Lab 5: RV64 缺页异常处理

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. 背景知识

下面介绍的是 Linux 中对于 VMA (virtual memory area) 和 Page Fault Handler 的介绍（顺便帮大家复习下期末考）。由于 Linux 巨大的体量，无论是 VMA 还是 Page Fault 的逻辑都较为复杂，这里只要求大家实现简化版本的，所以不要在阅读背景介绍的时候有太大的压力。

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。linux 具体某一 task 的虚拟内存区域映射关系可以通过 procfs 读取 `/proc/pid/maps` 的内容来获取:

比如，如下一个常规的 `bash` task，假设它的 task 号为 `7884`，则通过输入如下命令，就可以查看该 task 具体的虚拟地址内存映射情况(部分信息已省略)。

```
#cat /proc/7884/maps
556f22759000-556f22786000 r--p 00000000 08:05 16515165
/usr/bin/bash
556f22786000-556f22837000 r-xp 0002d000 08:05 16515165
/usr/bin/bash
556f22837000-556f2286e000 r--p 000de000 08:05 16515165
/usr/bin/bash
556f2286e000-556f22872000 r--p 00114000 08:05 16515165
/usr/bin/bash
556f22872000-556f2287b000 rw-p 00118000 08:05 16515165
/usr/bin/bash
556f22fa5000-556f2312c000 rw-p 00000000 00:00 0                          [heap]
7fb9edb0f000-7fb9edb12000 r--p 00000000 08:05 16517264
/usr/lib/x86_64-linux-gnu/libnss_files-2.31.so
7fb9edb12000-7fb9edb19000 r-xp 00003000 08:05 16517264
/usr/lib/x86_64-linux-gnu/libnss_files-2.31.so
...
7ffee5cdc000-7ffee5cfd000 rw-p 00000000 00:00 0                          [stack]
7ffee5dce000-7ffee5dd1000 r--p 00000000 00:00 0                          [vvar]
7ffee5dd1000-7ffee5dd2000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0
[vsyscall]
```

从中我们可以读取如下一些有关该 task 内虚拟内存映射的关键信息：

- `vm_start` ：(第1列) 指的是该段虚拟内存区域的开始地址

- `vm_end` ：(第2列) 指的是该段虚拟内存区域的结束地址
- `vm_flags` ：(第3列) 该 `vm_area` 的一组权限(rwx)标志，`vm_flags` 的具体取值定义可参考linux源代码的 linux/mm.h
- `vm_pgoff` ：(第4列) 虚拟内存映射区域在文件内的偏移量
- `vm_file` ：(第5/6/7列) 分别表示：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

注意这里记录的 `vm_start` 和 `vm_end` 都是用户态的虚拟地址，并且内核并不会将除了用户程序会用到的内存区域以外的部分添加成为 VMA。

可以看到，一段内存可以被用户程序当成某一个文件的一部分。如果这样的 VMA 产生了缺页异常，说明文件中对应的页不在操作系统的 buffer pool 中（回想起数据库课上学习的磁盘缓存了吗），或者是由于 buffer pool 的调度策略被换出到磁盘上了。这时候操作系统会用驱动读取硬盘上的内容，放入 buffer pool，然后修改当前 task 的页表来让其能够用原来的地址访问文件内容。而这一切对用户程序来说是完全透明的，除了访问延迟。

除了跟文件建立联系以外，VMA 还可能是一块匿名 (anonymous) 的区域。例如被标成 `[stack]` 的这一块区域，和文件之间并没有什么关系。

其它保存在 `vm_area_struct` 中的信息还有：

- `vm_ops`：该 `vm_area` 中的一组工作函数，其中是一系列函数指针，可以根据需要进行定制
- `vm_next/vm_prev`：同一 task 的所有虚拟内存区域由 **链表结构** 链接起来，这是分别指向前后两个 `vm_area_struct` 结构体的指针

可以发现，原本的 Linux 使用链表对一个 task 内的 VMA 进行管理。但是由于如今一个程序可能体量非常巨大，所以现在的 Linux 已经用虚拟地址为索引来建立红黑树了（如果你喜欢可以在这次实验中也手搓一棵红黑树）。

3.2 缺页异常 Page Fault

缺页异常是一种正在运行的程序访问当前未由内存管理单元（MMU）映射到虚拟内存的页面时，由计算机**硬件**引发的异常类型。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

总的说来，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 中对应了**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导致异常产生 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。

Page Fault 是一类比较复杂的异常，可以看到 Linux 内核中的处理时的逻辑是充满了 `if` `else` 乃至 `goto` 的：

<!-- #### 3.2.1 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

3.2.2 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即可 -->

4 实验步骤

4.0 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 Lab5 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 Lab5.

