浙江大学

操作系统实验五

课程名称：	操作系统
作业名称：	RV64 缺页异常处理与 fork 机制
姓　　名：	仇国智
学　　号：	3220102181
电子邮箱：	3220102181@zju.edu.cn
联系电话：	13714176104
指导教师：	申文博

2024 年 11 月 24 日

目录

1 实验内容

- 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

2 实验原理

2.1 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 具体某一进程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.

在本次实验中, 我们需要实现对进程多区域虚拟内存的管理, 即通过 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.

```
$ 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进程内虚拟内存映射的关键信息:

- vm_start:(第 1 列) 该段虚拟内存区域的开始地址
- vm_end:(第 2 列) 该段虚拟内存区域的结束地址
- vm_flags:(第 3 列) 该段虚拟内存区域的一组权限 (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

我们注意到, 一段内存中的内容可能是由磁盘中的文件映射的. 如果这样的内存的 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, 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, 然后修改当前进程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, 并没有对应的文件.

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: 同一进程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

2.2 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.

2.3 Demand paging

Demand paging 是一种虚拟内存管理技术, 它允许程序在需要时才将页面加载到内存中. 这样做的好处是, 仅加载执行进程所需的页面, 从而节省内存空间. 例如, 若一个页面从未被访问过, 那么它就不需要被放入内存中.

在 Lab4 的代码中, 我们在 task_init 的时候创建了用户栈,load_program 的时候拷贝了 load segment, 并通过 create_mapping 在页表中创建了映射. 在本次实验中, 我们将修改为 demand paging 的方式, 也就是在初始化 task 的时候不进行任何的映射

(除了内核栈以及页表以外也不需要开辟其他空间), 而是在发生缺页异常的时候检测到是记录在 vma 中的合法地址后, 再分配页面并进行映射.

2.4 RISC-V Page Faults

在 RISC-V 中, 当系统运行发生异常时, 可通过解析 scause 寄存器的值, 识别如下三种不同的 page fault:

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

2.5 处理 page fault 的方式

处理缺页异常时可能所需的信息如下:

- 触发 page fault 时访问的虚拟内存地址. 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, 保存在 scause 寄存器中

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

- 发生 page fault 时的指令执行位置, 保存在 sepc 中
- 当前进程合法的 VMA 映射关系, 保存在 vm_area_struct 链表中
- 发生异常的虚拟地址对应的 PTE (page table entry) 中记录的信息

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 中导致产生了异常的地址对应的 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, 则需要判断产生异常的原因:
  - 如果是匿名区域, 那么开辟一页内存, 然后把这一页映射到产生异常的 task 的页表中
  - 如果不是, 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

3. 返回到产生了该缺页异常的那条指令, 并继续执行程序

2.6 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.

3 实验具体过程与代码实现

3.1 缺页异常处理

3.1.1 实现虚拟内存管理功能

我们实现了下述两个函数:

- find_vma 函数: 实现对 vm_area_struct 的查找

```c
/*
* @mm : current thread's mm_struct
* @addr : the va to look up
*
* @return : the VMA if found or NULL if not found
*/
struct vm_area_struct *find_vma(struct mm_struct *mm,
    uint64_t addr);
```

- do_mmap 函数: 实现 vm_area_struct 的添加

```c
/*
* @mm : current thread's mm_struct
* @addr : the suggested va to map
* @len : memory size to map
* @vm_pgoff : phdr->p_offset
* @vm_filesz: phdr->p_filesz
* @flags : flags for the new VMA
*
* @return : start va
*/
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);
```

具体实现如下:

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

```
        }
        vma = vma->vm_next;
    }
    return vma;
}
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 *new_vma = (struct vm_area_struct
       *)kalloc();
    new_vma->vm_mm = mm;
    new_vma->vm_start = addr;
    new_vma->vm_end = addr + len;
    new_vma->vm_flags = flags;
    new_vma->vm_pgoff = vm_pgoff;
    new_vma->vm_filesz = vm_filesz;
    new_vma->vm_next = mm->mmap;
    new_vma->vm_prev = NULL;
    if (mm->mmap != NULL)
    {
        mm->mmap->vm_prev = new_vma;
    }
    mm->mmap = new_vma;
    Log("construct vma: [%p, %p) with flags %p for mm %p",
       new_vma->vm_start, new_vma->vm_end, new_vma->vm_flags, mm);
    return addr;
}
```

3.1.2 修改 tas_init

具体实现如下:

```
void task_init()
{
    ...
    for (int i = 0; i < NR_TASKS; i++)
```

```c
        {
            if (task[i] == NULL)
            {
                ...

                do_mmap(&(task[i]->mm), USER_END - PGSIZE, PGSIZE, 0, 0, VM_READ | VM_WRITE | VM_ANON);
                // load_binary(task[i]->pagetable, _sramdisk, _eramdisk);
                task[i]->thread.sepc = load_elf_lazy(task[i]->pagetable, _sramdisk,task[i]);
                break;
            }
        }
        printk("...task_init done!\n");
    }
```

我们在 task_init 的时候创建了用户栈,load_program 的时候拷贝了 load segment, 并通过 create_mapping 在页表中创建了映射. 在本次实验中, 我们将修改为 demand paging 的方式, 也就是在初始化 task 的时候不进行任何的映射, 仅仅将用户栈和所有所需的用户空间的 VMA 添加到 mm 中, 不进行内存分配和页表映射.

其中的 load_elf_lazy 函数的实现如下:

```c
uint64_t load_elf_lazy(pagetable_ptr_t pagetable, char *start, struct task_struct *new_proc)
{
    Elf64_Ehdr *elf = (Elf64_Ehdr *)start;
    for (int i = 0; i < elf->e_phnum; i++)
    {
        Elf64_Phdr *phdr = (Elf64_Phdr *)((uint64_t)start + elf->e_phoff + i * sizeof(Elf64_Phdr));
        if (phdr->p_type != PT_LOAD)
            continue;
        uint64_t flags = 0;
        if (phdr->p_flags & PF_X)
```

```c
                flags |= PTE_X;
            if (phdr->p_flags & PF_W)
                flags |= PTE_W;
            if (phdr->p_flags & PF_R)
                flags |= PTE_R;
            do_mmap(&(new_proc->mm), phdr->p_vaddr, phdr->p_memsz,
                phdr->p_offset, phdr->p_filesz, flags);
        }
        return elf->e_entry;
    }
```

可以看出, 我们在 load_elf_lazy 函数中, 仅仅将用户空间的 VMA 添加到 mm 中, 不进行内存分配和页表映射.

3.1.3 实现 page fault handler

按照上文2.5的步骤, 我们实现了 page fault handler 函数:

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

    if ((scause >> 63) == 0)
    {
        ...
        case 0xC:
            bad_addr = csr_read(stval);
            Log("Instruction page fault at %p", bad_addr);
            do_page_fault(bad_addr, VM_EXEC);
            break;
        case 0xD:
            bad_addr = csr_read(stval);
            Log("Load page fault at %p", bad_addr);
            do_page_fault(bad_addr, VM_READ);
            break;
        case 0xF:
```

```
                bad_addr = csr_read(stval);
                Log("Store/AMO page fault at %p", bad_addr);
                do_page_fault(bad_addr, VM_WRITE);
                break;
            ...
        }
        ...
}
void do_page_fault(uint64_t bad_addr, uint64_t flags)
{
    struct vm_area_struct *vma = find_vma(&(current->mm),
        bad_addr);
    if (vma == NULL)
    {
        Err("Page fault at %p out of vma", bad_addr);
        return;
    }
    if ((vma->vm_flags & flags) == 0)
    {
        if (flags == VM_EXEC)
        {
            Err("Page fault at %p, no exec permission", bad_addr);
        }
        else if (flags == VM_READ)
        {
            Err("Page fault at %p, no read permission", bad_addr);
        }
        else if (flags == VM_WRITE)
        {
            Err("Page fault at %p, no write permission", bad_addr);
        }
        return;
    }
    uint64_t pte_flags = PTE_V | PTE_U;
    if (vma->vm_flags & VM_EXEC)
```

```
            pte_flags |= PTE_X;
        if (vma->vm_flags & VM_WRITE)
        {
            ... // COW
            pte_flags |= PTE_W;
        }
        if (vma->vm_flags & VM_READ)
            pte_flags |= PTE_R;
        uint64_t new_pg = (uint64_t)kalloc();
        memset((void *)new_pg, 0, PGSIZE);

        if (!(vma->vm_flags & VM_ANON) && vma->vm_start <= bad_addr
            && vma->vm_end > bad_addr)
        {
            uint64_t copy_start = PGROUNDDOWN(bad_addr) <
                vma->vm_start ? vma->vm_start : PGROUNDDOWN(bad_addr);
            uint64_t copy_end = (PGROUNDDOWN(bad_addr) + PGSIZE) >
                vma->vm_start + vma->vm_filesz ? vma->vm_start +
                vma->vm_filesz : (PGROUNDDOWN(bad_addr) + PGSIZE);
            if (copy_end > copy_start)
            {
                uint64_t copy_size = copy_end - copy_start;
                uint64_t copy_offset = copy_start - vma->vm_start +
                    vma->vm_pgoff;
                memcpy((void *)new_pg + copy_start -
                    PGROUNDDOWN(bad_addr), (void *)(_sramdisk +
                    copy_offset), copy_size);
            }
        }
        create_mapping(current->pagetable, PGROUNDDOWN(bad_addr),
            VA2PA(new_pg), PGSIZE, pte_flags);
        Log("Page fault at %p, create mapping to %p", bad_addr,
            new_pg);
    }
```

注意中间有一段逻辑是用于处理 COW 暂且不提.

