浙江大学

《操作系统原理与实践》
实验报告

实验名称：	Lab 5: RV64 缺页异常处理与 fork 机制
姓　　名：	王晓宇
学　　号：	3220104364
电子邮箱：	3220104364@zju.edu.cn
联系电话：	19550222634
授课教师：	申文博
助　　教：	陈淦豪&王鹤翔&许昊瑞

2024 年 12 月 1 日

Lab 5: RV64 缺页异常处理与 fork 机制

1 实验内容及简要原理介绍

1.1 实验目的

1.2 简要原理介绍

1.2.1 缺页异常 page fault

1.2.2 Fork 系统调用

2 实验具体过程与代码实现

2.1 准备工作

2.2 缺页异常处理

2.2.1 实现虚拟内存管理功能

2.2.2 修改 task_init

2.2.3 实现 page fault handler

2.2.4 测试缺页处理

2.3 实现 fork 系统调用

2.3.1 表面的准备工作

2.3.2 思考 do_fork 要做什么

2.3.3 拷贝内核栈

2.3.4 创建子进程页表

2.3.5 处理进程返回逻辑

2.3.6 测试 fork

3 实验结果与分析

4 思考题与心得体会

Lab 5: RV64 缺页异常处理与 fork 机制

1 实验内容及简要原理介绍

1.1 实验目的

- 通过 vm_area_struct 数据结构实现对进程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
- 为进程加入 fork 机制，能够支持通过 fork 创建新的用户态进程

1.2 简要原理介绍

1.2.1 缺页异常 page fault

在一个启用了虚拟内存的系统上，若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，或访问权限不足，则会由计算机硬件引发的缺页异常（page fault）。

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）不在当前进程的 `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。

1.2.2 Fork 系统调用

Fork 是 Linux 中的重要系统调用，它的作用是将进行了该系统调用的 task 完整地复制一份，并加入 Ready Queue。这样在下一次调度发生时，调度器就能够发现多了一个 task。从这时候开始，新的 task 就可能被正式从 Ready 调度到 Running，而开始执行了。需留意，fork 具有以下特点：

- Fork 通过复制当前进程创建一个新的进程，新进程称为子进程，而原进程称为父进程
- 子进程和父进程在不同的内存空间上运行

- Fork 成功时，父进程返回子进程的 PID，子进程返回 `0`；失败时，父进程返回 `-1`
- 创建的子 task 需要深拷贝 `task_struct`，调整自己的页表、栈和 CSR 寄存器等信息，复制一份在用户态会用到的内存信息（用户态的栈、程序的代码和数据等），并且将自己伪装成是一个因为调度而加入了 Ready Queue 的普通程序来等待调度。在调度发生时，这个新 task 就像是原本就在等待调度一样，被调度器选择并调度。

2 实验具体过程与代码实现

2.1 准备工作

- 从仓库同步 `user/main.c` 文件并删除原来的 `getpid.c`
- 修改 `user/Makefile`：

```
# src/lab5/user/Makefile
TEST        = PFH1

CFLAG       = ...末尾添加 -D$(TEST)
```

2.2 缺页异常处理

2.2.1 实现虚拟内存管理功能

每块 vma 都有自己的 flag 来定义权限以及分类（是否匿名），在`proc.h`中添加以下宏定义，接下来要添加 vma 的数据结构，我们采用链表的实现（其实并不复杂，因为只用考虑插入和遍历）：

src > lab5 > arch > riscv > include > C proc.h > ...

```c
#ifndef __PROC_H__

//added in Lab5

#define VM_ANON 0x1
#define VM_READ 0x2
#define VM_WRITE 0x4
#define VM_EXEC 0x8

struct vm_area_struct {
    struct mm_struct *vm_mm;    // 所属的 mm_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
    struct vm_area_struct *vm_next, *vm_prev;   // 链表指针
    uint64_t vm_flags;          // VMA 对应的 flags
    // struct file *vm_file;    // 对应的文件（目前还没实现，而且我们只有一个 uapp 所以暂不需要）
    uint64_t vm_pgoff;          // 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
    uint64_t vm_filesz;         // 对应的文件内容的长度
};

struct mm_struct {
    struct vm_area_struct *mmap;
};

//end added in Lab5
```

```c
/* 线程数据结构 */
struct task_struct {
    uint64_t state;     // 线程状态
    uint64_t counter;   // 运行剩余时间
    uint64_t priority;  // 运行优先级 1 最低 10 最高
    uint64_t pid;       // 线程 id

    struct thread_struct thread;
    //added in Lab4
    //必须加在thread_struct后面，因为在__switch_to函数中会计算thread变量的位置
    uint64_t *pgd;  // 用户态页表
    //end added in Lab4

    //added in Lab5
    struct mm_struct mm;
    //end added in Lab5
};
```

每一个 `vm_area_struct` 都对应于 **task** 地址空间的唯一**连续**区间。

为了支持 **demand paging**，我们需要支持对 `vm_area_struct` 的添加和查找：

- **find_vma** 函数：实现对 `vm_area_struct` 的查找
  - 根据传入的地址 `addr`，遍历链表 `mm` 包含的 **VMA** 链表，找到该地址所在的 `vm_area_struct`
  - 如果链表中所有的 `vm_area_struct` 都不包含该地址，则返回 `NULL`

```c
struct vm_area_struct *find_vma(struct mm_struct *mm, uint64_t addr){
    for(struct vm_area_struct *vma = mm->mmap; vma; vma = vma->vm_next){
        if(vma->vm_start <= addr && addr < vma->vm_end){
            return vma;
        }
    }
    return NULL;
}
```

- `do_mmap` 函数：实现 `vm_area_struct` 的添加
    - 新建 `vm_area_struct` 结构体，根据传入的参数对结构体赋值，并添加到 `mm` 指向的 VMA 链表中

```c
uint64_t do_mmap(struct mm_struct *mm, uint64_t addr, uint64_t len, uint64_t vm_pgoff, uint64_t vm_filesz, uint64_t flags){
    struct vm_area_struct *vma_unit = (struct vm_area_struct*)alloc_page();
    vma_unit->vm_start = addr;
    vma_unit->vm_end = addr + len;
    vma_unit->vm_pgoff = vm_pgoff;
    vma_unit->vm_filesz = vm_filesz;
    vma_unit->vm_flags = flags;
    vma_unit->vm_prev = vma_unit->vm_next = NULL;
    // Log("New VMA: [0x%lx, 0x%lx), pgoff: 0x%lx, filesz: 0x%lx, flags: 0x%lx\n", vma_unit->vm_start, vma_unit->vm_end, vma_unit->vm_p
    struct vm_area_struct *vma;
    for(vma = mm->mmap; (vma && vma->vm_next); vma = vma->vm_next){}
    if(vma){
        vma->vm_next = (struct vm_area_struct *)vma_unit;
        vma_unit->vm_prev = vma;
    }else{
        mm->mmap = (struct vm_area_struct *)vma_unit;
    }
    vma_unit->vm_mm = mm;
    return addr;
}
```

2.2.2 修改 task_init

接下来我们要修改 `task_init` 来实现 demand paging。

为了减少 task 初始化时的开销，我们这样对一个 Segment 或者用户态的栈建立映射的操作只需改成分别建立一个 VMA 即可，具体的分配空间、填充数据的操作等后面再来完成。

我们需要修改 `task_init` 函数代码，更改为 demand paging：

- 删除（注释）掉之前实验中对用户栈、代码 load segment 的映射操作（alloc 和 create_mapping）

```c
src > lab5 > arch > riscv > kernel > C proc.c > task_init()
 87  void task_init() {
 97      for(int i = 1; i < nr_tasks; i++) {
112          //reuse swapper_pg_dir in user space
113          task[i]->pgd = (uint64_t*)alloc_page();
114          memcpy((void *)task[i]->pgd, (void *)&swapper_pg_dir, PGSIZE);
115
116              //load elf program++++++++++++++++++++++++++++++(deleted in Lab5)
117                 //
118                     // load_program(task[i]);
119                 //
120              //end load elf program++++++++++++++++++++++++++
121
122              //Or Or Or Or Or Or
123
124 >            //load simple program++++++++++++++++++++++++++(deleted in Lab5)···
133
134          //new stack for user space (deleted in Lab5)
135          //Size of stack is PGSIZE
136              // uint64_t* new_stack = (uint64_t*)alloc_page();
137              // create_mapping(task[i]->pgd, USER_END - PGSIZE, (uint64_t)new_stack - PA2VA_OFFSET, PGSIZE, 0x17);//U|W|R|V
138          //end added in Lab4-------------------------------------------
139
```

- 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 起始用户态虚拟地址 `phdr->p_vaddr` 开始，对应文件中偏移量为 `phdr->p_offset` 开始的部分
- 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（`VM_ANON`）

> 我们这里对于两块区域分别操作：
>
> 1. 对于代码和数据区域
>
>   我们新建了`load_program_vma`函数，其作用与`load_program`类似，只不过其映射操作由`create_mapping`变为了`do_mmap`，并且映射单位也从`一页`变为了`phdr->p_memsz`大小
>
> 2. 对于用户栈
>
>   我们直接使用`do_mmap`进行了映射

```c
void load_program_vma(struct task_struct *task) {
    Elf64_Ehdr *ehdr = (Elf64_Ehdr *)&_sramdisk;
    Elf64_Phdr *phdrs = (Elf64_Phdr *)((char*)(&_sramdisk) + ehdr->e_phoff);
    for (int i = 0; i < ehdr->e_phnum; ++i) {
        Elf64_Phdr *phdr = phdrs + i;
        if (phdr->p_type == PT_LOAD) {
            uint64_t perm = ((phdr->p_flags & PF_X) ? VM_EXEC :0x0) | ((phdr->p_flags & PF_W) ? VM_WRITE :0x0) |((phdr->p_flags & PF_R) ? VM_READ :0x0);//|XWR
            do_mmap(&task->mm, phdr->p_vaddr, phdr->p_memsz, phdr->p_offset, phdr->p_filesz, perm);
        }
    }
    task->thread.sepc = ehdr->e_entry;
}
```

```c
src > lab5 > arch > riscv > kernel > C proc.c > task_init()
 87   void task_init() {
 97           for(int i = 1; i < nr_tasks; i++) {
140           //added in Lab5
144               load_program_vma(task[i]);
145               do_mmap(&task[i]->mm, USER_END - PGSIZE, PGSIZE, 0, PGSIZE, VM_READ | VM_WRITE | VM_ANON);
146           //end added in Lab5
147       }
148       printk("...task_init done!\n");
149   }
150
```

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：

```
...buddy_init done!
...mm_init done!
New VMA: [0x100e8, 0x12648), pgoff: 0xe8, filesz: 0x1169, flags: 0xe
New VMA: [0x3ffffff000, 0x4000000000), pgoff: 0x0, filesz: 0x1000, flags: 0x7
New VMA: [0x100e8, 0x12648), pgoff: 0xe8, filesz: 0x1169, flags: 0xe
New VMA: [0x3ffffff000, 0x4000000000), pgoff: 0x0, filesz: 0x1000, flags: 0x7
New VMA: [0x100e8, 0x12648), pgoff: 0xe8, filesz: 0x1169, flags: 0xe
New VMA: [0x3ffffff000, 0x4000000000), pgoff: 0x0, filesz: 0x1000, flags: 0x7
New VMA: [0x100e8, 0x12648), pgoff: 0xe8, filesz: 0x1169, flags: 0xe
New VMA: [0x3ffffff000, 0x4000000000), pgoff: 0x0, filesz: 0x1000, flags: 0x7
...task_init done!
2024 ZJU Operating System
SET [PID = 1 PRIORITY = 7 COUNTER = 7]
SET [PID = 2 PRIORITY = 10 COUNTER = 10]
SET [PID = 3 PRIORITY = 4 COUNTER = 4]
SET [PID = 4 PRIORITY = 1 COUNTER = 1]

switch to [PID = 2 PRIORITY = 10 COUNTER = 10]
[trap.c,32,trap_handler] [S] Unhandled Exception: scause=12, sepc=100e8
QEMU: Terminated
axin0401@LAPTOP-0P208VUK:/mnt/d/cs/os24fall-stu/src/lab5$
```