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。

```
struct pt_regs {
    uint64_t zero;
    uint64_t ra;
    uint64_t sp;
    uint64_t gp;
    uint64_t tp;
    uint64_t t0;
    uint64_t t1;
    uint64_t t2;
    uint64_t s0;
    uint64_t s1;
    uint64_t a0;
    uint64_t a1;
    uint64_t a2;
    uint64_t a3;
    uint64_t a4;
    uint64_t a5;
    uint64_t a6;
    uint64_t a7;
    uint64_t s2;
    uint64_t s3;
    uint64_t s4;
    uint64_t s5;
    uint64_t s6;
    uint64_t s7;
    uint64_t s8;
    uint64_t s9;
    uint64_t s10;
    uint64_t s11;
    uint64_t t3;
    uint64_t t4;
    uint64_t t5;
    uint64_t t6;
    uint64_t sepc;
    uint64_t sstatus;
```

```
    uint64_t stval;
    uint64_t sscratch;
    uint64_t scause;
};


void trap_handler(uint64_t scause, uint64_t 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
    // 其他interrupt / exception 可以直接忽略
    if (scause == 0x8000000000000005) {
        printk("[S] Supervisor Mode Timer Interrupt\n");
        clock_set_next_event();
        do_timer();
    }
    else if (scause == 8) {
        {
            if (regs->a7 == SYS_WRITE) {
                regs->a0 = sys_write(regs->a0, (const char *)regs->a1, regs-
>a2);
            }
            else if (regs->a7 == SYS_GETPID) {
                regs->a0 = sys_getpid();
            }
            else {
                printk("[S] Unhandled syscall: %lx", regs->a7);
                while (1);
            }
            regs->sepc += 4;
        }
    }
    else if (scause == 12 || scause == 13 || scause == 15) {
        switch (scause) {
            case 12: printk("Instruction Page Fault "); break;
            case 13: printk("Load Page Fault "); break;
            case 15: printk("Store Page Fault "); break;
            default: break;
        }
        printk(", sepc = %lx, scause = %lx, stval = %lx\n", regs->sepc, regs-
>scause, regs->stval);
        do_page_fault(regs);
    }
    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。

4.1 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
```

- 在 `user/getpid.c` 中我们设置了四个 `main` 函数。在实现了 `Page Fault` 之后第一个 `main` 函数可以成功运行，在 lab6 实现了 `fork` 之后其余三个 `main` 函数可以成功运行。这些用户程序的行为需要同学们自行理解（估计期末考也一定会考到）。

4.2 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 */

    /* 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_t addr){
      // 在 task->vmas 数组中查找 addr 对应的 vma
      for (uint64_t 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;
  }
  ```

- `do_mmap` 创建一个新的 vma

  ```
  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){
      // 添加 vma 到 task->vmas 数组中
      struct vm_area_struct *vma = &(task->vmas[task->vma_cnt]);
      task->vma_cnt++;
      // 初始化 vma 的相关信息
      vma->vm_start = addr;
      vma->vm_end = addr + length;
      vma->vm_content_offset_in_file = vm_content_offset_in_file;
      vma->vm_content_size_in_file = vm_content_size_in_file;
      vma->vm_flags = flags;
  }
  ```

4.3 Page Fault Handler

4.3.1 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

4.3.2 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

4.3.3 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。

```
static uint64_t load_program(struct task_struct *task) {
    Elf64_Ehdr* ehdr = (Elf64_Ehdr*)_sramdisk;   // ELF 文件的起始地址

    uint64_t phdr_start = (uint64_t)ehdr + ehdr->e_phoff;   // 获得 phdr 的起始地址
    int phdr_cnt = ehdr->e_phnum;   // 获得段的个数

