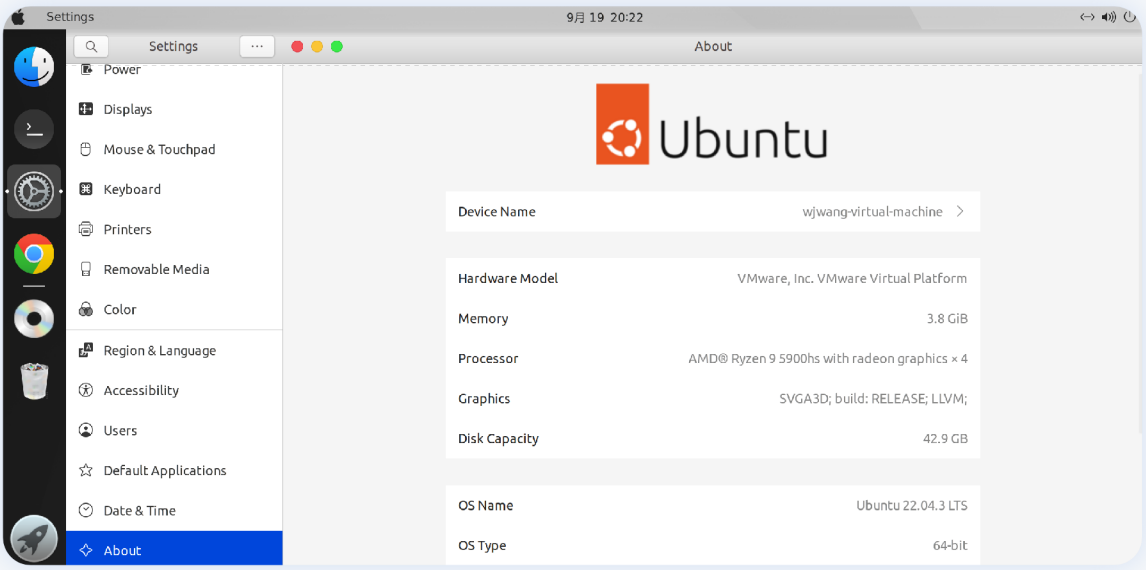
Lab 5: RV64 缺页异常处理

学号：3210106034
姓名：王伟杰

实验环境：

```
uname -a
Linux wjwang-virtual-machine 6.2.0-33-generic #33~22.04.1-Ubuntu SMP
PREEMPT_DYNAMIC Thu Sep  7 10:33:52 UTC 2 x86_64 x86_64 x86_64
GNU/Linux
```



准备工程

在lab4中我已经处理好了 `thread_info` 的问题，在此无需调整

实现 VMA

修改 `proc.h`，增加如下相关结构

```c
#define VM_X_MASK         0x0000000000000008
#define VM_W_MASK         0x0000000000000004
#define VM_R_MASK         0x0000000000000002
#define VM_ANONYM         0x0000000000000001

struct vm_area_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    */
    uint64_t vm_flags;          /* VMA 对应的 flags */

    /* uint64_t file_offset_on_disk */  /* 原本需要记录对应的文件在磁盘上的位置，
                                但是我们只有一个文件 uapp，所以暂时不需要记录 */

    uint64_t vm_content_offset_in_file;                 /* 如果对应了一个文件，
                        那么这块 VMA 起始地址对应的文件内容相对文件起始位置的偏移量，
                                            也就是 ELF 中各段的 p_offset 值 */

    uint64_t vm_content_size_in_file;                 /* 对应的文件内容的长度。
                                                        思考为什么还需要这个域？
                                                        和 (vm_end-vm_start)
                                                        一比，不是冗余了吗？ */
};

struct task_struct {
    uint64_t state;
    uint64_t counter;
    uint64_t priority;
    uint64_t pid;

    struct thread_struct thread;
    pagetable_t pgd;

    uint64_t vma_cnt;                       /* 下面这个数组里的元素的数量 */
    struct vm_area_struct vmas[0];          /* 为什么可以开大小为 0 的数组？
                                这个定义可以和前面的 vma_cnt 换个位置吗？ */
};
```