2.2.3 实现 page fault handler

接下来我们需要修改 `trap.c`，为 `trap_handler` 添加捕获 page fault 的逻辑，分别需要捕获 12，13，15 号异常。

> 注意到异常的Interrupt位是0

```
src > lab5 > arch > riscv > kernel > C trap.c > trap_handler(uint64_t, uint64_t, pt_regs *)
void trap_handler(uint64_t scause, uint64_t sepc, struct pt_regs *regs) {
    switch (highest_bit)
        if(scause == 0x8){
            switch (regs->reg17)
            regs->sepc += 0x4;
        }else if(scause == 12||scause == 13||scause == 15){
            Log("[PID = %d PC = 0x%lx] valid page fault at `0x%llx` with cause %d",current->pid,regs->sepc,regs->stval,scause);
            do_page_fault(regs);
        }else{
            // regs->sepc += 0x4;
            // Err("[S] Unhandled Exception: scause=%d, sepc=%llx", scause, sepc);
        }

    }

}
```

当捕获了 page fault 之后，需要实现缺页异常的处理函数 `do_page_fault`，它可以同时处理三种不同的 page fault。

```
src > lab5 > arch > riscv > kernel > C trap.c > do_page_fault(pt_regs *)
    void trap_handler(uint64_t scause, uint64_t sepc, struct pt_regs *regs) {

49
50  void do_page_fault(struct pt_regs *regs) {
51      struct vm_area_struct *vma = find_vma(&current->mm, regs->stval);
52      if(vma == NULL){
53          Err("Page Fault at [0x%lx]\n", regs->stval);
54      }else{
55          //如果非法（比如触发的是 instruction page fault 但 vma 权限不允许执行），则 Err 输出错误信息
56          // if(((vma->vm_flags & VM_EXEC) == 0) ){
57          //     Err("Instruction page Fault at [0x%lx] with no exec permission\n", regs->stval);
58          // }
59          //如果当前访问的虚拟地址在 VMA 中存在记录，则需要判断产生异常的原因：
60              //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
61              //如果不是，则访问的页是存在数据的（如代码），需要从相应位置读取出内容，然后映射到页表中
62          uint64_t new_pg = (uint64_t)alloc_page();
63          memset((void*)new_pg, 0, PGSIZE);
64          uint64_t offset = vma->vm_start & (PGSIZE-0x1);
65          uint64_t elf_beginning = (uint64_t)(&_sramdisk) + offset;
66          uint64_t end_offset_rev = PGSIZE - (vma->vm_start + vma->vm_filesz) & (PGSIZE-0x1);
67          uint64_t perm = ((vma->vm_flags & VM_EXEC) ? 0x8 :0x0) | ((vma->vm_flags & VM_WRITE) ? 0x4 :0x0) |((vma->vm_flags & VM_READ) ? 0x2 :0x0);//|XWR
68          perm |= 0x11; //U|V
69          bool is_in_fstpg = PGROUNDDOWN(regs->stval) == PGROUNDDOWN((uint64_t)vma->vm_start);
70          bool is_in_lstpg = PGROUNDDOWN(regs->stval) == PGROUNDDOWN((uint64_t)(vma->vm_start + vma->vm_filesz -1));
71          if(!(vma->vm_flags & VM_ANON)){
72              if(is_in_fstpg){
73                  if(is_in_lstpg){
74                      memcpy((void*)(new_pg+offset), (void*)((uint64_t)(&_sramdisk)+offset), PGSIZE-offset-end_offset_rev);
75                  }else{
76                      memcpy((void*)(new_pg+offset), (void*)((uint64_t)(&_sramdisk)+offset), PGSIZE-offset);
77                  }
78              }else{
79                  if(is_in_lstpg){
80                      memcpy((void*)new_pg, (void*)(elf_beginning+(PGROUNDDOWN(regs->stval) - vma->vm_start)), PGSIZE - end_offset_rev);
81                  }else{
82                      memcpy((void*)new_pg, (void*)(elf_beginning+(PGROUNDDOWN(regs->stval) - vma->vm_start)), PGSIZE);
83                  }
84              }
85          }
86          create_mapping(current->pgd, PGROUNDDOWN(regs->stval), (uint64_t)new_pg - PA2VA_OFFSET, PGSIZE, perm);
87      }
88  }
```

函数的具体逻辑为：

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
    - 如果不在，则出现非预期错误，可以通过 `Err` 宏输出错误信息
    - 如果在，则根据 vma 的 flags 权限判断当前 page fault 是否合法
        - 如果非法（比如触发的是 instruction page fault 但 vma 权限不允许执行），则 `Err` 输出错误信息（我们在本次实验种不会遇到这种情况，因此可以不实现）
        - 其他情况合法，需要我们按接下来的流程创建映射
3. 分配一个页，接下来要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
    - 如果是匿名空间（即**指明是用户栈**），则直接映射即可
    - 如果不是，则需要根据 `vma->vm_pgoff` 等信息从 ELF 中读取数据，填充后映射到用户空间

> 这里判断了发生Page Fault的位置使用了两个布尔变量，`is_in_fstpg`指明是否存在于ELF程序的第一页，`is_in_lstpg`指明是否存在于ELF程序的最后一页。

> 这两个变量决定了我们深拷贝的起始地址和终了地址以及拷贝大小，所以要进行分支判断，选择合适的地址和大小传入`memcpy`

2.2.4 测试缺页处理

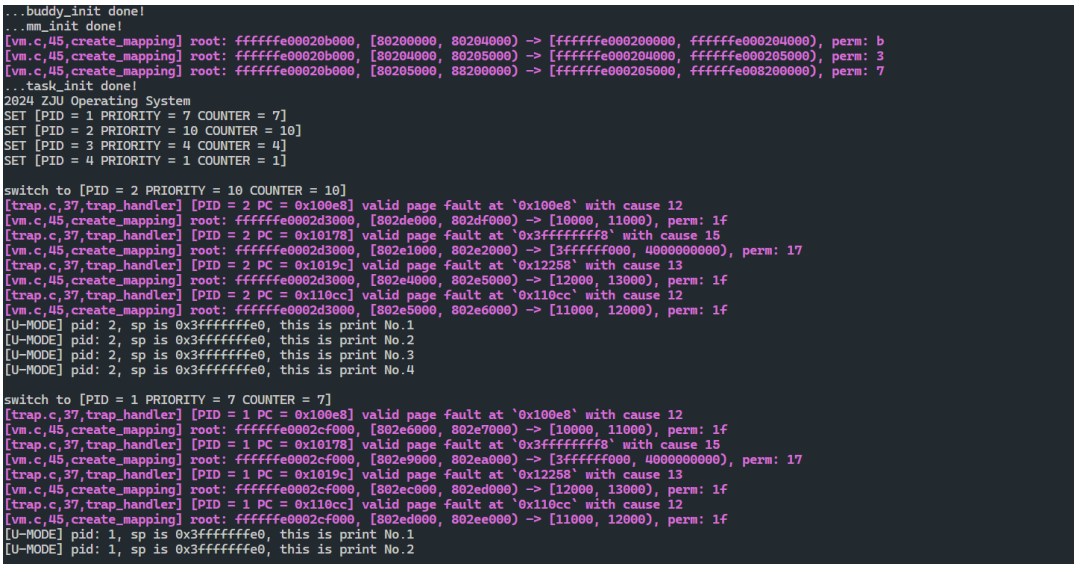
PS:

这里可能会遇到SYS_CLONE未定义的问题，这是由于我们到现在还没有设置处理Fork系统调用的处理函数，这个宏会在后面介绍，这里为了保证运行，直接将

```
#define SYS_CLONE 220
```

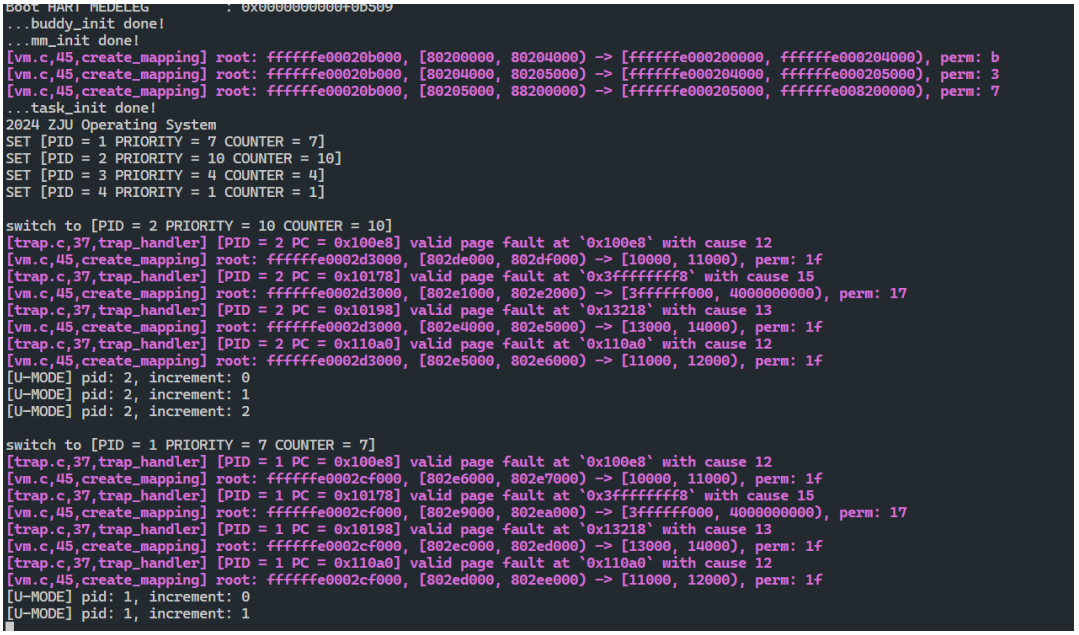
加入到`src/lab5/user/syscall.h`中即可

make run TEST=PFH1



可以看到直到 `task_init` 完成，都只有 `setup_vm_final` 的时候创建了映射，用户态进程的拷贝和映射都在调度之后遇到 page fault 才触发，并且只有第一次触发了：

make run TEST=PFH2



`make run TEST=PFH2` 的效果与前面类似，只不过它通过全局变量空出了一整页大小的未使用 `.data` 区域，这段区域在运行的时候也不会触发 page fault，所以在 log 中应该可以发现 `create_mapping` 映射的虚拟地址空间会缺少一页,即 `[12000,13000)` 区域。

2.3 实现 fork 系统调用

2.3.1 表面的准备工作

在实现较为复杂的 **fork** 流程之前，我们先将框架搭好，具体要做的有以下两件事：

- 修改 **proc** 相关代码，使其只初始化一个进程，其他进程保留为 **NULL** 等待 **fork** 创建

> 我们在`proc.h`中设置全局变量`nr_tasks`(这里初始化为2，包含IDLE和第一个初始化的线程),指明当前进程数，并作为 `task` 的栈顶指针来使用，这样 `task` 就是一个只考虑压栈操作的栈结构了。

