Lab 5: RV64 缺页异常处理

姓名：高铭健

学号：3210102322

日期：2023.12.20

1. 实验目的

- 通过 **vm_area_struct** 数据结构实现对 task **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**。

2. 实验环境

- 计算机（Intel Core i5以上，4GB内存以上）系统
- Ubuntu 22.04.2 LTS

3. 实验原理

3.1 vm_area_struct 介绍

在 linux 系统中，`vm_area_struct` 是虚拟内存管理的基本单元，`vm_area_struct` 保存了有关连续虚拟内存区域（简称 vma）的信息：

- `vm_start` ：该段虚拟内存区域的开始地址
- `vm_end` ：该段虚拟内存区域的结束地址
- `vm_flags` ：该 `vm_area` 的一组权限(rwx)标志
- `vm_pgoff` ：虚拟内存映射区域在文件内的偏移量
- `vm_file` ：映射文件所属设备号/以及指向关联文件结构的指针/以及文件名

3.2 缺页异常 Page Fault

在一个启用了虚拟内存的系统上，当正在运行的程序访问一个内存地址时，如果该地址当前未被内存管理单元（MMU）映射，则由计算机**硬件**引发**缺页异常**。访问未被映射的页或访问权限不足，都会导致该类异常的发生。处理缺页异常通常是操作系统内核的一部分，当处理缺页异常时，操作系统将尝试使所需页面在物理内存中的位置变得可访问（建立新的映射关系到虚拟内存）。而如果在非法访问内存的情况下，即发现触发 `Page Fault` 的虚拟内存地址（Bad Address）不在当前 task `vm_area_struct` 链表中所定义的允许访问的虚拟内存地址范围内，或访问位置的权限条件不满足时，缺页异常处理将终止该程序的继续运行。

3.2.1 Page Fault Handler

处理缺页异常需要进行以下步骤：

1. 捕获异常
2. 寻找当前 task 中**导致产生了异常的地址**对应的 VMA
3. 判断产生异常的原因

4. 如果是匿名区域，那么开辟**一页**内存，然后把这一页映射到产生异常的 task 的页表中。如果不是，那么首先将硬盘中的内容读入 buffer pool，将 buffer pool 中这段内存映射给 task。
5. 返回到产生了该缺页异常的那条指令，并继续执行程序

当 Linux 发生缺页异常并找到了当前 task 中对应的 `vm_area_struct` 后，可以根据以下信息来判断发生异常的原因

1. CSRs
2. `vm_area_struct` 中记录的信息
3. 发生异常的虚拟地址对应的 PTE (page table entry) 中记录的信息并对当前的异常进行处理。

3.2.2 常规处理 Page Fault 的方式介绍

处理缺页异常时所需的信息如下：

- 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
- 导致 **Page Fault** 的类型：
  - Exception Code = 12: page fault caused by an instruction fetch
  - Exception Code = 13: page fault caused by a read
  - Exception Code = 15: page fault caused by a write
- 发生 **Page Fault** 时的指令执行位置，保存在 `sepc` 中
- 当前 task 合法的 **VMA** 映射关系，保存在 `vm_area_struct` 链表中

处理缺页异常的方式：

- 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

4. 实验步骤

- 修改 `pt_regs` 和 `trap_handler`，来捕获异常
  - 在 `pt_regs` 中增加 `stval` 用于记录发生错误的地址；增加 `scause` 来记录错误的原因
  - 在 `trap_handler` 对没有做处理的错误输出其原因、错误地址、指令地址，并进入循环

```
struct pt_regs
{
    uint64 reg[32];
    uint64 sepc;
    uint64 sstatus;
```

```
    uint64 sscratch;
    uint64 stval;
    uint64 scause;
};
void trap_handler(unsigned long scause, unsigned long sepc, struct pt_regs
*regs) {
    // 通过 `scause` 判断trap类型
    // 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
    if(scause & 0x8000000000000000){//trap 类型为interrupt
        if(scause == 0x8000000000000005){//timer interrupt
//            printk("[S] Supervisor Mode Timer Interrupt\n");
            clock_set_next_event();
            do_timer();
        }
    }
    else if(scause == 8){
        if(regs->reg[17] == SYS_WRITE){
            char *buffer = (char *)(regs->reg[11]);
            int cnt = sys_write(1, buffer, regs->reg[12]);
            regs->sepc += 4;
        }
        else if(regs->reg[17] == SYS_PID){
            uint64 pid = sys_getpid();
            regs->sepc += 4;
            regs->reg[10] = pid;
        }
    }
    else {
        printk("[S] Unhandled trap, ");
        printk("scause: %lx, ", scause);
        printk("stval: %lx, ", regs->stval);
        printk("sepc: %lx\n", regs->sepc);
        while (1);
    }
}
```

