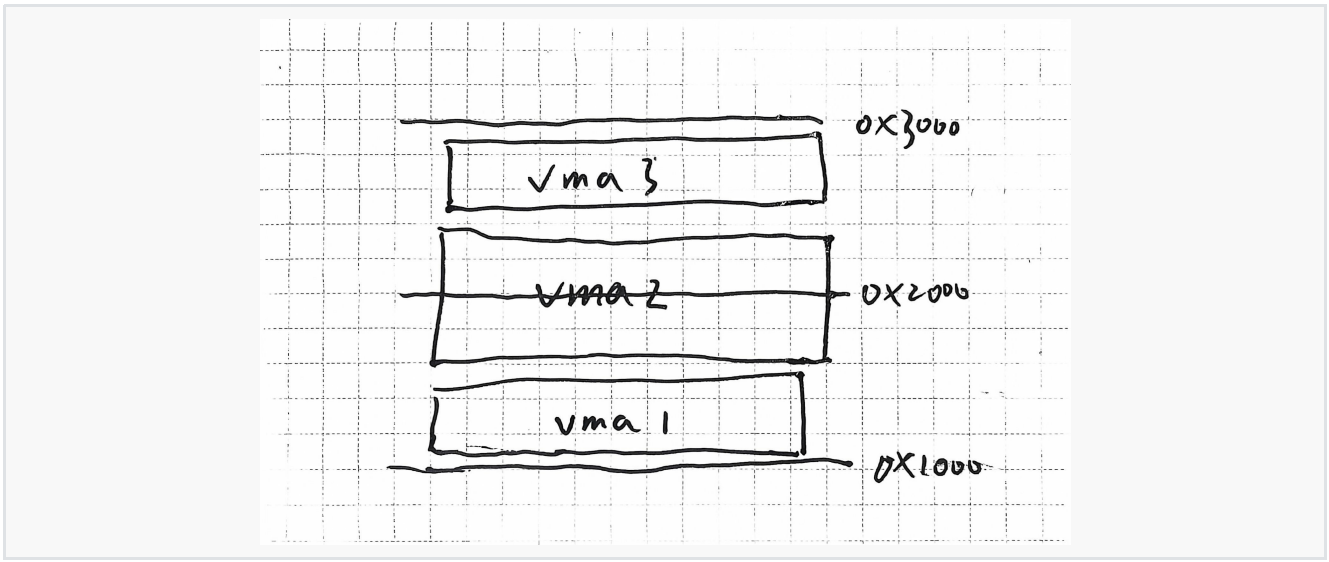
```
    struct vm_area_struct *tmp_vma = current->mm.mmap;
    while(tmp_vma != NULL){
        struct vm_area_struct *new_vma = alloc_page();
        memcpy(new_vma, tmp_vma, PGSIZE);
        new_vma->vm_next = _task->mm.mmap;
        new_vma->vm_prev = NULL;
        _task->mm.mmap = new_vma;
        //read and map
        uint64_t page_addr = new_vma->vm_start;
        for(;page_addr < new_vma->vm_end;){
            uint64_t *tmptbl = current->pgd;
            uint64_t vpn2 = (page_addr >> 30) & 0x1ff;
            uint64_t vpn1 = (page_addr >> 21) & 0x1ff;
            uint64_t vpn0 = (page_addr >> 12) & 0x1ff;
            uint64_t pte = tmptbl[vpn2];
            if ((pte & 0x1) == 0){
                page_addr = PGROUNDDOWN(page_addr + PGSIZE);
                continue;
            }
            tmptbl = (uint64_t*)(((pte >> 10) << 12) + PA2VA_OFFSET);
            pte = tmptbl[vpn1];
            if ((pte & 0x1) == 0){
                page_addr = PGROUNDDOWN(page_addr + PGSIZE);
                continue;
            }
            tmptbl = (uint64_t*)(((pte >> 10) << 12) + PA2VA_OFFSET);
            pte = tmptbl[vpn0];
            if ((pte & 0x1) == 0){
                page_addr = PGROUNDDOWN(page_addr + PGSIZE);
                continue;
            }
            uint64_t *copy_addr = (uint64_t*)(((pte >> 10) << 12) + PA2VA_OFFSET);
            uint64_t *phy_addr = alloc_page();
            memset(phy_addr, 0, PGSIZE);
            memcpy(phy_addr, copy_addr, PGSIZE);
            create_mapping(_task->pgd, PGROUNDDOWN(page_addr), (uint64_t)phy_addr -
PA2VA_OFFSET, PGSIZE, 0x1f);

            page_addr = PGROUNDDOWN(page_addr + PGSIZE);
        }

        tmp_vma = tmp_vma->vm_next;
    }
    _task->thread.sp = ((uint64_t)regs & 0xfff) + (uint64_t)_task;
    struct pt_regs *child_regs = (struct pt_regs*)_task->thread.sp;
    _task->thread.sepc = _task->thread.sepc + 4;
    child_regs->sepc = _task->thread.sepc;
    _task->thread.sscratch = csr_read(sscratch);

    return nr_tasks;
}
```

