# Lab 6: RV64 缺页异常处理以及 fork 机制

**课程名称** **:** 操作系统 **指导老师** **:** 寿黎但
**姓名:** 庄毅非

## 1、实验目的

- 通过 `vm_area_struct` 数据结构实现对 task **多区域**虚拟内存的管理。
- 在 **Lab5** 实现用户态程序的基础上，添加缺页异常处理 **Page Fault Handler**。
- 为 task 加入 **fork** 机制，能够支持通过 **fork** 创建新的用户态 task 。

## 2、实验环境

- Environment in previous lab5

## 3、实验步骤

### 3.1 实现vma

这里主要是对vma机制的实现，首先修改proc.h文件，添加如下的内容。

```c
// proc.h
// 添加 vm_area_struct 结构体
struct vm_area_struct {
    uint64_t vm_start;          /* VMA 对应的用户态虚拟地址的开始   */
    uint64_t vm_end;            /* VMA 对应的用户态虚拟地址的结束   */
    uint64_t vm_flags;          /* VMA 对应的 flags */
    uint64_t vm_content_offset_in_file; /* 对应内容在文件中的偏移量*/
    uint64_t vm_content_size_in_file; /* 对应内容在文件中的大小 */
};
// 在 task_struct 中添加 vmas 柔性数组
struct task_struct {
    uint64_t state;
    uint64_t counter;
    uint64_t priority;
    uint64_t pid;

    struct thread_struct thread;
    pagetable_t pgd;

    uint64_t vma_cnt; // vma的个数
    struct vm_area_struct vmas[0];
};
// 添加函数定义
```

```c
void do_mmap(struct task_struct *task, unsigned long addr, unsigned long length, unsigned long flags, unsigned long vm_content_offset_in_file, unsigned long vm_content_size_in_file);
struct vm_area_struct *find_vma(struct task_struct *task, unsigned long addr);
unsigned long clone(struct pt_regs *regs);
```

实现相关功能函数 find_vma 以及 do_mmap。

```c
//proc.c
// 寻找addr对应的vma
struct vm_area_struct *find_vma(struct task_struct *task, unsigned long addr)
{
    int vma_count = task->vma_cnt;
    for (int i = 0; i < vma_count; i++)
    {
        struct vm_area_struct *move = &(task->vmas[i]);
        if (addr >= move->vm_start && addr < move->vm_end)
        {
            return move;
        }
    }
    return NULL;
}
// 在进程中添加vma
void do_mmap(struct task_struct *current_task, unsigned long addr, unsigned long length, unsigned long flags, unsigned long vm_content_offset_in_file, unsigned long vm_content_size_in_file)
{
    struct vm_area_struct *newvma = &(current_task->vmas[current_task->vma_cnt]);
    newvma->vm_start = addr;
    newvma->vm_end = addr + length;
    newvma->vm_flags = flags;
    newvma->vm_content_offset_in_file = vm_content_offset_in_file;
    newvma->vm_content_size_in_file = vm_content_size_in_file;
    current_task->vma_cnt++;
}
```