3.2 实现 fork 系统调用

3.2.1 拷贝内核栈

```c
uint64_t fork(struct pt_regs *old_regs)
{
    int pid;
    for (pid = 1; pid < NR_TASKS; pid++)
    {
        if (task[pid] == NULL)
        {
            break;
        }
    }
    if (pid == NR_TASKS)
    {
        Err("No more process can be created\n");
        return -1;
    }
    uint64_t kernel_stack = (uint64_t)kalloc();
    pagetable_ptr_t pagetable = (pagetable_ptr_t)kalloc();
    memcpy((void *)kernel_stack, (void *)current, PGSIZE);
    task[pid] = (struct task_struct *)kernel_stack;
    task[pid]->pid = pid;
    task[pid]->pagetable = pagetable;
    task[pid]->mm.mmap = NULL;
    ...
}
```

我们在 fork 函数中首先找到一个空的进程位, 随后我们拷贝内核栈, 分配内核页表, 设置内核序号, 初始化进程 VMA.

3.2.2 创建子进程页表

流程如下:

- 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
- 复制父进程的 VMA

具体实现如下:

```
void fork(struct pt_regs *old_regs)
{
    ...
    copy_pgtbl(task[pid]->pagetable,
       (pagetable_ptr_t)swapper_pg_dir);
    vmas_deep_copy(&(task[pid]->mm), &(current->mm),
       task[pid]->pagetable, current->pagetable);
    ...
}
void vmas_deep_copy(struct mm_struct *new_mm, struct mm_struct
    *old_mm, pagetable_ptr_t new_pagetable, pagetable_ptr_t
    old_pagetable)
{
    for (struct vm_area_struct *vma = old_mm->mmap; vma != NULL;
       vma = vma->vm_next)
    {
        do_mmap(new_mm, vma->vm_start, vma->vm_end -
           vma->vm_start, vma->vm_pgoff, vma->vm_filesz,
           vma->vm_flags);
        vma_pages_copy(new_pagetable, old_pagetable,
           PGROUNDDOWN(vma->vm_start), PGROUNDUP(vma->vm_end));
    }
}
void vma_pages_copy(pagetable_ptr_t new_pagetable,
    pagetable_ptr_t old_pagetable, uint64_t start, uint64_t end)
{
```

```
        for (uint64_t va = start; va < end; va += PGSIZE)
        {
            int level = 3;
            pte_t *pte = walk(old_pagetable, va, 0, &level);
            if (pte == 0 || !(*pte & PTE_V))
            {
                continue;
            }
            uint64_t old_pa = PTE2ADDR(*pte, 3);
            uint64_t old_va = PA2VA(old_pa);
            uint64_t new_pa = (uint64_t)kalloc();
            memcpy((void *)new_pa, (void *)old_va, PGSIZE);
            create_mapping(new_pagetable, va, VA2PA(new_pa), PGSIZE, PTE2FLAG(*pte));
        }
    }
```

我们在这里实现了 vmas_deep_copy 函数, 用于复制父进程的 VMA, 并在其中调用了 vma_pages_copy 函数, 用于复制父进程已分配的所有页到新的进程中, 并映射到新页表中.

3.2.3 处理进程返回逻辑

代码实现如下:

```

uint64_t fork(struct pt_regs *old_regs)
{
    ...
    task[pid]->thread.ra = (uint64_t)__ret_from_fork;
    task[pid]->thread.sp = kernel_stack + PGSIZE - sizeof(struct pt_regs);
    task[pid]->thread.sscratch = 0;
    struct pt_regs *regs = (struct pt_regs *)(kernel_stack + PGSIZE - sizeof(struct pt_regs));
    regs->x[10] = 0;
    regs->sepc += 4;
```

```c
    Log("fork: pid = %d", pid);
    return pid;
}
```

我们这里直接设置新进程的返回地址为 call trap_handler 的返回地址, 因为在这里我们的保存和恢复函数只使用了 sp, 并且退出时的 sp 十分好计算即内核栈顶减去一个 pt_regs 的大小, 这样我们在配置子进程的内核上下文时, 只需维护 sp 即可. 同时还需要设置新进程的 sscratch 为 0, 因为处于内核态. 此外我们还需要设置 x[10] 为 0, 这是因为子进程 fork 的返回值为 0. 最后我们将 sepc 加 4, 这是系统调用的正常操作, 不然会重复执行 fork.

3.2.4 写时复制 COW

接下来在我们 do_fork 创建页面, 拷贝内容, 创建页表的时候, 只需要:

- 将物理页的引用计数加一
- 将父进程的该地址对应的页表项的 PTE_W 位置 0
- 注意因为修改了页表项权限, 所以全部修改完成后需要通过 sfence.vma 刷新 TLB
- 为子进程创建一个新的页表项, 指向父进程的物理页, 且权限不带 PTE_W

这样在父子进程想要写入的时候, 就会触发 page fault, 然后再由我们在 page fault handler 中进行 COW. 在 handler 中, 我们只需要判断, 如果发生了写错误, 且 vma 的 VM_WRITE 位为 1, 而且对应地址有 pte(进行了映射) 但 pte 的 PTE_W 位为 0, 那么就可以断定这是一个写时复制的页面, 我们只需要在这个时候拷贝一份原来的页面, 重新创建一个映射即可.

我们首先实现了 vmas_lazy_copy 函数, 用于在 fork 时进行拷贝:

```c
void vmas_lazy_copy(struct mm_struct *new_mm, struct mm_struct *old_mm, pagetable_ptr_t new_pagetable, pagetable_ptr_t old_pagetable)
{
    for (struct vm_area_struct *vma = old_mm->mmap; vma != NULL; vma = vma->vm_next)
    {
        do_mmap(new_mm, vma->vm_start, vma->vm_end - vma->vm_start, vma->vm_pgoff, vma->vm_filesz, vma->vm_flags);
```

```c
            vma_pages_lazy_copy(new_pagetable, old_pagetable,
                PGROUNDDOWN(vma->vm_start), PGROUNDUP(vma->vm_end));
        }
        flush_tlb();
    }
    void vma_pages_lazy_copy(pagetable_ptr_t new_pagetable,
        pagetable_ptr_t old_pagetable, uint64_t start, uint64_t end)
    {
        for (uint64_t va = start; va < end; va += PGSIZE)
        {
            int level = 3;
            pte_t *pte = walk(old_pagetable, va, 0, &level);
            if (pte == 0 || !(*pte & PTE_V))
            {
                continue;
            }
            if ((*pte & PTE_W))
            {
                *pte &= ~PTE_W;
            }
            get_page((void *)PA2VA(PTE2ADDR(*pte, 3)));
            create_mapping(new_pagetable, va, PTE2ADDR(*pte, 3),
                PGSIZE, PTE2FLAG(*pte));
        }
    }
```

在这个函数中, 我们首先遍历父进程的 VMA, 并在新进程中创建相同的 VMA, 然后遍历父进程的页表, 并在新进程的页表中创建相同的映射, 并将物理页的引用计数加一, 若发现 PTE_W 位为 1, 则将两张页表项的 PTE_W 位清零. 注意在修改页表项权限后, 我们需要通过 flush_tlb 刷新 TLB.

