Lab 5: RV64 缺页异常处理

3210105488 刘扬

1 实验过程

1.1 准备工作

从repo同步框架，不赘述。

在`trap_handler`中传入新的参数`stval`供本次实验使用：

```
void trap_handler(unsigned long scause, unsigned long sepc, unsigned long stval, struct
pt_regs *regs);
```

1.2 实现VMA

1.2.1 修改`proc.h`

结构体定义如下：

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

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？ */
};

/* 线程数据结构 */
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;
    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?
                               这个定义可以和前面的 vma_cnt 换个位置吗? */
};
```

添加`do_mmap`和`vm_area_struct`声明：

```
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 *find_vma(struct task_struct *task, uint64_t addr);
```

1.2.2 修改`task_init`

修改 `task_init` 函数代码，更改为 `Demand Paging`：

- 取消之前实验中对 `U-MODE` 代码以及栈进行的映射
- 调用`do_mmap`函数，建立用户 task 的虚拟地址空间信息。

在下述代码中，`load_program`调用了`do_mmap`建立用户 task 的虚拟地址空间信息。

```
void task_init() {
    test_init(NR_TASKS);
    ...
    for (int i = 1;i < NR_TASKS;i++ ){
        task[i] = (struct task_struct*)kalloc();
        task[i]->state = TASK_RUNNING;
        ...
        load_program(task[i], pgtbl); // 在load_program中调用do_mmap
        uint64 satp = (uint64)0x8 << 60;
        satp |= ((uint64)(pgtbl) - PA2VA_OFFSET) >> 12;
        task[i]->satp = satp;
    }
    printk("...proc_init done!\n");

}
```

`load_program`如下：

```
static uint64 load_program(struct task_struct* task, pagetable_t pgtbl) {
    Elf64_Ehdr* ehdr = (Elf64_Ehdr*)_sramdisk;

    uint64 phdr_start = (uint64)ehdr + ehdr->e_phoff;
    int phdr_cnt = ehdr->e_phnum;
```

```
    Elf64_Phdr* phdr;
    for (int i = 0; i < phdr_cnt; i++) {
        phdr = (Elf64_Phdr*)(phdr_start + sizeof(Elf64_Phdr) * i);
        if (phdr->p_type == PT_LOAD) {
            uint64 offset = (uint64)(phdr->p_vaddr) % PGSIZE;
            uint64 num_pages = (phdr->p_memsz + offset) / PGSIZE;
            if (num_pages * PGSIZE < (phdr->p_memsz + offset)){
                num_pages++;
            }
            uint64 length = num_pages * PGSIZE;
            do_mmap(task, phdr->p_vaddr, length, phdr->p_flags, phdr->p_offset, phdr-
>p_filesz);
        }
    }
    do_mmap(task, USER_END-PGSIZE, PGSIZE, VM_R_MASK | VM_W_MASK | VM_ANONYM, 0, 0);
    task->thread.sepc = ehdr->e_entry;
    unsigned long sstatus = csr_read(sstatus);
    sstatus &= ~(1 << 8); // sstatus[SPP] = 0
    sstatus |= 1 << 5; // sstatus[SPIE] = 1
    sstatus |= 1 << 18; // sstatus[SUM] = 1
    task->thread.sstatus = sstatus;
    task->thread.sscratch = USER_END;
}
```

在`load_program`中**只建立用户 task 的虚拟地址空间信息**，并不进行读取，当真正访问该页时，才读取对应的页。

1.2.3 `do_mmap` 实现

简单地将传入的参数进行存储即可。注意flags的对应关系。

```
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) {

    struct vm_area_struct* vma = &(task->vmas[task->vma_cnt]);
    task->vma_cnt++;
    vma->vm_start = addr;
    vma->vm_end = addr + length;
    vma->vm_flags = 0;
    if(flags & 1){
        vma->vm_flags += (1 << 3);
    }
    if(flags & 2){
        vma->vm_flags += (1 << 2);
    }
    if(flags & 4){
        vma->vm_flags += (1 << 1);
    }
    vma->vm_content_offset_in_file = vm_content_offset_in_file;
    vma->vm_content_size_in_file = vm_content_size_in_file;
```

