浙江大学

本科实验报告

课程名称：	操作系统
姓　名：	徐文皓
学　院：	计算机科学与技术学院
系：	软件工程系
专　业：	软件工程
学　号：	3210102377
指导教师：	夏莹杰

2024 年　01 月　03 日

浙江大学操作系统实验报告

实验名称：RV64 缺页异常处理

电子邮件地址：__手机：__.

实验地点：线上 实验日期：2024 年 01 月 03 日

一、实验目的和要求

本实验的要求是：通过 vm_area_struct 数据结构实现对 task 多区域虚拟内存的管理；在实现用户态程序的基础上，添加缺页异常处理 Page Fault Handler。

二、实验过程

按照实验指导书要求，更新 user 目录下的相关文件，编译运行结果如下。

```
2023[S-Mode] Hello RISC-V

SWITCH TO [pid=1 counter=4 priority=37]
[PID = 1] is running, variable: 0
[PID = 1] is running, variable: 1
[PID = 1] is running, variable: 2
[PID = 1] is running, variable: 3
[PID = 1] is running, variable: 4
[PID = 1] is running, variable: 5
[PID = 1] is running, variable: 6
```

修改 proc.h 内容。

```c
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) {
  uint64_t start_va = addr;
  uint64_t end_va = addr + length;

  uint64_t vma_cnt = task->vma_cnt;
  task->vmas[vma_cnt].vm_start = start_va;
  task->vmas[vma_cnt].vm_end = end_va;
  task->vmas[vma_cnt].vm_flags = flags;
  task->vmas[vma_cnt].vm_content_offset_in_file = vm_content_offset_in_file;
  task->vmas[vma_cnt].vm_content_size_in_file = vm_content_size_in_file;

  printk("VMA [%lx, %lx), flags: %lx\n", task->vmas[vma_cnt].vm_start, task->vmas[vma_cnt].vm_end, task->vmas[vma_cnt].vm_flags);
  task->vma_cnt++;
}

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 && task->vmas[i].vm_end > addr)
            return &(task->vmas[i]);
    }
    return NULL;
}
```

修改 pt_regs 及_traps 内容，使得在进入 trap_handler()时具有更多信息。

```
struct pt_regs {
    uint64 x[32];
    uint64 sepc;
    uint64 sstatus;
    uint64 stval;
    uint64 sscratch;
    uint64 scause;
};
```

```
# arch/riscv/kernel/entry.S

_traps:
    csrr t0, sscratch
    beq t0, zero, _traps_start
    csrw sscratch, sp
    add sp, zero, t0
_traps_start:
    # 1. save 32 registers and sepc to stack
    addi sp, sp, -8*37
    sd x0, 0(sp)
    sd x1, 8(sp)
    ...
    sd x31, 248(sp)
    csrr t0, sepc
    sd t0, 256(sp)
    csrr t0, sstatus
    sd t0, 264(sp)
    csrr t0, stval
    sd t0, 272(sp)
    csrr t0, sscratch
    sd t0, 280(sp)
    csrr t0, scause
    sd t0, 288(sp)

    # 2. call trap_handler
    csrr a0,scause
    csrr a1,sepc
    add a2, zero, sp
    call trap_handler

    # 3. restore sepc and 32 registers (x2(sp) should be restore last) from stack
_csrwrite:
    ld t0, 288(sp)
    csrw scause, t0
    ld t0, 280(sp)
    csrw sscratch, t0
    ld t0, 272(sp)
    csrw stval, t0
    ld t0, 264(sp)
    csrw sstatus, t0
    ld t0, 256(sp)

    csrw sepc, t0

    ld x31, 248(sp)
    ...
    ld x1, 8(sp)
    ld x0, 0(sp)
    ld x2, 16(sp)
    addi sp, sp, 8*37

    # 4. return from trap
    csrr t0, sscratch
    beq t0, zero, _traps_end
    csrw sscratch, sp
    add sp, zero, t0

_traps_end:
    sret
```