我们接着实现了 COW trap 处理逻辑, 我们首先捕获到 COW, 然后通过 handle_COW 函数处理:

```c
    void do_page_fault(uint64_t bad_addr, uint64_t flags)
    {
        ...
```

```c
    if (vma->vm_flags & VM_WRITE)
    {
        if (flags == VM_WRITE)
        {
            int level = 3;
            pte_t *pte = walk(current->pagetable, bad_addr, 0,
                &level);
            if (pte && !(*pte & PTE_W) && (*pte & PTE_V))
            {
                handle_COW(pte);
                Log("handle COW for process %d at address %p",
                    current->pid, bad_addr);
                return;
            }
        }
        pte_flags |= PTE_W;
    }
    ...
}
void handle_COW(pte_t *pte)
{
    uint64_t old_va = PA2VA(PTE2ADDR(*pte, 3));
    if (get_page_refcnt((void *)old_va) > 1)
    {
        uint64_t new_va = (uint64_t)kalloc();
        memcpy((void *)new_va, (void *)old_va, PGSIZE);
        put_page((void *)old_va);
        *pte = ADDR2PTE(VA2PA(new_va), PTE2FLAG(*pte), 3);
    }
    *pte |= PTE_W;
    flush_tlb();
}
```

捕获 COW 的逻辑是, 如果发生了写错误, 且 vma 的 VM_WRITE 位为 1, 而且对应地址有 pte(进行了映射) 但 pte 的 PTE_W 位为 0, 那么就可以断定这是一个写

时复制的页面.

在 handle_COW 函数中, 我们首先判断该物理页的引用计数是否大于 1, 若是则说明有多个进程共享这个物理页, 我们需要为这个物理页创建一个新的物理页, 并将原物理页的内容拷贝到新的物理页中, 否则直接用原物理页即可, 然后将原物理页的引用计数减一, 并将新的物理页映射到页表中, 并将 PTE_W 位置 1, 最后刷新 TLB.

4 实验结果与分析

4.1 测试缺页处理

我们首先运行 PFH1 测试:

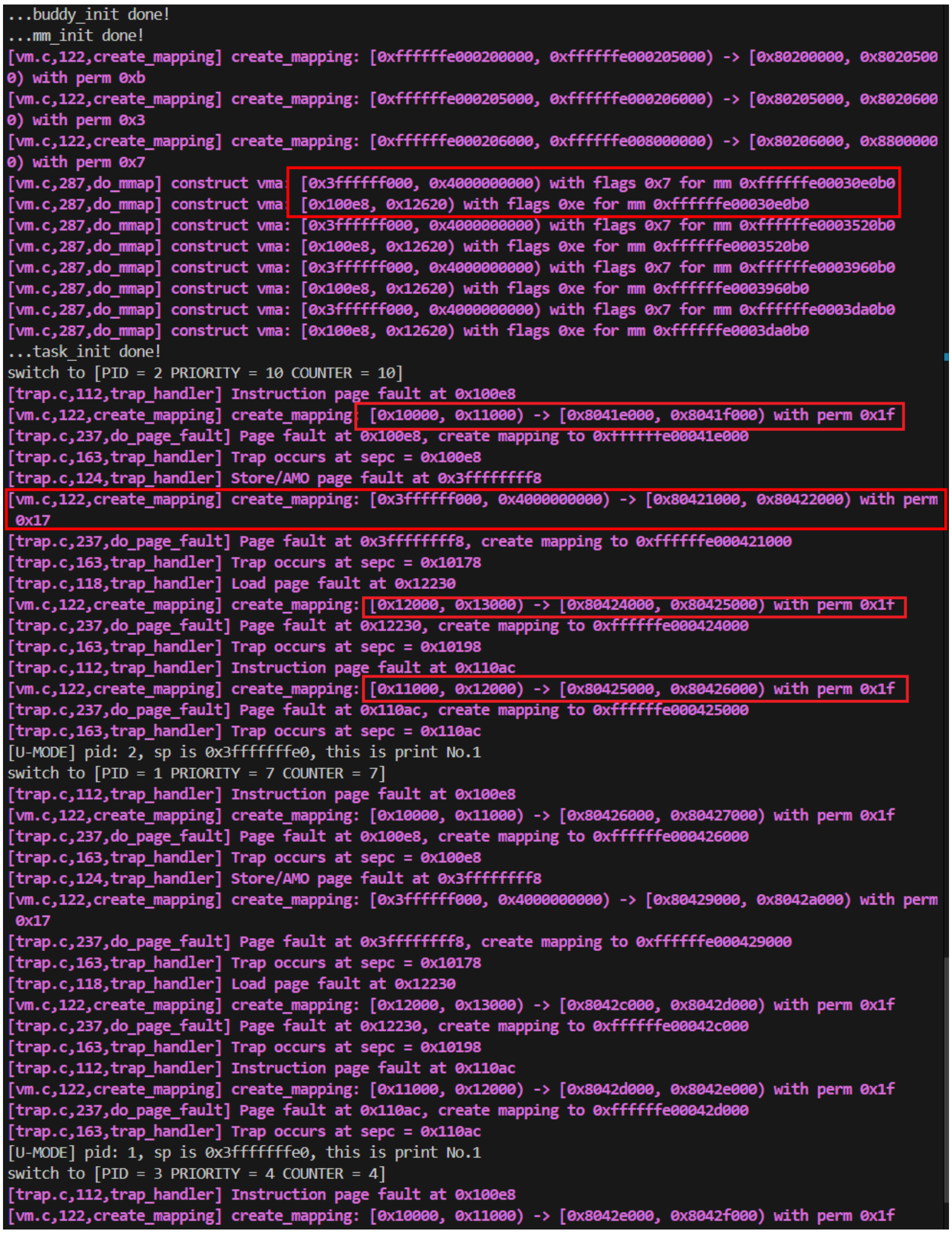


图 1: make run TEST=PFH1 image-1

```
[trap.c,163,trap_handler] Trap occurs at sepc = 0x110ac
[U-MODE] pid: 1, sp is 0x3fffffffe0, this is print No.1
switch to [PID = 3 PRIORITY = 4 COUNTER = 4]
[trap.c,112,trap_handler] Instruction page fault at 0x100e8
[vm.c,122,create_mapping] create_mapping: [0x10000, 0x11000) -> [0x8042e000, 0x8042f000) with perm 0x1f
[trap.c,237,do_page_fault] Page fault at 0x100e8, create mapping to 0xffffffe00042e000
[trap.c,163,trap_handler] Trap occurs at sepc = 0x100e8
[trap.c,124,trap_handler] Store/AMO page fault at 0x3ffffffff8
[vm.c,122,create_mapping] create_mapping: [0x3ffffff000, 0x4000000000) -> [0x80431000, 0x80432000) with perm
 0x17
[trap.c,237,do_page_fault] Page fault at 0x3ffffffff8, create mapping to 0xffffffe000431000
[trap.c,163,trap_handler] Trap occurs at sepc = 0x10178
[trap.c,118,trap_handler] Load page fault at 0x12230
[vm.c,122,create_mapping] create_mapping: [0x12000, 0x13000) -> [0x80434000, 0x80435000) with perm 0x1f
[trap.c,237,do_page_fault] Page fault at 0x12230, create mapping to 0xffffffe000434000
[trap.c,163,trap_handler] Trap occurs at sepc = 0x10198
[trap.c,112,trap_handler] Instruction page fault at 0x110ac
[vm.c,122,create_mapping] create_mapping: [0x11000, 0x12000) -> [0x80435000, 0x80436000) with perm 0x1f
[trap.c,237,do_page_fault] Page fault at 0x110ac, create mapping to 0xffffffe000435000
[trap.c,163,trap_handler] Trap occurs at sepc = 0x110ac
[U-MODE] pid: 3, sp is 0x3fffffffe0, this is print No.1
switch to [PID = 4 PRIORITY = 1 COUNTER = 1]
[trap.c,112,trap_handler] Instruction page fault at 0x100e8
[vm.c,122,create_mapping] create_mapping: [0x10000, 0x11000) -> [0x80436000, 0x80437000) with perm 0x1f
[trap.c,237,do_page_fault] Page fault at 0x100e8, create mapping to 0xffffffe000436000
[trap.c,163,trap_handler] Trap occurs at sepc = 0x100e8
[trap.c,124,trap_handler] Store/AMO page fault at 0x3ffffffff8
[vm.c,122,create_mapping] create_mapping: [0x3ffffff000, 0x4000000000) -> [0x80439000, 0x8043a000) with perm
 0x17
[trap.c,237,do_page_fault] Page fault at 0x3ffffffff8, create mapping to 0xffffffe000439000
[trap.c,163,trap_handler] Trap occurs at sepc = 0x10178
[trap.c,118,trap_handler] Load page fault at 0x12230
[vm.c,122,create_mapping] create_mapping: [0x12000, 0x13000) -> [0x8043c000, 0x8043d000) with perm 0x1f
[trap.c,237,do_page_fault] Page fault at 0x12230, create mapping to 0xffffffe00043c000
[trap.c,163,trap_handler] Trap occurs at sepc = 0x10198
[trap.c,112,trap_handler] Instruction page fault at 0x110ac
[vm.c,122,create_mapping] create_mapping: [0x11000, 0x12000) -> [0x8043d000, 0x8043e000) with perm 0x1f
[trap.c,237,do_page_fault] Page fault at 0x110ac, create mapping to 0xffffffe00043d000
[trap.c,163,trap_handler] Trap occurs at sepc = 0x110ac
[U-MODE] pid: 4, sp is 0x3fffffffe0, this is print No.1
switch to [PID = 2 PRIORITY = 10 COUNTER = 10]
[U-MODE] pid: 2, sp is 0x3fffffffe0, this is print No.2
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
switch to [PID = 3 PRIORITY = 4 COUNTER = 4]
switch to [PID = 4 PRIORITY = 1 COUNTER = 1]
switch to [PID = 2 PRIORITY = 10 COUNTER = 10]
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[U-MODE] pid: 1, sp is 0x3fffffffe0, this is print No.2
switch to [PID = 3 PRIORITY = 4 COUNTER = 4]
switch to [PID = 4 PRIORITY = 1 COUNTER = 1]
switch to [PID = 2 PRIORITY = 10 COUNTER = 10]
[U-MODE] pid: 2, sp is 0x3fffffffe0, this is print No.3
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
switch to [PID = 3 PRIORITY = 4 COUNTER = 4]
switch to [PID = 4 PRIORITY = 1 COUNTER = 1]
```

图 2: make run TEST=PFH1 image-2

可以看到所有的页错误都被正确处理了, 在运行中分配了新的页. 并且之后正常执行了.

