浙江大学

本科实验报告

课程名称：	操作系统
姓　　名：	夏尤楷
学　　院：	计算机科学与技术学院
系：	计算机科学与技术系
专　　业：	计算机科学与技术
学　　号：	3210104331
指导教师：	夏莹杰

2024 年 1 月 6 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：459510812@qq.com 手机：15058004449

实验地点： 玉泉曹光彪西楼 503 实验日期： 2024 年 1 月 1 日

一、实验目的和要求

1. 通过 vm_area_struct 数据结构实现对 task 多区域虚拟内存的管理。
2. 在 Lab4 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

（一） 整理已有程序

调整 pt_regs（与相应的在_traps 保存、恢复寄存器的逻辑）和 trap_handler，来更好地捕获异常并辅助调试。调整后大致如下：

```
typedef struct pt_regs {
    uint64 x[32];  //x0---x31
    uint64 sepc;
    uint64 sstatus;
    uint64 stval;//trap value
    uint64 sscratch;
    uint64 scause;
} pt_regs;
void trap_handler(uint64 scause, uint64 sepc, pt_regs* regs) {
    if (scause >> 63){ // 通过 `scause` 判断trap类型
        if (scause % 8 == 5) { //如果是interrupt 判断是否是
                                timer interrupt
            ...
        }
    } else if (scause == 8) {
        uint64_t ret;
        uint64_t syscall_id = regs->x[17];

```

```
        if (syscall_id == SYS_GETPID) {
            ...
        } else if (syscall_id == SYS_WRITE) {
            ...
        } else {
            printk("[S] Unhandled syscall: %lx\n", syscall_id);
            while (1);
        }
    } else if (scause == ...) {
        ...
    } else {
        printk("[S] Unhandled trap, scause: %lx, sstatus: %lx,
            sepc: %lx\n", scause, regs->sstatus, regs->sepc);
        while (1);
    }
}
```

这样发生了没有处理的异常、中断或者是系统调用的时候，内核会直接进入死循环。

然后，将 vmlinux.lds 和程序中的 uapp_start，uapp_end 分别换成 ramdisk_start 和 ramdisk_end，来提醒自己这一段内容是对硬盘的模拟，而不是可以直接使用的内存。需要拷贝进入 alloc_pages 分配出来的“真的”内存，才能够直接被使用。

（二） 准备工作

从 repo 同步 user 文件夹，并按照以下步骤将这些文件正确放置：

```
.
└── user
    ├── Makefile
    ├── getpid.c
    ├── link.lds
    ├── printf.c
    ├── start.S
    ├── stddef.h
    ├── stdio.h
    ├── syscall.h
    └── uapp.S
```

（三） 实现 VMA

修改 proc.h，增加如下相关结构：

```
#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 */
```

```
10.
11.     /* uint64_t file_offset_on_disk */
        /* 原本需要记录对应的文件在磁盘上的位置，
12.        但是我们只有一个文件 uapp，所以暂时不需要记录 */
13.
14.     uint64_t vm_content_offset_in_file;         /* 如果对应了一个文件，
15.                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，
16.                                        也就是 ELF 中各段的 p_offset 值 */
17.
18.     uint64_t vm_content_size_in_file;         /* 对应的文件内容的长度。
19.                                                思考为什么还需要这个域?
20.                                                  和 (vm_end-vm_start)
21.                                                  一比，不是冗余了吗? */
22. };
23.
24. struct task_struct {
25.     uint64_t state;
26.     uint64_t counter;
27.     uint64_t priority;
28.     uint64_t pid;
29.
30.     struct thread_struct thread;
31.     pagetable_t pgd;
32.
33.     uint64_t vma_cnt;                  /* 下面这个数组里的元素的数量 */
34.     struct vm_area_struct vmas[0];      /* 为什么可以开大小为 0 的数组?
35.                                 这个定义可以和前面的 vma_cnt 换个位置吗? */
36. };
```