### 3.2 page fault handle

### 3.2.1 demand paging

修改初始化进程代码，现在在一开始只初始化一个用户进程，并且不使用create_mapping进行映射，而是使用do_mmap进行映射。要注意的一点是我在vma中存储的flag是其对应页表中的flag，而不是其右移一位的结果，也就是说如果一个vma对应的页表项的permission是可读可写可执行有效的话，那么其vma对应的flag会是(1 << 1) | (1 << 2) | (1 << 3) | 1，而不是1 | (1 << 1) | (1 << 2)，这样做是为了方便do_page_fault函数的实现。对于匿名页的判断，因为只有匿名页对应的vm_content_offset_in_file和vm_content_size_in_file能够同时为0，所以我在初始化匿名页的时候会将这两个值设置为0，在处理page_fault的时候，如果这两个值都是0，那么就是匿名页的映射。

```c
void task_init()
{
    idle = (struct task_struct *)kalloc();
    idle->state = TASK_RUNNING;
    idle->counter = 0;
    idle->priority = 0;
    idle->pid = 0;
    current = idle;
    task[0] = idle;
    // 只初始化一个用户进程
    struct task_struct *launched = task[1] = (struct task_struct *)kalloc();
    launched->state = TASK_RUNNING;
    launched->counter = 0;
    launched->priority = rand();
    launched->pid = 1;
    launched->thread.ra = (unsigned long)__dummy;
    launched->thread.sp = (unsigned long)launched + PGSIZE;
    launched->thread_info = (struct thread_info *)(alloc_page());
    launched->thread_info->kernel_sp = launched->thread.sp;
    launched->thread_info->user_sp = (alloc_page());
    // 用户页表初始化
    pagetable_t pgtbl = (unsigned long *)(alloc_page());
    copy(swapper_pg_dir, pgtbl);
    // 使用do_mmap分配用户栈这个匿名页
    do_mmap(launched, USER_END - PGSIZE, PGSIZE, READABLE | WRITABLE | USER | VALID, 0, 0);
    // set sstatus
    unsigned long sstatus = csr_read(sstatus);
    // set spp bit to 0
    sstatus &= (~(1 << 8));
    // set SPIE bit to 1
    sstatus |= (1 << 5);
    // set SUM bit to 1
    sstatus |= (1 << 18);
    launched->thread.sstatus = sstatus;
    // user mode stack
    launched->thread.sscratch = USER_END;

    // printk("SET [PID = %d] COUNTER = %d\n", launched -> pid, launched -> counter);
    unsigned long satp = csr_read(satp);
    launched->pgd = (pagetable_t)(((satp >> 44) << 44) | (((unsigned long)(pgtbl)-PA2VA_OFFSET) >> 12));
    load_program(launched);
    /* YOUR CODE HERE */
    printk("...proc_init done!\n");
}
```

修改load_program如下,主要也是将create_mapping修改为do_mmap.

```c
static void load_program(struct task_struct *task)
{
    Elf64_Ehdr *ehdr = (Elf64_Ehdr *)uapp_start;

    unsigned long phdr_start = (unsigned long)ehdr + ehdr->e_phoff;
    int phdr_cnt = ehdr->e_phnum;

    Elf64_Phdr *phdr;
    for (int i = 0; i < phdr_cnt; i++)
    {
        phdr = (Elf64_Phdr *)(phdr_start + sizeof(Elf64_Phdr) * i);
        if (phdr->p_type == PT_LOAD)
        {
            unsigned long va = phdr->p_vaddr;
            do_mmap(task, PGROUNDDOWN(va), phdr->p_memsz, (phdr->p_flags << 1) | USER | VALID, phdr->p_offset, phdr->p_filesz);
        }
    }
    task->thread.sepc = ehdr->e_entry;
}
```

### 3.2.2 do_page_fault

这里是对page_fault的处理。

首先修改trap.c，添加对page_fault的处理，如果是异常，并且满足 `scause == PAGE_INSTRUCTION_FAULT || scause == PAGE_LOAD_FAULT || scause == PAGE_STORE_FAULT` ，那么说明是page_fault。

```c
#include "clock.h"
#include "printk.h"
```

```c
#include "proc.h"
#include "syscall.h"
#include "pagefault.h"
void trap_handler(unsigned long scause, unsigned long sepc, struct pt_regs *regs)
{
    if ((scause >> 63) & 1)
    {
        if ((scause & 5) == 5)
        {
            clock_set_next_event();
            do_timer();
        }
    }
    else
    {
        if (scause == 8)
        {
            syscall(regs);
        }
        else if (scause == PAGE_INSTRUCTION_FAULT || scause == PAGE_LOAD_FAULT || scause == PAGE_STORE_FAULT)
        {
            do_page_fault(regs);
        }
    }
}