我们再运行 PFH2 测试:

```
...buddy_init done!
...mm_init done!
[vm.c,122,create_mapping] create_mapping: [0xffffffe000200000, 0xffffffe000205000) -> [0x80200000, 0x8020500
0) with perm 0xb
[vm.c,122,create_mapping] create_mapping: [0xffffffe000205000, 0xffffffe000206000) -> [0x80205000, 0x8020600
0) with perm 0x3
[vm.c,122,create_mapping] create_mapping: [0xffffffe000206000, 0xffffffe008000000) -> [0x80206000, 0x8800000
0) with perm 0x7
[vm.c,287,do_mmap] construct vma: [0x3fffffff000, 0x4000000000) with flags 0x7 for mm 0xffffffe00030e0b0
[vm.c,287,do_mmap] construct vma: [0x100e8, 0x135e8) with flags 0xe for mm 0xffffffe00030e0b0
[vm.c,287,do_mmap] construct vma: [0x3fffffff000, 0x4000000000) with flags 0x7 for mm 0xffffffe0003520b0
[vm.c,287,do_mmap] construct vma: [0x100e8, 0x135e8) with flags 0xe for mm 0xffffffe0003520b0
[vm.c,287,do_mmap] construct vma: [0x3fffffff000, 0x4000000000) with flags 0x7 for mm 0xffffffe0003960b0
[vm.c,287,do_mmap] construct vma: [0x100e8, 0x135e8) with flags 0xe for mm 0xffffffe0003960b0
[vm.c,287,do_mmap] construct vma: [0x3fffffff000, 0x4000000000) with flags 0x7 for mm 0xffffffe0003da0b0
[vm.c,287,do_mmap] construct vma: [0x100e8, 0x135e8) with flags 0xe for mm 0xffffffe0003da0b0
...task_init done!
switch to [PID = 2 PRIORITY = 10 COUNTER = 10]
[trap.c,112,trap_handler] Instruction page fault at 0x100e8
[vm.c,122,create_mapping] create_mapping: [0x10000, 0x11000) -> [0x8041e000, 0x8041f000) with perm 0x1f
[trap.c,237,do_page_fault] Page fault at 0x100e8, create mapping to 0xffffffe00041e000
[trap.c,163,trap_handler] Trap occurs at sepc = 0x100e8
[trap.c,124,trap_handler] Store/AMO page fault at 0x3ffffffff8
[vm.c,122,create_mapping] create_mapping: [0x3fffffff000, 0x4000000000) -> [0x80421000, 0x80422000) with perm
 0x17
[trap.c,237,do_page_fault] Page fault at 0x3ffffffff8, create mapping to 0xffffffe000421000
[trap.c,163,trap_handler] Trap occurs at sepc = 0x10178
[trap.c,118,trap_handler] Load page fault at 0x131f0
[vm.c,122,create_mapping] create_mapping: [0x13000, 0x14000) -> [0x80424000, 0x80425000) with perm 0x1f
[trap.c,237,do_page_fault] Page fault at 0x131f0, create mapping to 0xffffffe000424000
[trap.c,163,trap_handler] Trap occurs at sepc = 0x10194
[trap.c,112,trap_handler] Instruction page fault at 0x1107c
[vm.c,122,create_mapping] create_mapping: [0x11000, 0x12000) -> [0x80425000, 0x80426000) with perm 0x1f
[trap.c,237,do_page_fault] Page fault at 0x1107c, create mapping to 0xffffffe000425000
[trap.c,163,trap_handler] Trap occurs at sepc = 0x1107c
[U-MODE] pid: 2, increment: 0
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[trap.c,112,trap_handler] Instruction page fault at 0x100e8
[vm.c,122,create_mapping] create_mapping: [0x10000, 0x11000) -> [0x80426000, 0x80427000) with perm 0x1f
[trap.c,237,do_page_fault] Page fault at 0x100e8, create mapping to 0xffffffe000426000
[trap.c,163,trap_handler] Trap occurs at sepc = 0x100e8
[trap.c,124,trap_handler] Store/AMO page fault at 0x3ffffffff8
[vm.c,122,create_mapping] create_mapping: [0x3fffffff000, 0x4000000000) -> [0x80429000, 0x8042a000) with perm
 0x17
[trap.c,237,do_page_fault] Page fault at 0x3ffffffff8, create mapping to 0xffffffe000429000
[trap.c,163,trap_handler] Trap occurs at sepc = 0x10178
[trap.c,118,trap_handler] Load page fault at 0x131f0
[vm.c,122,create_mapping] create_mapping: [0x13000, 0x14000) -> [0x8042c000, 0x8042d000) with perm 0x1f
[trap.c,237,do_page_fault] Page fault at 0x131f0, create mapping to 0xffffffe00042c000
```

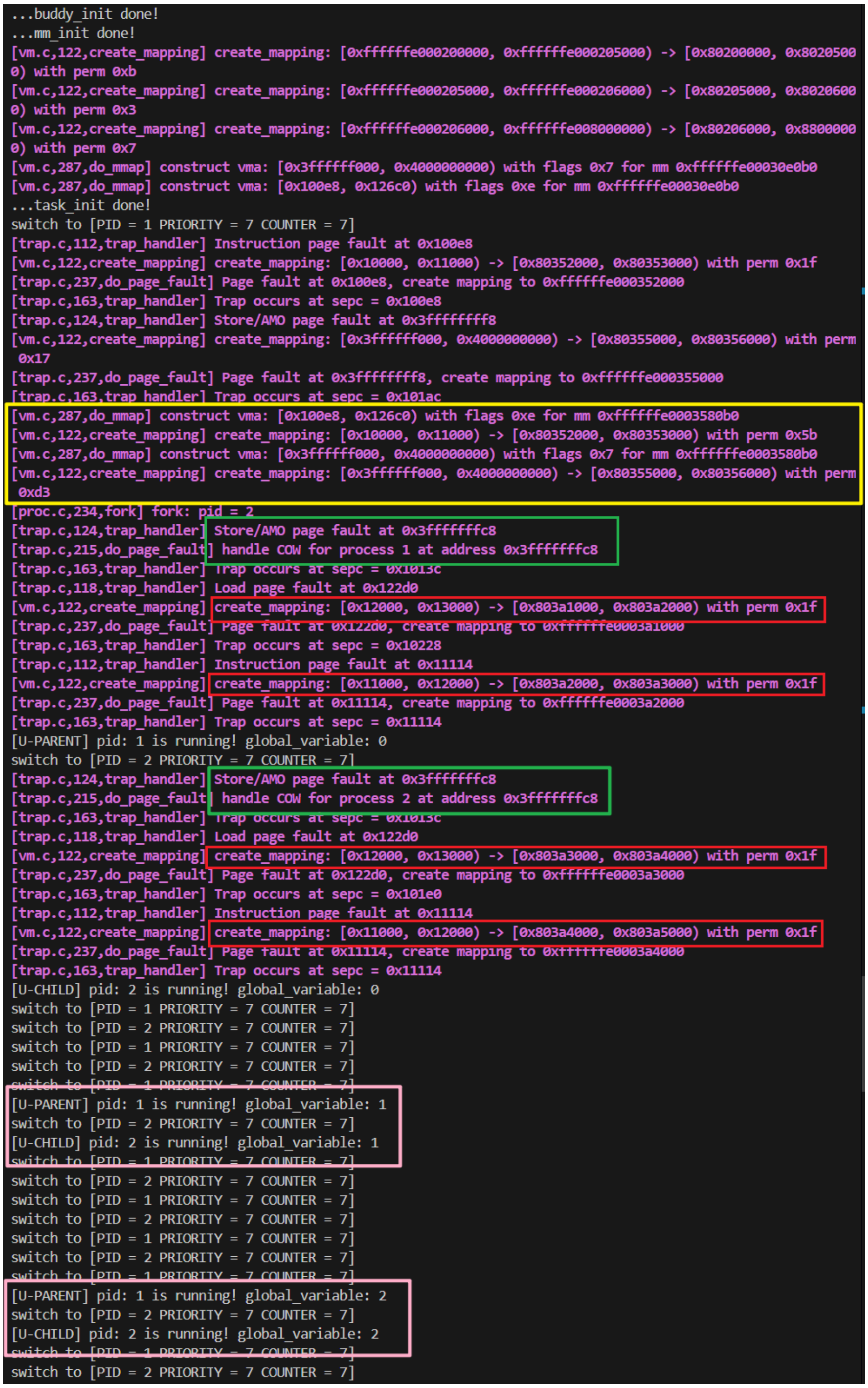
图 3: make run TEST=PFH2

可以看到页 [0x12000, 0x13000] 没有被映射.

