Lab 5: RV64 缺页异常处理

3210105944 黄锦骏

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

- Environment in previous labs.

3 实验步骤

3.0 在开始 Lab5 之前

我们的实验已经进行了将近一学期，在持续开发的代码上添加内容可能会让你的思维比较混乱。如果你认为你的代码可能需要整理，这里有一份简要的 Checklist，可以让你的代码更简洁，并让你在实现 Lab6 的时候思路更加清晰。如果你要按照以下的建议进行修改，请务必确认做好备份，并在改一小部分后就编译运行一次，不要让你辛苦写的代码 crash。当然，这一个步骤并不是强制的，完全复用之前的代码仍然可以完成 Lab6.

1. 由于一些历史遗留问题，在之前的实验指导中的 `task_struct` 中包含了一个 `thread_info` 域，但其实这个域并不必要，因为我们在内核态可以用 `sp` 和 `sscratch` 来存储内核态和用户态的两个指针，不需要借助 `thread_info` 中的两个域。因为 `switch_to` 中直接使用了汇编来访问 `task_struct` 中的内容，需要修改 `__switch_to` 中用于访问 `thread` 这个成员的一些 offset。当然如果你在别的地方也直接使用了汇编来访存 `task_struct` 中的值，你也需要一并修改。这里需要你善用 `grep` 命令。
2. 调整 `pt_regs` 和 `trap_handler`，来更好地捕获异常并辅助调试。比如我使用这样的 `pt_regs` 和 `trap_handler`：

```
struct pt_regs {
    uint64_t zero;
    ...
    uint64_t t6;
    uint64_t sepc;
    uint64_t sstatus;
    uint64_t stval;
    uint64_t sscratch;
    uint64_t scause;
};

void trap_handler(unsigned long scause, struct pt_regs *regs) {

    if (scause == 0x8000000000000005) {
        ...
   }  else if (scause == 8) {
        uint64_t sys_call_num = regs->a7;
        if (sys_call_num == 64) {
```

```
            ...
        } else if (sys_call_num == 172) {
            ...
        } else if (sys_call_num == 220) {
            ...
        } else {
            printk("[S] Unhandled syscall: %lx", sys_call_num);
            while (1);
        }
    } else if (scause == ...){
        ...
    } else {
        printk("[S] Unhandled trap, ");
        printk("scause: %lx, ", scause);
        printk("stval: %lx, ", regs->stval);
        printk("sepc: %lx\n", 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。你可以调整 `printk` 的内容来让内核给你输出你需要的信息

3. 将 `vmlinux.lds.S` 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。

3.1 准备工作

- 此次实验基于 lab4 同学所实现的代码进行。
- 从 repo 同步以下文件夹: user 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
```

3.2 实现 VMA

修改 `proc.h`，增加如下相关结构：(因为链表太麻烦了，这次让大家用数组存储 VMA)

```
#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? */
};
```

每一个 vm_area_struct 都对应于 task 地址空间的唯一**连续**区间。注意我们这里的 `vm_flag` 和 `p_flags` 并没有按 bit 进行对应，请同学们仔细对照 bit 的位置，以免出现问题。

为了支持 `Demand Paging`（见 4.3），我们需要支持对 vm_area_struct 的添加和查找。

```
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);
```

- `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。

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

```
    }
    if(vma_index == -1){
        printk("Not found the vma in the task: %lx\n", (uint64)task->pid);
        printk("addr: %lx", (uint64)addr);
        while(1);
        return NULL;
    }
    return &(task->vmas[vma_index]);
}
```

- `do_mmap` 创建一个新的 vma

```
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)
{
    task->vma_cnt++;
    task->vmas[task->vma_cnt-1].vm_start = addr;
    task->vmas[task->vma_cnt-1].vm_end = PGROUNDDOWN(addr) + length;
    task->vmas[task->vma_cnt-1].vm_flags = flags;
    task->vmas[task->vma_cnt-1].vm_content_offset_in_file = vm_content_offset_in_file;
    task->vmas[task->vma_cnt-1].vm_content_size_in_file = vm_content_size_in_file;

}
```