实现 do_mmap()和 find_vma()，用于对 vma 的创建和查找。

```
// arch/riscv/kernel/proc.c
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) {
  uint64_t start_va = addr;
  uint64_t end_va = addr + length;

  uint64_t vma_cnt = task->vma_cnt;
  task->vmas[vma_cnt].vm_start = start_va;
  task->vmas[vma_cnt].vm_end = end_va;
  task->vmas[vma_cnt].vm_flags = flags;
  task->vmas[vma_cnt].vm_content_offset_in_file = vm_content_offset_in_file;
  task->vmas[vma_cnt].vm_content_size_in_file = vm_content_size_in_file;

  printk("VMA [%lx, %lx), flags: %lx\n", task->vmas[vma_cnt].vm_start, task->vmas[vma_cnt].vm_end, task->vmas[vma_cnt].vm_flags);
  task->vma_cnt++;
}

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 && task->vmas[i].vm_end > addr)
            return &(task->vmas[i]);
    }
    return NULL;
}
```

修改 task_init()，在创建进程时初始化 vma 计数器为 0。

```c
// arch/riscv/kernel/proc.c
void task_init() {
  ...
  for (int i = 1; i < NR_TASKS; i++) {
    ...
    task[i]->vma_cnt = 0;

    load_program(task[i]);
    ...
```

在 load_program()中删去先前 lab 中关于内存分配和建立映射的内容，取而代之的是对每个段调用 do_mmap()建立用户进程的虚拟地址空间信息。

```c
// arch/riscv/kernel/proc.c
static uint64_t load_program(struct task_struct* task) {
  Elf64_Ehdr* ehdr = (Elf64_Ehdr*)ramdisk_start;

  uint64_t phdr_start = (uint64_t)ehdr + ehdr->e_phoff;
  int phdr_cnt = ehdr->e_phnum;

  Elf64_Phdr* phdr;
  int load_phdr_cnt = 0;
  for (int i = 0; i < phdr_cnt; i++) {
    phdr = (Elf64_Phdr*)(phdr_start + sizeof(Elf64_Phdr) * i);
    if (phdr->p_type == PT_LOAD) {

      uint64_t segment_va = phdr->p_vaddr;
      uint64_t page_offset = segment_va & 0xfff;
      uint64_t segment_size = phdr->p_memsz;

      uint64_t seg_flags = 0;
      if (phdr->p_flags & PF_X) seg_flags |= VM_X_MASK;
      if (phdr->p_flags & PF_W) seg_flags |= VM_W_MASK;
      if (phdr->p_flags & PF_R) seg_flags |= VM_R_MASK;

      do_mmap(task, segment_va, phdr->p_memsz, seg_flags, phdr->p_offset, phdr->p_filesz);

      load_phdr_cnt++;
    }
  }
  do_mmap(task, USER_END - PGSIZE, PGSIZE, VM_R_MASK | VM_W_MASK | VM_ANONYM, 0, PGSIZE);

  task->thread.sepc = ehdr->e_entry;
  task->thread.sstatus = (task->thread.sstatus | (uint64)0x40020) & (~((uint64)0x100));  // set 18(SUM), 5(SPIE), clear 8(SPP)
  task->thread.sscratch = USER_END;
}
```

修改 trap_handler()，增加捕获并处理缺页异常的逻辑，在发生缺页异常(scause[62:0]=0xc 或 0xd 或 0xf)时调用 do_page_fault()。