两个测试均符合预期.

4.2 测试 fork

我们运行 FORK1 测试:



```
...buddy_init done!
...mm_init done!
[vm.c,122,create_mapping] create_mapping: [0xffffffe000200000, 0xffffffe000205000) -> [0x80200000, 0x8020500
0) with perm 0xb
[vm.c,122,create_mapping] create_mapping: [0xffffffe000205000, 0xffffffe000206000) -> [0x80205000, 0x8020600
0) with perm 0x3
[vm.c,122,create_mapping] create_mapping: [0xffffffe000206000, 0xffffffe008000000) -> [0x80206000, 0x8800000
0) with perm 0x7
[vm.c,287,do_mmap] construct vma: [0x3ffffff000, 0x4000000000) with flags 0x7 for mm 0xffffffe00030e0b0
[vm.c,287,do_mmap] construct vma: [0x100e8, 0x126c0) with flags 0xe for mm 0xffffffe00030e0b0
...task_init done!
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[trap.c,112,trap_handler] Instruction page fault at 0x100e8
[vm.c,122,create_mapping] create_mapping: [0x10000, 0x11000) -> [0x80352000, 0x80353000) with perm 0x1f
[trap.c,237,do_page_fault] Page fault at 0x100e8, create mapping to 0xffffffe000352000
[trap.c,163,trap_handler] Trap occurs at sepc = 0x100e8
[trap.c,124,trap_handler] Store/AMO page fault at 0x3ffffffff8
[vm.c,122,create_mapping] create_mapping: [0x3ffffff000, 0x4000000000) -> [0x80355000, 0x80356000) with perm
 0x17
[trap.c,237,do_page_fault] Page fault at 0x3ffffffff8, create mapping to 0xffffffe000355000
[trap.c,163,trap_handler] Trap occurs at sepc = 0x101ac
[vm.c,287,do_mmap] construct vma: [0x100e8, 0x126c0) with flags 0xe for mm 0xffffffe0003580b0
[vm.c,122,create_mapping] create_mapping: [0x10000, 0x11000) -> [0x80352000, 0x80353000) with perm 0x5b
[vm.c,287,do_mmap] construct vma: [0x3ffffff000, 0x4000000000) with flags 0x7 for mm 0xffffffe0003580b0
[vm.c,122,create_mapping] create_mapping: [0x3ffffff000, 0x4000000000) -> [0x80355000, 0x80356000) with perm
 0xd3
[proc.c,234,fork] fork: pid = 2
[trap.c,124,trap_handler] Store/AMO page fault at 0x3fffffffc8
[trap.c,215,do_page_fault] handle COW for process 1 at address 0x3fffffffc8
[trap.c,163,trap_handler] Trap occurs at sepc = 0x1013c
[trap.c,118,trap_handler] Load page fault at 0x122d0
[vm.c,122,create_mapping] create_mapping: [0x12000, 0x13000) -> [0x803a1000, 0x803a2000) with perm 0x1f
[trap.c,237,do_page_fault] Page fault at 0x122d0, create mapping to 0xffffffe0003a1000
[trap.c,163,trap_handler] Trap occurs at sepc = 0x10228
[trap.c,112,trap_handler] Instruction page fault at 0x11114
[vm.c,122,create_mapping] create_mapping: [0x11000, 0x12000) -> [0x803a2000, 0x803a3000) with perm 0x1f
[trap.c,237,do_page_fault] Page fault at 0x11114, create mapping to 0xffffffe0003a2000
[trap.c,163,trap_handler] Trap occurs at sepc = 0x11114
[U-PARENT] pid: 1 is running! global_variable: 0
switch to [PID = 2 PRIORITY = 7 COUNTER = 7]
[trap.c,124,trap_handler] Store/AMO page fault at 0x3fffffffc8
[trap.c,215,do_page_fault] handle COW for process 2 at address 0x3fffffffc8
[trap.c,163,trap_handler] Trap occurs at sepc = 0x1013c
[trap.c,118,trap_handler] Load page fault at 0x122d0
[vm.c,122,create_mapping] create_mapping: [0x12000, 0x13000) -> [0x803a3000, 0x803a4000) with perm 0x1f
[trap.c,237,do_page_fault] Page fault at 0x122d0, create mapping to 0xffffffe0003a3000
[trap.c,163,trap_handler] Trap occurs at sepc = 0x101e0
[trap.c,112,trap_handler] Instruction page fault at 0x11114
[vm.c,122,create_mapping] create_mapping: [0x11000, 0x12000) -> [0x803a4000, 0x803a5000) with perm 0x1f
[trap.c,237,do_page_fault] Page fault at 0x11114, create mapping to 0xffffffe0003a4000
[trap.c,163,trap_handler] Trap occurs at sepc = 0x11114
[U-CHILD] pid: 2 is running! global_variable: 0
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
switch to [PID = 2 PRIORITY = 7 COUNTER = 7]
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
switch to [PID = 2 PRIORITY = 7 COUNTER = 7]
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[U-PARENT] pid: 1 is running! global_variable: 1
switch to [PID = 2 PRIORITY = 7 COUNTER = 7]
[U-CHILD] pid: 2 is running! global_variable: 1
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
switch to [PID = 2 PRIORITY = 7 COUNTER = 7]
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
switch to [PID = 2 PRIORITY = 7 COUNTER = 7]
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
switch to [PID = 2 PRIORITY = 7 COUNTER = 7]
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[U-PARENT] pid: 1 is running! global_variable: 2
switch to [PID = 2 PRIORITY = 7 COUNTER = 7]
[U-CHILD] pid: 2 is running! global_variable: 2
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
switch to [PID = 2 PRIORITY = 7 COUNTER = 7]
```

图 4: make run TEST=FORK1

可以看到 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.