```

然后是do_page_fault函数的实现，这里首先查询stval和cause，然后遍历进程的vma，找到stval所在的vma，如果没找到，那么输出错误信息并返回，如果找到，那么检查其是否满足权限，如果满足权限，那么检查是否是匿名页，如果是匿名页，因为在本实验中只有用户栈是匿名页映射，所以这里就直接将用户栈页进行映射；否则就对对应的磁盘文件进行映射。

```c
#include "pagefault.h"
#include "defs.h"
#include "proc.h"
#include "vm.h"
#include "printk.h"
#include "mm.h"
extern struct task_struct *current;
extern char uapp_start[];
void do_page_fault(struct pt_regs *regs)
{
    unsigned long stval = csr_read(stval);
    unsigned long scause = csr_read(scause);
    for (int i = 0; i < current->vma_cnt; i++)
    {
        struct vm_area_struct *move = (current->vmas + i);
        if (move->vm_start <= stval && move->vm_end > stval)
        {
            // 非法访问
            if (scause == PAGE_INSTRUCTION_FAULT && ((move->vm_flags & EXECUTABLE) == 0))
            {
                printk("invalid mem access for PAGE_INSTRUCTION_FAULT\n");
                return;
            }
            if (scause == PAGE_LOAD_FAULT && ((move->vm_flags & READABLE) == 0))
            {
                printk("invalid mem access for PAGE_LOAD_FAULT\n");
                return;
            }
            if (scause == PAGE_STORE_FAULT && ((move->vm_flags & WRITABLE) == 0))
            {
                printk("invalid mem access for PAGE_INSTRUCTION_FAULT\n");
                return;
            }
            pagetable_t pgtbl = (unsigned long *)(((((unsigned long)(current->pgd)) & 0xfffffffffff) << 12) + PA2VA_OFFSET);
            // 检查是否是匿名页
            if (move->vm_content_offset_in_file == 0 && move->vm_content_size_in_file == 0)
            {
                create_mapping(pgtbl, PGROUNDDOWN(move->vm_start), current->thread_info->user_sp - PA2VA_OFFSET, (unsigned long)(move->vm_end - move->vm_start), move->vm_flags);
                return;
            }
            else
            {
                // 磁盘文件
                unsigned long memPages = (PGROUNDUP(move->vm_end - move->vm_start)) / PGSIZE;
                // 复制代码段
                unsigned long mappedPage = alloc_pages(memPages);
                for (int j = 0; j < PGROUNDUP(move->vm_content_size_in_file) / 8; j++)
                {
                    ((unsigned long *)(mappedPage))[j] = ((unsigned long *)(PGROUNDDOWN(((unsigned long)(uapp_start + move->vm_content_offset_in_file)))))[j];
                }
                // 映射多余内存段
                create_mapping(pgtbl, move->vm_start, (unsigned long)(mappedPage - PA2VA_OFFSET), (memPages)*PGSIZE, move->vm_flags);
                return;
            }
        }
    }
    printk("[page fault] invalid access\n");
    return;
}
```

## 3.3 fork实现

在syscall.h中添加SYS_CLONE。

```c
...
    #define SYS_CLONE 220 // fork
...
```

在syscall函数中添加对sys_clone的处理。

```c
...
unsigned long sys_clone(struct pt_regs *regs)
{
    return clone(regs);
}
void syscall(struct pt_regs *regs)
{
    unsigned long sys_code = regs->a7;
    int return_value = -1;
    switch (sys_code)
    {
    ...
    case SYS_CLONE:
    {
        return_value = sys_clone(regs);
        break;
    }
    }
    // write result to a0
    regs->a0 = return_value;
    regs->sepc += 4;
}
```