    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) {  // 如果段的 type 是 PT_LOAD, 则装入内存

            // 先将 uapp 拷贝，再做地址映射
            uint64 offset = (uint64)(phdr->p_vaddr) - PGROUNDDOWN(phdr->p_vaddr);
            uint64 num_pages = (phdr->p_memsz + offset) / PGSIZE + 1;

            uint64 length = num_pages * PGSIZE;
            uint64 flags = phdr->p_flags << 1 | VM_X_MASK;
            // demand paging 的初始化方式，不再直接分配物理页
            do_mmap(task, phdr->p_vaddr, length, flags, phdr->p_offset, phdr-
>p_filesz);
        }
    }

    // 映射 U-Mode Stack, 权限为 U|-|W|R|V
    do_mmap(task, USER_END - PGSIZE, PGSIZE, VM_R_MASK | VM_ANONYM | VM_W_MASK,
0, 0);


    // 将 sepc 设置为 ELF 文件中可执行段的起始地址
    task->thread.sepc = ehdr->e_entry;
    // sstatus
    task->thread.sstatus = 0x40020;
    // 将 sscratch 设置为 U-mode 的 sp
    task->thread.sscratch = USER_END;
}
```

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

```
void trap_handler(uint64_t scause, uint64_t sepc, struct pt_regs *regs)
{

    if (scause == 0x8000000000000005) {
        ......
    } else if (scause == 12 || scause == 13 || scause == 15) {
        switch (scause) {
            case 12: printk("Instruction Page Fault "); break;
            case 13: printk("Load Page Fault "); break;
            case 15: printk("Store Page Fault "); break;
            default: break;
        }
        printk(", sepc = %lx, scause = %lx, stval = %lx\n", regs->sepc, regs-
>scause, regs->stval);
        do_page_fault(regs);
    }
    else {
        printk("[S] Unhandled trap, ");
        printk("scause: %lx, ", scause);
```

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

- 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。

```
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）
     2. 通过 find_vma() 查找 Bad Address 是否在某个 vma 中
     3. 分配一个页，将这个页映射到对应的用户地址空间
     4. 通过 (vma->vm_flags & VM_ANONYM) 获得当前的 VMA 是否是匿名空间
     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
    */
    uint64_t bad_addr = regs->stval; // 获得访问出错的虚拟内存地址
    struct vm_area_struct *vma = find_vma(current, bad_addr); //查找bad_addr是否在某个vma中
    if (vma != NULL) {
        // 若找到bad_addr所在的vma
        uint64 new_page = (uint64)alloc_page();
        uint64 perm = vma->vm_flags | 0x11; // PTN_U | PTN_V
        create_mapping(current->pgd, PGROUNDDOWN(bad_addr), new_page - PA2VA_OFFSET, PGSIZE, perm);
        if (!(vma->vm_flags & VM_ANONYM)) {
            // 非匿名page，需要拷贝
            uint64 src = (uint64)_sramdisk + vma->vm_content_offset_in_file;
            uint64 offset = vma->vm_start % PGSIZE;
            if (PGROUNDUP(bad_addr) - vma->vm_start < PGSIZE) {
                // bad_addr 在 vma 的开头
                uint64 size = vma->vm_end - offset >= PGROUNDUP(bad_addr) ? PGSIZE-offset : vma->vm_content_size_in_file;
                memcpy((void *)new_page + offset, src, size);
            }
            else if (vma->vm_end - PGROUNDDOWN(bad_addr) < PGSIZE) {
                // bad_addr 在 vma 的末尾
                uint64 offset = vma->vm_end % PGSIZE;
                memcpy((void *)new_page, src, offset);
            }
            else {
                // 匿名拷贝，只需要简单映射
                memcpy((void *)new_page, src, PGSIZE);
            }
```

```
        }
    } else {
        printk("Not find %lx in vma. pid = %d\n", bad_addr, current->pid);
        while (1);
    }
    return;
}
```

4.4 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。

第一次调度：