```
}
```

在实现完上述步骤后，`trap_handler`会检测到page fault，但还不能对page fault进行处理。

1.2.4 实现 Page Fault 的检测与处理

在`trap_handler`中检测page fault

```
void trap_handler(unsigned long scause, unsigned long sepc, unsigned long stval, struct
pt_regs *regs) {
    if (scause == 0x8000000000000005) {
        ...
   }
   else if (scause == 8) {
        ...
    }
    else if(scause == (uint64)0xc || scause == (uint64)0xd || scause == (uint64)0xf) {
        printk("[S] Supervisor Page Fault, scause: %lx, stval: %lx, sepc: %lx\n", scause,
stval, sepc);
        do_page_fault(regs, stval);
    }
    else {
        ...
    }
}
```

调用`do_page_fault`处理page fault：

- 需要先找到错误地址(stval)的vma，调用了`find_vma`

  ```
  struct vm_area_struct *find_vma(struct task_struct *task, uint64_t addr){
      for(int i = 0; i < task->vma_cnt; i++){
          if(addr >= task->vmas[i].vm_start && addr < task->vmas[i].vm_end){
              return &(task->vmas[i]);
          }
      }
      return NULL;
  }
  ```

- 调用`create_mapping`进行页表映射

  ```
  create_mapping(current->pgd, PGROUNDDOWN(stval), (uint64)new_page-PA2VA_OFFSET,
  PGSIZE, (vma->vm_flags | 0x11))
  ```

- 如果是非匿名的page，则需要将`uapp`对应地址的内容拷贝进来。注意要做对齐，如果是vm_start的所在的第一页，做拷贝时需要把vm_start的页偏移量考虑上，如果是后面的页，则拷贝一整页。

  ```
  if(!(vma->vm_flags & VM_ANONYM)){
      char *src_addr = (char*)(_sramdisk + vma->vm_content_offset_in_file);
      if(stval - PGROUNDDOWN(vma->vm_start) < PGSIZE){
  ```

```
            uint64 offset = stval % PGSIZE;
            for(int j = 0;j < PGSIZE - offset;j++){
                new_page[j + offset] = src_addr[j];
            }
        }
        else{
            uint64 pg_offset = (stval - PGROUNDDOWN(vma->vm_start)) / PGSIZE;
            uint64 offset = stval % PGSIZE;
            for(int j = 0;j < PGSIZE;j++){
                new_page[j] = src_addr[j + PGSIZE * pg_offset - offset];
            }
        }
    }
```

`do_page_fault`完整代码如下：

```
void do_page_fault(struct pt_regs *regs, unsigned long stval) {
    struct vm_area_struct *vma = find_vma(current, stval);
    char *new_page = alloc_page();
    create_mapping(current->pgd, PGROUNDDOWN(stval), (uint64)new_page-PA2VA_OFFSET,
PGSIZE, (vma->vm_flags | 0x11));
    if(!(vma->vm_flags & VM_ANONYM)){
        char *src_addr = (char*)(_sramdisk + vma->vm_content_offset_in_file);
        if(stval - PGROUNDDOWN(vma->vm_start) < PGSIZE){
            uint64 offset = stval % PGSIZE;
            for(int j = 0;j < PGSIZE - offset;j++){
                new_page[j + offset] = src_addr[j];
            }
        }
        else{
            uint64 pg_offset = (stval - PGROUNDDOWN(vma->vm_start)) / PGSIZE;
            uint64 offset = stval % PGSIZE;
            for(int j = 0;j < PGSIZE;j++){
                new_page[j] = src_addr[j + PGSIZE * pg_offset - offset];
            }
        }
    }
}
```

2 实验结果

我在实验中创建了4个user进程，每个进程发生3次page fault，依次分别是取指令错误，栈错误和访存错误，一共12次page fault。