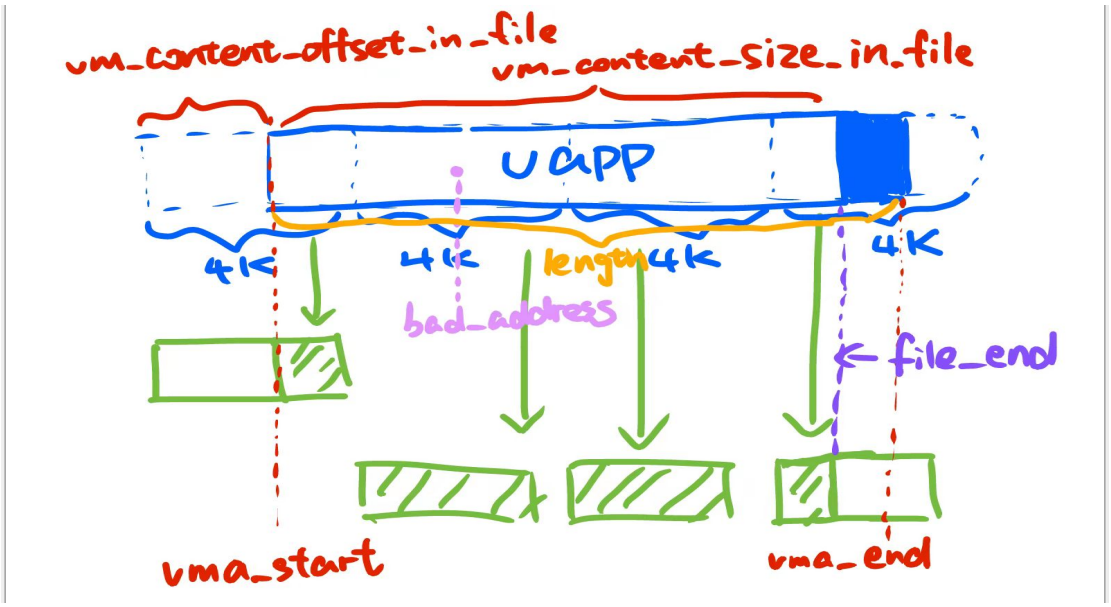
```c
void trap_handler(unsigned long scause, unsigned long sepc, struct pt_regs* regs) {
    unsigned long interrupt = scause & 0x8000000000000000; // 1 - Interrupt | 0 - Exception
    unsigned long cause = scause & 0x7fffffffffffffff;
    if (interrupt) { // interrupt
        if (cause == 5) { // timer interrupt
            // printk("[S] Supervisor Mode Timer Interrupt\n");
            clock_set_next_event();
            do_timer();
        }
        else {
            printk("[Error] Unexpected Interrupt with scause = 0x%16x\n", scause);
            printk("stval: %lx, ", regs->stval);
            printk("sepc: %lx\n", regs->sepc);
            while (1);
        }

    }
    else { // exception
        if (cause == 8) { // syscall
            syscall(regs);
        regs->sepc = regs->sepc + 4;
        }
        else if (cause == 0xc || cause == 0xd || cause == 0xf)  {
            printk("[S] Supervisor Page Fault, cause: 0x%lx, stval: 0x%lx, sepc: 0x%lx\n", cause, regs->stval, sepc);
            do_page_fault(regs);
        }
        else {
            printk("[Error] Unexpected Exception with scause = 0x%16x\n", scause);
            printk("stval: %lx, ", regs->stval);
            printk("sepc: %lx\n", regs->sepc);
            while (1);
        }
    }
    return;
}
```

```c
// arch/riscv/kernel/proc.c
void do_page_fault(struct pt_regs* regs) {
    uint64 bad_address = regs->stval;
    struct vm_area_struct* vma = find_vma(current, bad_address);
    if (vma == NULL){ // vma not found
        printk("[pid = %d] Page fault at [0x%lx] with scause = 0x%lx\n", current->pid, regs->stval, regs->scause);
        while(1);
    }
    else {
      uint64 new_pa = alloc_page();
      memset((char*)new_pa, 0, PGSIZE);
      uint64 perm = PTE_U | PTE_V;
      perm = perm | vma->vm_flags;
      if (!(vma->vm_flags & VM_ANONYM)) {
        uint64 uapp_addr = (uint64)(ramdisk_start) + vma->vm_content_offset_in_file;
        if(PGROUNDDOWN(bad_address) == PGROUNDDOWN(vma->vm_start)){
                uint64 sz1 = PGSIZE - vma->vm_content_offset_in_file;
                uint64 sz2 = vma->vm_content_size_in_file;
                uint64 size = sz1 < sz2 ? sz1 : sz2;
                memcpy((void*)(new_pa + vma->vm_content_offset_in_file), (void*)uapp_addr, size);
            }else{
                uint64 sz1 = PGSIZE;
                uint64 sz2 = vma->vm_start + vma->vm_content_size_in_file - bad_address;
                uint64 size = sz1 < sz2 ? sz1 : sz2;
                memcpy((void*)new_pa, (void*)(uapp_addr + PGROUNDDOWN(bad_address) - vma->vm_start),size);
            }

      }
      create_mapping(current->pgd, PGROUNDDOWN(bad_address), new_pa - PA2VA_OFFSET, PGSIZE, perm);
    }

}
```