> `task_init` 修改
>
> 将NR_TASKS修改为nr_tasks即可

```
void task_init() {

    //modified in Lab5 for `fork`
        // for(int i = 1; i < NR_TASKS; i++) {
        for(int i = 1; i < nr_tasks; i++) {
    //end modified in Lab5 for `fork`
        task[i] = (struct task_struct*)kalloc();
        task[i]->state = TASK_RUNNING;
        task[i]->pid = i;
```

`schedule`

将NR_TASKS修改为nr_tasks即可

```
void schedule() {
    while(1){
        uint64_t max = 0;
        struct task_struct* max_task = NULL;
    //modified in Lab5 for `fork`
        // for(int i = 0; i < NR_TASKS; i++) {
        for(int i = 0; i < nr_tasks; i++) {
    //end modified in Lab5 for `fork`
            if(task[i]->counter > max) {
                max = task[i]->counter;
                max_task = task[i];
            }
        }
        if(max == 0){
            printk("\n");
        //modified in Lab5 for `fork`
            // for(int i = 1; i < NR_TASKS; i++) {
            for(int i = 1; i < nr_tasks; i++) {
        //end modified in Lab5 for `fork`
                task[i]->counter = task[i]->priority;
                printk("SET [PID = %d PRIORITY = %d COUNTER = %d]\n", task[i]->pid, task[i]->priority, task[i]->counter);
            }
            continue;
        } else {
            switch_to(max_task);
            return;
        }
    }
}
```

- 添加系统调用处理

在合适的位置加上下面这句（包括 `user/` 下的 `syscall.h`）。

`#define SYS_CLONE 220`

src > lab5 > user > C syscall.h > SYS_CLONE

```
1   #define SYS_WRITE   64
2
3   #define SYS_GETPID  172
4
5   #define SYS_CLONE 220
```

src > lab5 > arch > riscv > include > C trap.h > SYS_CLONE

```c
#include "stdint.h"
#include "printk.h"
#include "clock.h"
#include "syscall.h"
#include "proc.h"

#define SYS_WRITE   64

#define SYS_GETPID  172

#define SYS_CLONE 220

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

void do_page_fault(struct pt_regs *regs);
```

然后在系统调用的处理函数中，检测到 `regs->a7 == SYS_CLONE` 时，调用 `do_fork` 函数来完成 fork 的工作。

src > lab5 > arch > riscv > kernel > C trap.c > trap_handler(uint64_t, uint64_t, pt_regs *)

```c
void trap_handler(uint64_t scause, uint64_t sepc, struct pt_regs *regs) {
    switch (highest_bit)
    {
        if(scause == 0x8){
            // printk("Environment call from U-mode\n");
            switch (regs->reg17)
            {
            case SYS_WRITE://sys_write
                sys_write(regs->reg10, (const char*)regs->reg11, (uint64_t)regs->reg12, regs);
                break;
            case SYS_GETPID://sys_getpid
                // Log("[PID = %d PC = 0x%lx] sys_getpid",current->pid,regs->sepc);
                sys_getpid(regs);
                break;
            case SYS_CLONE://sys_clone
                // Log("[PID = %d PC = 0x%lx] sys_clone",current->pid,regs->sepc);
                regs->reg10 = do_fork(regs);
                break;
            }
            regs->sepc += 0x4;
        }else if(scause == 12||scause == 13||scause == 15){
            Log("[PID = %d PC = 0x%lx] valid page fault at `0x%llx` with cause %d",current->pid,regs->sepc,regs->stval,scause);
            do_page_fault(regs);
        }else{
            // regs->sepc += 0x4;
            // Err("[S] Unhandled Exception: scause=%d, sepc=%llx", scause, sepc);
        }

    }
```

2.3.2 思考 do_fork 要做什么

在了解了 fork 的原理之后，我们可以梳理一下 fork 的工作：

- 创建一个新进程：
  - 拷贝内核栈（包括了 `task_struct` 等信息）
  - 创建一个新的页表
    - 拷贝内核页表 `swapper_pg_dir`
    - 遍历父进程 vma，并遍历父进程页表
      - 将这个 vma 也添加到新进程的 vma 链表中
      - 如果该 vma 项有对应的页表项存在（说明已经创建了映射），则需要深拷贝一整页的内容并映射到新页表中

- 将新进程加入调度队列
- 处理父子进程的返回值
    - 父进程通过 `do_fork` 函数直接返回子进程的 `pid`，并回到自身运行
    - 子进程通过被调度器调度后（跳到 `thread.ra`），开始执行并返回 `0`

2.3.3 拷贝内核栈

因为内核栈和 `task_struct` 在同一个页的高低地址上，所以我们直接 `memcpy` 深拷贝这个页就可以得到我们需要的所有信息了。

```
src > lab5 > arch > riscv > kernel > C syscall.c > do_fork(pt_regs *)
26  uint64_t do_fork(struct pt_regs *regs){
27      printk("\n");
28      struct task_struct *new_task = (struct task_struct*)alloc_page();
29      task[nr_tasks] = new_task;
30      memcpy((void*)new_task, (void*)current, PGSIZE);
```

但除此之外还要略微修改 `task_struct` 内容，假设新的一页为指针 `_task`，则需要修改：

> 这里的各项参数说明我们放到思考题中

- `_task->pid` 根据 `nr_tasks` 来赋值
- `_task->thread.ra/sp/sscratch` 根据后面的指导赋值
- `_task->pgd` 为新分配的页表地址
- `_task->mm.mmap` 为 `NULL`，因为新进程还没有任何映射
- 为子进程 `pt_regs` 的 `a0` 设置返回值 `0`，为 `sepc` 手动加四。

```
new_task->pid = nr_tasks++;//nr_tasks must be unused in the future
new_task->thread.ra = (uint64_t)&__ret_from_fork;

new_task->thread.sp = (uint64_t)new_task + ((uint64_t)regs - PGROUNDDOWN((uint64_t)regs));
new_task->thread.sscratch = csr_read(sscratch);//same as parent

new_task->mm.mmap = NULL;

struct pt_regs *forked_regs = (struct pt_regs *)(new_task->thread.sp);
forked_regs->reg10 = 0;
forked_regs->sepc += 0x4;
forked_regs->reg2 = new_task->thread.sp;
```

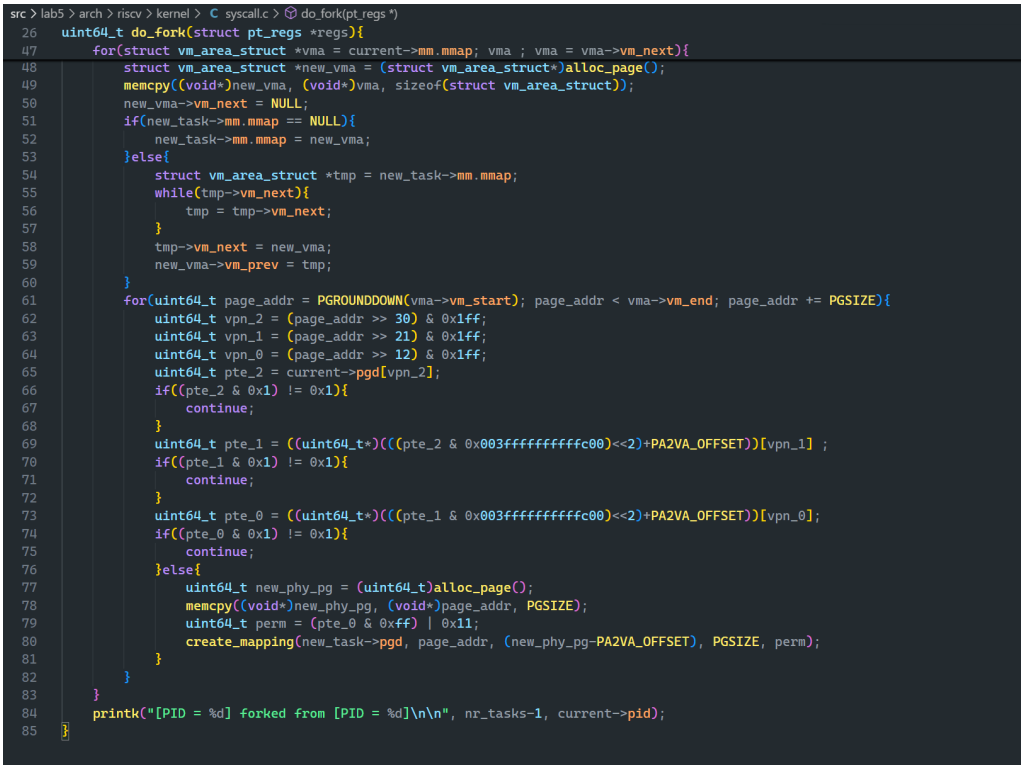
2.3.4 创建子进程页表

根据前面所说，流程为：

- 拷贝内核页表 `swapper_pg_dir`

```
    new_task->pgd = (uint64_t*)alloc_page();
    memcpy((void *)new_task->pgd, (void *)&swapper_pg_dir, PGSIZE);
```

- 遍历父进程 `vma`，并遍历父进程页表
  - 将这个 `vma` 也添加到新进程的 `vma` 链表中
  - 如果该 `vma` 项有对应的页表项存在（说明已经创建了映射），则需要深拷贝一整页的内容并映射到新页表中



```
src > lab5 > arch > riscv > kernel > C syscall.c > do_fork(pt_regs *)
26  uint64_t do_fork(struct pt_regs *regs){
47      for(struct vm_area_struct *vma = current->mm.mmap; vma ; vma = vma->vm_next){
48          struct vm_area_struct *new_vma = (struct vm_area_struct*)alloc_page();
49          memcpy((void*)new_vma, (void*)vma, sizeof(struct vm_area_struct));
50          new_vma->vm_next = NULL;
51          if(new_task->mm.mmap == NULL){
52              new_task->mm.mmap = new_vma;
53          }else{
54              struct vm_area_struct *tmp = new_task->mm.mmap;
55              while(tmp->vm_next){
56                  tmp = tmp->vm_next;
57              }
58              tmp->vm_next = new_vma;
59              new_vma->vm_prev = tmp;
60          }
61          for(uint64_t page_addr = PGROUNDDOWN(vma->vm_start); page_addr < vma->vm_end; page_addr += PGSIZE){
62              uint64_t vpn_2 = (page_addr >> 30) & 0x1ff;
63              uint64_t vpn_1 = (page_addr >> 21) & 0x1ff;
64              uint64_t vpn_0 = (page_addr >> 12) & 0x1ff;
65              uint64_t pte_2 = current->pgd[vpn_2];
66              if((pte_2 & 0x1) != 0x1){
67                  continue;
68              }
69              uint64_t pte_1 = ((uint64_t*)(((pte_2 & 0x003ffffffffffc00)<<2)+PA2VA_OFFSET))[vpn_1] ;
70              if((pte_1 & 0x1) != 0x1){
71                  continue;
72              }
73              uint64_t pte_0 = ((uint64_t*)(((pte_1 & 0x003ffffffffffc00)<<2)+PA2VA_OFFSET))[vpn_0];
74              if((pte_0 & 0x1) != 0x1){
75                  continue;
76              }else{
77                  uint64_t new_phy_pg = (uint64_t)alloc_page();
78                  memcpy((void*)new_phy_pg, (void*)page_addr, PGSIZE);
79                  uint64_t perm = (pte_0 & 0xff) | 0x11;
80                  create_mapping(new_task->pgd, page_addr, (new_phy_pg-PA2VA_OFFSET), PGSIZE, perm);
81              }
82          }
83      }
84      printk("[PID = %d] forked from [PID = %d]\n\n", nr_tasks-1, current->pid);
85  }
```