3.3 Page Fault Handler

3.3.1 RISC-V Page Faults

RISC-V 异常处理：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。

SCAUSE 寄存器指示发生异常的种类：

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

3.3.2 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即可

3.3.3 Demand Paging

在前面的实验中提到，Linux 在 Page Fault Handler 中需要考虑三类数据的值。我们的实验经过简化，只需要根据 `vm_area_struct` 中的 `vm_flags` 来确定当前发生了什么样的错误，并且需要如何处理。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。

例如，我们原本要为用户态虚拟地址映射一个页，需要进行如下操作：

1. 使用 `alloc_page` 分配一个页的空间
2. 对这个页中的数据进行填充
3. 将这个页映射到用户空间，供用户程序访问。并设置好对应的 U, W, X, R 权限，最后将 V 置为 1，代表其有效。

而为了减少 task 初始化时的开销，我们对一个 **Segment** 或者 **用户态的栈** 只需分别建立一个 VMA。

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，在本次实验中仅包括两个区域:
    - 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。
    - 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。

load_program函数修改如下：

```
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) {
```

```
        //     // alloc space and copy content
            uint64 page_num = (phdr->p_vaddr - PGROUNDDOWN((uint64)phdr->p_vaddr) + phdr->p_memsz + PGSIZE - 1) / PGSIZE;
            // uint64 uapp = alloc_pages(page_num);
            // memset(uapp, 0, page_num*PGSIZE);
            // for(int t = 0; t < page_num; t++){
            //     uint64* src = (uint64*)(ramdisk_start + t * PGSIZE + phdr->p_offset);
            //     uint64* dst = (uint64*)(uapp + t * PGSIZE + phdr->p_vaddr - PGROUNDDOWN((uint64)phdr->p_vaddr));
            //     for (int k = 0; k < PGSIZE / 8; k++) {
            //         dst[k] = src[k];
            //     }
            // }
            // // do mapping  X|W|R|V
            // create_mapping(task->pgd, PGROUNDDOWN((uint64)phdr->p_vaddr), uapp - PA2VA_OFFSET, PGSIZE*page_num,
            // ((phdr->p_flags & PF_X) << 3) | ((phdr->p_flags & PF_W) << 1) | ((phdr->p_flags & PF_R) >> 1) | 0b10001);
            do_mmap(task, (uint64)phdr->p_vaddr, PGSIZE*page_num,
                    ((phdr->p_flags & PF_X) << 3) | ((phdr->p_flags & PF_W) << 1) | ((phdr->p_flags & PF_R) >> 1), phdr->p_offset, phdr->p_memsz);
            printk("pid: %lx, task_pgd: %lx\n", task->pid, task->pgd);

        }
    }
```

对以下两个区域创建对应的 `vm_area_struct`，并且映射到用户空间，但是不真的分配页。映射需要一个物理地址，我们可以先使用 NULL，而在之后真的分配了页之后，再填充真正的物理地址的值。

- 代码区域：该区域从 ELF 给出的起始地址 `ehdr->e_entry` 开始，大小为 `uapp_end - uapp_start`(`ramdisk_end - ramdisk_start`).
- 用户栈

在完成上述修改之后，如果运行代码我们就可以截获一个 page fault，如下所示：

```
[S] Switch to: pid: 1, priority: 1, counter: 4
[S] Unhandled trap, scause: 000000000000000c, stval: 00000000000100e8, sepc: 00000000000100e8

************************** uapp elf_header **************************

❯ readelf -a uapp
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           RISC-V
  Version:                           0x1
  Entry point address:               0x100e8

```

```
......
************************** uapp elf_header **************************
```

可以看到，发生了缺页异常的 `sepc` 是 `0x100e8`，说明我们在 `sret` 来执行用户态程序的时候，第一条指令就因为 `V-bit` 为 0 表征其映射的地址无效而发生了异常，并且发生的异常是 Insturction Page Fault。