每一个 vm_area_struct 都对应于 task 地址空间的唯一连续区间。这里的 vm_flag 和 p_flags 并没有按 bit 进行对应。、

为了支持 Demand Paging，我们需要支持对 vm_area_struct 的添加和查找。在 proc.h 中加入如下声明：

```
1. 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);
2.
3. struct vm_area_struct *find_vma(struct task_struct *task, uint64_t addr);
```

do_mmap 创建一个新的 vma。

find_vma 查找包含某个 addr 的 vma，该函数主要在 Page Fault 处理时起作用。

在 proc.c 中作相应实现如下：

```
1.  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){
2.      struct vm_area_struct temp;
3.      temp.vm_start = addr;
```

```
    temp.vm_end = addr + length;
    temp.vm_flags = flags;
    temp.vm_content_offset_in_file = vm_content_offset_in_file;
    temp.vm_content_size_in_file = vm_content_size_in_file; //在file中的大小

    task->vmas[task->vma_cnt++] = temp;
}

struct vm_area_struct *find_vma(struct task_struct *task, uint64_t addr){
    struct vm_area_struct *tmp;
    for(int i = 0;i < task->vma_cnt;i++){
        tmp = & task->vmas[i];
        if( addr >= tmp->vm_start && addr <= tmp->vm_end) //满足地址范围条件
            return tmp;
    }
}
```

（四） Page Fault Handler

当系统运行发生异常时，可即时地通过解析 scause 寄存器的值，识别如下三种不同的 Page Fault：

Interrupt	Exception Code	Description
0	12	Instruction Page Fault
0	13	Load Page Fault
0	15	Store/AMO Page Fault

处理缺页异常时所需的信息如下：

1. 触发 Page Fault 时访问的虚拟内存地址 VA。当触发 page fault 时，stval 寄存器被被硬件自动设置为该出错的 VA 地址。
2. 导致 Page Fault 的类型：
   （1） Exception Code = 12: page fault caused by an instruction fetch。
   （2） Exception Code = 13: page fault caused by a read。
   （3） Exception Code = 15: page fault caused by a write。
3. 发生 Page Fault 时的指令执行位置，保存在 sepc 中。
4. 当前 task 合法的 VMA 映射关系，保存在 vm_area_struct 链表中。

当缺页异常发生时，检查 VMA。

如果当前访问的虚拟地址在 VMA 中没有记录，即是不合法的地址，则运行出错（本实验不涉及）。

如果当前访问的虚拟地址在 VMA 中存在记录，则进行相应的映射即可。如果访问的页是存在数据的，如访问的是代码，则需要从文件系统中读取内容，随后进行映射；否则是匿名映射，即找一个可用的帧映射上去即可。

在本实验中初始化一个 task 时，我们既不分配内存，又不更改页表项来建立映射。这样的话，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。因此，按照如下思路，修改 task_init 函数代码，更改为 Demand Paging：

1. 取消之前实验中对 U-MODE 代码以及栈进行的映射；
2. 调用 do_mmap 函数，建立用户 task 的虚拟地址空间信息，在本次实验中仅包括两个区域：
   （1）代码和数据区域：该区域从 ELF 给出的 Segment 起始地址 phdr->p_offset 开始，权限参考 phdr->p_flags 进行设置。
   （2）用户栈：范围为[USER_END - PGSIZE, USER_END)，权限为 VM_READ | VM_WRITE，并且是匿名的区域。