clone函数在proc.c中，其调用proc.c中的do_fork函数。

```c
unsigned long clone(struct pt_regs *regs)
{
    return do_fork(regs);
}
unsigned long do_fork(struct pt_regs *regs)
{
    // 进程数++
    task_count++;
    int i = task_count;
    struct task_struct *child = task[i] = (struct task_struct *)kalloc();
    // 拷贝父进程整个page
    for (int i = 0; i < PGSIZE / 8; i++)
    {
        ((unsigned long *)child)[i] = ((unsigned long *)current)[i];
    }
    // 设置子进程相关属性
    child->counter = 0;
    child->priority = rand();
    child->pid = i;
    // 设置ra为__ret_from_fork函数地址
    child->thread.ra = (unsigned long)(__ret_from_fork);
    // 这里设置sp为一个看上去比较奇怪的量，解释如下，事实上父进程在进入trap_handler之前，将自己的中断上下文，包括32个寄存器和几个状态寄存器都存储在了地址为current + ((unsigned long)(regs)-PGROUNDDOWN((unsigned long)(current)))的位置，由于子进程这时候已经对父进程进行了整个page的拷贝，所以子进程可以通过地址(unsigned long)child + (unsigned long)(((unsigned long)(regs)-PGROUNDDOWN((unsigned long)(current))))来获取父进程的中断上下文，这里将sp设置为它的地址，方便__ret_form_fork的逻辑实现
    child->thread.sp = (unsigned long)child + (unsigned long)(((unsigned long)(regs)-PGROUNDDOWN((unsigned long)(current))));
    child->thread_info = (struct thread_info *)(alloc_page());
    child->thread_info->kernel_sp = child->thread.sp;
    // 拷贝父进程栈
    child->thread_info->user_sp = (alloc_page());
    unsigned long parent_user_stack = (unsigned long)(csr_read(sscratch));
    for (int j = 0; j < PGSIZE / 8; j++)
    {
        ((unsigned long *)(child->thread_info->user_sp))[j] = ((unsigned long *)(PGROUNDDOWN(parent_user_stack)))[j];
    }

    // 用户页表初始化
    pagetable_t pgtbl = (unsigned long *)(alloc_page());
    copy(swapper_pg_dir, pgtbl);
    // 拷贝父进程的mmap
    child->vma_cnt = current->vma_cnt;
    for (int j = 0; j < current->vma_cnt; j++)
    {
        child->vmas[j] = current->vmas[j];
    }
    // 如果有mmap已经映射，那么进行深拷贝
    for (int i = 0; i < current->vma_cnt; i++)
    {
        unsigned long pte = walk(current, current->vmas[i].vm_start);
        if (pte & VALID)
        {
            // 已经mapping，进行拷贝
            unsigned long *child_pages = (unsigned long *)(alloc_pages(PGROUNDUP((unsigned long)(current->vmas[i].vm_end) - (unsigned long)(current->vmas[i].vm_start)) / PGSIZE));
            for (int j = 0; j < (PGROUNDUP((unsigned long)(current->vmas[i].vm_end) - (unsigned long)(current->vmas[i].vm_start)) / 8); j++)
            {
                child_pages[j] = ((unsigned long *)((unsigned long)current->vmas[i].vm_start))[j];
            }
            // 执行mapping
            create_mapping(pgtbl, current->vmas[i].vm_start, (unsigned long)child_pages - PA2VA_OFFSET, PGROUNDUP((unsigned long)(current->vmas[i].vm_end) - (unsigned long)(current->vmas[i].vm_start)), current->vmas[i].vm_flags);
        }
    }
    // 修改frame中对应的值
    struct pt_regs *frame = (struct pt_regs *)((unsigned long)child + (unsigned long)(regs) - (unsigned long)(current));
    frame->a0 = 0;
    // 修改frame中sp的值为sscratch，事实上就是父进程用户栈的位置
    frame->sp = csr_read(sscratch);
    frame->sepc += 4;

    unsigned long sstatus = csr_read(sstatus);
    child->thread.sstatus = sstatus;

    child->thread.sscratch = child->thread_info->kernel_sp;

    unsigned long satp = csr_read(satp);
    child->pgd = (pagetable_t)(((satp >> 44) << 44) | (((unsigned long)(pgtbl)-PA2VA_OFFSET) >> 12));
    return i;
}
```

实现__ret_from_fork函数，在entry.S文件中实现。

```asm
.global __ret_from_fork
__ret_from_fork:
    # 这里的sp就是上面说的中断上下文的首地址，子进程从这里对自己的寄存器进行设置。
    ld t0,32 * 8(sp)
    csrw sepc,t0
    ld t0,33 * 8(sp)
    csrw sstatus, t0

    ld x1, 1 * 8(sp)
    ld x3, 3 * 8(sp)
    ld x4, 4 * 8(sp)
    ld x5, 5 * 8(sp)
    ld x6, 6 * 8(sp)
    ld x7, 7 * 8(sp)
    ld x8, 8 * 8(sp)
    ld x9, 9 * 8(sp)
    ld x10, 10 * 8(sp)
    ld x11, 11 * 8(sp)
    ld x12, 12 * 8(sp)
    ld x13, 13 * 8(sp)
    ld x14, 14 * 8(sp)
    ld x15, 15 * 8(sp)
    ld x16, 16 * 8(sp)
    ld x17, 17 * 8(sp)
    ld x18, 18 * 8(sp)
    ld x19, 19 * 8(sp)
    ld x20, 20 * 8(sp)
    ld x21, 21 * 8(sp)
    ld x22, 22 * 8(sp)
    ld x23, 23 * 8(sp)
    ld x24, 24 * 8(sp)
    ld x25, 25 * 8(sp)
    ld x26, 26 * 8(sp)
    ld x27, 27 * 8(sp)
    ld x28, 28 * 8(sp)
    ld x29, 29 * 8(sp)
    ld x30, 30 * 8(sp)
    ld x31, 31 * 8(sp)
    ld sp, 2 * 8(sp)
    #-- -- -- -- -- -
    sret
```