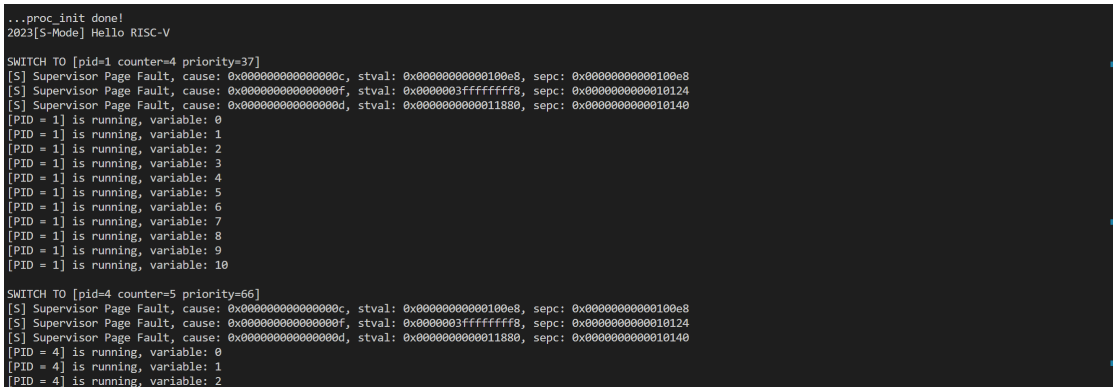
其中，在该 vma 不是匿名时，我们需要将 uapp 内容拷贝到该段。这里判断的逻辑是，如果用户需要操作而产生缺页异常的错误地址是在 uapp 的第一页，即 bad_address 同 vm_start 在同一页，则需要待拷贝的是 uapp 所占用的第一页，则要将本页的[offset, PGSIZE)这块内容拷贝到物理页相同位置；否则，用户需要拷贝的不在第一页，则从本页的 0 位置处开始拷贝。

这里我们使用 sz1 和 sz2 用来判断分配的物理页是否能够装下 uapp 需要拷贝的剩余内容，如果能装下则全部拷贝，否则将本页填满即可，后续部分留待下次缺页异常处理。