完整的`do_fork`：

> 别忘了外部引用：

```
extern struct task_struct *current;
extern struct task_struct *task[];
extern uint64_t nr_tasks;
extern uint64_t* swapper_pg_dir;
extern void __ret_from_fork();
```

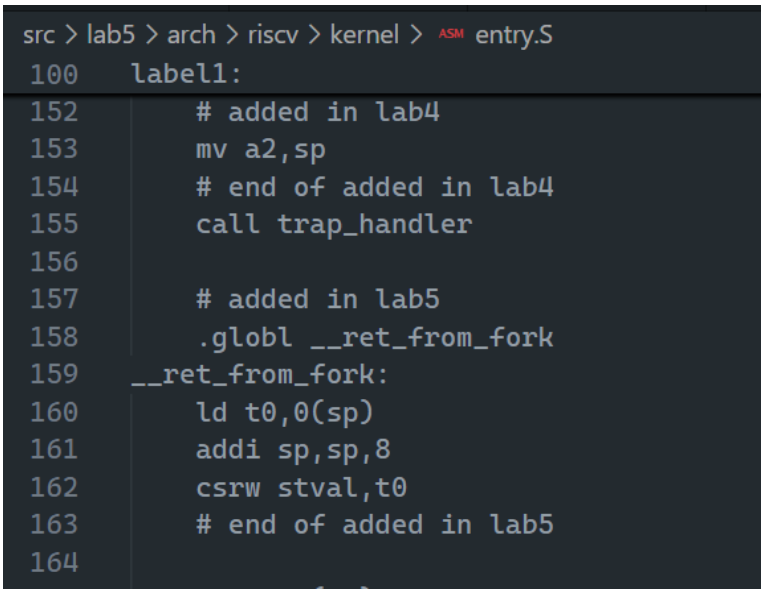
```
uint64_t do_fork(struct pt_regs *regs){
    printk("\n");
    struct task_struct *new_task = (struct task_struct*)alloc_page();
    task[nr_tasks] = new_task;
```

```
    memcpy((void*)new_task, (void*)current, PGSIZE);
    new_task->pid = nr_tasks++;//nr_tasks must be unused in the future
    new_task->thread.ra = (uint64_t)&__ret_from_fork;

    new_task->thread.sp = (uint64_t)new_task + ((uint64_t)regs - PGROUNDDOWN((uint64_t)regs));
    new_task->thread.sscratch = csr_read(sscratch);//same as parent


    new_task->mm.mmap = NULL;

    struct pt_regs *forked_regs = (struct pt_regs *)(new_task->thread.sp);
    forked_regs->reg10 = 0;
    forked_regs->sepc += 0x4;
    forked_regs->reg2 = new_task->thread.sp;

    new_task->pgd = (uint64_t*)alloc_page();
    memcpy((void *)new_task->pgd, (void *)&swapper_pg_dir, PGSIZE);


    for(struct vm_area_struct *vma = current->mm.mmap; vma ; vma = vma->vm_next){
        struct vm_area_struct *new_vma = (struct vm_area_struct*)alloc_page();
        memcpy((void*)new_vma, (void*)vma, sizeof(struct vm_area_struct));
        new_vma->vm_next = NULL;
        if(new_task->mm.mmap == NULL){
            new_task->mm.mmap = new_vma;
        }else{
            struct vm_area_struct *tmp = new_task->mm.mmap;
            while(tmp->vm_next){
                tmp = tmp->vm_next;
            }
```

```
            tmp->vm_next = new_vma;
            new_vma->vm_prev = tmp;
        }
        for(uint64_t page_addr = PGROUNDDOWN(vma->vm_start); page_addr < vma->vm_end; page_addr += PGSIZE){
            uint64_t vpn_2 = (page_addr >> 30) & 0x1ff;
            uint64_t vpn_1 = (page_addr >> 21) & 0x1ff;
            uint64_t vpn_0 = (page_addr >> 12) & 0x1ff;
            uint64_t pte_2 = current->pgd[vpn_2];
            if((pte_2 & 0x1) != 0x1){
                continue;
            }
            uint64_t pte_1 = ((uint64_t*)(((pte_2 & 0x003ffffffffffc00)<<2)+PA2VA_OFFSET))[vpn_1] ;
            if((pte_1 & 0x1) != 0x1){
                continue;
            }
            uint64_t pte_0 = ((uint64_t*)(((pte_1 & 0x003ffffffffffc00)<<2)+PA2VA_OFFSET))[vpn_0];
            if((pte_0 & 0x1) != 0x1){
                continue;
            }else{
                uint64_t new_phy_pg = (uint64_t)alloc_page();
                memcpy((void*)new_phy_pg, (void*)page_addr, PGSIZE);
                uint64_t perm = (pte_0 & 0xff) | 0x11;
                create_mapping(new_task->pgd, page_addr, (new_phy_pg-PA2VA_OFFSET), PGSIZE, perm);
            }
        }
    }
    printk("[PID = %d] forked from [PID = %d]\n\n", nr_tasks-1, current->pid);
}
```

2.3.5 处理进程返回逻辑

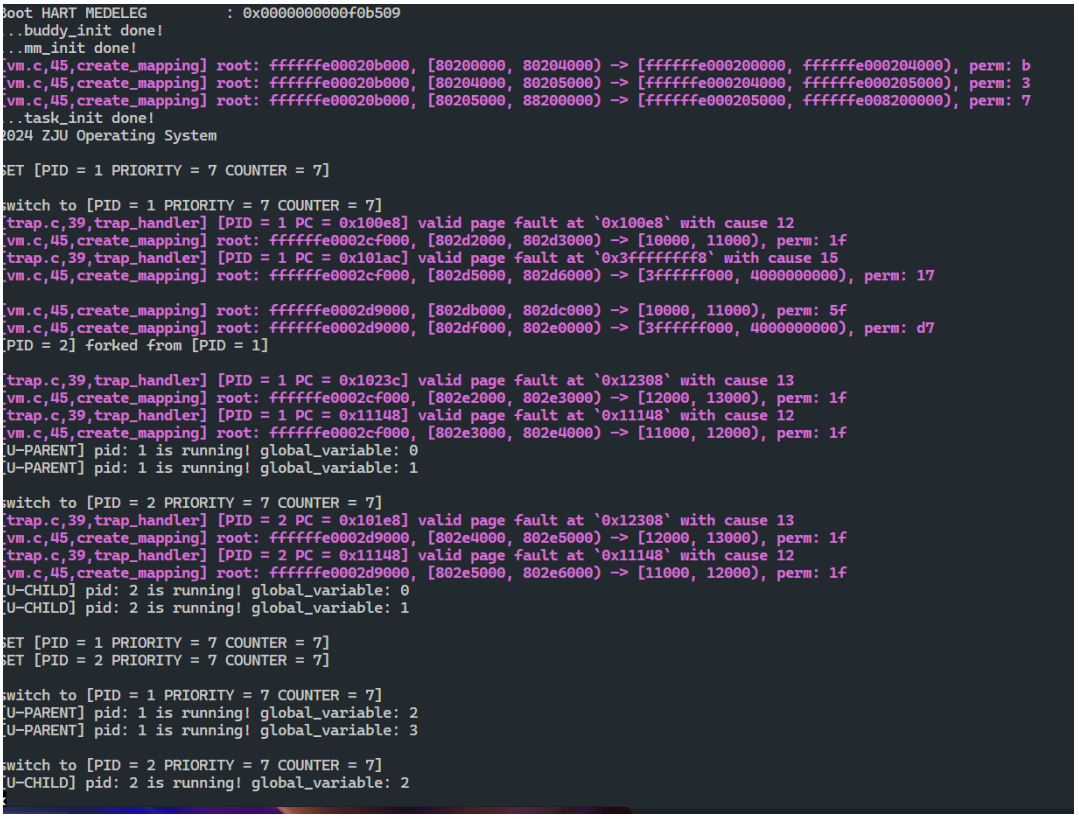
利用 `__switch_to` 时恢复的 `ra` 与 `sp`，我们可以直接跳转到 `_traps` 中从 `trap_handler` 返回的位置，只需要加一个标号：



这样我们的 `_task->thread.ra` 就很显然是 `__ret_from_fork` 的地址了。

2.3.6 测试 fork

```
make run TEST=FORK1
```



可以看到 PID 1 在 fork 出 PID 2 时将现有 `create_mapping` 过的两个页拷贝并在子进程的页表中创建了映射，然后调度后 PID 2 开始运行，而且 `global_variable` 的值互不影响，后续 page fault 也是各自为自己的页表添加映射。

```
make run TEST=FORK2
```

```
...buddy_init done!
...mm_init done!
[vm.c,45,create_mapping] root: ffffffe00020b000, [80200000, 80204000) -> [ffffffe000200000, ffffffe000204000), perm: b
[vm.c,45,create_mapping] root: ffffffe00020b000, [80204000, 80205000) -> [ffffffe000204000, ffffffe000205000), perm: 3
[vm.c,45,create_mapping] root: ffffffe00020b000, [80205000, 88200000) -> [ffffffe000205000, ffffffe008200000), perm: 7
...task_init done!
2024 ZJU Operating System

SET [PID = 1 PRIORITY = 7 COUNTER = 7]

switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[trap.c,39,trap_handler] [PID = 1 PC = 0x100e8] valid page fault at `0x100e8` with cause 12
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802d2000, 802d3000) -> [10000, 11000), perm: 1f
[trap.c,39,trap_handler] [PID = 1 PC = 0x101ac] valid page fault at `0x3fffffff8` with cause 15
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802d5000, 802d6000) -> [3ffffff000, 4000000000), perm: 17
[trap.c,39,trap_handler] [PID = 1 PC = 0x101d4] valid page fault at `0x12568` with cause 13
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802d8000, 802d9000) -> [12000, 13000), perm: 1f
[trap.c,39,trap_handler] [PID = 1 PC = 0x11318] valid page fault at `0x11318` with cause 12
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802d9000, 802da000) -> [11000, 12000), perm: 1f
[trap.c,39,trap_handler] [PID = 1 PC = 0x10468] valid page fault at `0x14570` with cause 13
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802da000, 802db000) -> [14000, 15000), perm: 1f
[U] pid: 1 is running! global_variable: 0
[U] pid: 1 is running! global_variable: 1
[U] pid: 1 is running! global_variable: 2
[trap.c,39,trap_handler] [PID = 1 PC = 0x10230] valid page fault at `0x13570` with cause 15
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802db000, 802dc000) -> [13000, 14000), perm: 1f

[vm.c,45,create_mapping] root: ffffffe0002dd000, [802df000, 802e0000) -> [10000, 11000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe0002dd000, [802e2000, 802e3000) -> [11000, 12000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe0002dd000, [802e3000, 802e4000) -> [12000, 13000), perm: df
[vm.c,45,create_mapping] root: ffffffe0002dd000, [802e4000, 802e5000) -> [13000, 14000), perm: df
[vm.c,45,create_mapping] root: ffffffe0002dd000, [802e5000, 802e6000) -> [14000, 15000), perm: df
[vm.c,45,create_mapping] root: ffffffe0002dd000, [802e7000, 802e8000) -> [3ffffff000, 4000000000), perm: d7
[PID = 2] forked from [PID = 1]

[U-PARENT] pid: 1 is running! Message: ZJU OS Lab5
[U-PARENT] pid: 1 is running! global_variable: 3
[U-PARENT] pid: 1 is running! global_variable: 4

switch to [PID = 2 PRIORITY = 7 COUNTER = 7]
[U-CHILD] pid: 2 is running! Message: ZJU OS Lab5
[U-CHILD] pid: 2 is running! global_variable: 3
[U-CHILD] pid: 2 is running! global_variable: 4

SET [PID = 1 PRIORITY = 7 COUNTER = 7]
SET [PID = 2 PRIORITY = 7 COUNTER = 7]

switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[U-PARENT] pid: 1 is running! global_variable: 5
[U-PARENT] pid: 1 is running! global_variable: 6

switch to [PID = 2 PRIORITY = 7 COUNTER = 7]
[U-CHILD] pid: 2 is running! global_variable: 5
QEMU: Terminated
zaxin0401@LAPTOP-0P208VUK:/mnt/d/cs/os24fall-stu/src/lab5$
```