- 在 `_traps` 中调用 `trap_handler` 前后保存 `stval` 和 `scause`

```
        ......
        csrr t0, stval
        sd t0, 280(sp)
        csrr t0, scause
        sd t0, 288(sp)
        ......
        ld t0, 288(sp)
        csrw scause, t0
        ld t0, 280(sp)
        csrw stval, t0
```

4.1 准备工作

- 同步以下文件夹`user`并将其放到以下目录：

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

4.2 实现 VMA

- 修改`proc.h`，增加如下结构：

```
struct vm_area_struct {
    uint64 vm_start;          // VMA 对应的用户态虚拟地址的开始    */
    uint64 vm_end;            // VMA 对应的用户态虚拟地址的结束    */
    uint64 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 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 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 thread_info {
    uint64 kernel_sp;
    uint64 user_sp;
};

typedef unsigned long* pagetable_t;
struct thread_struct {
    uint64 ra;
    uint64 sp;
    uint64 s[12];

```

```
    uint64 sepc, sstatus, sscratch;
};

struct task_struct {
    struct thread_info *thread_info;
    uint64 state;    // 线程状态
    uint64 counter;  // 运行剩余时间
    uint64 priority; // 运行优先级 1最低 10最高
    uint64 pid;      // 线程id
    struct thread_struct thread;
    uint64 satp;
    uint64 kernel_sp;
    uint64 user_sp;
    pagetable_t pgd;
    uint64 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?
};
```

- `find_vma` 查找包含某个 `addr` 的 `vma`：

  获得该进程的 VMA 数组的首地址，循环遍历进程的所有 VMA，如果所查找的地址 addr 大于等于 vm_start，小于结束地址 vm_end，即在当前 VMA 的范围内，那么返回当前 VMA 的地址；如果没有找到匹配的 VMA，返回值NULL

```
struct vm_area_struct *find_vma(struct task_struct *task, uint64 addr){
    struct vm_area_struct *vmas = &(task->vmas[0]);
    int judge = 0;
    for(int i = 0 ; i < task->vma_cnt ; i++){
        if(addr >= vmas[i].vm_start){
            if(addr < vmas[i].vm_end){
                judge = 1;
                return &(vmas[i]);
            }
        }
    }
    if(judge == 0){
        return NULL;
    }
}
```

- `do_mmap` 创建一个新的 vma：

  计算出新 VMA 起始地址和结束地址的页对齐地址，将起始地址 `vm_start`、结束地址 `vm_end`、标志位 `flags`、文件偏移 `vm_content_offset_in_file` 和文件大小 `vm_content_size_in_file` 赋给新 VMA 。

  把 `vma_new` 加到进程的 `task->vmas` 数组，增加 `task->vma_cnt` 计数器。

```
```

```
void do_mmap(struct task_struct *task, uint64 addr, uint64 length, uint64
flags,
             uint64 vm_content_offset_in_file, uint64 vm_content_size_in_file){
    uint64 vm_start = addr/PGSIZE *PGSIZE;
    //printk("vm_start= %|x\n",vm_start);
    uint64 vm_end;
    if((addr + length) % PGSIZE != 0){
        vm_end = ((addr + length) / PGSIZE + 1) * PGSIZE;
    }
    else{
        vm_end  = ((addr + length) / PGSIZE) * PGSIZE;
    }
    int judge = 0;
    for(uint64 i = 0 ; i < vm_end - vm_start ; i += PGSIZE){
        if(find_vma(task, vm_start + i)){
            judge = 1;
            break;
        }
    }
        struct vm_area_struct *vma_new = (struct vm_area_struct *)alloc_page();
        vma_new->vm_start = vm_start;
        vma_new->vm_end = vm_end;
        vma_new->vm_flags = flags;
        vma_new->vm_content_offset_in_file = vm_content_offset_in_file;
        vma_new->vm_content_size_in_file = vm_content_size_in_file;
        task->vmas[task->vma_cnt] = *vma_new;
        task->vma_cnt++;
        // printk("task->vmas->start_va= %|x task->vmas->vm_end= %|x task->vmas-
>vma_cnt= %d\n", task->vmas[task->vma_cnt-1].vm_start,
        // task->vmas[task->vma_cnt-1].vm_end, task->vma_cnt-1);
        return;
}
```

4.3 Page Fault Handler

- 修改 `task_init`，调用 `do_mmap` 函数，建立用户 task 的虚拟地址空间信息

  将原来的`create_mapping`改为`do_mmap`，分别添加用户栈和uapp，其权限分别为`VM_ANONYM|VM_R_MASK|VM_W_MASK`和`VM_R_MASK|VM_W_MASK|VM_X_MASK`