程序运行效果如下。

```
SET [PID = 4 COUNTER = 9]
switch to [PID = 3 COUNTER = 4]
[U] pid: 3 is running!, global_variable: 201
[U] pid: 3 is running!, global_variable: 202
[U] pid: 3 is running!, global_variable: 203
[U] pid: 3 is running!, global_variable: 204
[U] pid: 3 is running!, global_variable: 205
switch to [PID = 1 COUNTER = 8]
[U] pid: 1 is running!, global_variable: 229
[U] pid: 1 is running!, global_variable: 230
[U] pid: 1 is running!, global_variable: 231
[U] pid: 1 is running!, global_variable: 232
[U] pid: 1 is running!, global_variable: 233
[U] pid: 1 is running!, global_variable: 234
[U] pid: 1 is running!, global_variable: 235
[U] pid: 1 is running!, global_variable: 236
[U] pid: 1 is running!, global_variable: 237
[U] pid: 1 is running!, global_variable: 238
switch to [PID = 4 COUNTER = 9]
[U] pid: 4 is running!, global_variable: 184
[U] pid: 4 is running!, global_variable: 185
[U] pid: 4 is running!, global_variable: 186
[U] pid: 4 is running!, global_variable: 187
[U] pid: 4 is running!, global_variable: 188
[U] pid: 4 is running!, global_variable: 189
[U] pid: 4 is running!, global_variable: 190
[U] pid: 4 is running!, global_variable: 191
[U] pid: 4 is running!, global_variable: 192
[U] pid: 4 is running!, global_variable: 193
[U] pid: 4 is running!, global_variable: 194
[U] pid: 4 is running!, global_variable: 195
switch to [PID = 2 COUNTER = 10]
[U] pid: 2 is running!, global_variable: 243
[U] pid: 2 is running!, global_variable: 244
[U] pid: 2 is running!, global_variable: 245
[U] pid: 2 is running!, global_variable: 246
[U] pid: 2 is running!, global_variable: 247
[U] pid: 2 is running!, global_variable: 248
[U] pid: 2 is running!, global_variable: 249
[U] pid: 2 is running!, global_variable: 250
[U] pid: 2 is running!, global_variable: 251
```

## 4. 思考题

### 4.1 `uint64_t vm_content_size_in_file;` 对应的文件内容的长度。为什么还需要这个域?

答案：这个问题事实上在lab5的报告中已经回答了，在装载用户程序的时候，lab6使用的是将elf文件加载到内存中，因为在 elf 文件中的load的段的p_memsz和p_filesz一般并不相等，一个 load 段可能有为无需初始化的变量或者数组等数据，这些数据都会在程序装载的时候 初始化为 0，如果在 elf 文件中为这些数据预留空间的话，就会造成磁盘空间的浪费。但是这些数据在内存中是需要对应的空间来保障程序运行的，所以在许多情况下 p_memsz 会稍大于 p_filesz 。所以我们必须分别记录每个段的文件中的大小和内存中的起止位置，才能够进行page_fault的处理。

### 4.2 `struct vm_area_struct vmas[0];` 为什么可以开大小为 0 的数组? 这个定义可以和前面的 vma_cnt 换个位置吗?

答案：这是gcc支持的一个特性，叫做柔性数组，允许我们在运行的时候通过申请不同的内存大小来动态开辟数组，这个定义不能和vma_cnt交换，这是柔性数组本身规定的，其必须在结构的最后。