- 测试结果如下：

make run TEST=FORK1	make run TEST=FORK2	make run TEST=FORK3
		

4. 讨论与心得

- 大多数心得其实已经在`3.2.0`和`3.3.0`两个总括中写过。此处着重讲一下`3.2.3`中提到的问题。本次实验中的vma过于简单，只有两块内容，且地址相差甚远——一块位于用户虚拟空间开始，另一块位于用户虚拟空间结尾。但对于复杂的vma，如果出现两块虚拟地址相邻的vma，则会出现一个无法解决的映射问题：映射vma1时，只会映射0x1000-0x2000中vma1部分的内容；此后页表中0x1000至0x2000的部分valid位为1。这样一来，vma2处与0x1000-0x2000的部分将永远无法映射成功，会永远陷入缺页异常。



    - 这个问题没有解决的办法。目前唯一认为可能的解释就是，我们的程序太过简单，未考虑这种情况。实际操作中，或许elf文件的格局分布会避免有用的需要load的文件存在页重合。

5. 思考题

5.1. 呈现出你在 page fault 的时候拷贝 ELF 程序内容的逻辑

- vma中的程序开始地址vm_start（也就是elf中的e_entry）并不一定是4kb页对齐的。若直接把从vm_start开始的4kb页复制到新分配的4kb页中可能导致一条指令被截断在两页中。

- 从e_filesz到e_memsz之间还有一段空闲空间。这段空间虽然置0，但也需要分配空间并进行映射。
- 故而当deman mapping加载一页至内存并映射时，需要考虑缺页地址的页下界/页上界和vm_start /vm_end /vm_start+vm_filesz的关系。要实现的效果是：内存中分配一页，将vm_start开始到vm_end某一整页的内容对齐的加载到分配页上，建立映射。

```
  uint64_t *paddr = alloc_page();
  memset(paddr, 0, PGSIZE);
  uint64_t VUP = PGROUNDUP(stval);
  uint64_t VDOWN = PGROUNDDOWN(stval);
  struct vm_area_struct *vma = tmp_vma;
  void *page = paddr;
  uint64_t page_start;
  uint64_t file_start;
  uint64_t map_start = MAX(vma->vm_start, PGROUNDDOWN(stval));
  uint64_t map_end = MIN(vma->vm_start + vma->vm_filesz, PGROUNDDOWN(stval)+PGSIZE);
  if (map_start < map_end) {
    page_start = map_start - PGROUNDDOWN(stval);
    file_start = vma->vm_pgoff + map_start - vma->vm_start;
    memcpy(page + page_start, _sramdisk + file_start, map_end - map_start);
    create_mapping(current->pgd, PGROUNDDOWN(stval), (uint64_t)(page - PA2VA_OFFSET),
PGSIZE, 0x1f);
  } else {
    if (vma->vm_end >= PGROUNDDOWN(stval))
      create_mapping(current->pgd, PGROUNDDOWN(stval), (uint64_t)(page -
PA2VA_OFFSET), PGSIZE, 0x1f);
    else
      Err("absolute wrong addr");
  }
```

5.2. 回答问题

- 在 do_fork 中，父进程的内核栈和用户栈指针分别是什么?
  - 父进程的内核栈指针是sp，是进入系统调用前在`_trap`中保存了寄存器后指向的位置。用户态指针是sscratch，是系统调度发生前的用户态指针的值。
- 在 do_fork 中，子进程的内核栈和用户栈指针的值应该是什么?
  - 子进程的内核栈指针应该和父进程在栈内（页内）指向的相对位置相同，只不过是指向子进程的栈空间（页）。
  - 子进程的用户栈指针应该是父进程在系统调度发生前的用户态指针的值，即读取sscratch。注意，这里不能用task里存的用户栈上的sscratch，因为这个sscratch是父进程刚刚被调度执行时sscratch的值，而这时距离系统调度发生还有一段要运行的代码，sscratch会发生变化。
- 在 do_fork 中，子进程的内核栈和用户栈指针分别应该赋值给谁?
  - 子进程的内核栈指针赋给sp，用户态指针赋给sscratch，因为程序第一次调度后会跳转至ra，而ra中应该模拟父进程完成系统调用后的状态，写入完成系统调用回到`_trap`后的`__ret_from_fork`处，继续完成对调用前栈上寄存器的恢复，并交换sp与sscratch，使sp指向用户栈，sscratch指向内核栈。