为了支持 `Demand Paging`，我们支持对 `vm_area_struct` 的添加和查找。

```c
// proc.c
```

```
/* 创建一个新的 vma */
void do_mmap(struct task_struct *task, uint64_t addr, uint64_t length,
uint64_t flags,
            uint64_t vm_content_offset_in_file, uint64_t
vm_content_size_in_file) {
    struct vm_area_struct *vma = &(task->vmas[task->vma_cnt++]); // find last
vma
    vma->vm_start = addr;
    vma->vm_end = addr + length;
    vma->vm_flags = flags;
    vma->vm_content_offset_in_file = vm_content_offset_in_file;
    vma->vm_content_size_in_file = vm_content_size_in_file;
}

/* 查找包含某个 addr 的 vma，该函数主要在 Page Fault 处理时起作用 */
struct vm_area_struct *find_vma(struct task_struct *task, uint64_t addr) {
    for (int i = 0; i < task->vma_cnt; ++i) {
        if (task->vmas[i].vm_start <= addr && addr < task->vmas[i].vm_end) {
            return &(task->vmas[i]);
        }
    }
    return NULL;
}
```

Demand Paging

在初始化一个 task 时我们既**不分配内存**，又**不更改页表项来建立映射**。回退到用户态进行程序执行的时候就会因为没有映射而发生 Page Fault，进入我们的 Page Fault Handler 后，我们再分配空间（按需要拷贝内容）进行映射。

我们需要修改 `task_init` 中的 `load_program` 函数代码，更改为 `Demand Paging`，将两次 `create_mapping` 改为 `do_mmap`，取消分配内存的操作：

```
static uint64_t load_program(struct task_struct* task) {
    Elf64_Ehdr* ehdr = (Elf64_Ehdr*)_sramdisk;

    if (ehdr->e_ident[0] != 0x7f || ehdr->e_ident[1] != 'E' ||
        ehdr->e_ident[2] != 'L' || ehdr->e_ident[3] != 'F') {
        printk("not elf format\n");
        return -1;
    }
```

```
9
10      uint64_t phdr_start = (uint64_t)ehdr + ehdr->e_phoff;
11      int phdr_cnt = ehdr->e_phnum;
12
13      Elf64_Phdr* phdr;
14      int load_phdr_cnt = 0;
15      for (int i = 0; i < phdr_cnt; i++) {
16          phdr = (Elf64_Phdr*)(phdr_start + sizeof(Elf64_Phdr) * i);
17          if (phdr->p_type == PT_LOAD) {
18              // alloc space and copy content
19              // do mapping
20              uint64 flags = (phdr->p_flags & PF_R) >> 1 | (phdr->p_flags &
    PF_W) << 1 | (phdr->p_flags & PF_X) << 3;
21              // char* va = (char*)alloc_pages((size + PGSIZE - 1) / PGSIZE);
22              // // memcpy(va + offset, (char*)ehdr + phdr->p_offset, phdr-
    >p_filesz);
23              // for (int j = 0; j < phdr->p_filesz; ++j) {
24              //     va[offset + j] = ((char*)ehdr)[phdr->p_offset + j];
25              // }
26              // memset(va + offset + phdr->p_filesz, 0, phdr->p_memsz - phdr-
    >p_filesz);
27              do_mmap(task, phdr->p_vaddr, phdr->p_memsz, flags, phdr-
    >p_offset, phdr->p_filesz);
28              // create_mapping(task->pgd, phdr->p_vaddr, (uint64)va -
    PA2VA_OFFSET, size, 1 << 4 | 1 << 0 | (phdr->p_flags & 0x4) >> 1 | (phdr-
    >p_flags & 0x2) << 1 | (phdr->p_flags & 0x1) << 3);
29          }
30      }
31
32      // allocate user stack and do mapping
33
34      // uint64 user_stack = (uint64)alloc_page();
35      do_mmap(task, USER_END - PGSIZE, PGSIZE, 0x7, 0, 0);
36      // create_mapping(task->pgd, USER_END - PGSIZE, user_stack -
    PA2VA_OFFSET, PGSIZE, 0x17);
37      // pc for the user program
38      task->thread.sepc = ehdr->e_entry;
39      return 0;
40  }
```