下图描述了 vma 的属性的含义。



至此，我们实现了缺页异常处理逻辑，可以观察到运行状态符合预期。



不包含 fork()的 main()每个线程在创建时会产生 3 次缺页异常。这四个线程一共产生了 12 次缺页异常。

```
...proc_init done!
2023[S-Mode] Hello RISC-V

SWITCH TO [pid=1 counter=4 priority=37]
[S] Supervisor Page Fault, cause: 0x000000000000000c, stval: 0x00000000000100e8, sepc: 0x00000000000100e8
[S] Supervisor Page Fault, cause: 0x000000000000000f, stval: 0x00000003ffffffff8, sepc: 0x0000000000010124
[S] Supervisor Page Fault, cause: 0x000000000000000d, stval: 0x0000000000011880, sepc: 0x0000000000010140
[PID = 1] is running, variable: 0
[PID = 1] is running, variable: 1
[PID = 1] is running, variable: 2
[PID = 1] is running, variable: 3
[PID = 1] is running, variable: 4
[PID = 1] is running, variable: 5
[PID = 1] is running, variable: 6
[PID = 1] is running, variable: 7
[PID = 1] is running, variable: 8
[PID = 1] is running, variable: 9
[PID = 1] is running, variable: 10

SWITCH TO [pid=4 counter=5 priority=66]
[S] Supervisor Page Fault, cause: 0x000000000000000c, stval: 0x00000000000100e8, sepc: 0x00000000000100e8
[S] Supervisor Page Fault, cause: 0x000000000000000f, stval: 0x00000003ffffffff8, sepc: 0x0000000000010124
[S] Supervisor Page Fault, cause: 0x000000000000000d, stval: 0x0000000000011880, sepc: 0x0000000000010140
[PID = 4] is running, variable: 0
[PID = 4] is running, variable: 1
[PID = 4] is running, variable: 2
[PID = 4] is running, variable: 3
[PID = 4] is running, variable: 4
[PID = 4] is running, variable: 5
[PID = 4] is running, variable: 6
[PID = 4] is running, variable: 7
[PID = 4] is running, variable: 8
[PID = 4] is running, variable: 9
[PID = 4] is running, variable: 10
[PID = 4] is running, variable: 11
[PID = 4] is running, variable: 12
[PID = 4] is running, variable: 13

SWITCH TO [pid=3 counter=8 priority=52]
[S] Supervisor Page Fault, cause: 0x000000000000000c, stval: 0x00000000000100e8, sepc: 0x00000000000100e8
[S] Supervisor Page Fault, cause: 0x000000000000000f, stval: 0x00000003ffffffff8, sepc: 0x0000000000010124
[S] Supervisor Page Fault, cause: 0x000000000000000d, stval: 0x0000000000011880, sepc: 0x0000000000010140
[PID = 3] is running, variable: 0
[PID = 3] is running, variable: 1
[PID = 3] is running, variable: 2
[PID = 3] is running, variable: 3
[PID = 3] is running, variable: 4
[PID = 3] is running, variable: 5
[PID = 3] is running, variable: 6
[PID = 3] is running, variable: 7
[PID = 3] is running, variable: 8
[PID = 3] is running, variable: 9
[PID = 3] is running, variable: 10
[PID = 3] is running, variable: 11
[PID = 3] is running, variable: 12
[PID = 3] is running, variable: 13
[PID = 3] is running, variable: 14
[PID = 3] is running, variable: 15
[PID = 3] is running, variable: 16
[PID = 3] is running, variable: 17
[PID = 3] is running, variable: 18
[PID = 3] is running, variable: 19
[PID = 3] is running, variable: 20
[PID = 3] is running, variable: 21

SWITCH TO [pid=2 counter=9 priority=88]
[S] Supervisor Page Fault, cause: 0x000000000000000c, stval: 0x00000000000100e8, sepc: 0x00000000000100e8
[S] Supervisor Page Fault, cause: 0x000000000000000f, stval: 0x00000003ffffffff8, sepc: 0x0000000000010124
[S] Supervisor Page Fault, cause: 0x000000000000000d, stval: 0x0000000000011880, sepc: 0x0000000000010140
[PID = 2] is running, variable: 0
[PID = 2] is running, variable: 1
[PID = 2] is running, variable: 2
[PID = 2] is running, variable: 3
[PID = 2] is running, variable: 4
[PID = 2] is running, variable: 5
[PID = 2] is running, variable: 6
[PID = 2] is running, variable: 7
[PID = 2] is running, variable: 8
[PID = 2] is running, variable: 9
[PID = 2] is running, variable: 10
[PID = 2] is running, variable: 11
[PID = 2] is running, variable: 12
[PID = 2] is running, variable: 13
[PID = 2] is running, variable: 14
[PID = 2] is running, variable: 15
[PID = 2] is running, variable: 16
[PID = 2] is running, variable: 17
[PID = 2] is running, variable: 18
[PID = 2] is running, variable: 19
[PID = 2] is running, variable: 20
[PID = 2] is running, variable: 21
[PID = 2] is running, variable: 22
[PID = 2] is running, variable: 23
[PID = 2] is running, variable: 24

SWITCH TO [pid=1 counter=1 priority=37]
[PID = 1] is running, variable: 11
[PID = 1] is running, variable: 12
[PID = 1] is running, variable: 13

SWITCH TO [pid=2 counter=4 priority=88]
[PID = 2] is running, variable: 25
[PID = 2] is running, variable: 26
[PID = 2] is running, variable: 27
[PID = 2] is running, variable: 28
[PID = 2] is running, variable: 29
[PID = 2] is running, variable: 30
[PID = 2] is running, variable: 31
[PID = 2] is running, variable: 32
[PID = 2] is running, variable: 33
[PID = 2] is running, variable: 34

SWITCH TO [pid=4 counter=4 priority=66]
[PID = 4] is running, variable: 14
[PID = 4] is running, variable: 15
[PID = 4] is running, variable: 16
[PID = 4] is running, variable: 17
[PID = 4] is running, variable: 18
[PID = 4] is running, variable: 19
[PID = 4] is running, variable: 20
[PID = 4] is running, variable: 21
[PID = 4] is running, variable: 22
[PID = 4] is running, variable: 23
[PID = 4] is running, variable: 24

SWITCH TO [pid=3 counter=10 priority=52]
[PID = 3] is running, variable: 22
[PID = 3] is running, variable: 23
```