修改完成后代码内容如下：

```
1.  static uint64_t load_program(struct task_struct* task) {
2.      Elf64_Ehdr* ehdr = (Elf64_Ehdr*)uapp_start;
3.
4.      uint64_t phdr_start = (uint64_t)ehdr + ehdr->e_phoff;
5.      int phdr_cnt = ehdr->e_phnum;
6.
7.      Elf64_Phdr* phdr;
8.      for (int i = 0; i < phdr_cnt; i++) {
9.          phdr = (Elf64_Phdr*)(phdr_start + sizeof(Elf64_Phdr) * i);
10.         if (phdr->p_type == PT_LOAD) {
11.             uint64_t perm = 0;
12.             perm |= (phdr->p_flags & PF_X) ? VM_X_MASK : 0;
13.             perm |= (phdr->p_flags & PF_W) ? VM_W_MASK : 0;
14.             perm |= (phdr->p_flags & PF_R) ? VM_R_MASK : 0;
15.
16.             do_mmap(task, phdr->p_vaddr, phdr->p_memsz, perm,
                        phdr->p_offset, phdr->p_filesz);
17.         }
18.     }
19.
20.     // Set up the rest of the task structure.
21.     task->thread.sepc = ehdr->e_entry;
22.     task->thread.sstatus = csr_read(sstatus);
```

```
    task->thread.sstatus &= ~(1 << 8);  // Clear SIE
    task->thread.sstatus |= (1 << 18);  // Set SUM
    task->thread.sstatus |= (1 << 5);   // Set SPIE
    task->thread.sscratch = USER_END;

    return ehdr->e_entry;
}

void task_init() {
    // 1. 调用 kalloc() 为 idle 分配一个物理页
    // 2. 设置 state 为 TASK_RUNNING;
    // 3. 由于 idle 不参与调度 可以将其 counter / priority 设置为 0
    // 4. 设置 idle 的 pid 为 0
    // 5. 将 current 和 task[0] 指向 idle
    idle = (struct task_struct*)kalloc();
    idle->state = TASK_RUNNING;
    idle->counter = 0;
    idle->priority = 0;
    idle->pid = 0;
    current = idle;
    task[0] = idle;

    for (int i = 1; i < NR_TASKS; ++i) { // 初始化其他进程
        task[i] = (struct task_struct*)kalloc();
        task[i]->pid = i;
        task[i]->state = TASK_RUNNING;
        task[i]->counter = 0;
        task[i]->priority = rand();
        task[i]->thread.ra = (uint64)&__dummy;
        task[i]->thread.sp = (uint64)task[i] + PGSIZE;
        //创建进程自己的页表并拷贝
        task[i]->pgd = (pagetable_t)alloc_page();
        for (int j = 0; j < 512; ++j)
            task[i]->pgd[j] = swapper_pg_dir[j];
        task[i]->vma_cnt = 0;
        do_mmap(task[i], USER_END - PGSIZE, PGSIZE,
                VM_R_MASK | VM_W_MASK | VM_ANONYM, 0, 0);

        task[i]->thread.sepc = load_program(task[i]);
    }
    printk("...proc_init done!\n");
}
```

在完成 Demand Paging 后，实现 Page Fault 的检测与处理。

首先修改 trap.c，添加捕获 Page Fault 的逻辑，修改后 trap_handler 代码如下：

```
void trap_handler(uint64 scause, uint64 sepc, pt_regs* regs) {
    if (scause >> 63){ // 通过 `scause` 判断 trap 类型
        if (scause % 8 == 5) { // 如果是 interrupt 判断是否是
                               timer interrupt
            // 如果是 timer interrupt 则打印输出相关信息，并通过
                    `clock_set_next_event()` 设置下一次时钟中断
```

```
            //printk("[S] Supervisor mode time interrupt!\n");
            clock_set_next_event();
            do_timer();
        }
    } else if (scause == 8) {
        uint64_t ret;
        uint64_t syscall_id = regs->x[17];

        if (syscall_id == SYS_GETPID) {
            ret = (uint64_t)sys_getpid();
        } else if (syscall_id == SYS_WRITE) {
            uint64_t fd = (uint64_t)regs->x[10];
            char* buffer = (char*)regs->x[11];
            size_t siz = (size_t)regs->x[12];

            ret = sys_write(fd, buffer, siz);
        } else {
            printk("[S] Unhandled syscall: %lx\n", syscall_id);
            while (1);
        }
        regs->x[10] = ret;
        regs->sepc += 4;
    } else if (scause == 12) {
        // inst page fault
        printk("Instruction page fault.\n");
        printk("sepc: %lx, scause: %lx, stval: %lx.\n",
            csr_read(sepc), csr_read(scause), csr_read(stval));
        do_page_fault(regs);
    } else if (scause == 13) {
        // ld page fault
        printk("LD page fault.\n");
        printk("sepc: %lx, scause: %lx, stval: %lx.\n",
            csr_read(sepc), csr_read(scause), csr_read(stval));
        do_page_fault(regs);
    } else if (scause == 15) {
        // sd/amo page fault
        printk("SD/SMO page fault.\n");
        printk("sepc: %lx, scause: %lx, stval: %lx.\n",
            csr_read(sepc), csr_read(scause), csr_read(stval));
        do_page_fault(regs);
    } else {
        printk("[S] Unhandled trap, scause: %lx, sstatus: %lx,
            sepc: %lx\n", scause, regs->sstatus, regs->sepc);
        while (1);
    }
}
```