```
SET [PID = 1 COUNTER = 1]
SET [PID = 2 COUNTER = 4]
SET [PID = 3 COUNTER = 10]
SET [PID = 4 COUNTER = 4]

switch to [PID = 1 COUNTER = 1]
Instruction Page Fault
sepc = 00000000000100e8, scause = 000000000000000c, stval = 00000000000100e8
Store Page Fault
sepc = 0000000000010124, scause = 000000000000000f, stval = 0000003fffffffe8
Store Page Fault
sepc = 00000000000107cc, scause = 000000000000000f, stval = 000000000001189c
UAPP
[PID = 1] is running, this is print No. 0
[PID = 1] is running, this is print No. 1
[PID = 1] is running, this is print No. 2
[PID = 1] is running, this is print No. 3
[S] Supervisor Mode Timer Interrupt

switch to [PID = 2 COUNTER = 4]
Instruction Page Fault
sepc = 00000000000100e8, scause = 000000000000000c, stval = 00000000000100e8
Store Page Fault
sepc = 0000000000010124, scause = 000000000000000f, stval = 0000003fffffffe8
Store Page Fault
sepc = 00000000000107cc, scause = 000000000000000f, stval = 000000000001189c
UAPP
[PID = 2] is running, this is print No. 0
[PID = 2] is running, this is print No. 1
[PID = 2] is running, this is print No. 2
[PID = 2] is running, this is print No. 3
[S] Supervisor Mode Timer Interrupt
[PID = 2] is running, this is print No. 4
[PID = 2] is running, this is print No. 5
[PID = 2] is running, this is print No. 6
[PID = 2] is running, this is print No. 7
[S] Supervisor Mode Timer Interrupt
[PID = 2] is running, this is print No. 8
[PID = 2] is running, this is print No. 9
[PID = 2] is running, this is print No. 10
[PID = 2] is running, this is print No. 11
[S] Supervisor Mode Timer Interrupt
[PID = 2] is running, this is print No. 12
[PID = 2] is running, this is print No. 13
[PID = 2] is running, this is print No. 14
[S] Supervisor Mode Timer Interrupt
```

```
switch to [PID = 4 COUNTER = 4]
Instruction Page Fault
sepc = 00000000000100e8, scause = 000000000000000c, stval = 00000000000100e8
Store Page Fault
sepc = 0000000000010124, scause = 000000000000000f, stval = 0000003fffffffff8
Store Page Fault
sepc = 00000000000107cc, scause = 000000000000000f, stval = 000000000001189c
UAPP
[PID = 4] is running, this is print No. 0
[PID = 4] is running, this is print No. 1
[PID = 4] is running, this is print No. 2
[PID = 4] is running, this is print No. 3
[S] Supervisor Mode Timer Interrupt
[PID = 4] is running, this is print No. 4
[PID = 4] is running, this is print No. 5
[PID = 4] is running, this is print No. 6
[PID = 4] is running, this is print No. 7
[S] Supervisor Mode Timer Interrupt
[PID = 4] is running, this is print No. 8
[PID = 4] is running, this is print No. 9
[PID = 4] is running, this is print No. 10
[PID = 4] is running, this is print No. 11
[S] Supervisor Mode Timer Interrupt
[PID = 4] is running, this is print No. 12
[PID = 4] is running, this is print No. 13
[PID = 4] is running, this is print No. 14
[S] Supervisor Mode Timer Interrupt

switch to [PID = 3 COUNTER = 10]
Instruction Page Fault
sepc = 00000000000100e8, scause = 000000000000000c, stval = 00000000000100e8
Store Page Fault
sepc = 0000000000010124, scause = 000000000000000f, stval = 0000003fffffffff8
Store Page Fault
sepc = 00000000000107cc, scause = 000000000000000f, stval = 000000000001189c
UAPP
[PID = 3] is running, this is print No. 0
[PID = 3] is running, this is print No. 1
[PID = 3] is running, this is print No. 2
[PID = 3] is running, this is print No. 3
[S] Supervisor Mode Timer Interrupt
```