Page Fault Handler

修改 `trap_handler`，添加跳转 `page_fault` 函数的部分，并在必要的时候进入 `unhandled_trap`：

```
void trap_handler(unsigned long scause, unsigned long sepc, struct pt_regs *regs) {
    unsigned long temp = 1;
    if(scause & (temp << 63)) { // interrupt
        switch (scause & ~(temp << 63))
        {
        case 0x5:
            printk("[INTERRUPT] S mode timer interrupt!\n");
            clock_set_next_event();
            do_timer();
            break;

        default:
            unhandled_trap(regs);
            break;
        }
    } else {
        switch (scause & ~(temp << 63))
        {
        case 0x8: // ECALL_FROM_U_MODE
            uint64 syscall_id = regs->a7;
            switch (syscall_id)
            {
            case SYS_WRITE: // sys_write
                regs->a0 = sys_write(regs->a0, regs->a1, regs->a2);
                break;
            case SYS_GETPID: // sys_getpid
                regs->a0 = sys_getpid();
                break;
            }
            regs->sepc += 4;
            break;

        case 0xc:
        case 0xd:
        case 0xf:
            do_page_fault(regs);
            break;

        default:
```

```
40                 printk("Unhandled exception in %lx!\n", scause);
41                 unhandled_trap(regs);
42                 break;
43             }
44         } // exception
45     }
```

`unhandled_trap` 定义了发生未知错误时的处理方式：暂停等待：

```
1   void unhandled_trap(struct pt_regs *regs) {
2       printk("Unhandled trap:\n");
3       printk("sepc: %lx, scause: %lx, stval: %lx\n", regs->sepc, regs->scause,
    regs->stval);
4       while(1);
5   }
```

进入 `page_fault` 之后处理步骤如下：

- 通过 stval 获得访问出错的虚拟内存地址（Bad Address）
- 通过 `find_vma()` 查找 Bad Address 是否在某个 vma 中
  - 找不到这个vma，或者检查是否是权限错误，如果是这样我们不进行处理，直接进入 `unhandled_trap` .STOP
  - 找到了对应vma，继续下一步
- 分配一个页，将这个页映射到对应的用户地址空间，并将这个页全部清空（之后不需要进行清零处理）
- 通过 (`vma->vm_flags & VM_ANONYM`) 获得当前的 VMA 是否是匿名空间
  - 如果VMA 匿名，由于上一步已经进行清零操作，直接返回 .STOP
  - 如果不是匿名，并且拷贝页面存在（ `end_seg>start_bad_seg_page` ）我们将 `bad_addr` 所在的那一页拷贝过来，拷贝数据的开头有两种情况：
    - `start_seg` 和 `bad_addr` 在同一页，这时拷贝数据的开头不需要是整页的起始地址，而是需要从 `start_seg` 开始拷贝，即算出对应 `offset=start_seg后12位`
    - `start_seg` 和 `bad_addr` 不在同一页，这时直接从 `bad_addr` 对应文件中的那一页起始地址，从 `start_bad_seg_page` 开始拷贝，即置 `offset` 为零