本测试的主要输出现象为，父进程在给 `global_variable` 自增了三次，为 `placeholder` 中赋值了字符串之后才 fork 出子进程，子进程应该要通过深拷贝页表来保留这些信息。PID 2 开始运行时也应该正确输出 `ZJU OS Lab5` 字符串，并且 `global_variable` 从 3 开始自增，且后续和父进程互不影响。

```
make run TEST=FORK3
```

```
...task_init done!
2024 ZJU Operating System

SET [PID = 1 PRIORITY = 7 COUNTER = 7]

switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[trap.c,39,trap_handler] [PID = 1 PC = 0x100e8] valid page fault at `0x100e8` with cause 12
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802d2000, 802d3000) -> [10000, 11000), perm: 1f
[trap.c,39,trap_handler] [PID = 1 PC = 0x101ac] valid page fault at `0x3ffffffff8` with cause 15
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802d5000, 802d6000) -> [3ffffff000, 4000000000), perm: 17
[trap.c,39,trap_handler] [PID = 1 PC = 0x101cc] valid page fault at `0x122f0` with cause 13
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802d8000, 802d9000) -> [12000, 13000), perm: 1f
[trap.c,39,trap_handler] [PID = 1 PC = 0x11170] valid page fault at `0x11170` with cause 12
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802d9000, 802da000) -> [11000, 12000), perm: 1f
[U] pid: 1 is running! global_variable: 0

[vm.c,45,create_mapping] root: ffffffe0002db000, [802dd000, 802de000) -> [10000, 11000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe0002db000, [802e0000, 802e1000) -> [11000, 12000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe0002db000, [802e1000, 802e2000) -> [12000, 13000), perm: df
[vm.c,45,create_mapping] root: ffffffe0002db000, [802e3000, 802e4000) -> [3ffffff000, 4000000000), perm: d7
[PID = 2] forked from [PID = 1]


[vm.c,45,create_mapping] root: ffffffe0002e7000, [802e9000, 802ea000) -> [10000, 11000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe0002e7000, [802ec000, 802ed000) -> [11000, 12000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe0002e7000, [802ed000, 802ee000) -> [12000, 13000), perm: df
[vm.c,45,create_mapping] root: ffffffe0002e7000, [802ef000, 802f0000) -> [3ffffff000, 4000000000), perm: d7
[PID = 3] forked from [PID = 1]

[U] pid: 1 is running! global_variable: 1

[vm.c,45,create_mapping] root: ffffffe0002f3000, [802f5000, 802f6000) -> [10000, 11000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe0002f3000, [802f8000, 802f9000) -> [11000, 12000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe0002f3000, [802f9000, 802fa000) -> [12000, 13000), perm: df
[vm.c,45,create_mapping] root: ffffffe0002f3000, [802fb000, 802fc000) -> [3ffffff000, 4000000000), perm: d7
[PID = 4] forked from [PID = 1]

[U] pid: 1 is running! global_variable: 2
[U] pid: 1 is running! global_variable: 3

switch to [PID = 2 PRIORITY = 7 COUNTER = 7]

[vm.c,45,create_mapping] root: ffffffe0002ff000, [80301000, 80302000) -> [10000, 11000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe0002ff000, [80304000, 80305000) -> [11000, 12000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe0002ff000, [80305000, 80306000) -> [12000, 13000), perm: df
[vm.c,45,create_mapping] root: ffffffe0002ff000, [80307000, 80308000) -> [3ffffff000, 4000000000), perm: d7
[PID = 5] forked from [PID = 2]

[U] pid: 2 is running! global_variable: 1

[vm.c,45,create_mapping] root: ffffffe00030b000, [8030d000, 8030e000) -> [10000, 11000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe00030b000, [80310000, 80311000) -> [11000, 12000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe00030b000, [80311000, 80312000) -> [12000, 13000), perm: df
[vm.c,45,create_mapping] root: ffffffe00030b000, [80313000, 80314000) -> [3ffffff000, 4000000000), perm: d7
[PID = 6] forked from [PID = 2]
```

```
[U] pid: 2 is running! global_variable: 1

[vm.c,45,create_mapping] root: ffffffe00030b000, [8030d000, 8030e000) -> [10000, 11000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe00030b000, [80310000, 80311000) -> [11000, 12000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe00030b000, [80311000, 80312000) -> [12000, 13000), perm: df
[vm.c,45,create_mapping] root: ffffffe00030b000, [80313000, 80314000) -> [3ffffff000, 4000000000), perm: d7
[PID = 6] forked from [PID = 2]

[U] pid: 2 is running! global_variable: 2
[U] pid: 2 is running! global_variable: 3

switch to [PID = 3 PRIORITY = 7 COUNTER = 7]
[U] pid: 3 is running! global_variable: 1

[vm.c,45,create_mapping] root: ffffffe000317000, [80319000, 8031a000) -> [10000, 11000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe000317000, [8031c000, 8031d000) -> [11000, 12000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe000317000, [8031d000, 8031e000) -> [12000, 13000), perm: df
[vm.c,45,create_mapping] root: ffffffe000317000, [8031f000, 80320000) -> [3ffffff000, 4000000000), perm: d7
[PID = 7] forked from [PID = 3]

[U] pid: 3 is running! global_variable: 2
[U] pid: 3 is running! global_variable: 3

switch to [PID = 4 PRIORITY = 7 COUNTER = 7]
[U] pid: 4 is running! global_variable: 2
[U] pid: 4 is running! global_variable: 3

switch to [PID = 5 PRIORITY = 7 COUNTER = 7]
[U] pid: 5 is running! global_variable: 1

[vm.c,45,create_mapping] root: ffffffe000323000, [80325000, 80326000) -> [10000, 11000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe000323000, [80328000, 80329000) -> [11000, 12000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe000323000, [80329000, 8032a000) -> [12000, 13000), perm: df
[vm.c,45,create_mapping] root: ffffffe000323000, [8032b000, 8032c000) -> [3ffffff000, 4000000000), perm: d7
[PID = 8] forked from [PID = 5]

[U] pid: 5 is running! global_variable: 2
[U] pid: 5 is running! global_variable: 3

switch to [PID = 6 PRIORITY = 7 COUNTER = 7]
[U] pid: 6 is running! global_variable: 2
[U] pid: 6 is running! global_variable: 3

switch to [PID = 7 PRIORITY = 7 COUNTER = 7]
[U] pid: 7 is running! global_variable: 2
[U] pid: 7 is running! global_variable: 3

switch to [PID = 8 PRIORITY = 7 COUNTER = 7]
[U] pid: 8 is running! global_variable: 2
[U] pid: 8 is running! global_variable: 3
```

结合main.c的代码解释：

```c
int global_variable = 0;

int main() {

 printf("[U] pid: %ld is running! global_variable: %d\n", getpid(), global_variable++);
 fork();
 fork();

 printf("[U] pid: %ld is running! global_variable: %d\n", getpid(), global_variable++);
 fork();

 while(1) {
```

```
13      printf("[U] pid: %ld is running! global_variable: %d\n",
    getpid(), global_variable++);
14      wait(WAIT_TIME);
15  }
16 }
```

为了表述方便，我们将父进程pid设为1，其余fork的进程名字即为其pid号

1. 父进程(pid = 1)自增`global_variable`

   > [U] pid: 1 is running! global_variable: 0

   1. 开始fork，创建了新进程2：

      > [PID = 2] forked from [PID = 1]

   2. 由于现在的进程仍为进程1，继续fork：

      > [PID = 3] forked from [PID = 1]

   3. 进程1自增`global_variable`：

      > [U] pid: 1 is running! global_variable: 1

   4. 由于现在的进程仍为进程1，继续fork：

      > [PID = 4] forked from [PID = 1]

   5. 进程1自增`global_variable`：

      > [U] pid: 1 is running! global_variable: 2
      > [U] pid: 1 is running! global_variable: 3

2. 切换到进程2

   1. 现在处于的位置是进行第二次fork的位置：

      > [PID = 5] forked from [PID = 2]

   2. 进程2自增`global_variable`：

      由于父进程是1，fork进程2的时候`global_variable == 1`，因此在这里输出1

      > [U] pid: 2 is running! global_variable: 1

3. 由于现在的进程仍为进程2，继续fork：

   > [PID = 6] forked from [PID = 1]

4. 进程2自增`global_variable`：

   > [U] pid: 2 is running! global_variable: 2
   >
   > [U] pid: 2 is running! global_variable: 3

3. 切换到线程3

   其变化与进程2保持一致，只是fork的线程是7：

   > [PID = 7] forked from [PID = 3]

4. 切换到线程4

   由于没有fork操作，只是输出自增`global_variable`：

   由于父进程是1，fork进程4的时候`global_variable == 2`，因此在这里输出2

   > switch to [PID = 4 PRIORITY = 7 COUNTER = 7]
   > [U] pid: 4 is running! global_variable: 2
   > [U] pid: 4 is running! global_variable: 3

5. 切换到线程5

   1. 进程5自增`global_variable`：

      由于父进程是2，fork进程5的时候`global_variable == 1`，因此在这里输出1

      > [U] pid: 5 is running! global_variable: 1

   2. 由于现在的进程仍为进程5，继续fork：

      > [PID = 8] forked from [PID = 5]

   3. 进程5自增`global_variable`：

      > [U] pid: 5 is running! global_variable: 2
      > [U] pid: 5 is running! global_variable: 3