```
[PID = 3] is running, this is print No. 4
[PID = 3] is running, this is print No. 5
[PID = 3] is running, this is print No. 6
[PID = 3] is running, this is print No. 7
[S] Supervisor Mode Timer Interrupt
[PID = 3] is running, this is print No. 8
[PID = 3] is running, this is print No. 9
[PID = 3] is running, this is print No. 10
[PID = 3] is running, this is print No. 11
[S] Supervisor Mode Timer Interrupt
[PID = 3] is running, this is print No. 12
[PID = 3] is running, this is print No. 13
[PID = 3] is running, this is print No. 14
[PID = 3] is running, this is print No. 15
[S] Supervisor Mode Timer Interrupt
[PID = 3] is running, this is print No. 16
[PID = 3] is running, this is print No. 17
[PID = 3] is running, this is print No. 18
[PID = 3] is running, this is print No. 19
[S] Supervisor Mode Timer Interrupt
[PID = 3] is running, this is print No. 20
[PID = 3] is running, this is print No. 21
[PID = 3] is running, this is print No. 22
[S] Supervisor Mode Timer Interrupt
[PID = 3] is running, this is print No. 23
[PID = 3] is running, this is print No. 24
[PID = 3] is running, this is print No. 25
[PID = 3] is running, this is print No. 26
[S] Supervisor Mode Timer Interrupt
[PID = 3] is running, this is print No. 27
[PID = 3] is running, this is print No. 28
[PID = 3] is running, this is print No. 29
[PID = 3] is running, this is print No. 30
[S] Supervisor Mode Timer Interrupt
[PID = 3] is running, this is print No. 31
[PID = 3] is running, this is print No. 32
[PID = 3] is running, this is print No. 33
[PID = 3] is running, this is print No. 34
[S] Supervisor Mode Timer Interrupt
[PID = 3] is running, this is print No. 35
[PID = 3] is running, this is print No. 36
[PID = 3] is running, this is print No. 37
[PID = 3] is running, this is print No. 38
[S] Supervisor Mode Timer Interrupt
```

第二次调度：

```
switch to [PID = 4 COUNTER = 2]
[PID = 4] is running, this is print No. 15
[PID = 4] is running, this is print No. 16
[PID = 4] is running, this is print No. 17
[PID = 4] is running, this is print No. 18
[S] Supervisor Mode Timer Interrupt
[PID = 4] is running, this is print No. 19
[PID = 4] is running, this is print No. 20
[PID = 4] is running, this is print No. 21
[PID = 4] is running, this is print No. 22
[S] Supervisor Mode Timer Interrupt

switch to [PID = 3 COUNTER = 5]
[PID = 3] is running, this is print No. 39
[PID = 3] is running, this is print No. 40
[PID = 3] is running, this is print No. 41
[S] Supervisor Mode Timer Interrupt
[PID = 3] is running, this is print No. 42
[PID = 3] is running, this is print No. 43
[PID = 3] is running, this is print No. 44
[PID = 3] is running, this is print No. 45
[S] Supervisor Mode Timer Interrupt
[PID = 3] is running, this is print No. 46
[PID = 3] is running, this is print No. 47
[PID = 3] is running, this is print No. 48
[PID = 3] is running, this is print No. 49
[S] Supervisor Mode Timer Interrupt
[PID = 3] is running, this is print No. 50
[PID = 3] is running, this is print No. 51
[PID = 3] is running, this is print No. 52
[PID = 3] is running, this is print No. 53
[S] Supervisor Mode Timer Interrupt
[PID = 3] is running, this is print No. 54
[PID = 3] is running, this is print No. 55
[PID = 3] is running, this is print No. 56
[PID = 3] is running, this is print No. 57
[S] Supervisor Mode Timer Interrupt

switch to [PID = 1 COUNTER = 10]
[PID = 1] is running, this is print No. 4
[PID = 1] is running, this is print No. 5
[PID = 1] is running, this is print No. 6
[PID = 1] is running, this is print No. 7
[S] Supervisor Mode Timer Interrupt
[PID = 1] is running, this is print No. 8
[PID = 1] is running, this is print No. 9
[PID = 1] is running, this is print No. 10
[PID = 1] is running, this is print No. 11
```