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

答案：对于那些父进程已经分配和映射的页面，很有可能父进程已经对其进行了修改等写入行为，如果我们在fork的时候不进行拷贝的话，那么子进程在返回用户区间运行的时候，就可能因为读取到和父进程不同的值，导致运行出错，一个例子就是父进程在0x10200写入了一个32（原来是-1），然后在fork之后通过读取这个32来执行程序，子进程如果不进行拷贝，从磁盘上拷贝0x10020对应的vma，之后读取到了这个-1，那么本来子进程应该和父进程读取到一样的值的，这里却不一样了，这就可能导致程序运行出错。

　　对于那些还没被映射的页，由于父进程没有通过do_page_fault进行映射，那么其肯定也没有修改对应page中的值，子进程也不需要拷贝，直接从磁盘中读取即可。

### 4.4 参考 task_init 创建一个新的 task，将的 parent task 的整个页复制到新创建的 task_struct 页上，这一步复制了哪些东西?

答案：事实上这一步是对整个task_struct的复制，从task_struct的结构定义可以看出，这里主要是复制了线程的ra、sp、12个s寄存器，sepc寄存器，sstatus寄存器，sscratch寄存器，以及父进程的vma。当然也复制了父进程状态，计数器，页表等。

### 4.5 将 thread.ra 设置为 `__ret_from_fork` ，并正确设置 `thread.sp` 。仔细想想，这个应该设置成什么值?可以根据 child task 的返回路径来倒推。

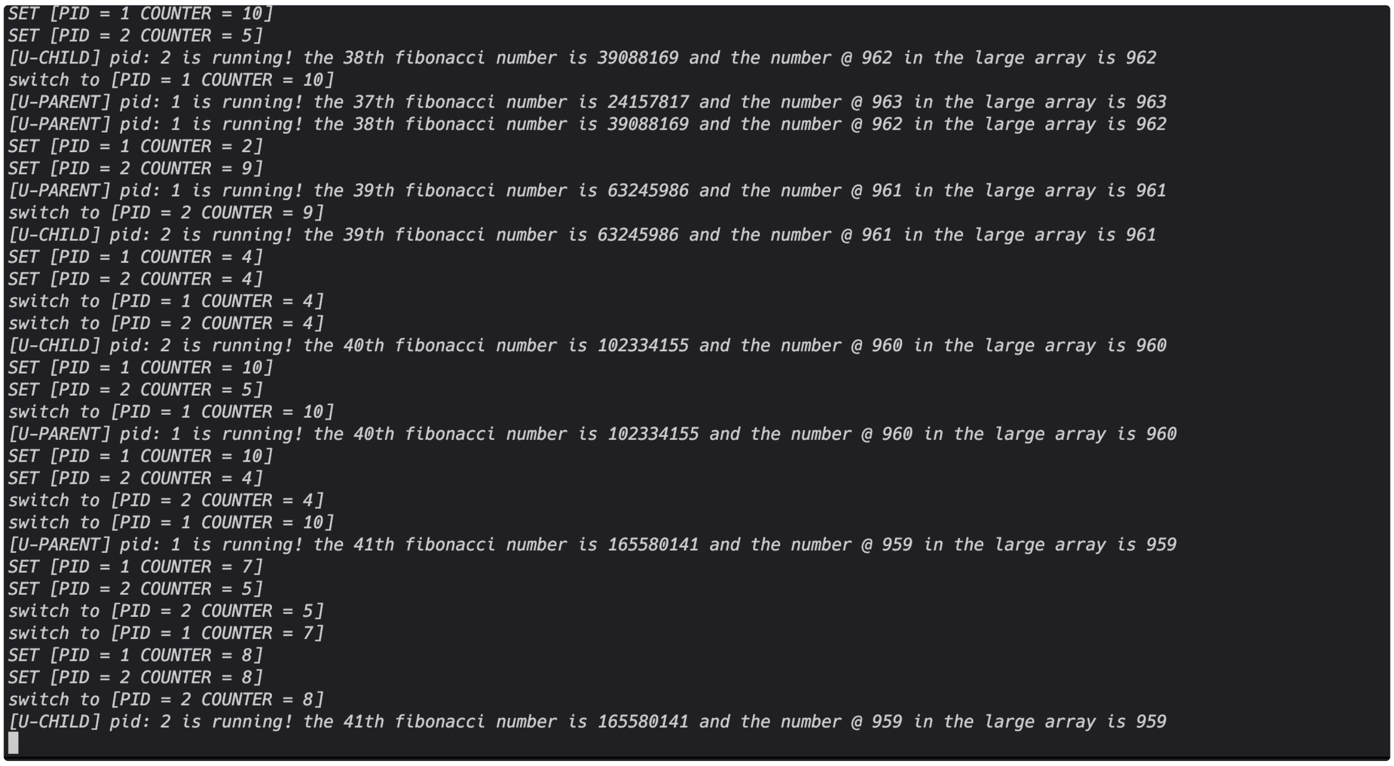
答案：这里我将sp设置为 `(unsigned long)child + (unsigned long)(((unsigned long)(regs)-PGROUNDDOWN((unsigned long)(current))))` ，事实上，regs是父进程存储中断上下文的地址，由于child在此前已经对父进程的整个task_struct结构体进行了拷贝，所以这里 `(unsigned long)child + (unsigned long)(((unsigned long)(regs)-PGROUNDDOWN((unsigned long)(current))))` 指向的就是子进程中存储父进程中断上下文的地方，我们后续可以通过这个地址，在__ret_from_fork中进行子进程状态的恢复。