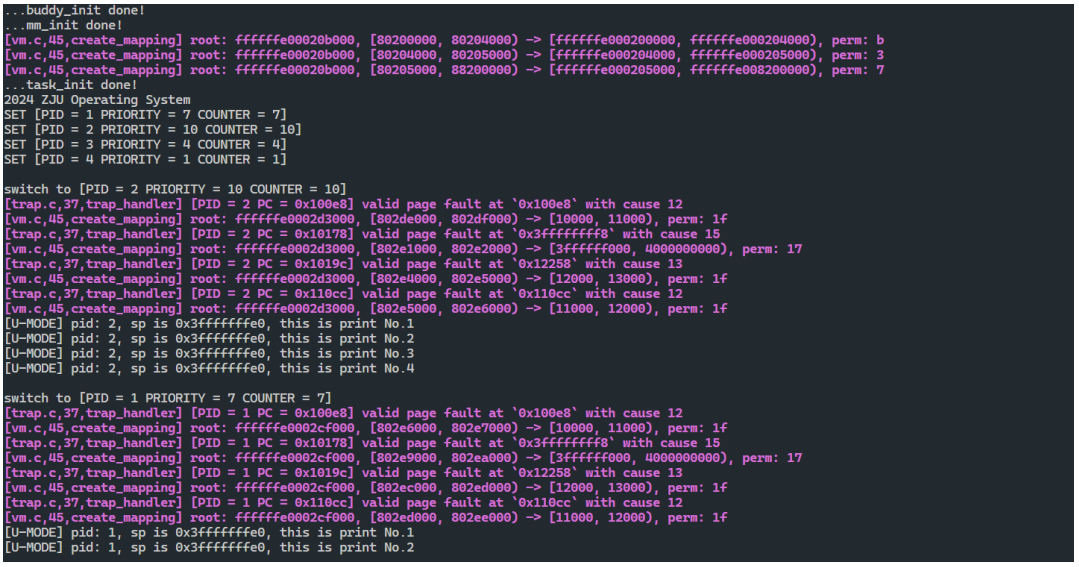
6. 切换到线程6、7、8

   状态与线程4变化保持一致，只是单纯地输出自增变量

```
switch to [PID = 6 PRIORITY = 7 COUNTER = 7]
[U] pid: 6 is running! global_variable: 2
[U] pid: 6 is running! global_variable: 3

switch to [PID = 7 PRIORITY = 7 COUNTER = 7]
[U] pid: 7 is running! global_variable: 2

.....
```

3 实验结果与分析

```
make run TEST=PFH1
```



可以看到直到 `task_init` 完成，都只有 `setup_vm_final` 的时候创建了映射，用户态进程的拷贝和映射都在调度之后遇到 page fault 才触发，并且只有第一次触发了：

```
make run TEST=PFH2
```

```
Boot HART MEDELEG         : 0x0000000000f0b509
...buddy_init done!
...mm_init done!
[vm.c,45,create_mapping] root: ffffffe00020b000, [80200000, 80204000) -> [ffffffe000200000, ffffffe000204000), perm: b
[vm.c,45,create_mapping] root: ffffffe00020b000, [80204000, 80205000) -> [ffffffe000204000, ffffffe000205000), perm: 3
[vm.c,45,create_mapping] root: ffffffe00020b000, [80205000, 88200000) -> [ffffffe000205000, ffffffe008200000), perm: 7
...task_init done!
2024 ZJU Operating System
SET [PID = 1 PRIORITY = 7 COUNTER = 7]
SET [PID = 2 PRIORITY = 10 COUNTER = 10]
SET [PID = 3 PRIORITY = 4 COUNTER = 4]
SET [PID = 4 PRIORITY = 1 COUNTER = 1]

switch to [PID = 2 PRIORITY = 10 COUNTER = 10]
[trap.c,37,trap_handler] [PID = 2 PC = 0x100e8] valid page fault at `0x100e8` with cause 12
[vm.c,45,create_mapping] root: ffffffe0002d3000, [802de000, 802df000) -> [10000, 11000), perm: 1f
[trap.c,37,trap_handler] [PID = 2 PC = 0x10178] valid page fault at `0x3ffffffff8` with cause 15
[vm.c,45,create_mapping] root: ffffffe0002d3000, [802e1000, 802e2000) -> [3ffffff000, 4000000000), perm: 17
[trap.c,37,trap_handler] [PID = 2 PC = 0x10198] valid page fault at `0x13218` with cause 13
[vm.c,45,create_mapping] root: ffffffe0002d3000, [802e4000, 802e5000) -> [13000, 14000), perm: 1f
[trap.c,37,trap_handler] [PID = 2 PC = 0x110a0] valid page fault at `0x110a0` with cause 12
[vm.c,45,create_mapping] root: ffffffe0002d3000, [802e5000, 802e6000) -> [11000, 12000), perm: 1f
[U-MODE] pid: 2, increment: 0
[U-MODE] pid: 2, increment: 1
[U-MODE] pid: 2, increment: 2

switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[trap.c,37,trap_handler] [PID = 1 PC = 0x100e8] valid page fault at `0x100e8` with cause 12
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802e6000, 802e7000) -> [10000, 11000), perm: 1f
[trap.c,37,trap_handler] [PID = 1 PC = 0x10178] valid page fault at `0x3ffffffff8` with cause 15
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802e9000, 802ea000) -> [3ffffff000, 4000000000), perm: 17
[trap.c,37,trap_handler] [PID = 1 PC = 0x10198] valid page fault at `0x13218` with cause 13
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802ec000, 802ed000) -> [13000, 14000), perm: 1f
[trap.c,37,trap_handler] [PID = 1 PC = 0x110a0] valid page fault at `0x110a0` with cause 12
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802ed000, 802ee000) -> [11000, 12000), perm: 1f
[U-MODE] pid: 1, increment: 0
[U-MODE] pid: 1, increment: 1
```

`make run TEST=PFH2` 的效果与前面类似，只不过它通过全局变量空出了一整页大小的未使用 `.data` 区域，这段区域在运行的时候也不会触发 page fault，所以在 log 中应该可以发现 `create_mapping` 映射的虚拟地址空间会缺少一页,即 `[12000,13000)` 区域。

make run TEST=FORK1

```
Boot HART MEDELEG         : 0x0000000000f0b509
...buddy_init done!
...mm_init done!
[vm.c,45,create_mapping] root: ffffffe00020b000, [80200000, 80204000) -> [ffffffe000200000, ffffffe000204000), perm: b
[vm.c,45,create_mapping] root: ffffffe00020b000, [80204000, 80205000) -> [ffffffe000204000, ffffffe000205000), perm: 3
[vm.c,45,create_mapping] root: ffffffe00020b000, [80205000, 88200000) -> [ffffffe000205000, ffffffe008200000), perm: 7
...task_init done!
2024 ZJU Operating System

SET [PID = 1 PRIORITY = 7 COUNTER = 7]

switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[trap.c,39,trap_handler] [PID = 1 PC = 0x100e8] valid page fault at `0x100e8` with cause 12
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802d2000, 802d3000) -> [10000, 11000), perm: 1f
[trap.c,39,trap_handler] [PID = 1 PC = 0x101ac] valid page fault at `0x3ffffffff8` with cause 15
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802d5000, 802d6000) -> [3ffffff000, 4000000000), perm: 17

[vm.c,45,create_mapping] root: ffffffe0002d9000, [802db000, 802dc000) -> [10000, 11000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe0002d9000, [802df000, 802e0000) -> [3ffffff000, 4000000000), perm: d7
[PID = 2] forked from [PID = 1]

[trap.c,39,trap_handler] [PID = 1 PC = 0x1023c] valid page fault at `0x12308` with cause 13
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802e2000, 802e3000) -> [12000, 13000), perm: 1f
[trap.c,39,trap_handler] [PID = 1 PC = 0x11148] valid page fault at `0x11148` with cause 12
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802e3000, 802e4000) -> [11000, 12000), perm: 1f
[U-PARENT] pid: 1 is running! global_variable: 0
[U-PARENT] pid: 1 is running! global_variable: 1

switch to [PID = 2 PRIORITY = 7 COUNTER = 7]
[trap.c,39,trap_handler] [PID = 2 PC = 0x101e8] valid page fault at `0x12308` with cause 13
[vm.c,45,create_mapping] root: ffffffe0002d9000, [802e4000, 802e5000) -> [12000, 13000), perm: 1f
[trap.c,39,trap_handler] [PID = 2 PC = 0x11148] valid page fault at `0x11148` with cause 12
[vm.c,45,create_mapping] root: ffffffe0002d9000, [802e5000, 802e6000) -> [11000, 12000), perm: 1f
[U-CHILD] pid: 2 is running! global_variable: 0
[U-CHILD] pid: 2 is running! global_variable: 1

SET [PID = 1 PRIORITY = 7 COUNTER = 7]
SET [PID = 2 PRIORITY = 7 COUNTER = 7]

switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[U-PARENT] pid: 1 is running! global_variable: 2
[U-PARENT] pid: 1 is running! global_variable: 3

switch to [PID = 2 PRIORITY = 7 COUNTER = 7]
[U-CHILD] pid: 2 is running! global_variable: 2
```

可以看到 PID 1 在 fork 出 PID 2 时将现有 `create_mapping` 过的两个页拷贝并在子进程的页表中创建了映射，然后调度后 PID 2 开始运行，而且 `global_variable` 的值互不影响，后续 page fault 也是各自为自己的页表添加映射。

make run TEST=FORK2

```
...buddy_init done!
...mm_init done!
[vm.c,45,create_mapping] root: ffffffe00020b000, [80200000, 80204000) -> [ffffffe000200000, ffffffe000204000), perm: b
[vm.c,45,create_mapping] root: ffffffe00020b000, [80204000, 80205000) -> [ffffffe000204000, ffffffe000205000), perm: 3
[vm.c,45,create_mapping] root: ffffffe00020b000, [80205000, 88200000) -> [ffffffe000205000, ffffffe008200000), perm: 7
...task_init done!
2024 ZJU Operating System

SET [PID = 1 PRIORITY = 7 COUNTER = 7]

switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[trap.c,39,trap_handler] [PID = 1 PC = 0x100e8] valid page fault at `0x100e8` with cause 12
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802d2000, 802d3000) -> [10000, 11000), perm: 1f
[trap.c,39,trap_handler] [PID = 1 PC = 0x101ac] valid page fault at `0x3fffffff8` with cause 15
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802d5000, 802d6000) -> [3fffffff000, 4000000000), perm: 17
[trap.c,39,trap_handler] [PID = 1 PC = 0x101d4] valid page fault at `0x12568` with cause 13
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802d8000, 802d9000) -> [12000, 13000), perm: 1f
[trap.c,39,trap_handler] [PID = 1 PC = 0x11318] valid page fault at `0x11318` with cause 12
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802d9000, 802da000) -> [11000, 12000), perm: 1f
[trap.c,39,trap_handler] [PID = 1 PC = 0x10468] valid page fault at `0x14570` with cause 13
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802da000, 802db000) -> [14000, 15000), perm: 1f
[U] pid: 1 is running! global_variable: 0
[U] pid: 1 is running! global_variable: 1
[U] pid: 1 is running! global_variable: 2
[trap.c,39,trap_handler] [PID = 1 PC = 0x10230] valid page fault at `0x13570` with cause 15
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802db000, 802dc000) -> [13000, 14000), perm: 1f

[vm.c,45,create_mapping] root: ffffffe0002dd000, [802df000, 802e0000) -> [10000, 11000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe0002dd000, [802e2000, 802e3000) -> [11000, 12000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe0002dd000, [802e3000, 802e4000) -> [12000, 13000), perm: df
[vm.c,45,create_mapping] root: ffffffe0002dd000, [802e4000, 802e5000) -> [13000, 14000), perm: df
[vm.c,45,create_mapping] root: ffffffe0002dd000, [802e5000, 802e6000) -> [14000, 15000), perm: df
[vm.c,45,create_mapping] root: ffffffe0002dd000, [802e7000, 802e8000) -> [3fffffff000, 4000000000), perm: d7
[PID = 2] forked from [PID = 1]

[U-PARENT] pid: 1 is running! Message: ZJU OS Lab5
[U-PARENT] pid: 1 is running! global_variable: 3
[U-PARENT] pid: 1 is running! global_variable: 4

switch to [PID = 2 PRIORITY = 7 COUNTER = 7]
[U-CHILD] pid: 2 is running! Message: ZJU OS Lab5
[U-CHILD] pid: 2 is running! global_variable: 3
[U-CHILD] pid: 2 is running! global_variable: 4

SET [PID = 1 PRIORITY = 7 COUNTER = 7]
SET [PID = 2 PRIORITY = 7 COUNTER = 7]

switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[U-PARENT] pid: 1 is running! global_variable: 5
[U-PARENT] pid: 1 is running! global_variable: 6

switch to [PID = 2 PRIORITY = 7 COUNTER = 7]
[U-CHILD] pid: 2 is running! global_variable: 5
QEMU: Terminated
zaxin0401@LAPTOP-0P208VUK:/mnt/d/cs/os24fall-stu/src/lab5$
```