三、讨论和心得

经过一次又一次的实验，感觉和操作系统越来越亲近了。结合和同学们的讨论，自认为本次实验中的理解是最为深刻和透彻的。在上文出现的由我绘制的 vma 示意图就体现了这一点。相比于之前的实验，这次理解之后做起来就会更加顺利些。

本次实验基本没有遇到什么问题，唯一一个卡得比较久的点就是我之前在异常处理后默认将 sepc 移动到下一地址，导致缺页异常处理一直不符合预期，在排查之后修改为只在系统调用后移动 sepc，就解决了这一问题。

四、思考题

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

和上一个 lab 中提到的 phdr->p_memsz 与 phdr->p_filesz 的关系相同，length(即 vma_end-vma_start 的值)对应 phdr->p_memsz，是用户所需要的内存的长度，而 vm_content_size_in_file 是用户文件的长度，后者一般会小于前者，差出的空间可能用于存放.bss 段等，在进程中使用，并不存储在磁盘上。

因此，在 vma 中，我们需要标记多出来的这段内存以供用户使用，但同样不需要将其进行内存拷贝。

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?

大小为 0 的数组只是表征一个地址，并没有实际存放内容，这部分内存可以在内核运行时添加内容。因为我们为 task_struct 分配了一页的内存，这页内存并没有被其他成员使用完，因此可以直接在 vmas 后动态的添加内容而不需要经过 malloc()等过程。

我们将 vmas 放在了 task_struct 的所有成员中最后的位置，它才能够起到标记地址而实际不占用内存空间的作用，而如果将 vma_cnt 和其调换位置，在稍后向 vmas 添加元素后将覆盖 vma_cnt 的内存，造成 vma_cnt 的值可能与预期不

同(不难得出，它将和 vmas[0].vm_start 共享内存)，从而内核无法按预期运行。

这里，我本来觉得会因为 vmas 没有指定长度而导致 vma_cnt 内存不明从而报编译错误之类的，但经过实验发现程序是可以正常编译运行的，大概就是将 vmas[0]视作一个 0 长度的地址标记了吧。