5.3. 为什么要为子进程 `pt_regs` 的 `sepc` 手动加四?

- 因为子进程会通过`ra`返回到`_trap`中的`__ret_from_fork`，恢复栈上存储的sepc，这一步会覆盖掉直接在task中写的sepc的值；这解释了为什么要修改`pt_regs`中的sepc。
- 而修改sepc为+4后的值，则是因为sepc是用户程序中fork系统调用处。子进程应该从fork后的下一行代码开始执行。

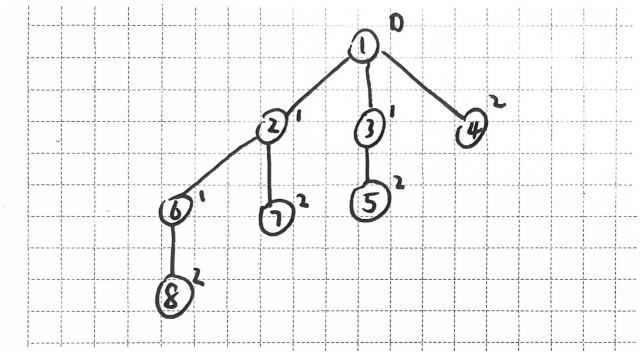
5.4. 对于 `Fork main #2`（即 `FORK2`），在运行时，`ZJU OS Lab5` 位于内存的什么位置？是否在读取的时候产生了 page fault？请给出必要的截图以说明。

位于内存的13000-14000部分，产生了page fault。连续三个[U]输出后的缺页异常就是ZJU OS lab5导致的

```
...task_init done!
2024 ZJU Operating System
SET [PID = 1 PRIORITY = 7 COUNTER = 7]
switch to [pid = 1, priority = 7, priority = 7]
[trap.c,85,do_page_fault] [pid = 1, PC = 100e8], valid page fault at '100e8' with scause c

[vm.c,149,create_mapping] root: ffffffe0002d0000, paddr: [802d3000, 802d4000), vaddr: [10000, 11000), perm: 1f

[trap.c,85,do_page_fault] [pid = 1, PC = 101c4], valid page fault at '3ffffffff8' with scause f

[vm.c,149,create_mapping] root: ffffffe0002d0000, paddr: [802d6000, 802d7000), vaddr: [3ffffff000, 4000000000), perm: 17

[trap.c,85,do_page_fault] [pid = 1, PC = 101ec], valid page fault at '12000' with scause d

[vm.c,149,create_mapping] root: ffffffe0002d0000, paddr: [802d9000, 802da000), vaddr: [12000, 13000), perm: 1f

[trap.c,85,do_page_fault] [pid = 1, PC = 1136c], valid page fault at '1136c' with scause c

[vm.c,149,create_mapping] root: ffffffe0002d0000, paddr: [802da000, 802db000), vaddr: [11000, 12000), perm: 1f

[trap.c,85,do_page_fault] [pid = 1, PC = 1048c], valid page fault at '14008' with scause d

[vm.c,149,create_mapping] root: ffffffe0002d0000, paddr: [802db000, 802dc000), vaddr: [14000, 15000), perm: 1f

[U] pid: 1 is running! global_variable: 0
[U] pid: 1 is running! global_variable: 1
[U] pid: 1 is running! global_variable: 2
[trap.c,85,do_page_fault] [pid = 1, PC = 10248], valid page fault at '13008' with scause f

[vm.c,149,create_mapping] root: ffffffe0002d0000, paddr: [802dc000, 802dd000), vaddr: [13000, 14000), perm: 1f

addr of reg/sp: ffffffe0002cfef8
now nrtasks: 3
[vm.c,149,create_mapping] root: ffffffe0002de000, paddr: [802e0000, 802e1000), vaddr: [3ffffff000, 4000000000), perm: 1f

[vm.c,149,create_mapping] root: ffffffe0002de000, paddr: [802e4000, 802e5000), vaddr: [10000, 11000), perm: 1f

[vm.c,149,create_mapping] root: ffffffe0002de000, paddr: [802e7000, 802e8000), vaddr: [11000, 12000), perm: 1f

[vm.c,149,create_mapping] root: ffffffe0002de000, paddr: [802e8000, 802e9000), vaddr: [12000, 13000), perm: 1f

[vm.c,149,create_mapping] root: ffffffe0002de000, paddr: [802e9000, 802ea000), vaddr: [13000, 14000), perm: 1f

[vm.c,149,create_mapping] root: ffffffe0002de000, paddr: [802ea000, 802eb000), vaddr: [14000, 15000), perm: 1f
```