实现 Page Fault 的检测与处理

- 修改 `trap.c`，添加捕获 Page Fault 的逻辑

trap.c修改如下：

```c
// trap.c
#include "sbi.h"
#include "proc.h"
#include "printk.h"
#include "syscall.h"

void trap_handler(uint64 scause, uint64 sepc, struct pt_regs *regs) {

    if(scause == 0x8000000000000005)
    {
        // printk("[S] Supervisor Mode Timer Interrupt\n");
        clock_set_next_event();
        do_timer();
    }
    //Interrupt = 0 && exception code == 8
    else if(scause == 0x0000000000000008)
    {
        // printk("[S] System call Trap, ");
        // printk("scause=: %lx, ",scause);
        // printk("stval: %lx, ", regs->stval);
        // printk("sepc: %lx\n", sepc);
        uint64 sys_call_num = regs->x[17];
        regs->sepc += 4;
        if(sys_call_num == 172 || sys_call_num == 64)
            sys_call(regs, sys_call_num);
        else{
            printk("[S] Unhandled syscall: %lx", sys_call_num);
            printk("scause=%lx__",scause);
            printk("stval: %lx, ", regs->stval);
            printk("sepc: %lx\n", sepc);
            while (1);
        }
    }
    else if(scause == 0x000000000000000c || scause == 0x000000000000000f || scause == 0x000000000000000d){
        printk("[S] Page Fault Trap, ");
        printk("scause=: %lx, ",scause);
        printk("stval: %lx, ", regs->stval);
        printk("sepc: %lx\n", sepc);
        do_page_fault(regs);
    }
    else{
        printk("[S] Unhandled trap, ");
```

```
        printk("scause=: %lx, ",scause);
        printk("stval: %lx, ", regs->stval);
        printk("sepc: %lx\n", sepc);
        while(1);
    }
    return;
}


```

- 当捕获了 `Page Fault` 之后，需要实现缺页异常的处理函数 `do_page_fault`。 我们最先捕获到了一条指令页错误异常，这个异常需要你新分配一个页，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。
- 我们之后还会捕获到 `0xd, 0xf` 类型的异常，处理的逻辑可以参考这个流程：
  <!-- * 对于第一次 Page Fault，即缺失代码页导致的 Instruction Page Fault，原则上说，我们需要先到磁盘上读取程序到内存中，随后再对这块内存进行映射。但本次实验中不涉及文件操作，uapp 已经在内存中了，所以我们只需要把代码映射到相应的位置即可。
  - 对于第二次 Page Fault，即缺失栈页导致的 Store/AMO Page Fault，我们只用分配一个匿名的页（通过 kalloc），随后将其映射上去即可。
  - 如何区别这两者?可以思考一下。

do_page_fault实现如下：

```
void do_page_fault(struct pt_regs *regs){
     //1. 通过 stval 获得访问出错的虚拟内存地址（Bad Address）
     uint64 bad_addr = csr_read(stval);
     //2. 通过 find_vma() 查找 Bad Address 是否在某个 vma 中
     struct vm_area_struct* vma = find_vma(current, bad_addr);
     if(vma){
        //3. 分配一个页，将这个页映射到对应的用户地址空间
        uint64 page = alloc_page();
        uint64 page_index = ((bad_addr - PGROUNDDOWN((uint64)(vma->vm_start))) / PGSIZE);
        uint64 page_start = PGROUNDDOWN((uint64)(vma->vm_start)) + page_index * PGSIZE;
        //4. 通过 (vma->vm_flags | VM_ANONYM) 获得当前的 VMA 是否是匿名空间
        int annom_flag = vma->vm_flags & VM_ANONYM;
        //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
        memset(page, 0, PGSIZE);
        if(!annom_flag){
            uint64* src = (uint64*)(ramdisk_start + page_index * PGSIZE + vma->vm_content_offset_in_file);
            uint64* dst = (uint64*)(page + vma->vm_start - PGROUNDDOWN((uint64)(vma->vm_start)));
            for (int k = 0; k < PGSIZE / 8; k++) {
                dst[k] = src[k];
            }
            create_mapping(current->pgd, (uint64)page_start, (uint64)page - PA2VA_OFFSET, (uint64)PGSIZE, vma->vm_flags | 0b10001);
        }
        else{
            create_mapping(current->pgd, (uint64)page_start, (uint64)page - PA2VA_OFFSET, (uint64)PGSIZE, 23);
```

```
        }
    }
}
```

