浙江大学

# 本科实验报告

| | |
|---|---|
| 课程名称： | 操作系统 |
| 姓　　名： | 董佳鑫 |
| 学　　院： | 计算机学院 |
| 系： | 计算机系 |
| 专　　业： | 计算机科学与技术 |
| 学　　号： | 3210102181 |
| 指导教师： | 寿黎但 |

# Lab 6：RV64 fork 机制

## 实验目的

1.为 task 加入 fork 机制，能够支持通过 fork 创建新的用户态 task。

## 实验过程和操作步骤

### 1.添加 fork 系统调用

在 syscall.c 中添加新的系统调用 fork。

```
struct task_struct* child = (struct task_struct*)kalloc();
memcpy(child,current,PGSIZE);
child->thread.ra = (uint64_t)(&__ret_from_fork);
int i;
for(i = 2;i < NR_TASKS;i++) if(task[i] == NULL) break;
child->pid = i;
task[i] = child;
uint64 offset = PGOFFSET(((uint64)regs));
struct pt_regs* child_regs = (struct pt_regs*)(child + offset);
child->thread.sp = (uint64_t)child_regs;
memcpy(child_regs,regs,sizeof(struct pt_regs));
child_regs->reg[10-1] = 0;
child_regs->reg[2-1] = (uint64)child_regs;
child_regs->sepc = regs->sepc ;
uint64 u_stack = kalloc();
memcpy(u_stack,USER_END - PGSIZE,PGSIZE);
pagetable_t pgtbl = kalloc();
memset(pgtbl,0,PGSIZE);
memcpy(pgtbl,swapper_pg_dir,PGSIZE);
create_mapping_sub(pgtbl,USER_END - PGSIZE,u_stack - PA2VA_OFFSET, PGSIZE, 0x17);
child->pgd = (8 << 60) | ((unsigned long)pgtbl - PA2VA_OFFSET) >> 12;
for(int j = 0;j < current->vma_cnt;j++){
  struct vm_area_struct* vma = &(current->vmas[j]);
  uint64 addr = vma->vm_start;
  while ((addr < vma->vm_end))
  {
   if(IsMapped(pgtbl,addr)){
   uint64 page = kalloc();
   create_mapping_sub(pgtbl,PGROUNDDOWN(addr),page - PA2VA_OFFSET,PGSIZE,(vma->vm_flags & (~(uint64_t)VM_ANONYM)) | 0x11);
   memcpy(page,PGROUNDDOWN(addr),PGSIZE);
   }
   addr += PGSIZE;
  }
}
printk("[S-MODE] New task:PID = %d\n",i);
regs->reg[10-1] = i;
```

按照实验手册的逻辑完成 fork 系统调用。其中值得注意的地方有：关于子进程 thread.sp 的值应该按照父进程 pt_regs 的位置进行设置，这在下面的思考题会具体阐述；然后我们要对一块新申请的内存作为子进程的用户态栈，并且把这个栈建立映射，然后拷贝父进程的内容。在设置好必要的变量后，申请并建立对应的页表，这和 lab3 时建立页表映射类似。然后要根据父进程 VMA 的映射情况，子进程也随之进行映射。这里会调用函数 IsMapped，这个函数的作用主要就是查看目标地址是否能根据三级页表找到对应的物理地址，如果找到了就说明在父进程中已经建立起了页表，此时子进程也要随之建立页表。IsMapped 函数实现如下：