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

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

    uint64* new_pgtbl = (uint64*)alloc_page();
    //将内核页表 （ swapper_pg_dir ） 复制到每个进程的页表中
    memcopy(new_pgtbl, swapper_pg_dir, PGSIZE);
    task->pgd = new_pgtbl;
```

```
    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) {
            // alloc space and copy content
            int page_cnt;
            if(phdr->p_memsz % PGSIZE == 0){
                page_cnt = phdr->p_memsz / PGSIZE;
            }
            else{
                page_cnt = phdr->p_memsz / PGSIZE + 1;
            }
            uint64* new_space = (uint64 *)alloc_pages(page_cnt);
            uint64* src = (uint64 *)(_sramdisk);
            memcopy(new_space , src, phdr->p_memsz);
            // do mapping
            uint64 va = phdr->p_vaddr;
            uint64 pa = (uint64)new_space - PA2VA_OFFSET;
            //printk("p_offset = %lx\n",phdr->p_offset);
            // printk("phdr->p_memsz = %|x\n",phdr->p_memsz);
            // printk("phdr->p_filesz = %|x\n",phdr->p_filesz);
            do_mmap(task, phdr->p_vaddr, phdr->p_memsz,14 , phdr->p_offset,
                    phdr->p_filesz);
        }
    }

    // allocate user stack and do mapping
    // code...
    // following code has been written for you
    // set user stack
    // pc for the user program
    //设置用户态栈,通过 alloc_page 接口申请一个空的页面来作为用户态栈，并映射到进程的页表中
    task->user_sp = alloc_page();
    uint64 va = USER_END - PGSIZE;
    uint64 pa = task->user_sp - PA2VA_OFFSET;
    uint64 sz = PGSIZE;
    uint64 perm = 0b10111;
    do_mmap(task, USER_END - PGSIZE, PGSIZE, 7 , 0, 0);
    task->satp = (csr_read(satp) & 0xfffff00000000000) |
        (((uint64)new_pgtbl - PA2VA_OFFSET) >> 12);
    //将 sepc 设置为 ehdr->e_entry
    task->thread.sepc = ehdr->e_entry;
    // printk("sepc= %|x",ehdr->e_entry);
    //配置 sstatus 中的 SPP（使得 sret 返回至 U-Mode）， SPIE （sret 之后开启中断），
SUM（S-Mode 可以访问 User 页面）
    uint64 sstatus = csr_read(sstatus);
    sstatus |= 0x0000000000040020;
    sstatus &= 0xffffffffffffeff;
    task->thread.sstatus = sstatus;
    //将 sscratch 设置为 U-Mode 的 sp，其值为 USER_END （即，用户态栈被放置在 user
space 的最后一个页面）
```

```
    task->thread.sscratch = USER_END;
}

```

- 修改 `trap.c`，添加捕获 Page Fault 的逻辑：

  通过 `csr_read(stval)` 获取发生错误的地址，调用 `find_vma()` 查找当前进程中是否存在包含 `stval` 的 VMA

  如果找到的 VMA 的 `VM_ANONYM` 位为1，表示匿名映射，使用 `alloc_page()` 分配一页内存 `new_page` ，调用 `create_mapping()` ，将出错的那一页地址映射到新分配的一页，并将权限V和U位设为1，使用 `memset()` 将 `new_page` 清零

  否则，使用 `alloc_page()` 分配一页内存，将出错的那一页地址映射到新分配的一页 `new_space` ，并将权限V和U位设为1，使用 `memcopy()` 函数将uapp对应页的内容从 复制到 `new_space` 中。

```c
void do_page_fault(struct pt_regs *regs) {
    //通过 stval 获得访问出错的虚拟内存地址（Bad Address）
    uint64 stval = csr_read(stval);
    //通过 find_vma() 查找 Bad Address 是否在某个 vma 中
    struct vm_area_struct * vma = find_vma(current, stval);
    if(vma->vm_flags & VM_ANONYM){
        uint64* new_page = (uint64*)alloc_page();
        uint64 va = (stval / PGSIZE) * PGSIZE;
        uint64 pa = (uint64)new_page - PA2VA_OFFSET;//心得
        uint64 sz = PGSIZE;
        uint64 perm;
        memset(new_page, 0, PGSIZE);
        create_mapping(current->pgd, va, pa, sz, 23);