```
switch to [PID = 1 PRIORITY = 37 COUNTER = 4]
[S] Supervisor Page Fault, scause: 000000000000000c, stval: 00000000000100e8, sepc: 0
0000000000100e8
[S] Supervisor Page Fault, scause: 000000000000000f, stval: 0000003ffffffff8, sepc: 0
000000000010124
[S] Supervisor Page Fault, scause: 000000000000000d, stval: 0000000000011880, sepc: 0
000000000010140
[PID = 1] is running, variable: 0
[PID = 1] is running, variable: 1
[PID = 1] is running, variable: 2
[PID = 1] is running, variable: 3
switch to [PID = 4 PRIORITY = 66 COUNTER = 5]
[S] Supervisor Page Fault, scause: 000000000000000c, stval: 00000000000100e8, sepc: 0
0000000000100e8
[S] Supervisor Page Fault, scause: 000000000000000f, stval: 0000003ffffffff8, sepc: 0
000000000010124
[S] Supervisor Page Fault, scause: 000000000000000d, stval: 0000000000011880, sepc: 0
000000000010140
[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
```

```
switch to [PID = 3 PRIORITY = 52 COUNTER = 8]
[S] Supervisor Page Fault, scause: 000000000000000c, stval: 00000000000100e8, sepc: 0
0000000000100e8
[S] Supervisor Page Fault, scause: 000000000000000f, stval: 0000003ffffffff8, sepc: 0
000000000010124
[S] Supervisor Page Fault, scause: 000000000000000d, stval: 0000000000011880, sepc: 0
000000000010140
[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
switch to [PID = 2 PRIORITY = 88 COUNTER = 9]
[S] Supervisor Page Fault, scause: 000000000000000c, stval: 00000000000100e8, sepc: 0
0000000000100e8
[S] Supervisor Page Fault, scause: 000000000000000f, stval: 0000003ffffffff8, sepc: 0
000000000010124
[S] Supervisor Page Fault, scause: 000000000000000d, stval: 0000000000011880, sepc: 0
000000000010140
[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
SET [PID = 4 PRIORITY = 66 COUNTER = 1]
SET [PID = 3 PRIORITY = 52 COUNTER = 4]
SET [PID = 2 PRIORITY = 88 COUNTER = 10]
SET [PID = 1 PRIORITY = 37 COUNTER = 4]
switch to [PID = 4 PRIORITY = 66 COUNTER = 1]
[PID = 4] is running, variable: 6
switch to [PID = 1 PRIORITY = 37 COUNTER = 4]
[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
switch to [PID = 3 PRIORITY = 52 COUNTER = 4]
[PID = 3] is running, variable: 9
```

```
switch to [PID = 1 PRIORITY = 37 COUNTER = 4]
[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
switch to [PID = 3 PRIORITY = 52 COUNTER = 4]
[PID = 3] is running, variable: 9
[PID = 3] is running, variable: 10
[PID = 3] is running, variable: 11
[PID = 3] is running, variable: 12
switch to [PID = 2 PRIORITY = 88 COUNTER = 10]
[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
QEMU: Terminated
parallels@ubuntu-linux-22-04-02-desktop:~/oslab/lab5$
```

3 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?

在`do_mapp`中，`vm_content_size_in_file`映射的是`uapp`中的`phdr->p_filesz`，而本段内容在内容镜像中的大小是大于`phdr->p_filesz`的。在上个lab的思考题中也提到过这二者的区别：段的文件映像是指存储在文件中的段的部分。段的内存映像是指在程序执行期间加载到内存中的段的部分。`p_memsz`(即`length`)大于`p_filesz`(即`vm_content_size_in_file`)的原因是，可加载段可能包含`.bss`节，该节包含未初始化的数据。将此数据存储在磁盘上会很浪费，因此，仅在ELF文件加载到内存后才占用空间。`struct`

2. `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?

参考：[你了解C语言的“柔性数组”吗？看完你就知道了 - 知乎 (zhihu.com)]()

该数组是柔性数组。这个特性允许你在定义结构体的时候创建一个空数组，而这个数组的大小可以在程序运行的过程中根据你的需求进行更改。特别注意的一点是：这个空数组必须声明为结构体的最后一个成员，并且还要求这样的结构体至少包含一个其他类型的成员。

可以看到柔性数组必须在结构体的最后声明。在本实验中，如果和vma_cnt交换位置，在后续写入vmas数组时，会将vma_cnt的值覆盖掉，导致错误。