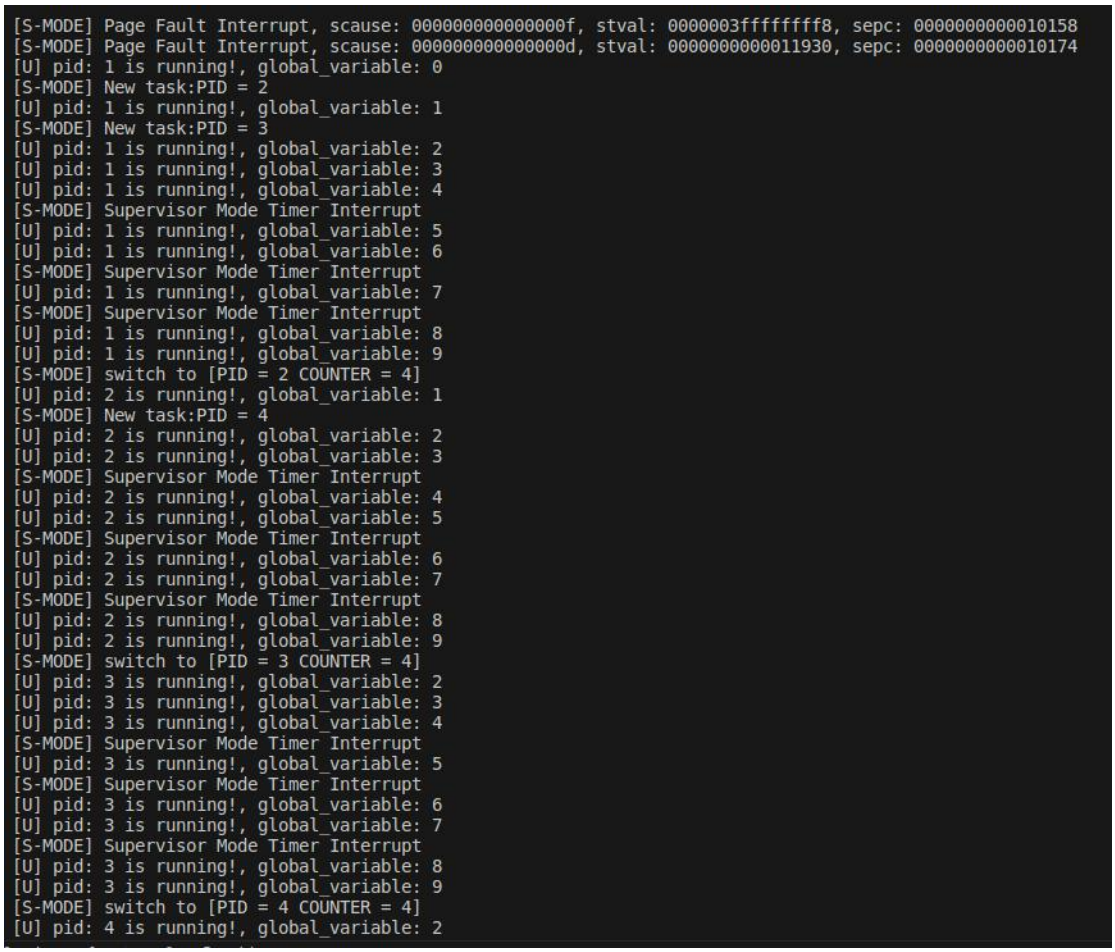
```c
int IsMapped(uint64* ptb,uint64 addr){

    uint64 vpn_2 = (addr >> 30) & 0x1ff;
    uint64 *level2;
    if(ptb[vpn_2] & 0x1 ){
        uint64 ppn = (ptb[vpn_2] >> 10) & 0xfffffffffff;
        level2 = (uint64)(ppn << 12);
        level2 = (uint64)level2 + PA2VA_OFFSET;
    }else{
        return 0;
    }
    uint64 *level3;
    uint64 vpn_1 = (addr >> 21) & 0x1ff;
    if(level2[vpn_1] & 0x1 ){
        uint64 ppn = (level2[vpn_1] >> 10) & 0xfffffffffff;
        level3 = (uint64)(ppn << 12);
        level3 = (uint64)level3 + PA2VA_OFFSET;
    }else{
        return 0;
    }
    uint64 vpn_0 = (addr >> 12) & 0x1ff;
    if( level3[vpn_0] & 0x1) {
        return 1;
    }else{
        return 0;
    }
}
```