本测试的主要输出现象为，父进程在给 `global_variable` 自增了三次，为 `placeholder` 中赋值了字符串之后才 fork 出子进程，子进程应该要通过深拷贝页表来保留这些信息。PID 2 开始运行时也应该正确输出 `ZJU OS Lab5` 字符串，并且 `global_variable` 从 3 开始自增，且后续和父进程互不影响。

make run TEST=FORK3

```
...task_init done!
2024 ZJU Operating System

SET [PID = 1 PRIORITY = 7 COUNTER = 7]

switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[trap.c,39,trap_handler] [PID = 1 PC = 0x100e8] valid page fault at `0x100e8` with cause 12
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802d2000, 802d3000) -> [10000, 11000), perm: 1f
[trap.c,39,trap_handler] [PID = 1 PC = 0x101ac] valid page fault at `0x3ffffffff8` with cause 15
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802d5000, 802d6000) -> [3ffffff000, 4000000000), perm: 17
[trap.c,39,trap_handler] [PID = 1 PC = 0x101cc] valid page fault at `0x122f0` with cause 13
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802d8000, 802d9000) -> [12000, 13000), perm: 1f
[trap.c,39,trap_handler] [PID = 1 PC = 0x11170] valid page fault at `0x11170` with cause 12
[vm.c,45,create_mapping] root: ffffffe0002cf000, [802d9000, 802da000) -> [11000, 12000), perm: 1f
[U] pid: 1 is running! global_variable: 0

[vm.c,45,create_mapping] root: ffffffe0002db000, [802dd000, 802de000) -> [10000, 11000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe0002db000, [802e0000, 802e1000) -> [11000, 12000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe0002db000, [802e1000, 802e2000) -> [12000, 13000), perm: df
[vm.c,45,create_mapping] root: ffffffe0002db000, [802e3000, 802e4000) -> [3ffffff000, 4000000000), perm: d7
[PID = 2] forked from [PID = 1]


[vm.c,45,create_mapping] root: ffffffe0002e7000, [802e9000, 802ea000) -> [10000, 11000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe0002e7000, [802ec000, 802ed000) -> [11000, 12000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe0002e7000, [802ed000, 802ee000) -> [12000, 13000), perm: df
[vm.c,45,create_mapping] root: ffffffe0002e7000, [802ef000, 802f0000) -> [3ffffff000, 4000000000), perm: d7
[PID = 3] forked from [PID = 1]

[U] pid: 1 is running! global_variable: 1

[vm.c,45,create_mapping] root: ffffffe0002f3000, [802f5000, 802f6000) -> [10000, 11000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe0002f3000, [802f8000, 802f9000) -> [11000, 12000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe0002f3000, [802f9000, 802fa000) -> [12000, 13000), perm: df
[vm.c,45,create_mapping] root: ffffffe0002f3000, [802fb000, 802fc000) -> [3ffffff000, 4000000000), perm: d7
[PID = 4] forked from [PID = 1]

[U] pid: 1 is running! global_variable: 2
[U] pid: 1 is running! global_variable: 3

switch to [PID = 2 PRIORITY = 7 COUNTER = 7]

[vm.c,45,create_mapping] root: ffffffe0002ff000, [80301000, 80302000) -> [10000, 11000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe0002ff000, [80304000, 80305000) -> [11000, 12000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe0002ff000, [80305000, 80306000) -> [12000, 13000), perm: df
[vm.c,45,create_mapping] root: ffffffe0002ff000, [80307000, 80308000) -> [3ffffff000, 4000000000), perm: d7
[PID = 5] forked from [PID = 2]

[U] pid: 2 is running! global_variable: 1

[vm.c,45,create_mapping] root: ffffffe00030b000, [8030d000, 8030e000) -> [10000, 11000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe00030b000, [80310000, 80311000) -> [11000, 12000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe00030b000, [80311000, 80312000) -> [12000, 13000), perm: df
[vm.c,45,create_mapping] root: ffffffe00030b000, [80313000, 80314000) -> [3ffffff000, 4000000000), perm: d7
[PID = 6] forked from [PID = 2]
```

```
[U] pid: 2 is running! global_variable: 1

[vm.c,45,create_mapping] root: ffffffe00030b000, [8030d000, 8030e000) -> [10000, 11000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe00030b000, [80310000, 80311000) -> [11000, 12000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe00030b000, [80311000, 80312000) -> [12000, 13000), perm: df
[vm.c,45,create_mapping] root: ffffffe00030b000, [80313000, 80314000) -> [3ffffff000, 4000000000), perm: d7
[PID = 6] forked from [PID = 2]

[U] pid: 2 is running! global_variable: 2
[U] pid: 2 is running! global_variable: 3

switch to [PID = 3 PRIORITY = 7 COUNTER = 7]
[U] pid: 3 is running! global_variable: 1

[vm.c,45,create_mapping] root: ffffffe000317000, [80319000, 8031a000) -> [10000, 11000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe000317000, [8031c000, 8031d000) -> [11000, 12000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe000317000, [8031d000, 8031e000) -> [12000, 13000), perm: df
[vm.c,45,create_mapping] root: ffffffe000317000, [8031f000, 80320000) -> [3ffffff000, 4000000000), perm: d7
[PID = 7] forked from [PID = 3]

[U] pid: 3 is running! global_variable: 2
[U] pid: 3 is running! global_variable: 3

switch to [PID = 4 PRIORITY = 7 COUNTER = 7]
[U] pid: 4 is running! global_variable: 2
[U] pid: 4 is running! global_variable: 3

switch to [PID = 5 PRIORITY = 7 COUNTER = 7]
[U] pid: 5 is running! global_variable: 1

[vm.c,45,create_mapping] root: ffffffe000323000, [80325000, 80326000) -> [10000, 11000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe000323000, [80328000, 80329000) -> [11000, 12000), perm: 5f
[vm.c,45,create_mapping] root: ffffffe000323000, [80329000, 8032a000) -> [12000, 13000), perm: df
[vm.c,45,create_mapping] root: ffffffe000323000, [8032b000, 8032c000) -> [3ffffff000, 4000000000), perm: d7
[PID = 8] forked from [PID = 5]

[U] pid: 5 is running! global_variable: 2
[U] pid: 5 is running! global_variable: 3

switch to [PID = 6 PRIORITY = 7 COUNTER = 7]
[U] pid: 6 is running! global_variable: 2
[U] pid: 6 is running! global_variable: 3

switch to [PID = 7 PRIORITY = 7 COUNTER = 7]
[U] pid: 7 is running! global_variable: 2
[U] pid: 7 is running! global_variable: 3

switch to [PID = 8 PRIORITY = 7 COUNTER = 7]
[U] pid: 8 is running! global_variable: 2
[U] pid: 8 is running! global_variable: 3
```

结合main.c的代码解释：

```c
int global_variable = 0;

int main() {

printf("[U] pid: %ld is running! global_variable: %d\n", getpid(), global_variable++);
fork();
fork();

printf("[U] pid: %ld is running! global_variable: %d\n", getpid(), global_variable++);
fork();

while(1) {
```

```
13    printf("[U] pid: %ld is running! global_variable: %d\n",
    getpid(), global_variable++);
14    wait(WAIT_TIME);
15  }
16  }
```

为了表述方便，我们将父进程pid设为1，其余fork的进程名字即为其pid号

1. 父进程(pid = 1)自增`global_variable`

   > [U] pid: 1 is running! global_variable: 0

   1. 开始fork，创建了新进程2：

      > [PID = 2] forked from [PID = 1]

   2. 由于现在的进程仍为进程1，继续fork：

      > [PID = 3] forked from [PID = 1]

   3. 进程1自增`global_variable`：

      > [U] pid: 1 is running! global_variable: 1

   4. 由于现在的进程仍为进程1，继续fork：

      > [PID = 4] forked from [PID = 1]

   5. 进程1自增`global_variable`：

      > [U] pid: 1 is running! global_variable: 2
      > [U] pid: 1 is running! global_variable: 3

2. 切换到进程2

   1. 现在处于的位置是进行第二次fork的位置：

      > [PID = 5] forked from [PID = 2]

   2. 进程2自增`global_variable`：

      由于父进程是1，fork进程2的时候`global_variable == 1`，因此在这里输出1

      > [U] pid: 2 is running! global_variable: 1

3. 由于现在的进程仍为进程2，继续fork：

   > [PID = 6] forked from [PID = 1]

4. 进程2自增`global_variable`：

   > [U] pid: 2 is running! global_variable: 2
   >
   > [U] pid: 2 is running! global_variable: 3

3. 切换到线程3

   其变化与进程2保持一致，只是fork的线程是7：

   > [PID = 7] forked from [PID = 3]

4. 切换到线程4

   由于没有fork操作，只是输出自增`global_variable`：

   由于父进程是1，fork进程4的时候`global_variable == 2`，因此在这里输出2

   > switch to [PID = 4 PRIORITY = 7 COUNTER = 7]
   > [U] pid: 4 is running! global_variable: 2
   > [U] pid: 4 is running! global_variable: 3

5. 切换到线程5

   1. 进程5自增`global_variable`：

      由于父进程是2，fork进程5的时候`global_variable == 1`，因此在这里输出1

      > [U] pid: 5 is running! global_variable: 1

   2. 由于现在的进程仍为进程5，继续fork：

      > [PID = 8] forked from [PID = 5]

   3. 进程5自增`global_variable`：

      > [U] pid: 5 is running! global_variable: 2
      > [U] pid: 5 is running! global_variable: 3

6. 切换到线程6、7、8

   状态与线程4变化保持一致，只是单纯地输出自增变量

```
switch to [PID = 6 PRIORITY = 7 COUNTER = 7]
[U] pid: 6 is running! global_variable: 2
[U] pid: 6 is running! global_variable: 3
switch to [PID = 7 PRIORITY = 7 COUNTER = 7]
[U] pid: 7 is running! global_variable: 2
.....
```

4 思考题与心得体会

1. 呈现出你在 page fault 的时候拷贝 ELF 程序内容的逻辑。

> 这里我们仅说明非匿名区域(非栈，即ELF)的拷贝

1. 首先分配一页新的内存，并将其清零。

```
uint64_t new_pg = (uint64_t)alloc_page();
memset((void*)new_pg, 0, PGSIZE);
```

2. 计算ELF文件第一条指令在文件中的偏移量、 ELF 文件在内存中的起始地址、ELF程序末尾离页表末尾的字节数

```
uint64_t offset = vma->vm_start & (PGSIZE-0x1);
uint64_t elf_beginning = (uint64_t)(&_sramdisk) + offset;
uint64_t end_offset_rev = PGSIZE - (vma->vm_start + vma->vm_filesz) & (PGSIZE-0x1);
```