我们再运行 FORK2 测试:

```
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[trap.c,112,trap_handler] Instruction page fault at 0x100e8
[vm.c,122,create_mapping] create_mapping: [0x10000, 0x11000) -> [0x80352000, 0x80353000) with perm 0x1f
[trap.c,237,do_page_fault] Page fault at 0x100e8, create mapping to 0xffffffe000352000
[trap.c,163,trap_handler] Trap occurs at sepc = 0x100e8
[trap.c,124,trap_handler] Store/AMO page fault at 0x3ffffffff8
[vm.c,122,create_mapping] create_mapping: [0x3ffffff000, 0x4000000000) -> [0x80355000, 0x80356000) with perm
 0x17
[trap.c,237,do_page_fault] Page fault at 0x3ffffffff8, create mapping to 0xffffffe000355000
[trap.c,163,trap_handler] Trap occurs at sepc = 0x101ac
[trap.c,118,trap_handler] Load page fault at 0x12518
[vm.c,122,create_mapping] create_mapping: [0x12000, 0x13000) -> [0x80358000, 0x80359000) with perm 0x1f
[trap.c,237,do_page_fault] Page fault at 0x12518, create mapping to 0xffffffe000358000
[trap.c,163,trap_handler] Trap occurs at sepc = 0x101d0
[trap.c,112,trap_handler] Instruction page fault at 0x112cc
[vm.c,122,create_mapping] create_mapping: [0x11000, 0x12000) -> [0x80359000, 0x8035a000) with perm 0x1f
[trap.c,237,do_page_fault] Page fault at 0x112cc, create mapping to 0xffffffe000359000
[trap.c,163,trap_handler] Trap occurs at sepc = 0x112cc
[trap.c,118,trap_handler] Load page fault at 0x14520
[vm.c,122,create_mapping] create_mapping: [0x14000, 0x15000) -> [0x8035a000, 0x8035b000) with perm 0x1f
[trap.c,237,do_page_fault] Page fault at 0x14520, create mapping to 0xffffffe00035a000
[trap.c,163,trap_handler] Trap occurs at sepc = 0x10434
[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,124,trap_handler] Store/AMO page fault at 0x13520
[vm.c,122,create_mapping] create_mapping: [0x13000, 0x14000) -> [0x8035b000, 0x8035c000) with perm 0x1f
[trap.c,237,do_page_fault] Page fault at 0x13520, create mapping to 0xffffffe00035b000
[trap.c,163,trap_handler] Trap occurs at sepc = 0x10228
[vm.c,287,do_mmap] construct vma: [0x100e8, 0x14910) with flags 0xe for mm 0xffffffe00035c0b0
[vm.c,122,create_mapping] create_mapping: [0x10000, 0x11000) -> [0x80352000, 0x80353000) with perm 0x5b
[vm.c,122,create_mapping] create_mapping: [0x11000, 0x12000) -> [0x80359000, 0x8035a000) with perm 0x5b
[vm.c,122,create_mapping] create_mapping: [0x12000, 0x13000) -> [0x80358000, 0x80359000) with perm 0xdb
[vm.c,122,create_mapping] create_mapping: [0x13000, 0x14000) -> [0x8035b000, 0x8035c000) with perm 0xdb
[vm.c,122,create_mapping] create_mapping: [0x14000, 0x15000) -> [0x8035a000, 0x8035b000) with perm 0xdb
[vm.c,287,do_mmap] construct vma: [0x3ffffff000, 0x4000000000) with flags 0x7 for mm 0xffffffe00035c0b0
[vm.c,122,create_mapping] create_mapping: [0x3ffffff000, 0x4000000000) -> [0x80355000, 0x80356000) with perm
 0xd3
[proc.c,234,fork] fork: pid = 2
[trap.c,124,trap_handler] Store/AMO page fault at 0x3fffffffc8
[trap.c,215,do_page_fault] handle COW for process 1 at address 0x3fffffffc8
[trap.c,163,trap_handler] Trap occurs at sepc = 0x1013c
[trap.c,124,trap_handler] Store/AMO page fault at 0x14520
[trap.c,215,do_page_fault] handle COW for process 1 at address 0x14520
[trap.c,163,trap_handler] Trap occurs at sepc = 0x10448
[U-PARENT] pid: 1 is running! Message: ZJU OS Lab5
[trap.c,124,trap_handler] Store/AMO page fault at 0x12518
[trap.c,215,do_page_fault] handle COW for process 1 at address 0x12518
[trap.c,163,trap_handler] Trap occurs at sepc = 0x103f4
[U-PARENT] pid: 1 is running! global_variable: 3
switch to [PID = 2 PRIORITY = 7 COUNTER = 7]
[trap.c,124,trap_handler] Store/AMO page fault at 0x3fffffffc8
[trap.c,215,do_page_fault] handle COW for process 2 at address 0x3fffffffc8
[trap.c,163,trap_handler] Trap occurs at sepc = 0x1013c
[trap.c,124,trap_handler] Store/AMO page fault at 0x14520
[trap.c,215,do_page_fault] handle COW for process 2 at address 0x14520
[trap.c,163,trap_handler] Trap occurs at sepc = 0x10448
[U-CHILD] pid: 2 is running! Message: ZJU OS Lab5
[trap.c,124,trap_handler] Store/AMO page fault at 0x12518
[trap.c,215,do_page_fault] handle COW for process 2 at address 0x12518
[trap.c,163,trap_handler] Trap occurs at sepc = 0x1038c
[U-CHILD] pid: 2 is running! global_variable: 3
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
switch to [PID = 2 PRIORITY = 7 COUNTER = 7]
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
switch to [PID = 2 PRIORITY = 7 COUNTER = 7]
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[U-PARENT] pid: 1 is running! global_variable: 4
switch to [PID = 2 PRIORITY = 7 COUNTER = 7]
[U-CHILD] pid: 2 is running! global_variable: 4
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
switch to [PID = 2 PRIORITY = 7 COUNTER = 7]
```

图 5: make run TEST=FORK2

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, 符合预期.

最后我们运行 FORK3 测试, 测试结果在下文思考题6.1中.

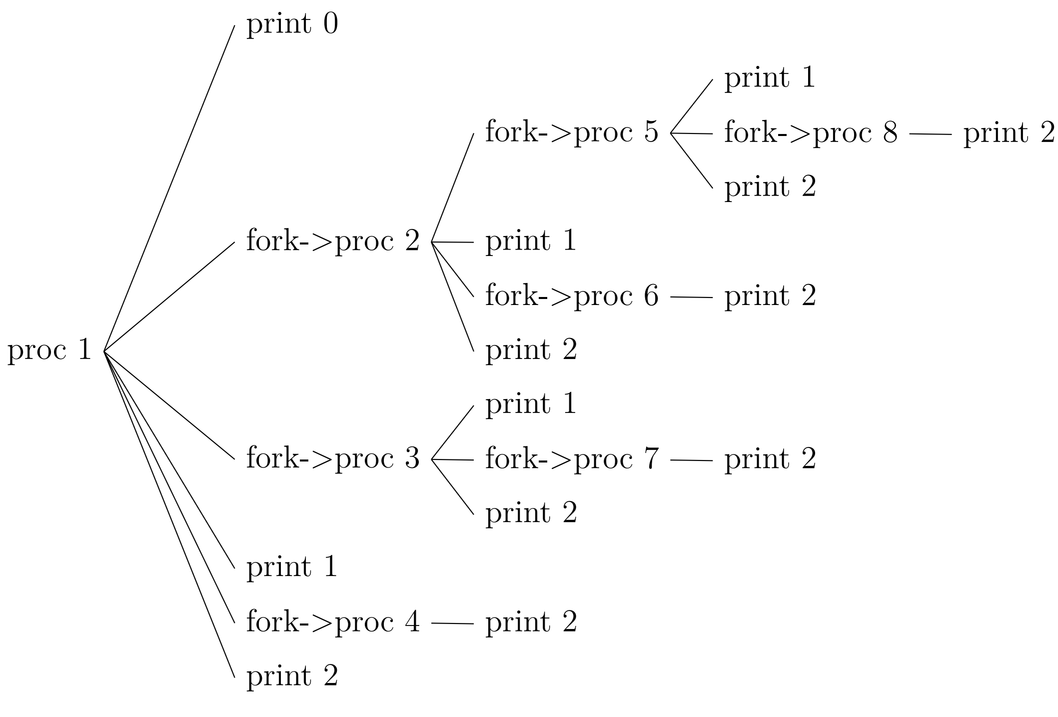
5 遇到的问题及解决方法

问题: 在 FORK3 测试中, 子进程不是从 3 开始输出解决方法: 一开始我还以为是因为 COW 没有正确处理双引用页和单引用页, 结果发现是因为没有刷新 TLB.

6 总结与心得

本次实验与上两次实验相比较为简单, 这是因为本次实验的 debug 难度大大降低. 不过我感到有迷惑的一点是, 目前引入了 COW 的页引用机制后, buddy system 只能支持单页分配和释放了, 因为页引用只能支持单页, 混用的话很可能会出错. 那这样的话, 我们引入 buddy system 的目的是什么? 还不如只用链表.

6.1 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.