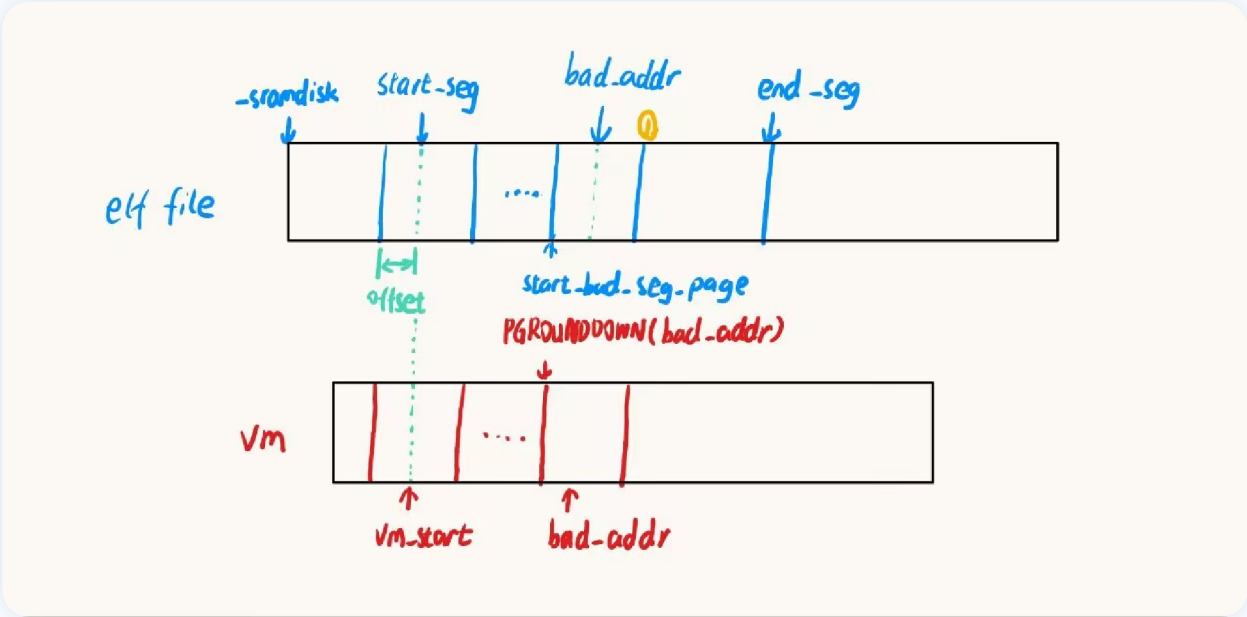
    结尾有以下两种情况（其中下方参考图的①号位置为 `start_bad_seg_page + PGSIZE` ）：
    - `end_seg > start_bad_seg_page + PGSIZE` ，这时我们直接拷贝[`start_bad_seg_page + offset` , `start_bad_seg_page + offset + PGSIZE`]

- `start_bad_seg_page < end_seg < start_bad_seg_page + PGSIZE`，这时我们拷贝[[`start_bad_seg_page + offset`，`end_seg`]

  .STOP

- 如果不是匿名，并且拷贝页面不存在（ `end_seg ≤ start_bad_seg_page` ），则不需要做处理，上一步已经进行清零操作 .STOP

对于找到对应vma、 `start_seg` 和 `bad_addr` 不在同一页、 `start_bad_seg_page < end_seg < start_bad_seg_page + PGSIZE` 的情况，我绘制了拷贝参考图进行辅助说明：




```
void do_page_fault(struct pt_regs *regs) {
    uint64 bad_addr = regs->stval;
    struct vm_area_struct *vma = find_vma(current, bad_addr);
    // 范围检查 权限检查
    if(vma == NULL ||
        (regs->scause == 0xc && !(vma->vm_flags & VM_X_MASK)) ||
        (regs->scause == 0xd && !(vma->vm_flags & VM_R_MASK)) ||
        (regs->scause == 0xf && !(vma->vm_flags & VM_W_MASK))) {
        printk("Bad Address: %lx\n", bad_addr);
        unhandled_trap(regs);
    }
    uint64 page = alloc_page();
    create_mapping(current->pgd, bad_addr, page - PA2VA_OFFSET, PGSIZE, (vma->vm_flags & 0b1110) | 0x11); // UXWRV
    memset((void *)page, 0, PGSIZE);
    if (!(vma->vm_flags & VM_ANONYM)) {
```

```
16          // segment start location in file
17          uint64 start_seg = (uint64)(_sramdisk) + vma->vm_content_offset_in_file;
18          uint64 offset = 0;
19          uint64 end_seg= (uint64)(_sramdisk) + vma→vm_content_offset_in_file+vma→vm_content_size_in_file;
20          uint64 start_bad_seg_page = start_seg + PGROUNDDOWN(bad_addr) - vma->vm_start;
21          uint64 start_bad_seg = start_bad_seg_page;
22          if (PGROUNDDOWN(start_seg) == PGROUNDDOWN(bad_addr)) {
23              start_bad_seg = start_seg;
24              offset = start_seg & 0xfff;
25          }
26          if (end_seg ⩽ start_bad_seg_page + PGSIZE && end_seg > start_bad_seg_page){
27              uint64 valid_seg = end_seg - start_bad_seg_page - offset;
28              memcpy((void *)(page + offset), (void *)(start_bad_seg), valid_seg);
29          } else if (end_seg > start_bad_seg_page + PGSIZE) {
30              memcpy((void *)(page + offset), (void *)(start_bad_seg), PGSIZE - offset);
31          }
32      }
33  }
```