完成上述代码，并且完成必要的修改以后就能直接运行测试，下面是分别运行 3 个 main 函数后的结果：

```
[S-MODE] Page Fault Interrupt, scause: 000000000000000d, stval: 0000000000011000, sepc: ffffffe000202dc8
[S-MODE] New task:PID = 2
[U-PARENT] pid: 1 is running!, global_variable: 0
[U-PARENT] pid: 1 is running!, global_variable: 1
[S-MODE] Supervisor Mode Timer Interrupt
[U-PARENT] pid: 1 is running!, global_variable: 2
[U-PARENT] pid: 1 is running!, global_variable: 3
[S-MODE] Supervisor Mode Timer Interrupt
[U-PARENT] pid: 1 is running!, global_variable: 4
[U-PARENT] pid: 1 is running!, global_variable: 5
[S-MODE] Supervisor Mode Timer Interrupt
[U-PARENT] pid: 1 is running!, global_variable: 6
[S-MODE] switch to [PID = 2 COUNTER = 4]
[U-CHILD] pid: 2 is running!, global_variable: 0
[U-CHILD] pid: 2 is running!, global_variable: 1
[S-MODE] Supervisor Mode Timer Interrupt
[U-CHILD] pid: 2 is running!, global_variable: 2
[U-CHILD] pid: 2 is running!, global_variable: 3
[S-MODE] Supervisor Mode Timer Interrupt
[U-CHILD] pid: 2 is running!, global_variable: 4
[U-CHILD] pid: 2 is running!, global_variable: 5
[S-MODE] Supervisor Mode Timer Interrupt
```

```
[S-MODE] Page Fault Interrupt, scause: 000000000000000c, stval: 00000000000100e8, sepc: 00000000000100e8
[S-MODE] Page Fault Interrupt, scause: 000000000000000f, stval: 0000003fffffff8, sepc: 0000000000010158
[S-MODE] Page Fault Interrupt, scause: 000000000000000d, stval: 0000000000011a00, sepc: 000000000001017c
[U] pid: 1 is running!, global_variable: 0
[U] pid: 1 is running!, global_variable: 1
[U] pid: 1 is running!, global_variable: 2
[S-MODE] New task:PID = 2
[U-PARENT] pid: 1 is running!, global_variable: 3
[U-PARENT] pid: 1 is running!, global_variable: 4
[S-MODE] Supervisor Mode Timer Interrupt
[U-PARENT] pid: 1 is running!, global_variable: 5
[U-PARENT] pid: 1 is running!, global_variable: 6
[S-MODE] Supervisor Mode Timer Interrupt
[U-PARENT] pid: 1 is running!, global_variable: 7
[U-PARENT] pid: 1 is running!, global_variable: 8
[S-MODE] Supervisor Mode Timer Interrupt
[U-PARENT] pid: 1 is running!, global_variable: 9
[U-PARENT] pid: 1 is running!, global_variable: 10
[S-MODE] switch to [PID = 2 COUNTER = 4]
[U-CHILD] pid: 2 is running!, global_variable: 3
[U-CHILD] pid: 2 is running!, global_variable: 4
[S-MODE] Supervisor Mode Timer Interrupt
[U-CHILD] pid: 2 is running!, global_variable: 5
```




## 思考题

1. 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?

每次创建 task 的时候，都会申请一个 page，这个 page 中包括各种构成 struct_task 的变量，以及内核态栈的数据。因此我们在复制的时候，会把这个 page 中的所有数据全部复制到新的 task 中，包括 task_struct 和 pt_regs 结构体中的数据。

2. 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。

thread.sp 应该利用参数 regs 来计算出 child task 的对应的 pt_regs 的地址，也就是 child + PGOFFSET(regs)。Child task 的返回路径是 __switch_to->__ret_from_fork->user program，而在__ret_from_fork（traps）中，所用到的 sp 应该是内核态的指针，也就是指向各自结构体 pt_regs 的指针，因此应该利用参数的 regs 来计算 child task 的对应的 pt_regs 地址。

3.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?

因为我们在返回到__ret_from_fork(_traps)中，需要 restore 寄存器的值，这就包括了 sp。因此我们的 regs->reg[2](sp)需要设置为指向 pt_regs 的指针，这才能保证 restore 以后还能正常找到正确的 sp。

## 讨论心得

本次实验遇到的问题是，在设置 child task 的时候，有很多变量的设置需要慎重考虑。我在这里遇到的问题是，对于父进程和子进程的程序流不太清楚，在认真学习和思考以后，搞懂了其中的原理后也顺便复习了操作系统的期末知识。

本次实验是我们完成的最后一个 OS 实验，回想这一个学期的实验旅程，实在是颇为感慨。难度实在是不小，不过收获也颇丰。从内核引导，到时钟中断处理，到虚拟内存创建，到用户态线程，再到缺页异常处理和 fork 机制。自己在这一个学期一步步建立了一个操作系统，也是蛮有成就感的一件事。