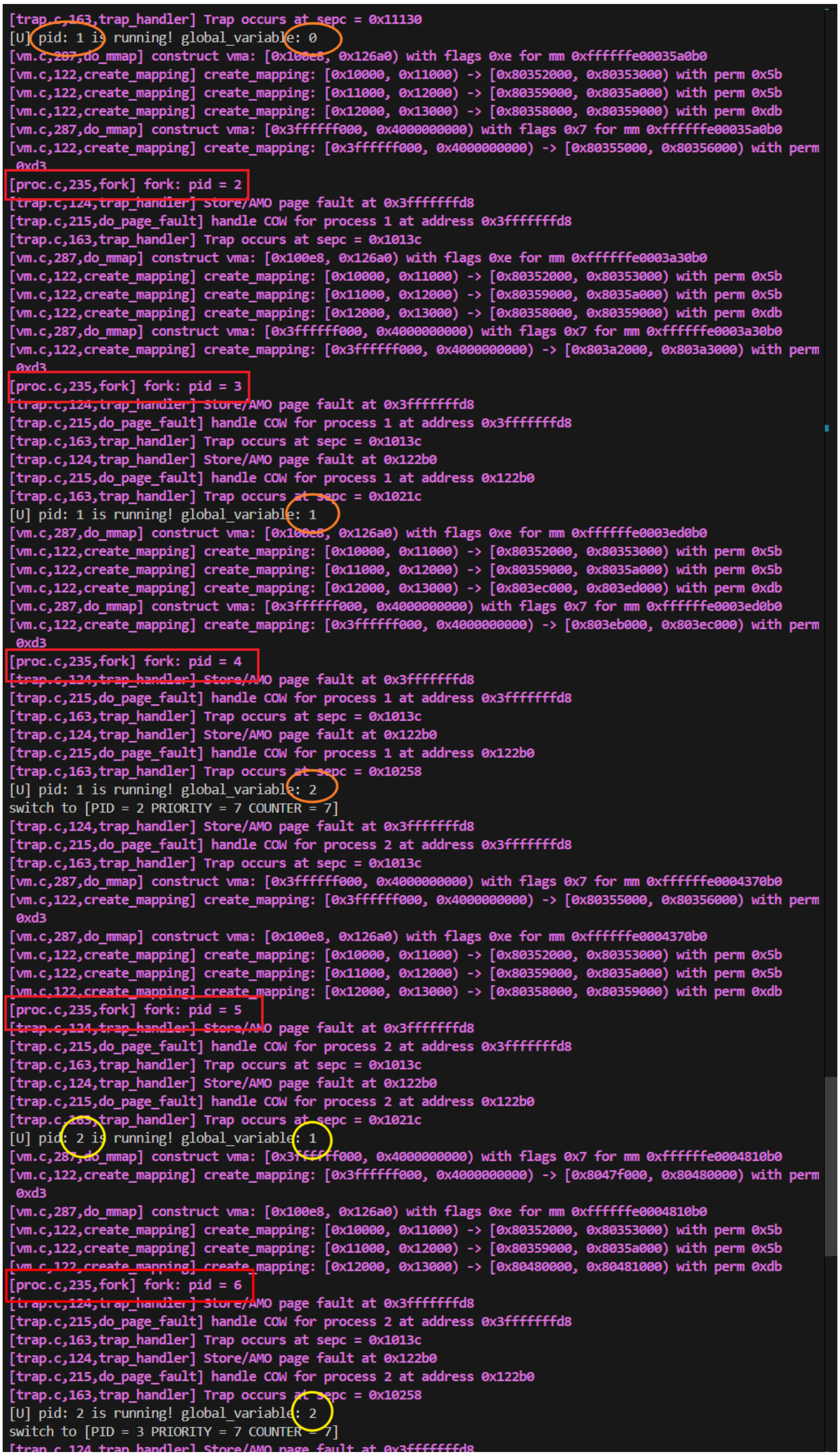


由于我们时钟周期设置较大, 我们可以确保在第一个时钟周期内执行完 WAIT 前的指令, 并且我们的调度策略是同优先级序列号越小优先级越高, 所以我们的进程是按照序号从小到大的顺序执行的. 据此, 我们可以画出上述的进程树.

我们可以看出各个进程第一个输出的 global_variable 如下:

proc 1	proc 2	proc 3	proc 4	proc 5	proc 6	proc 7	proc 8
0	1	1	2	1	2	2	2

实际测试如下图:



```
[trap.c,163,trap_handler] Trap occurs at sepc = 0x11130
[U] pid: 1 is running! global_variable: 0
[vm.c,287,do_mmap] construct vma: [0x100e8, 0x126a0) with flags 0xe for mm 0xffffffe00035a0b0
[vm.c,122,create_mapping] create_mapping: [0x10000, 0x11000) -> [0x80352000, 0x80353000) with perm 0x5b
[vm.c,122,create_mapping] create_mapping: [0x11000, 0x12000) -> [0x80359000, 0x8035a000) with perm 0x5b
[vm.c,122,create_mapping] create_mapping: [0x12000, 0x13000) -> [0x80358000, 0x80359000) with perm 0xdb
[vm.c,287,do_mmap] construct vma: [0x3ffffff000, 0x4000000000) with flags 0x7 for mm 0xffffffe00035a0b0
[vm.c,122,create_mapping] create_mapping: [0x3ffffff000, 0x4000000000) -> [0x80355000, 0x80356000) with perm
 0xd3
[proc.c,235,fork] fork: pid = 2
[trap.c,124,trap_handler] Store/AMO page fault at 0x3fffffffd8
[trap.c,215,do_page_fault] handle COW for process 1 at address 0x3fffffffd8
[trap.c,163,trap_handler] Trap occurs at sepc = 0x1013c
[vm.c,287,do_mmap] construct vma: [0x100e8, 0x126a0) with flags 0xe for mm 0xffffffe0003a30b0
[vm.c,122,create_mapping] create_mapping: [0x10000, 0x11000) -> [0x80352000, 0x80353000) with perm 0x5b
[vm.c,122,create_mapping] create_mapping: [0x11000, 0x12000) -> [0x80359000, 0x8035a000) with perm 0x5b
[vm.c,122,create_mapping] create_mapping: [0x12000, 0x13000) -> [0x80358000, 0x80359000) with perm 0xdb
[vm.c,287,do_mmap] construct vma: [0x3ffffff000, 0x4000000000) with flags 0x7 for mm 0xffffffe0003a30b0
[vm.c,122,create_mapping] create_mapping: [0x3ffffff000, 0x4000000000) -> [0x803a2000, 0x803a3000) with perm
 0xd3
[proc.c,235,fork] fork: pid = 3
[trap.c,124,trap_handler] Store/AMO page fault at 0x3fffffffd8
[trap.c,215,do_page_fault] handle COW for process 1 at address 0x3fffffffd8
[trap.c,163,trap_handler] Trap occurs at sepc = 0x1013c
[trap.c,124,trap_handler] Store/AMO page fault at 0x122b0
[trap.c,215,do_page_fault] handle COW for process 1 at address 0x122b0
[trap.c,163,trap_handler] Trap occurs at sepc = 0x1021c
[U] pid: 1 is running! global_variable: 1
[vm.c,287,do_mmap] construct vma: [0x100e8, 0x126a0) with flags 0xe for mm 0xffffffe0003ed0b0
[vm.c,122,create_mapping] create_mapping: [0x10000, 0x11000) -> [0x80352000, 0x80353000) with perm 0x5b
[vm.c,122,create_mapping] create_mapping: [0x11000, 0x12000) -> [0x80359000, 0x8035a000) with perm 0x5b
[vm.c,122,create_mapping] create_mapping: [0x12000, 0x13000) -> [0x803ec000, 0x803ed000) with perm 0xdb
[vm.c,287,do_mmap] construct vma: [0x3ffffff000, 0x4000000000) with flags 0x7 for mm 0xffffffe0003ed0b0
[vm.c,122,create_mapping] create_mapping: [0x3ffffff000, 0x4000000000) -> [0x803eb000, 0x803ec000) with perm
 0xd3
[proc.c,235,fork] fork: pid = 4
[trap.c,124,trap_handler] Store/AMO page fault at 0x3fffffffd8
[trap.c,215,do_page_fault] handle COW for process 1 at address 0x3fffffffd8
[trap.c,163,trap_handler] Trap occurs at sepc = 0x1013c
[trap.c,124,trap_handler] Store/AMO page fault at 0x122b0
[trap.c,215,do_page_fault] handle COW for process 1 at address 0x122b0
[trap.c,163,trap_handler] Trap occurs at sepc = 0x10258
[U] pid: 1 is running! global_variable: 2
switch to [PID = 2 PRIORITY = 7 COUNTER = 7]
[trap.c,124,trap_handler] Store/AMO page fault at 0x3fffffffd8
[trap.c,215,do_page_fault] handle COW for process 2 at address 0x3fffffffd8
[trap.c,163,trap_handler] Trap occurs at sepc = 0x1013c
[vm.c,287,do_mmap] construct vma: [0x3ffffff000, 0x4000000000) with flags 0x7 for mm 0xffffffe0004370b0
[vm.c,122,create_mapping] create_mapping: [0x3ffffff000, 0x4000000000) -> [0x80355000, 0x80356000) with perm
 0xd3
[vm.c,287,do_mmap] construct vma: [0x100e8, 0x126a0) with flags 0xe for mm 0xffffffe0004370b0
[vm.c,122,create_mapping] create_mapping: [0x10000, 0x11000) -> [0x80352000, 0x80353000) with perm 0x5b
[vm.c,122,create_mapping] create_mapping: [0x11000, 0x12000) -> [0x80359000, 0x8035a000) with perm 0x5b
[vm.c,122,create_mapping] create_mapping: [0x12000, 0x13000) -> [0x80358000, 0x80359000) with perm 0xdb
[proc.c,235,fork] fork: pid = 5
[trap.c,124,trap_handler] Store/AMO page fault at 0x3fffffffd8
[trap.c,215,do_page_fault] handle COW for process 2 at address 0x3fffffffd8
[trap.c,163,trap_handler] Trap occurs at sepc = 0x1013c
[trap.c,124,trap_handler] Store/AMO page fault at 0x122b0
[trap.c,215,do_page_fault] handle COW for process 2 at address 0x122b0
[trap.c,163,trap_handler] Trap occurs at sepc = 0x1021c
[U] pid: 2 is running! global_variable: 1
[vm.c,287,do_mmap] construct vma: [0x3ffffff000, 0x4000000000) with flags 0x7 for mm 0xffffffe0004810b0
[vm.c,122,create_mapping] create_mapping: [0x3ffffff000, 0x4000000000) -> [0x8047f000, 0x80480000) with perm
 0xd3
[vm.c,287,do_mmap] construct vma: [0x100e8, 0x126a0) with flags 0xe for mm 0xffffffe0004810b0
[vm.c,122,create_mapping] create_mapping: [0x10000, 0x11000) -> [0x80352000, 0x80353000) with perm 0x5b
[vm.c,122,create_mapping] create_mapping: [0x11000, 0x12000) -> [0x80359000, 0x8035a000) with perm 0x5b
[vm.c,122,create_mapping] create_mapping: [0x12000, 0x13000) -> [0x80480000, 0x80481000) with perm 0xdb
[proc.c,235,fork] fork: pid = 6
[trap.c,124,trap_handler] Store/AMO page fault at 0x3fffffffd8
[trap.c,215,do_page_fault] handle COW for process 2 at address 0x3fffffffd8
[trap.c,163,trap_handler] Trap occurs at sepc = 0x1013c
[trap.c,124,trap_handler] Store/AMO page fault at 0x122b0
[trap.c,215,do_page_fault] handle COW for process 2 at address 0x122b0
[trap.c,163,trap_handler] Trap occurs at sepc = 0x10258
[U] pid: 2 is running! global_variable: 2
switch to [PID = 3 PRIORITY = 7 COUNTER = 7]
[trap.c,124,trap_handler] Store/AMO page fault at 0x3fffffffd8
```