编译及测试

进程第一次被调度，`main` 作为 `uapp` 的情况下，一共会发生3次 Page Fault：

```
...proc_init done!
2022 Hello RISC-V
sstatus = 8000000000006000
sscratch = 0

switch to [PID = 1 COUNTER = 4 PRIORITY = 37]
Page Fault: sepc: 00000000000100e8, scause: 000000000000000c, stval: 00000000000100e8
Page Fault: sepc: 0000000000010124, scause: 000000000000000f, stval: 0000003fffffffff8
Page Fault: sepc: 0000000000010140, scause: 000000000000000d, stval: 0000000000011880
[PID = 1] is running, variable: 0
[PID = 1] is running, variable: 1
[INTERRUPT] S mode timer interrupt!
[PID = 1] is running, variable: 2
[PID = 1] is running, variable: 3
[INTERRUPT] S mode timer interrupt!
[PID = 1] is running, variable: 4
[PID = 1] is running, variable: 5
[INTERRUPT] S mode timer interrupt!
[PID = 1] is running, variable: 6
[PID = 1] is running, variable: 7
[PID = 1] is running, variable: 8
[INTERRUPT] S mode timer interrupt!

switch to [PID = 3 COUNTER = 8 PRIORITY = 52]
Page Fault: sepc: 00000000000100e8, scause: 000000000000000c, stval: 00000000000100e8
Page Fault: sepc: 0000000000010124, scause: 000000000000000f, stval: 0000003fffffffff8
Page Fault: sepc: 0000000000010140, scause: 000000000000000d, stval: 0000000000011880
[PID = 3] is running, variable: 0
[PID = 3] is running, variable: 1
```

第二次被调度，不会发生page fault：

```
[INTERRUPT] S mode timer interrupt!
SET [PID = 1 COUNTER = 1]
SET [PID = 2 COUNTER = 4]
SET [PID = 3 COUNTER = 10]

switch to [PID = 1 COUNTER = 1 PRIORITY = 37]
[PID = 1] is running, variable: 9
[PID = 1] is running, variable: 10
[INTERRUPT] S mode timer interrupt!

switch to [PID = 2 COUNTER = 4 PRIORITY = 88]
[PID = 2] is running, variable: 19
[PID = 2] is running, variable: 20
[INTERRUPT] S mode timer interrupt!
[PID = 2] is running, variable: 21
[PID = 2] is running, variable: 22
[INTERRUPT] S mode timer interrupt!
[PID = 2] is running, variable: 23
[PID = 2] is running, variable: 24
[INTERRUPT] S mode timer interrupt!
[PID = 2] is running, variable: 25
[PID = 2] is running, variable: 26
[INTERRUPT] S mode timer interrupt!

switch to [PID = 3 COUNTER = 10 PRIORITY = 52]
[PID = 3] is running, variable: 17
[PID = 3] is running, variable: 18
[INTERRUPT] S mode timer interrupt!
[PID = 3] is running, variable: 19
```

思考题

1. `uint64_t vm_content_size_in_file;` 对应的文件内容的长度。为什么还需要这个域?

   实验四中，我们知道 `p_filesz` < `p_memsz` ，因为elf文件不会为一些特定段预留空间。`vm_start - vm_end` 相当于 `p_memsz` ，而 `vm_content_size_in_file` 相当于 `p_filesz` 。我们使用 `vm_content_size_in_file` 来确定需要清零的区域，内存中非匿名区域大于 `vm_content_size_in_file` 的部分需要清零。

2. `struct vm_area_struct vmas[0];` 为什么可以开大小为 0 的数组? 这个定义可以和前面的 vma_cnt 换个位置吗?

   C语言中，声明大小为0的数组是非标准的做法，但是C99标准中引入柔性数组成员，允许在结构体的最后一个成员中使用未知大小的数组，最后一个零长数组实际上充当了一个指向结构体末尾连续内存块的指针，可以根据需要动态地分配更多的内存给这个结构体，以存储可变长度的数据。所以这个定义与前面的元素不可更换位置，如果更换,首先不能根据 `task_struct` 的地址直接找到 `vma_cnt` 以及 `vmas` 的位置，也不符合C语言的标准。