**4.6 利用参数 regs 来计算出 child task 的对应的 pt_regs 的地址，并将其中的 a0, sp, sepc 设置成正确的值。为什么还要设置 sp?**

答案：这里的sp事实上需要是父进程用户栈的虚拟地址，如果不设置的话，那么存储的会是父进程kernel栈在存储中断上下文时的地址，导致程序运行出错。在我的实现中，是在fork的逻辑里将sp设置为csr_read(sscratch)，也就是父进程栈的地址。

## 5. 更多测试用例

使用了实验手册中提供的斐波那契函数进行测试，程序运行无误。



```
SET [PID = 1 COUNTER = 10]
SET [PID = 2 COUNTER = 5]
[U-CHILD] pid: 2 is running! the 38th fibonacci number is 39088169 and the number @ 962 in the large array is 962
switch to [PID = 1 COUNTER = 10]
[U-PARENT] pid: 1 is running! the 37th fibonacci number is 24157817 and the number @ 963 in the large array is 963
[U-PARENT] pid: 1 is running! the 38th fibonacci number is 39088169 and the number @ 962 in the large array is 962
SET [PID = 1 COUNTER = 2]
SET [PID = 2 COUNTER = 9]
[U-PARENT] pid: 1 is running! the 39th fibonacci number is 63245986 and the number @ 961 in the large array is 961
switch to [PID = 2 COUNTER = 9]
[U-CHILD] pid: 2 is running! the 39th fibonacci number is 63245986 and the number @ 961 in the large array is 961
SET [PID = 1 COUNTER = 4]
SET [PID = 2 COUNTER = 4]
switch to [PID = 1 COUNTER = 4]
switch to [PID = 2 COUNTER = 4]
[U-CHILD] pid: 2 is running! the 40th fibonacci number is 102334155 and the number @ 960 in the large array is 960
SET [PID = 1 COUNTER = 10]
SET [PID = 2 COUNTER = 5]
switch to [PID = 1 COUNTER = 10]
[U-PARENT] pid: 1 is running! the 40th fibonacci number is 102334155 and the number @ 960 in the large array is 960
SET [PID = 1 COUNTER = 10]
SET [PID = 2 COUNTER = 4]
switch to [PID = 2 COUNTER = 4]
switch to [PID = 1 COUNTER = 10]
[U-PARENT] pid: 1 is running! the 41th fibonacci number is 165580141 and the number @ 959 in the large array is 959
SET [PID = 1 COUNTER = 7]
SET [PID = 2 COUNTER = 5]
switch to [PID = 2 COUNTER = 5]
switch to [PID = 1 COUNTER = 7]
SET [PID = 1 COUNTER = 8]
SET [PID = 2 COUNTER = 8]
switch to [PID = 2 COUNTER = 8]
[U-CHILD] pid: 2 is running! the 41th fibonacci number is 165580141 and the number @ 959 in the large array is 959
```