图 6: make run TEST=FORK3 proc 1-2

```
switch to [PID = 3 PRIORITY = 7 COUNTER = 7]
[trap.c,124,trap_handler] Store/AMO page fault at 0x3fffffffd8
[trap.c,215,do_page_fault] handle COW for process 3 at address 0x3fffffffd8
[trap.c,163,trap_handler] Trap occurs at sepc = 0x1013c
[trap.c,124,trap_handler] Store/AMO page fault at 0x122b0
[trap.c,215,do_page_fault] handle COW for process 3 at address 0x122b0
[trap.c,163,trap_handler] Trap occurs at sepc = 0x1021c
[U] pid: 3 is running! global_variable: 1
[vm.c,287,do_mmap] construct vma: [0x3ffffff000, 0x4000000000) with flags 0x7 for mm 0xffffffe0004cc0b0
[vm.c,122,create_mapping] create_mapping: [0x3ffffff000, 0x4000000000) -> [0x803a2000, 0x803a3000) with perm
 0xd3
[vm.c,287,do_mmap] construct vma: [0x100e8, 0x126a0) with flags 0xe for mm 0xffffffe0004cc0b0
[vm.c,122,create_mapping] create_mapping: [0x10000, 0x11000) -> [0x80352000, 0x80353000) with perm 0x5b
[vm.c,122,create_mapping] create_mapping: [0x11000, 0x12000) -> [0x80359000, 0x8035a000) with perm 0x5b
[vm.c,122,create_mapping] create_mapping: [0x12000, 0x13000) -> [0x804cb000, 0x804cc000) with perm 0xdb
[proc.c,235,fork] fork: pid = 7
[trap.c,124,trap_handler] Store/AMO page fault at 0x3fffffffd8
[trap.c,215,do_page_fault] handle COW for process 3 at address 0x3fffffffd8
[trap.c,163,trap_handler] Trap occurs at sepc = 0x1013c
[trap.c,124,trap_handler] Store/AMO page fault at 0x122b0
[trap.c,215,do_page_fault] handle COW for process 3 at address 0x122b0
[trap.c,163,trap_handler] Trap occurs at sepc = 0x10258
[U] pid: 3 is running! global_variable: 2
switch to [PID = 4 PRIORITY = 7 COUNTER = 7]
[trap.c,124,trap_handler] Store/AMO page fault at 0x3fffffffd8
[trap.c,215,do_page_fault] handle COW for process 4 at address 0x3fffffffd8
[trap.c,163,trap_handler] Trap occurs at sepc = 0x1013c
[trap.c,124,trap_handler] Store/AMO page fault at 0x122b0
[trap.c,215,do_page_fault] handle COW for process 4 at address 0x122b0
[trap.c,163,trap_handler] Trap occurs at sepc = 0x10258
[U] pid: 4 is running! global_variable: 2
switch to [PID = 5 PRIORITY = 7 COUNTER = 7]
[trap.c,124,trap_handler] Store/AMO page fault at 0x3fffffffd8
[trap.c,215,do_page_fault] handle COW for process 5 at address 0x3fffffffd8
[trap.c,163,trap_handler] Trap occurs at sepc = 0x1013c
[trap.c,124,trap_handler] Store/AMO page fault at 0x122b0
[trap.c,215,do_page_fault] handle COW for process 5 at address 0x122b0
[trap.c,163,trap_handler] Trap occurs at sepc = 0x1021c
[U] pid: 5 is running! global_variable: 1
[vm.c,287,do_mmap] construct vma: [0x100e8, 0x126a0) with flags 0xe for mm 0xffffffe0005160b0
[vm.c,122,create_mapping] create_mapping: [0x10000, 0x11000) -> [0x80352000, 0x80353000) with perm 0x5b
[vm.c,122,create_mapping] create_mapping: [0x11000, 0x12000) -> [0x80359000, 0x8035a000) with perm 0x5b
[vm.c,122,create_mapping] create_mapping: [0x12000, 0x13000) -> [0x80358000, 0x80359000) with perm 0xdb
[vm.c,287,do_mmap] construct vma: [0x3ffffff000, 0x4000000000) with flags 0x7 for mm 0xffffffe0005160b0
[vm.c,122,create_mapping] create_mapping: [0x3ffffff000, 0x4000000000) -> [0x80355000, 0x80356000) with perm
 0xd3
[proc.c,235,fork] fork: pid = 8
[trap.c,124,trap_handler] Store/AMO page fault at 0x3fffffffd8
[trap.c,215,do_page_fault] handle COW for process 5 at address 0x3fffffffd8
[trap.c,163,trap_handler] Trap occurs at sepc = 0x1013c
[trap.c,124,trap_handler] Store/AMO page fault at 0x122b0
[trap.c,215,do_page_fault] handle COW for process 5 at address 0x122b0
[trap.c,163,trap_handler] Trap occurs at sepc = 0x10258
[U] pid: 5 is running! global_variable: 2
switch to [PID = 6 PRIORITY = 7 COUNTER = 7]
[trap.c,124,trap_handler] Store/AMO page fault at 0x3fffffffd8
[trap.c,215,do_page_fault] handle COW for process 6 at address 0x3fffffffd8
[trap.c,163,trap_handler] Trap occurs at sepc = 0x1013c
[trap.c,124,trap_handler] Store/AMO page fault at 0x122b0
[trap.c,215,do_page_fault] handle COW for process 6 at address 0x122b0
[trap.c,163,trap_handler] Trap occurs at sepc = 0x10258
[U] pid: 6 is running! global_variable: 2
switch to [PID = 7 PRIORITY = 7 COUNTER = 7]
[trap.c,124,trap_handler] Store/AMO page fault at 0x3fffffffd8
[trap.c,215,do_page_fault] handle COW for process 7 at address 0x3fffffffd8
[trap.c,163,trap_handler] Trap occurs at sepc = 0x1013c
[trap.c,124,trap_handler] Store/AMO page fault at 0x122b0
[trap.c,215,do_page_fault] handle COW for process 7 at address 0x122b0
[trap.c,163,trap_handler] Trap occurs at sepc = 0x10258
[U] pid: 7 is running! global_variable: 2
switch to [PID = 8 PRIORITY = 7 COUNTER = 7]
[trap.c,124,trap_handler] Store/AMO page fault at 0x3fffffffd8
[trap.c,215,do_page_fault] handle COW for process 8 at address 0x3fffffffd8
[trap.c,163,trap_handler] Trap occurs at sepc = 0x1013c
[trap.c,124,trap_handler] Store/AMO page fault at 0x122b0
[trap.c,215,do_page_fault] handle COW for process 8 at address 0x122b0
[trap.c,163,trap_handler] Trap occurs at sepc = 0x10258
[U] pid: 8 is running! global_variable: 2
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
```

图 7: make run TEST=FORK3 proc 3-8

这与上面我们推测的进程树结果是一致的.