```
[S] Supervisor Mode Timer Interrupt
[PID = 1] is running, this is print No. 16
[PID = 1] is running, this is print No. 17
[PID = 1] is running, this is print No. 18
[PID = 1] is running, this is print No. 19
[S] Supervisor Mode Timer Interrupt
[PID = 1] is running, this is print No. 20
[PID = 1] is running, this is print No. 21
[PID = 1] is running, this is print No. 22
[S] Supervisor Mode Timer Interrupt
[PID = 1] is running, this is print No. 23
[PID = 1] is running, this is print No. 24
[PID = 1] is running, this is print No. 25
[PID = 1] is running, this is print No. 26
[S] Supervisor Mode Timer Interrupt
[PID = 1] is running, this is print No. 27
[PID = 1] is running, this is print No. 28
[PID = 1] is running, this is print No. 29
[PID = 1] is running, this is print No. 30
[S] Supervisor Mode Timer Interrupt
[PID = 1] is running, this is print No. 31
[PID = 1] is running, this is print No. 32
[PID = 1] is running, this is print No. 33
[PID = 1] is running, this is print No. 34
[S] Supervisor Mode Timer Interrupt
[PID = 1] is running, this is print No. 35
[PID = 1] is running, this is print No. 36
[PID = 1] is running, this is print No. 37
[PID = 1] is running, this is print No. 38
[S] Supervisor Mode Timer Interrupt
[PID = 1] is running, this is print No. 39
[PID = 1] is running, this is print No. 40
[PID = 1] is running, this is print No. 41
[S] Supervisor Mode Timer Interrupt

switch to [PID = 2 COUNTER = 10]
[PID = 2] is running, this is print No. 15
[PID = 2] is running, this is print No. 16
[PID = 2] is running, this is print No. 17
[PID = 2] is running, this is print No. 18
[S] Supervisor Mode Timer Interrupt
[PID = 2] is running, this is print No. 19
[PID = 2] is running, this is print No. 20
[PID = 2] is running, this is print No. 21
[PID = 2] is running, this is print No. 22
[S] Supervisor Mode Timer Interrupt
[PID = 2] is running, this is print No. 23
[PID = 2] is running, this is print No. 24
```

```
[S] Supervisor Mode Timer Interrupt
[PID = 2] is running, this is print No. 27
[PID = 2] is running, this is print No. 28
[PID = 2] is running, this is print No. 29
[PID = 2] is running, this is print No. 30
[S] Supervisor Mode Timer Interrupt
[PID = 2] is running, this is print No. 31
[PID = 2] is running, this is print No. 32
[PID = 2] is running, this is print No. 33
[S] Supervisor Mode Timer Interrupt
[PID = 2] is running, this is print No. 34
[PID = 2] is running, this is print No. 35
[PID = 2] is running, this is print No. 36
[PID = 2] is running, this is print No. 37
[S] Supervisor Mode Timer Interrupt
[PID = 2] is running, this is print No. 38
[PID = 2] is running, this is print No. 39
[PID = 2] is running, this is print No. 40
[PID = 2] is running, this is print No. 41
[S] Supervisor Mode Timer Interrupt
[PID = 2] is running, this is print No. 42
[PID = 2] is running, this is print No. 43
[PID = 2] is running, this is print No. 44
[PID = 2] is running, this is print No. 45
[S] Supervisor Mode Timer Interrupt
[PID = 2] is running, this is print No. 46
[PID = 2] is running, this is print No. 47
[PID = 2] is running, this is print No. 48
[S] Supervisor Mode Timer Interrupt
[PID = 2] is running, this is print No. 49
[PID = 2] is running, this is print No. 50
[PID = 2] is running, this is print No. 51
[PID = 2] is running, this is print No. 52
[S] Supervisor Mode Timer Interrupt
```

可见，在 main 函数作为 uapp 时，一共发生了3 * 4 = 12 次 page_fault

5.思考题

使用 Ctrl-f 来搜寻当前页面中的问号，根据上下文来回答这些问题：

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?

   这个域的作用是帮助定位 uapp 中的 text 段的确切位置。在指令缺页时，需要将 uapp 中的 text 段复制并装 入进程的内存中，而定位 text 的位置时需要跳过 elf 文件头以及前面部分内容，找到 text. 该值并不等于 (vm_end-vm_start)。

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?

这是为了在结构体末尾定义一个可变长度的数组而不占用结构体的空间。可以在运行时动态的根据需要分配 更多空间，而不必在编译时确定数组的大小。 不可以调换位置。如果将 vma_cnt 放在该数组后面，每当数组扩展时，vma_cnt 的地址都会改变，会导致地 址的错误。