其中，do_page_fault 函数是用来处理捕获到的缺页异常的。在本实验中，该函数用来在捕获到了一条指令页错误异常的时候，新分配一个页，并拷贝 uapp 这个 ELF 文件中的对应内容到新分配的页内，然后将这个页映射到用户空间中。

实现函数的代码如下：

```
1. void do_page_fault(pt_regs* regs) {
2.     /*
3.      1. 通过 stval 获得访问出错的虚拟内存地址（Bad Address）
4.      2. 通过 find_vma() 查找 Bad Address 是否在某个 vma 中
5.      3. 分配一个页，将这个页映射到对应的用户地址空间
6.      4. 通过 (vma->vm_flags | VM_ANONYM) 获得当前的 VMA 是否是匿名空间
7.      5. 根据 VMA 匿名与否决定将新的页清零或是拷贝 uapp 中的内容
8.     */
9.     uint64 stval = csr_read(stval);
10.    struct vm_area_struct* pgf_vm_area = find_vma(current, stval);
11.    if (pgf_vm_area == NULL) {
12.        printk("illegal address, run time error.\n");
13.        return;
14.    }
15.
16.    uint64 va = PGROUNDDOWN(stval);
17.    uint64 sz = PGROUNDUP(stval + PGSIZE) - va;
18.    uint64 pa = alloc_pages(sz / PGSIZE);
19.    uint64 perm = !!(pgf_vm_area->vm_flags & VM_R_MASK) * PTE_R |
20.                  !!(pgf_vm_area->vm_flags & VM_W_MASK) * PTE_W |
21.                  !!(pgf_vm_area->vm_flags & VM_X_MASK) * PTE_X |
22.                  PTE_U | PTE_V;
23.    memset((void*)pa, 0, sz);
24.
25.    if (pgf_vm_area->vm_flags & VM_ANONYM) {
26.        // For anonymous mapping, the allocated space is already zeroed
27.    } else {
28.        // For file mapping
29.        uint64_t src_uapp = (uint64_t)uapp_start +
               pgf_vm_area->vm_content_offset_in_file;
30.        uint64_t offset = stval - pgf_vm_area->vm_start;
31.        uint64_t src_uapp1 = PGROUNDDOWN(src_uapp + offset);
32.
33.        for (int j = 0; j < sz; ++j) {
34.            ((char*)(pa))[j] = ((char*)src_uapp1)[j];
               // Copy contents from the file
35.        }
36.    }
37.
38.    create_mapping(current->pgd, va, pa - PA2VA_OFFSET, sz, perm);
39.}
```