> 这里使用了&_sramdisk直接访问到ELF程序的虚拟地址，不必再进行转换，&_sramdisk加上offset即是第一条指令的虚拟地址，我们利用memcpy拷贝的时候便可以直接寻址寻到。
>
> end_offset_rev的作用是后续用来计算拷贝字节数的，因为一旦指令在最后一页中，我们不能全部拷贝(避免拷贝到不属于ELF的部分)，只需要拿PGSIZE减去end_offset_rev即可。
>
> > PS:
> >
> > 当ELF文件只占一页并且没有占满时，我们拷贝的字数应该是(PGSIZE-offset-end_offset_rev)

3. 根据 VMA 的标志设置页面权限。

```
uint64_t perm = ((vma->vm_flags & VM_EXEC) ? 0x8 :0x0) |
((vma->vm_flags & VM_WRITE) ? 0x4 :0x0) |((vma->vm_flags &
VM_READ) ? 0x2 :0x0);
perm |= 0x11; //U|V
```

> 我们根据虚拟内存单元VMA中的flag 来定义权限，调用对应宏便可以知道读写执行权限，再对应到PTE的权限位，我们将PTE的权限保存在perm变量中

4. 判断当前访问的地址是否在 VMA 的第一页或最后一页。

```
bool is_in_fstpg = PGROUNDDOWN(regs->stval) ==
PGROUNDDOWN((uint64_t)vma->vm_start);
bool is_in_lstpg = PGROUNDDOWN(regs->stval) ==
PGROUNDDOWN((uint64_t)(vma->vm_start + vma->vm_filesz
-1));
```

> 这里判断发生Page Fault的位置使用了两个布尔变量，`is_in_fstpg`指明是否存在于ELF程序的第一页，`is_in_lstpg`指明是否存在于ELF程序的最后一页。
>
> 这两个变量决定了我们深拷贝的起始地址和终了地址以及拷贝大小，便于我们进行分支判断，选择合适的地址和大小传入`memcpy`
>
> `is_in_fstpg`判断逻辑是：
>
> 如果Bad Address所在页面和第一条指令所在页面一致，就说明访问地址确实在第一页，我们置为True
>
> `is_in_lstpg`判断逻辑是：
>
> 如果Bad Address所在页面和程序最后一条指令所在页面一致，就说明访问地址确实在第一页，我们置为True

5. 根据是否在第一页或最后一页，拷贝 ELF 程序内容到新分配的页面中。

```
if(!(vma->vm_flags & VM_ANON)){
    if(is_in_fstpg){
        if(is_in_lstpg){
            memcpy((void*)(new_pg+offset), (void*)
((uint64_t)(&_sramdisk)+offset), PGSIZE-offset-
end_offset_rev);
        } else {
```

```
            memcpy((void*)(new_pg+offset), (void*)((uint64_t)(&_sramdisk)+offset), PGSIZE-offset);
        }
    } else {
        if(is_in_lstpg){
            memcpy((void*)new_pg, (void*)(elf_beginning+(PGROUNDDOWN(regs->stval) - vma->vm_start)), PGSIZE - end_offset_rev);
        } else {
            memcpy((void*)new_pg, (void*)(elf_beginning+(PGROUNDDOWN(regs->stval) - vma->vm_start)), PGSIZE);
        }
    }
}
```

> 1. 只存在于第一页：
>
>   拷贝地址从ELF第一条指令开始，拷贝大小是PGSIZE-offset
> 2. 只存在于最后一页：
>
>   拷贝地址从ELF最后一页0地址开始，拷贝大小是PGSIZE-end_offset_rev
> 3. 既不在第一页，也不在最后一页
>
>   拷贝地址从访问异常地址的所在页0地址开始，拷贝大小是PGSIZE
> 4. 既是第一页，又是最后一页
>
>   > 说明ELF比较小巧，一页足够放得下Header和所有指令
>
>   拷贝地址从ELF第一条指令开始，拷贝大小是PGSIZEPGSIZE-offset-end_offset_rev

6. **创建映射**：

   最后，将新分配的页面映射到当前任务的页表中。

```
create_mapping(current->pgd, PGROUNDDOWN(regs->stval), (uint64_t)new_pg - PA2VA_OFFSET, PGSIZE, perm);
```

> 创造映射标准像之前那样，只给定页的0地址(包括虚拟地址和物理地址)，映射到当前进程的pgd即可

2. **回答4.3.5中的问题**：

- 在 do_fork 中，父进程的内核栈和用户栈指针分别是什么？

  内核栈指针：**等价于传入参数`regs`指针的值**

  用户栈指针：**当前`sscratch`寄存器的保存值**

- 在 do_fork 中，子进程的内核栈和用户栈指针的值应该是什么？

  内核栈指针：**forked_regs = `(uint64_t)new_task + ((uint64_t)regs - PGROUNDDOWN((uint64_t)regs))`**

  > 当拷贝内核栈时，我们按照父线程的内核栈指针相对于其task_struct的偏移量，即当前新增线程的内核栈指针相对于新task_struct的偏移量，我们便可以依靠加上偏移量来算出内核栈指针

  用户栈指针：**当前`sscratch`寄存器的保存值**

  > 虽然其虚拟地址与父进程的用户栈指针一致，但是我们切换线程时会改变pgd，不同线程页表下同一虚拟地址对应物理地址不同，我们对于新的用户栈会重新拷贝一份给新进程

- 在 do_fork 中，子进程的内核栈和用户栈指针分别应该赋值给谁？

  子进程的内核栈指针赋值给**子进程的内核栈(forked_regs)内的sp**

  子进程的用户栈指针赋值给**子进程的内核栈(forked_regs)内的sscratch**，这里的用户栈指针虚拟地址并没有发生改变，所以我们没有显式赋值

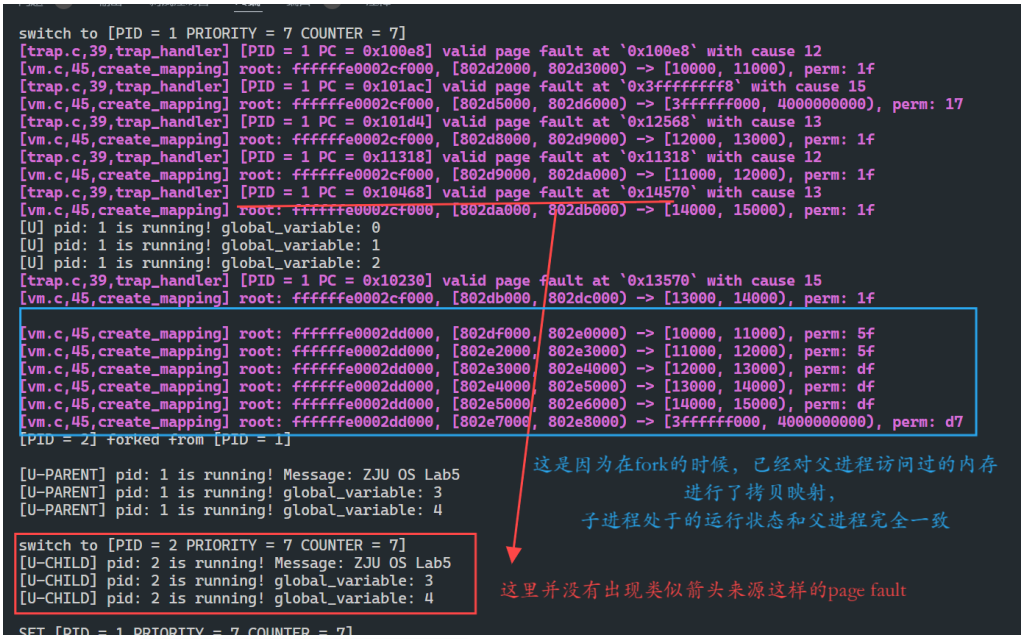
3. 为什么要为子进程 `pt_regs` 的 `sepc` 手动加四？

   sepc此时对应的是**fork**系统调用，如果不**+4**跳到下一条指令，在切换到这个新建的子线程之后，系统会再次执行**fork**，**sepc**依然不变，新新产生的子进程又会**fork**，一生二，二生三，三生万物，到时候便会产生无数多个这样的子进程。

4. 对于 `Fork main #2`（即 `FORK2`），在运行时，`ZJU OS Lab5` 位于内存的什么位置？是否在读取的时候产生了 page fault？请给出必要的截图以说明。

   全局变量存放在程序空间的`.data`区域，因此`ZJU OS Lab5`位于当前线程对应程序空间的`.data`区域。

   读取时并不产生**page fault**，在**fork**后切换到此线程时并没有出现**do_page_fault:**

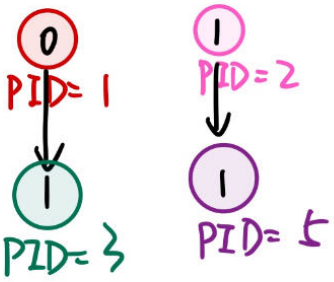


5. 画图分析 `make run TEST=FORK3` 的进程 **fork** 过程，并呈现出各个进程的 `global_variable` 应该从几开始输出，再与你的输出进行对比验证。

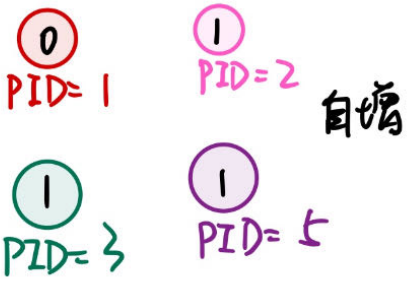
```c
int global_variable = 0;

int main() {

  printf("[U] pid: %ld is running! global_variable: %d\n", getpid(), global_variable++);
  fork();
  fork();

  printf("[U] pid: %ld is running! global_variable: %d\n", getpid(), global_variable++);
  fork();

  while(1) {
    printf("[U] pid: %ld is running! global_variable: %d\n", getpid(), global_variable++);
    wait(WAIT_TIME);
  }
}
```


```c
int global_variable = 0;

int main() {

  printf("[U] pid: %ld is running! global_variable: %d\n", getpid(), global_variable++);
  fork();
  fork();

  printf("[U] pid: %ld is running! global_variable: %d\n", getpid(), global_variable++);
  fork();

  while(1) {
    printf("[U] pid: %ld is running! global_variable: %d\n", getpid(), global_variable++);
    wait(WAIT_TIME);
  }
}
```


```c
int global_variable = 0;

int main() {

  printf("[U] pid: %ld is running! global_variable: %d\n", getpid(), global_variable++);
  fork();
  fork();

  printf("[U] pid: %ld is running! global_variable: %d\n", getpid(), global_variable++);
  fork();

  while(1) {
    printf("[U] pid: %ld is running! global_variable: %d\n", getpid(), global_variable++);
    wait(WAIT_TIME);
  }
}
```




```c
int global_variable = 0;

int main() {

  printf("[U] pid: %ld is running! global_variable: %d\n", getpid(), global_variable++);
  fork();
  fork();

  printf("[U] pid: %ld is running! global_variable: %d\n", getpid(), global_variable++);
  fork();

  while(1) {
    printf("[U] pid: %ld is running! global_variable: %d\n", getpid(), global_variable++);
    wait(WAIT_TIME);
  }
}
```




```c
int global_variable = 0;

int main() {

  printf("[U] pid: %ld is running! global_variable: %d\n", getpid(), global_variable++);
  fork();
  fork();

  printf("[U] pid: %ld is running! global_variable: %d\n", getpid(), global_variable++);
  fork();

  while(1) {
    printf("[U] pid: %ld is running! global_variable: %d\n", getpid(), global_variable++);
    wait(WAIT_TIME);
  }
}
```