5.5. 画图分析 `make run TEST=FORK3` 的进程 fork 过程，并呈现出各个进程的 `global_variable` 应该从几开始输出，再与你的输出进行对比验证。

预计输出



实际输出

```
now nrtasks: 8
[vm.c,149,create_mapping] root: ffffffe000317000, paddr: [80319000, 8031a000), vaddr: [10000, 11000), perm: 1f

[vm.c,149,create_mapping] root: ffffffe000317000, paddr: [8031c000, 8031d000), vaddr: [11000, 12000), perm: 1f

[vm.c,149,create_mapping] root: ffffffe000317000, paddr: [8031d000, 8031e000), vaddr: [12000, 13000), perm: 1f

[vm.c,149,create_mapping] root: ffffffe000317000, paddr: [8031f000, 80320000), vaddr: [3fffffff000, 4000000000), perm: 1f

[U] pid: 2 is running! global_variable: 2
[U] pid: 2 is running! global_variable: 3
switch to [pid = 7, priority = 7, priority = 7]
[U] pid: 7 is running! global_variable: 2
[U] pid: 7 is running! global_variable: 3
switch to [pid = 6, priority = 7, priority = 7]
[U] pid: 6 is running! global_variable: 1
addr of reg/sp: ffffffe00030aef8
now nrtasks: 9
[vm.c,149,create_mapping] root: ffffffe000323000, paddr: [80325000, 80326000), vaddr: [3fffffff000, 4000000000), perm: 1f

[vm.c,149,create_mapping] root: ffffffe000323000, paddr: [80329000, 8032a000), vaddr: [10000, 11000), perm: 1f

[vm.c,149,create_mapping] root: ffffffe000323000, paddr: [8032c000, 8032d000), vaddr: [11000, 12000), perm: 1f

[vm.c,149,create_mapping] root: ffffffe000323000, paddr: [8032d000, 8032e000), vaddr: [12000, 13000), perm: 1f

[U] pid: 6 is running! global_variable: 2
[U] pid: 6 is running! global_variable: 3
switch to [pid = 8, priority = 7, priority = 7]
[U] pid: 8 is running! global_variable: 2
[U] pid: 8 is running! global_variable: 3
SET [PID = 8 PRIORITY = 7 COUNTER = 7]
SET [PID = 7 PRIORITY = 7 COUNTER = 7]
SET [PID = 6 PRIORITY = 7 COUNTER = 7]
SET [PID = 5 PRIORITY = 7 COUNTER = 7]
SET [PID = 4 PRIORITY = 7 COUNTER = 7]
SET [PID = 3 PRIORITY = 7 COUNTER = 7]
SET [PID = 2 PRIORITY = 7 COUNTER = 7]
SET [PID = 1 PRIORITY = 7 COUNTER = 7]
[U] pid: 8 is running! global_variable: 4
switch to [pid = 7, priority = 7, priority = 7]
[U] pid: 7 is running! global_variable: 4
switch to [pid = 6, priority = 7, priority = 7]
[U] pid: 6 is running! global_variable: 4
switch to [pid = 5, priority = 7, priority = 7]
[U] pid: 5 is running! global_variable: 4
switch to [pid = 4, priority = 7, priority = 7]
[U] pid: 4 is running! global_variable: 4
switch to [pid = 3, priority = 7, priority = 7]
[U] pid: 3 is running! global_variable: 4
switch to [pid = 2, priority = 7, priority = 7]
[U] pid: 2 is running! global_variable: 4
switch to [pid = 1, priority = 7, priority = 7]
[U] pid: 1 is running! global_variable: 4
```