编译并运行的结果如下页图所示。该图给出了使用第 1 个 main 函数时，从程序开始执行到每个进程被调度两次时为止的输出。可以看到，在使用 3 个用户态进程和第 1 个 main 函数进行调度的时候，一共会发生 6 次 Page Fault（其中，每个用户态进程在第一次被调度时，分别产生 1 次 Instruction Page Fault 和 1 次 Store/AMO Page Fault）。

```
OpenSBI v0.9
```

```
Platform Name             : riscv-virtio,qemu
Platform Features         : timer,mfdeleg
Platform HART Count       : 1
Firmware Base             : 0x80000000
Firmware Size             : 100 KB
Runtime SBI Version       : 0.2

Domain0 Name              : root
Domain0 Boot HART         : 0
Domain0 HARTs             : 0*
Domain0 Region00          : 0x0000000080000000-0x000000008001ffff ()
Domain0 Region01          : 0x0000000000000000-0xffffffffffffffff (R,W,X)
Domain0 Next Address      : 0x0000000080200000
Domain0 Next Arg1         : 0x0000000087000000
Domain0 Next Mode         : S-mode
Domain0 SysReset          : yes

Boot HART ID              : 0
Boot HART Domain          : root
Boot HART ISA             : rv64imafdcsu
Boot HART Features        : scounteren,mcounteren,time
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 0
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109
...buddy_init done!
...proc_init done!
[S-MODE] 2022 Hello RISC-V
SET [PID = 1 COUNTER = 4]
SET [PID = 2 COUNTER = 10]
SET [PID = 3 COUNTER = 10]

switch to [PID = 1 COUNTER = 4]
Instruction page fault.
sepc: 00000000000100e8, scause: 000000000000000c, stval: 00000000000100e8.
SD/SMO page fault.
sepc: 0000000000010124, scause: 000000000000000f, stval: 0000003ffffffff8.
[PID = 1] is running, variable: 0
[PID = 1] is running, variable: 1

switch to [PID = 2 COUNTER = 10]
Instruction page fault.
sepc: 00000000000100e8, scause: 000000000000000c, stval: 00000000000100e8.
SD/SMO page fault.
sepc: 0000000000010124, scause: 000000000000000f, stval: 0000003ffffffff8.
[PID = 2] is running, variable: 0
[PID = 2] is running, variable: 1
[PID = 2] is running, variable: 2
[PID = 2] is running, variable: 3

switch to [PID = 3 COUNTER = 10]
Instruction page fault.
sepc: 00000000000100e8, scause: 000000000000000c, stval: 00000000000100e8.
SD/SMO page fault.
sepc: 0000000000010124, scause: 000000000000000f, stval: 0000003ffffffff8.
[PID = 3] is running, variable: 0
[PID = 3] is running, variable: 1
[PID = 3] is running, variable: 2
[PID = 3] is running, variable: 3
SET [PID = 1 COUNTER = 5]
SET [PID = 2 COUNTER = 2]
SET [PID = 3 COUNTER = 9]

switch to [PID = 2 COUNTER = 2]
[PID = 2] is running, variable: 4

switch to [PID = 1 COUNTER = 5]
[PID = 1] is running, variable: 2
[PID = 1] is running, variable: 3

switch to [PID = 3 COUNTER = 9]
[PID = 3] is running, variable: 4
[PID = 3] is running, variable: 5
[PID = 3] is running, variable: 6
SET [PID = 1 COUNTER = 4]
SET [PID = 2 COUNTER = 4]
SET [PID = 3 COUNTER = 10]
```

三、讨论和心得

这次实验让我对多区域虚拟内存的实现有了一些了解，对缺页异常的产生原理和处理方式有了更深的理解。

四、思考题

1. 因为在文件内可能还有一些空洞（比如.bss 节，该节包含的是未初始化或是要初始化为 0 的数据，这些数据没必要存在磁盘中，因此磁盘中的文件不存储这些数据，以节省空间；而把文件加载进内存时，这些数据要在内存中有相应的空间），这些空洞所包含的数据不被存储在磁盘内，但要在内存中被分配到相应大小的空间。因此，需要 uint64_t vm_content_size_in_file 这个域标明文件所占内存空间的大小，这个大小不一定是（vm_end – vm_start）。
2. 在 C99 及之后的标准中加入了“柔性数组”，在定义结构体时，结构体内已声明有其他成员的情况下，可在结构体内最后面在形式上声明一个长度为 0 的数组，从而声明了一个可变长度的数组。由于数组长度未知，这个数组的声明只能放在结构体定义的最后，不可以和前面的 vma_cnt 换个位置（不然数组后面的空间被分配给结构体内其他成员变量，数组就没有拓展的空间了）。

五、附录

无。