3.4 编译及测试

在测试时，由于大家电脑性能都不一样，如果出现了时钟中断频率比用户打印频率高很多的情况，可以减少用户程序里的 while 循环的次数来加快打印，只要 OS 和用户态程序运行符合你的预期，那就是正确的。

测试结果截图：

Main: 每个进程都会发生三次Page Fault

```
...proc_init done!
[S-MODE] Hello RISC-V
switch to [PID = 1 COUNTER = 4]
[S] Page Fault Trap, scause=: 000000000000000c, stval: 00000000000100e8, sepc: 00000000000100e8
[S] Page Fault Trap, scause=: 000000000000000f, stval: 0000003fffffffe8, sepc: 0000000000010124
[S] Page Fault Trap, scause=: 000000000000000d, stval: 00000000000118a0, sepc: 0000000000010144
[U-MODE] pid: 1, sp is 0000003fffffffe0, this is print No.1
switch to [PID = 4 COUNTER = 5]
[S] Page Fault Trap, scause=: 000000000000000c, stval: 00000000000100e8, sepc: 00000000000100e8
[S] Page Fault Trap, scause=: 000000000000000f, stval: 0000003fffffffe8, sepc: 0000000000010124
[S] Page Fault Trap, scause=: 000000000000000d, stval: 00000000000118a0, sepc: 0000000000010144
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.2
switch to [PID = 3 COUNTER = 8]
[S] Page Fault Trap, scause=: 000000000000000c, stval: 00000000000100e8, sepc: 00000000000100e8
[S] Page Fault Trap, scause=: 000000000000000f, stval: 0000003fffffffe8, sepc: 0000000000010124
[S] Page Fault Trap, scause=: 000000000000000d, stval: 00000000000118a0, sepc: 0000000000010144
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.2
switch to [PID = 2 COUNTER = 9]
[S] Page Fault Trap, scause=: 000000000000000c, stval: 00000000000100e8, sepc: 00000000000100e8
[S] Page Fault Trap, scause=: 000000000000000f, stval: 0000003fffffffe8, sepc: 0000000000010124
[S] Page Fault Trap, scause=: 000000000000000d, stval: 00000000000118a0, sepc: 0000000000010144
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.2
switch to [PID = 1 COUNTER = 1]
[U-MODE] pid: 1, sp is 0000003fffffffe0, this is print No.2
switch to [PID = 2 COUNTER = 4]
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.3
switch to [PID = 4 COUNTER = 4]
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.3
switch to [PID = 3 COUNTER = 5]
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.3
switch to [PID = 2 COUNTER = 4]
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.4
switch to [PID = 3 COUNTER = 4]
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.4
```

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?

   因为实际分配内存时，往往由于内存对齐等原因，实际分配的虚拟空间大小往往大于文件实际在虚拟内存中所占的长度，所以该变量记录了对应文件实际在虚拟内存中的长度

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?

   这是C语言中的柔性数组，可以动态分配空间。不能和vma_cnt换位置，因为柔性数组vmas的大小是不固定的，交换之后我们无法准确获得vma_cnt的对应地址。

3. 想想为什么只要拷贝那些已经分配并映射的页，那些本来应该被分配并映射，但是暂时还没有因为 Page Fault 而被分配并映射的页怎么办?

   因为已经分配并映射的页在物理内存中有对应的数据，所以可以直接进行拷贝。而那些尚未被分配并映射的页，采取了mmap的内存映射机制，为每个进程保存了映射的相关数据，之后在触发Page fault的时候再实际进行按需分配