    }
    else{
        //拷贝 uapp 中的内容
        uint64* new_space = (uint64 *)alloc_page();
        // do mapping
        uint64 va = (stval / PGSIZE) * PGSIZE;
        uint64 pa = (uint64)new_space - PA2VA_OFFSET;
        create_mapping(current->pgd, va, pa, PGSIZE,31);
        uint64* src = (uint64 *)((_sramdisk) + (stval - (vma->vm_start)) /
PGSIZE * PGSIZE);
        memcopy(new_space , src, PGSIZE);
    }
}
```

4.4 编译及测试

- 为了打印效果将用户程序的循环次数改为下图：

```c
while(1) {
    printf("[PID = %d] is running, variable: %d\n", getpid(), global_variable++);
    for (unsigned int i = 0; i < (0x8FFFFFF); i++);
}
```

- 输出如下图所示（为了方便输出，直接把时间片和优先级都设为3）

  第一轮运行时会出现`page fault`，`page fault`的数量就是每个进程出现三个，一共12次

```
...proc_init done!
2023 Hello RISC-V

switch to [PID = 4 COUNTER = 3 PRIORITY = 3]
[S] trap, scause: 000000000000000c, stval: 00000000000100e8, sepc: 00000000000100e8
[S] trap, scause: 000000000000000f, stval: 0000003fffffffe8, sepc: 0000000000010124
[S] trap, scause: 000000000000000d, stval: 0000000000011880, sepc: 0000000000010140
[PID = 4] is running, variable: 0
[PID = 4] is running, variable: 1
[PID = 4] is running, variable: 2

switch to [PID = 3 COUNTER = 3 PRIORITY = 3]
[S] trap, scause: 000000000000000c, stval: 00000000000100e8, sepc: 00000000000100e8
[S] trap, scause: 000000000000000f, stval: 0000003fffffffe8, sepc: 0000000000010124
[S] trap, scause: 000000000000000d, stval: 0000000000011880, sepc: 0000000000010140
[PID = 3] is running, variable: 0
[PID = 3] is running, variable: 1
[PID = 3] is running, variable: 2

switch to [PID = 2 COUNTER = 3 PRIORITY = 3]
[S] trap, scause: 000000000000000c, stval: 00000000000100e8, sepc: 00000000000100e8
[S] trap, scause: 000000000000000f, stval: 0000003fffffffe8, sepc: 0000000000010124
[S] trap, scause: 000000000000000d, stval: 0000000000011880, sepc: 0000000000010140
[PID = 2] is running, variable: 0
[PID = 2] is running, variable: 1
[PID = 2] is running, variable: 2
```

```
switch to [PID = 1 COUNTER = 3 PRIORITY = 3]
[S] trap, scause: 000000000000000c, stval: 00000000000100e8, sepc: 00000000000100e8
[S] trap, scause: 000000000000000f, stval: 0000003fffffffe8, sepc: 0000000000010124
[S] trap, scause: 000000000000000d, stval: 0000000000011880, sepc: 0000000000010140
[PID = 1] is running, variable: 0
[PID = 1] is running, variable: 1
[PID = 1] is running, variable: 2
```

  第二轮由于已经映射到新分配的页，正常运行

```
switch to [PID = 4 COUNTER = 3 PRIORITY = 3]
[PID = 4] is running, variable: 3
[PID = 4] is running, variable: 4
[PID = 4] is running, variable: 5

switch to [PID = 3 COUNTER = 3 PRIORITY = 3]
[PID = 3] is running, variable: 3
[PID = 3] is running, variable: 4
[PID = 3] is running, variable: 5

switch to [PID = 2 COUNTER = 3 PRIORITY = 3]
[PID = 2] is running, variable: 3
[PID = 2] is running, variable: 4
[PID = 2] is running, variable: 5

switch to [PID = 1 COUNTER = 3 PRIORITY = 3]
[PID = 1] is running, variable: 3
[PID = 1] is running, variable: 4
[PID = 1] is running, variable: 5

switch to [PID = 4 COUNTER = 3 PRIORITY = 3]
[PID = 4] is running, variable: 6
[PID = 4] is running, variable: 7
QEMU: Terminated
```

5. 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?

   答：这个域直接给出了文件内容的大小，可以避免每次使用时计算文件的大小，提高性能；可以检查vm_start和vm_end的正确性；更加灵活，可以通过该字段来判断加载文件的方式

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?

   答：因为这个数组在结构体的末尾，它可以使用结构体剩余的内存，这实现了在结构体的尾部动态分配可变长的数组，比如我们每个`task`分配一页的内存，那么除了数组前面的内容，这一页的其余空间就是数组的大小。

   不可以换位置，因为换了之后数组无法使用结构体剩余的内存，大小就不能是0.